Python Documentation

Release 2.5 19th September, 2006

- **Tutorial**
  (start here)

- **What's New in Python**
  (changes since the last major release)

- **Global Module Index**
  (for quick access to all documentation)

- **Library Reference**
  (keep this under your pillow)

- **Macintosh Module Reference**
  (this too, if you use a Macintosh)

- **Installing Python Modules**
  (for administrators)

- **Distributing Python Modules**
  (for developers and packagers)

- **Language Reference**
  (for language lawyers)

- **Extending and Embedding**
  (tutorial for C/C++ programmers)

- **Python/C API**
  (reference for C/C++ programmers)

- **Documenting Python**
  (information for documentation authors)

- **Documentation Central**
  (for everyone)

- **Python How-To Guides**
  (special topics)

---

See [About the Python Documentation](#) for information on suggesting changes.
Global Module Index

Some module names are followed by an annotation indicating what platform they are available on.

```python
__builtin__
__future__
__main__
_winreg (Windows)
aepack (Mac)
aetools (Mac)
aetypes (Mac)
aifc
AL (IRIX)
al (IRIX)
anydbm
applesingle (Mac)
array
asynchat
asyncore
atexit
audioop
autoGIL (Mac)
base64
BaseHTTPServer
Bastion
binascii
binhex
bisect
bsddb (Unix, Windows)
bz2
calendar
Carbon.AE (Mac)
Carbon.AH (Mac)
Carbon.App (Mac)
distutils.emxccompiler
distutils.errors
distutils.extension
distutils.fancy_getopt
distutils.file_util
distutils.filelist
distutils.log
distutils.msvccompiler
distutils.mwerkscomp
distutils.spawn
distutils.sysconfig
distutils.text_file
distutils.unixxcompil
distutils.util
distutils.version
dl (Unix)
doctest
DocXMLRPCServer
dumbdbm
dummy_thread
dummy_threading
EasyDialogs (Mac)
email
email.charset
e-mail.encoders
e-mail.errors
e-mail.generator
e-mail.header
e-mail.iterators
e-mail.message
```
<table>
<thead>
<tr>
<th>Module</th>
<th>macOS</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>distutils.command.build</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.command.build_clib</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.command.build_ext</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.command.build_py</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.command.build_scripts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.command.clean</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.command.config</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.command.install</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.command.install_data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.command.install_headers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.command.install_lib</td>
<td></td>
<td>Nav (Mac)</td>
</tr>
<tr>
<td>distutils.command.install_scripts</td>
<td></td>
<td>netrc</td>
</tr>
<tr>
<td>distutils.command.register</td>
<td></td>
<td>new</td>
</tr>
<tr>
<td>distutils.command.sdist</td>
<td>mimetypes</td>
<td></td>
</tr>
<tr>
<td>distutils.core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.cygwinccompiler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.debug</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.dep_util</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.dir_util</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distutils.dist</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See [About this document...](...) for information on suggesting changes.
1 PEP 308: Conditional Expressions

For a long time, people have been requesting a way to write conditional expressions, which are expressions that return value A or value B depending on whether a Boolean value is true or false. A conditional expression lets you write a single assignment statement that has the same effect as the following:

```python
if condition:
    x = true_value
else:
    x = false_value
```

There have been endless tedious discussions of syntax on both python-dev and comp.lang.python. A vote was even held that found the majority of voters wanted conditional expressions in some form, but there was no syntax that was preferred by a clear majority. Candidates included C's `cond ? true_v : false_v`, `if cond then true_v else false_v`, and 16 other variations.

Guido van Rossum eventually chose a surprising syntax:

```python
x = true_value if condition else false_value
```

Evaluation is still lazy as in existing Boolean expressions, so the order of evaluation jumps around a bit. The `condition` expression in the middle is evaluated first, and the `true_value` expression is evaluated only if the condition was true. Similarly, the `false_value` expression is only evaluated when the condition is false.

This syntax may seem strange and backwards; why does the condition go in the `middle` of the expression, and not in the front as in C's `c ? x : y`? The decision was checked by applying the new syntax to the modules in the standard library and seeing how the resulting code read. In many cases where a conditional expression is used, one value seems to be the 'common case' and one value is an 'exceptional case', used only on rarer occasions when the condition isn't met. The conditional syntax makes this pattern a bit more obvious:

```python
contents = ((doc + '\n') if doc else '')
```
I read the above statement as meaning "here contents is usually assigned a value of doc+' \n '; sometimes doc is empty, in which special case an empty string is returned." I doubt I will use conditional expressions very often where there isn't a clear common and uncommon case.

There was some discussion of whether the language should require surrounding conditional expressions with parentheses. The decision was made to not require parentheses in the Python language's grammar, but as a matter of style I think you should always use them. Consider these two statements:

```python
# First version -- no parens
level = 1 if logging else 0

# Second version -- with parens
level = (1 if logging else 0)
```

In the first version, I think a reader's eye might group the statement into 'level = 1', 'if logging', 'else 0', and think that the condition decides whether the assignment to level is performed. The second version reads better, in my opinion, because it makes it clear that the assignment is always performed and the choice is being made between two values.

Another reason for including the brackets: a few odd combinations of list comprehensions and lambdas could look like incorrect conditional expressions. See PEP 308 for some examples. If you put parentheses around your conditional expressions, you won't run into this case.

See Also:

PEP 308, Conditional Expressions

PEP written by Guido van Rossum and Raymond D. Hettinger; implemented by Thomas Wouters.
2 PEP 309: Partial Function Application

The `functools` module is intended to contain tools for functional-style programming.

One useful tool in this module is the `partial()` function. For programs written in a functional style, you'll sometimes want to construct variants of existing functions that have some of the parameters filled in. Consider a Python function \( f(a, b, c) \); you could create a new function \( g(b, c) \) that was equivalent to \( f(1, b, c) \). This is called "partial function application".

`partial` takes the arguments `(function, arg1, arg2, ... kwarg1=value1, kwarg2=value2)`. The resulting object is callable, so you can just call it to invoke `function` with the filled-in arguments.

Here's a small but realistic example:

```python
import functools

def log(message, subsystem):
    "Write the contents of 'message' to the specified subsystem."
    print '%s: %s' % (subsystem, message)
...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

Here's another example, from a program that uses PyGTK. Here a context-sensitive pop-up menu is being constructed dynamically. The callback provided for the menu option is a partially applied version of the `open_item()` method, where the first argument has been provided.

```python
...
class Application:
    def open_item(self, path):
        ...
    def init (self):
        open_func = functools.partial(self.open_item, item_path)
        popup_menu.append( ('Open', open_func, 1) )
```
Another function in the `functools` module is the `update_wrapper(wrapper, wrapped)` function that helps you write well-behaved decorators. `update_wrapper()` copies the name, module, and docstring attribute to a wrapper function so that tracebacks inside the wrapped function are easier to understand. For example, you might write:

```python
def my_decorator(f):
    def wrapper(*args, **kwds):
        print 'Calling decorated function'
        return f(*args, **kwds)
    functools.update_wrapper(wrapper, f)
    return wrapper
```

`wraps()` is a decorator that can be used inside your own decorators to copy the wrapped function's information. An alternate version of the previous example would be:

```python
def my_decorator(f):
    @functools.wraps(f)
    def wrapper(*args, **kwds):
        print 'Calling decorated function'
        return f(*args, **kwds)
    return wrapper
```

See Also:

**PEP 309, Partial Function Application**

PEP proposed and written by Peter Harris; implemented by Hye-Shik Chang and Nick Coghlan, with adaptations by Raymond Hettinger.
3 PEP 314: Metadata for Python Software Packages v1.1

Some simple dependency support was added to Distutils. The setup() function now has requires, provides, and obsoletes keyword parameters. When you build a source distribution using the sdist command, the dependency information will be recorded in the PKG-INFO file.

Another new keyword parameter is download_url, which should be set to a URL for the package's source code. This means it's now possible to look up an entry in the package index, determine the dependencies for a package, and download the required packages.

```python
VERSION = '1.0'
setup(name='PyPackage',
      version=VERSION,
      requires=['numarray', 'zlib (>=1.1.4)'],
      obsoletes=['OldPackage']
      download_url=('http://www.example.com/pypackage/dist/pkg-%s.tar.gz' % VERSION),
)
```

Another new enhancement to the Python package index at [http://cheeseshop.python.org](http://cheeseshop.python.org) is storing source and binary archives for a package. The new upload Distutils command will upload a package to the repository.

Before a package can be uploaded, you must be able to build a distribution using the sdist Distutils command. Once that works, you can run python setup.py upload to add your package to the PyPI archive. Optionally you can GPG-sign the package by supplying the --sign and --identity options.

Package uploading was implemented by Martin von Löwis and Richard Jones.

See Also:

[PEP 314, Metadata for Python Software Packages v1.1](#)

Release 1.0.
See About this document... for information on suggesting changes.
4 PEP 328: Absolute and Relative Imports

The simpler part of PEP 328 was implemented in Python 2.4: parentheses could now be used to enclose the names imported from a module using the `from ... import ...` statement, making it easier to import many different names.

The more complicated part has been implemented in Python 2.5: importing a module can be specified to use absolute or package-relative imports. The plan is to move toward making absolute imports the default in future versions of Python.

Let's say you have a package directory like this:

```
pkg/
pkg/__init__.py
pkg/main.py
pkg/string.py
```

This defines a package named `pkg` containing the `pkg.main` and `pkg.string` submodules.

Consider the code in the `main.py` module. What happens if it executes the statement `import string`? In Python 2.4 and earlier, it will first look in the package's directory to perform a relative import, finds `pkg/string.py`, imports the contents of that file as the `pkg.string` module, and that module is bound to the name "string" in the `pkg.main` module's namespace.

That's fine if `pkg.string` was what you wanted. But what if you wanted Python's standard `string` module? There's no clean way to ignore `pkg.string` and look for the standard module; generally you had to look at the contents of `sys.modules`, which is slightly unclean. Holger Krekel's `py.std` package provides a tidier way to perform imports from the standard library, `import py ; py.std.string.join()`, but that package isn't available on all Python installations.

Reading code which relies on relative imports is also less clear, because a reader
may be confused about which module, `string` or `pkg.string`, is intended to be used. Python users soon learned not to duplicate the names of standard library modules in the names of their packages' submodules, but you can't protect against having your submodule's name being used for a new module added in a future version of Python.

In Python 2.5, you can switch `import`'s behaviour to absolute imports using a `from __future__ import absolute_import` directive. This absolute-import behaviour will become the default in a future version (probably Python 2.7). Once absolute imports are the default, `import string` will always find the standard library's version. It's suggested that users should begin using absolute imports as much as possible, so it's preferable to begin writing `from pkg import string` in your code.

Relative imports are still possible by adding a leading period to the module name when using the `from ... import` form:

```python
# Import names from pkg.string
from .string import name1, name2
# Import pkg.string
from . import string
```

This imports the `string` module relative to the current package, so in `pkg.main` this will import `name1` and `name2` from `pkg.string`. Additional leading periods perform the relative import starting from the parent of the current package. For example, code in the `A.B.C` module can do:

```python
from . import D  # Imports A.B.D
from .. import E  # Imports A.E
from ...F import G  # Imports A.F.G
```

Leading periods cannot be used with the `import modname` form of the import statement, only the `from ... import` form.

**See Also:**

[PEP 328, Imports: Multi-Line and Absolute/Relative](http://codespeak.net/py/current/doc/index.html)

PEP written by Aahz; implemented by Thomas Wouters.
The py library by Holger Krekel, which contains the py.std package.

Release 1.0.
See About this document... for information on suggesting changes.
5 PEP 338: Executing Modules as Scripts

The -m switch added in Python 2.4 to execute a module as a script gained a few more abilities. Instead of being implemented in C code inside the Python interpreter, the switch now uses an implementation in a new module, runpy.

The runpy module implements a more sophisticated import mechanism so that it's now possible to run modules in a package such as pychecker.checker. The module also supports alternative import mechanisms such as the zipimport module. This means you can add a .zip archive's path to sys.path and then use the -m switch to execute code from the archive.

See Also:

PEP 338, Executing modules as scripts
PEP written and implemented by Nick Coghlan.

Release 1.0.
See About this document... for information on suggesting changes.
6 PEP 341: Unified try/except/finally

Until Python 2.5, the `try` statement came in two flavours. You could use a `finally` block to ensure that code is always executed, or one or more `except` blocks to catch specific exceptions. You couldn't combine both `except` blocks and a `finally` block, because generating the right bytecode for the combined version was complicated and it wasn't clear what the semantics of the combined should be.

Guido van Rossum spent some time working with Java, which does support the equivalent of combining `except` blocks and a `finally` block, and this clarified what the statement should mean. In Python 2.5, you can now write:

```python
try:
    block-1 ...
except Exception1:
    handler-1 ...
except Exception2:
    handler-2 ...
else:
    else-block
finally:
    final-block
```

The code in `block-1` is executed. If the code raises an exception, the various `except` blocks are tested: if the exception is of class `Exception1`, `handler-1` is executed; otherwise if it's of class `Exception2`, `handler-2` is executed, and so forth. If no exception is raised, the `else-block` is executed.

No matter what happened previously, the `final-block` is executed once the code block is complete and any raised exceptions handled. Even if there's an error in an exception handler or the `else-block` and a new exception is raised, the code in the `final-block` is still run.

---

**See Also:**

PEP 341, *Unifying try-except and try-finally*  
PEP written by Georg Brandl; implementation by Thomas Lee.
Release 1.0.
See About this document... for information on suggesting changes.
7 PEP 342: New Generator Features

Python 2.5 adds a simple way to pass values into a generator. As introduced in Python 2.3, generators only produce output; once a generator's code was invoked to create an iterator, there was no way to pass any new information into the function when its execution is resumed. Sometimes the ability to pass in some information would be useful. Hackish solutions to this include making the generator's code look at a global variable and then changing the global variable's value, or passing in some mutable object that callers then modify.

To refresh your memory of basic generators, here's a simple example:

```python
def counter(maximum):
    i = 0
    while i < maximum:
        yield i
        i += 1
```

When you call `counter(10)`, the result is an iterator that returns the values from 0 up to 9. On encountering the `yield` statement, the iterator returns the provided value and suspends the function's execution, preserving the local variables. Execution resumes on the following call to the iterator's `next()` method, picking up after the `yield` statement.

In Python 2.3, `yield` was a statement; it didn't return any value. In 2.5, `yield` is now an expression, returning a value that can be assigned to a variable or otherwise operated on:

```python
val = (yield i)
```

I recommend that you always put parentheses around a `yield` expression when you're doing something with the returned value, as in the above example. The parentheses aren't always necessary, but it's easier to always add them instead of having to remember when they're needed.

(PEP 342 explains the exact rules, which are that a `yield`-expression must always be parenthesized except when it occurs at the top-level expression on the right-hand side of an assignment. This means you can write `val = yield i` but have to use parentheses when there's an operation, as in `val = (yield i)`
Values are sent into a generator by calling its `send(value)` method. The generator's code is then resumed and the `yield` expression returns the specified value. If the regular `next()` method is called, the `yield` returns `None`.

Here's the previous example, modified to allow changing the value of the internal counter.

```python
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1
```

And here's an example of changing the counter:

```python
>>> it = counter(10)
>>> print it.next()
0
>>> print it.next()
1
>>> print it.send(8)
8
>>> print it.next()
9
>>> print it.next()
Traceback (most recent call last):
  File `t.py', line 15, in ?
    print it.next()
StopIteration
```

Because `yield` will often be returning `None`, you should always check for this case. Don't just use its value in expressions unless you're sure that the `send()` method will be the only method used resume your generator function.

In addition to `send()`, there are two other new methods on generators:

- `throw(type, value=None, traceback=None)` is used to raise an exception inside the generator; the exception is raised by the `yield` expression where the generator's execution is paused.
close() raises a new GeneratorExit exception inside the generator to terminate the iteration. On receiving this exception, the generator's code must either raise GeneratorExit or StopIteration; catching the exception and doing anything else is illegal and will trigger a RuntimeException. close() will also be called by Python's garbage collector when the generator is garbage-collected.

If you need to run cleanup code when a GeneratorExit occurs, I suggest using a try: ... finally: suite instead of catching GeneratorExit.

The cumulative effect of these changes is to turn generators from one-way producers of information into both producers and consumers.

Generators also become coroutines, a more generalized form of subroutines. Subroutines are entered at one point and exited at another point (the top of the function, and a return statement), but coroutines can be entered, exited, and resumed at many different points (the yield statements). We'll have to figure out patterns for using coroutines effectively in Python.

The addition of the close() method has one side effect that isn't obvious. close() is called when a generator is garbage-collected, so this means the generator's code gets one last chance to run before the generator is destroyed. This last chance means that try...finally statements in generators can now be guaranteed to work; the finally clause will now always get a chance to run. The syntactic restriction that you couldn't mix yield statements with a try...finally suite has therefore been removed. This seems like a minor bit of language trivia, but using generators and try...finally is actually necessary in order to implement the with statement described by PEP 343. I'll look at this new statement in the following section.

Another even more esoteric effect of this change: previously, the gi_frame attribute of a generator was always a frame object. It's now possible for gi_frame to be None once the generator has been exhausted.

See Also:
PEP 342, *Coroutines via Enhanced Generators*

PEP written by Guido van Rossum and Phillip J. Eby; implemented by Phillip J. Eby. Includes examples of some fancier uses of generators as coroutines.

Earlier versions of these features were proposed in PEP 288 by Raymond Hettinger and PEP 325 by Samuele Pedroni.


The Wikipedia entry for coroutines.

http://www.sidhe.org/~dan/blog/archives/000178.html

An explanation of coroutines from a Perl point of view, written by Dan Sugalski.

---

What's New in Python 2.5


Release 1.0.

See About this document... for information on suggesting changes.
Subsections

- 8.1 Writing Context Managers
- 8.2 The contextlib module
8 PEP 343: The 'with' statement

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

The 'with' statement is a new control-flow structure whose basic structure is:

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

The expression is evaluated, and it should result in an object that supports the context management protocol. This object may return a value that can optionally be bound to the name variable. (Note carefully that variable is not assigned the result of expression.) The object can then run set-up code before with-block is executed and some clean-up code is executed after the block is done, even if the block raised an exception.

To enable the statement in Python 2.5, you need to add the following directive to your module:

```
from __future__ import with_statement
```

The statement will always be enabled in Python 2.6.

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

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

After this statement has executed, the file object in f will have been automatically closed, even if the 'for' loop raised an exception part-way through the block.
The threading module's locks and condition variables also support the 'with' statement:

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

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

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

```
from decimal import Decimal, Context, localcontext

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

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

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

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

- The expression is evaluated and should result in an object called a ``context manager``. The context manager must have __enter__( ) and __exit__( ) methods.

- The context manager's __enter__( ) method is called. The value returned is assigned to VAR. If no 'as VAR' clause is present, the value is simply discarded.

- The code in BLOCK is executed.

- If BLOCK raises an exception, the __exit__( type, value, traceback ) is called with the exception details, the same values returned by sys.exc_info(). The method’s return value controls whether the exception is re-raised: any false value re-raises the exception, and True will result in suppressing it. You'll only rarely want to suppress the exception, because if you do the author of the code containing the 'with' statement will never realize anything went wrong.

- If BLOCK didn't raise an exception, the __exit__( ) method is still called, but type, value, and traceback are all None.

Let's think through an example. I won't present detailed code but will only sketch the methods necessary for a database that supports transactions.

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

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

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

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

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

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

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

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

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

class DatabaseConnection:

    def __exit__(self, type, value, tb):
        if tb is None:
            # No exception, so commit
            self.commit()
        else:
            # Exception occurred, so rollback.
            self.rollback()
            # return False
8.2 The `contextlib` module

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

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

Our database example from the previous section could be written using this decorator as:

```python
from contextlib import contextmanager

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

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

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

```python
lock = threading.Lock()
with nested (db_transaction(db), lock) as (cursor, locked):
    ...
```
Finally, the `closing(object) function returns `object` so that it can be bound to a variable, and calls `object.close()` at the end of the block.

```python
import urllib, sys
from contextlib import closing

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

See Also:

**PEP 343, *The ``with'' statement***

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

`../lib/module-contextlib.html`

The documentation for the `contextlib` module.

---

**What's New in Python 2.5**


Release 1.0.

See *About this document...* for information on suggesting changes.
9 PEP 352: Exceptions as New-Style Classes

Exception classes can now be new-style classes, not just classic classes, and the built-in Exception class and all the standard built-in exceptions (NameError, ValueError, etc.) are now new-style classes.

The inheritance hierarchy for exceptions has been rearranged a bit. In 2.5, the inheritance relationships are:

```
BaseException          # New in Python 2.5
    |- KeyboardInterrupt
    |- SystemExit
    |- Exception
        |- (all other current built-in exceptions)
```

This rearrangement was done because people often want to catch all exceptions that indicate program errors. KeyboardInterrupt and SystemExit aren't errors, though, and usually represent an explicit action such as the user hitting Control-C or code calling sys.exit(). A bare except: will catch all exceptions, so you commonly need to list KeyboardInterrupt and SystemExit in order to re-raise them. The usual pattern is:

```
try:
    ...
except (KeyboardInterrupt, SystemExit):
    raise
except:
    # Log error...
    # Continue running program...
```

In Python 2.5, you can now write except Exception to achieve the same result, catching all the exceptions that usually indicate errors but leaving KeyboardInterrupt and SystemExit alone. As in previous versions, a bare except: still catches all exceptions.

The goal for Python 3.0 is to require any class raised as an exception to derive from BaseException or some descendant of BaseException, and future releases in the Python 2.x series may begin to enforce this constraint. Therefore,
I suggest you begin making all your exception classes derive from `Exception` now. It's been suggested that the bare `except:` form should be removed in Python 3.0, but Guido van Rossum hasn't decided whether to do this or not.

Raising of strings as exceptions, as in the statement `raise "Error occurred"`, is deprecated in Python 2.5 and will trigger a warning. The aim is to be able to remove the string-exception feature in a few releases.

**See Also:**

- [PEP 352, Required Superclass for Exceptions](#)
  PEP written by Brett Cannon and Guido van Rossum; implemented by Brett Cannon.

---

What's New in Python 2.5


See [About this document...](#) for information on suggesting changes.
10 PEP 353: Using ssize_t as the index type

A wide-ranging change to Python's C API, using a new Py_ssize_t type definition instead of int, will permit the interpreter to handle more data on 64-bit platforms. This change doesn't affect Python's capacity on 32-bit platforms.

Various pieces of the Python interpreter used C's int type to store sizes or counts; for example, the number of items in a list or tuple were stored in an int. The C compilers for most 64-bit platforms still define int as a 32-bit type, so that meant that lists could only hold up to \(2^{**31} - 1 = 2147483647\) items. (There are actually a few different programming models that 64-bit C compilers can use - see http://www.unix.org/version2/whatsnew/lp64_wp.html for a discussion - but the most commonly available model leaves int as 32 bits.)

A limit of 2147483647 items doesn't really matter on a 32-bit platform because you'll run out of memory before hitting the length limit. Each list item requires space for a pointer, which is 4 bytes, plus space for a PyObject representing the item. 2147483647*4 is already more bytes than a 32-bit address space can contain.

It's possible to address that much memory on a 64-bit platform, however. The pointers for a list that size would only require 16 GiB of space, so it's not unreasonable that Python programmers might construct lists that large. Therefore, the Python interpreter had to be changed to use some type other than int, and this will be a 64-bit type on 64-bit platforms. The change will cause incompatibilities on 64-bit machines, so it was deemed worth making the transition now, while the number of 64-bit users is still relatively small. (In 5 or 10 years, we may all be on 64-bit machines, and the transition would be more painful then.)

This change most strongly affects authors of C extension modules. Python strings and container types such as lists and tuples now use Py_ssize_t to store their size. Functions such as PyList_Size() now return Py_ssize_t. Code in extension modules may therefore need to have some variables changed to Py_ssize_t.
The `PyArg_ParseTuple()` and `Py_BuildValue()` functions have a new conversion code, "n", for `Py_ssize_t`. `PyArg_ParseTuple()`'s "s#" and "t#" still output `int` by default, but you can define the macro `PY_SSIZE_T_CLEAN` before including `Python.h` to make them return `Py_ssize_t`.

**PEP 353** has a section on conversion guidelines that extension authors should read to learn about supporting 64-bit platforms.

---

**See Also:**

**PEP 353, Using ssize_t as the index type**

PEP written and implemented by Martin von Löwis.

---

Release 1.0.

See [About this document...](#) for information on suggesting changes.
11 PEP 357: The '__index__' method

The NumPy developers had a problem that could only be solved by adding a new special method, __index__. When using slice notation, as in [start:stop:step], the values of the start, stop, and step indexes must all be either integers or long integers. NumPy defines a variety of specialized integer types corresponding to unsigned and signed integers of 8, 16, 32, and 64 bits, but there was no way to signal that these types could be used as slice indexes.

Slicing can't just use the existing __int__ method because that method is also used to implement coercion to integers. If slicing used __int__, floating-point numbers would also become legal slice indexes and that's clearly an undesirable behaviour.

Instead, a new special method called __index__ was added. It takes no arguments and returns an integer giving the slice index to use. For example:

```python
class C:
    def __index__ (self):
        return self.value
```

The return value must be either a Python integer or long integer. The interpreter will check that the type returned is correct, and raises a TypeError if this requirement isn't met.

A corresponding nb_index slot was added to the C-level PyNumberMethods structure to let C extensions implement this protocol. PyNumber_Index(obj) can be used in extension code to call the __index__ function and retrieve its result.

See Also:

PEP 357, Allowing Any Object to be Used for Slicing

PEP written and implemented by Travis Oliphant.
Release 1.0.
See About this document... for information on suggesting changes.
Subsections

- 12.1 Interactive Interpreter Changes
- 12.2 Optimizations
12 Other Language Changes

Here are all of the changes that Python 2.5 makes to the core Python language.

- The `dict` type has a new hook for letting subclasses provide a default value when a key isn't contained in the dictionary. When a key isn't found, the dictionary's `__missing__(key)` method will be called. This hook is used to implement the new `defaultdict` class in the `collections` module. The following example defines a dictionary that returns zero for any missing key:

  ```python
  class zerodict(dict):
    def __missing__(self, key):
      return 0
  
  d = zerodict({1:1, 2:2})
  print d[1], d[2]  # Prints 1, 2
  print d[3], d[4]  # Prints 0, 0
  ```

- Both 8-bit and Unicode strings have new `partition(sep)` and `rpartition(sep)` methods that simplify a common use case.

  The `find(S)` method is often used to get an index which is then used to slice the string and obtain the pieces that are before and after the separator. `partition(sep)` condenses this pattern into a single method call that returns a 3-tuple containing the substring before the separator, the separator itself, and the substring after the separator. If the separator isn't found, the first element of the tuple is the entire string and the other two elements are empty. `rpartition(sep)` also returns a 3-tuple but starts searching from the end of the string; the "r" stands for 'reverse'.

  Some examples:

  ```python
  >>> ('http://www.python.org').partition('://')
  ('http', '://', 'www.python.org')
  >>> ('file:/usr/share/doc/index.html').partition('://')
  ('file:/usr/share/doc/index.html', '', '')
  >>> (u'Subject: a quick question').partition(':')
  (u'Subject', u':', u'a quick question')
  >>> 'www.python.org'.rpartition('.
  ('www.python', '.', 'org')
  ```
>>> 'www.python.org'.rpartition(':
('', '', 'www.python.org')

(Implemented by Fredrik Lundh following a suggestion by Raymond Hettinger.)

- The `startswith()` and `endswith()` methods of string types now accept tuples of strings to check for.
  
  ```python
def is_image_file (filename):
    return filename.endswith(('.gif', '.jpg', '.tiff'))
  ```

  (Implemented by Georg Brandl following a suggestion by Tom Lynn.)

- The `min()` and `max()` built-in functions gained a `key` keyword parameter analogous to the `key` argument for `sort()`. This parameter supplies a function that takes a single argument and is called for every value in the list; `min()`/`max()` will return the element with the smallest/largest return value from this function. For example, to find the longest string in a list, you can do:

  ```python
  L = ['medium', 'longest', 'short']
  # Prints 'longest'
  print max(L, key=len)
  # Prints 'short', because lexicographically 'short' has the large
  print max(L)
  ```

  (Contributed by Steven Bethard and Raymond Hettinger.)

- Two new built-in functions, `any()` and `all()`, evaluate whether an iterator contains any true or false values. `any()` returns `True` if any value returned by the iterator is true; otherwise it will return `False`. `all()` returns `True` only if all of the values returned by the iterator evaluate as true. (Suggested by Guido van Rossum, and implemented by Raymond Hettinger.)

- The result of a class's `__hash__()` method can now be either a long integer or a regular integer. If a long integer is returned, the hash of that value is taken. In earlier versions the hash value was required to be a regular integer, but in 2.5 the `id()` built-in was changed to always return non-negative numbers, and users often seem to use `id(self)` in `__hash__()` methods (though this is discouraged).
• ASCII is now the default encoding for modules. It's now a syntax error if a
module contains string literals with 8-bit characters but doesn't have an
encoding declaration. In Python 2.4 this triggered a warning, not a syntax
token. See PEP 263 for how to declare a module's encoding; for example,
you might add a line like this near the top of the source file:

    # -*- coding: latin1 -*-

• A new warning, UnicodeWarning, is triggered when you attempt to
compare a Unicode string and an 8-bit string that can't be converted to
Unicode using the default ASCII encoding. The result of the comparison is
false:

    >>> chr(128) == unichr(128)    # Can't convert chr(128) to Unicode
    __main__:1: UnicodeWarning: Unicode equal comparison failed
to convert both arguments to Unicode - interpreting them
as being unequal
    False
    >>> chr(127) == unichr(127)    # chr(127) can be converted
    True

Previously this would raise a UnicodeDecodeError exception, but in
2.5 this could result in puzzling problems when accessing a dictionary. If
you looked up unichr(128) and chr(128) was being used as a key,
you'd get a UnicodeDecodeError exception. Other changes in 2.5
resulted in this exception being raised instead of suppressed by the code in
dictobject.c that implements dictionaries.

Raising an exception for such a comparison is strictly correct, but the
change might have broken code, so instead UnicodeWarning was
introduced.

(Implemented by Marc-André Lemburg.)

• One error that Python programmers sometimes make is forgetting to
include an __init__.py module in a package directory. Debugging this
mistake can be confusing, and usually requires running Python with the -v
switch to log all the paths searched. In Python 2.5, a new
ImportWarning warning is triggered when an import would have picked
up a directory as a package but no __init__.py was found. This warning is
silently ignored by default; provide the -Wd option when running the
Python executable to display the warning message. (Implemented by Thomas Wouters.)

- The list of base classes in a class definition can now be empty. As an example, this is now legal:

```python
class C():
    pass
```

(Implemented by Brett Cannon.)
## 12.1 Interactive Interpreter Changes

In the interactive interpreter, `quit` and `exit` have long been strings so that new users get a somewhat helpful message when they try to quit:

```python
>>> quit
'Use Ctrl-D (i.e. EOF) to exit.'
```

In Python 2.5, `quit` and `exit` are now objects that still produce string representations of themselves, but are also callable. Newbies who try `quit()` or `exit()` will now exit the interpreter as they expect. (Implemented by Georg Brandl.)

The Python executable now accepts the standard long options `--help` and `--version`; on Windows, it also accepts the `/?` option for displaying a help message. (Implemented by Georg Brandl.)
12.2 Optimizations

Several of the optimizations were developed at the NeedForSpeed sprint, an event held in Reykjavik, Iceland, from May 21-28 2006. The sprint focused on speed enhancements to the CPython implementation and was funded by EWT LLC with local support from CCP Games. Those optimizations added at this sprint are specially marked in the following list.

- When they were introduced in Python 2.4, the built-in `set` and `frozenset` types were built on top of Python's dictionary type. In 2.5 the internal data structure has been customized for implementing sets, and as a result sets will use a third less memory and are somewhat faster. (Implemented by Raymond Hettinger.)

- The speed of some Unicode operations, such as finding substrings, string splitting, and character map encoding and decoding, has been improved. (Substring search and splitting improvements were added by Fredrik Lundh and Andrew Dalke at the NeedForSpeed sprint. Character maps were improved by Walter Dörwald and Martin von Löwis.)

- The `long(str, base)` function is now faster on long digit strings because fewer intermediate results are calculated. The peak is for strings of around 800-1000 digits where the function is 6 times faster. (Contributed by Alan McIntyre and committed at the NeedForSpeed sprint.)

- The `struct` module now compiles structure format strings into an internal representation and caches this representation, yielding a 20% speedup. (Contributed by Bob Ippolito at the NeedForSpeed sprint.)

- The `re` module got a 1 or 2% speedup by switching to Python's allocator functions instead of the system's `malloc()` and `free()`. (Contributed by Jack Diederich at the NeedForSpeed sprint.)

- The code generator's peephole optimizer now performs simple constant folding in expressions. If you write something like `a = 2+3`, the code generator will do the arithmetic and produce code corresponding to `a = 5`. (Proposed and implemented by Raymond Hettinger.)
• Function calls are now faster because code objects now keep the most recently finished frame (a `zombie frame`) in an internal field of the code object, reusing it the next time the code object is invoked. (Original patch by Michael Hudson, modified by Armin Rigo and Richard Jones; committed at the NeedForSpeed sprint.)

Frame objects are also slightly smaller, which may improve cache locality and reduce memory usage a bit. (Contributed by Neal Norwitz.)

• Python's built-in exceptions are now new-style classes, a change that speeds up instantiation considerably. Exception handling in Python 2.5 is therefore about 30% faster than in 2.4. (Contributed by Richard Jones, Georg Brandl and Sean Reifschneider at the NeedForSpeed sprint.)

• Importing now caches the paths tried, recording whether they exist or not so that the interpreter makes fewer open() and stat() calls on startup. (Contributed by Martin von Löwis and Georg Brandl.)
Subsections

- 13.1 The ctypes package
- 13.2 The ElementTree package
- 13.3 The hashlib package
- 13.4 The sqlite3 package
- 13.5 The wsgiref package
13 New, Improved, and Removed Modules

The standard library received many enhancements and bug fixes in Python 2.5. Here's a partial list of the most notable changes, sorted alphabetically by module name. Consult the Misc/NEWS file in the source tree for a more complete list of changes, or look through the SVN logs for all the details.

- The audioop module now supports the a-LAW encoding, and the code for u-LAW encoding has been improved. (Contributed by Lars Immisch.)

- The codecs module gained support for incremental codecs. The codec.lookup() function now returns a CodecInfo instance instead of a tuple. CodecInfo instances behave like a 4-tuple to preserve backward compatibility but also have the attributes encode, decode, incrementalencoder, incrementaldecoder, streamwriter, and streamreader. Incremental codecs can receive input and produce output in multiple chunks; the output is the same as if the entire input was fed to the non-incremental codec. See the codecs module documentation for details. (Designed and implemented by Walter Dörwald.)

- The collections module gained a new type, defaultdict, that subclasses the standard dict type. The new type mostly behaves like a dictionary but constructs a default value when a key isn't present, automatically adding it to the dictionary for the requested key value.

The first argument to defaultdict's constructor is a factory function that gets called whenever a key is requested but not found. This factory function receives no arguments, so you can use built-in type constructors such as list() or int(). For example, you can make an index of words based on their initial letter like this:

```python
words = """Nel mezzo del cammin di nostra vita
mi ritrovo per una selva oscura
che la diritta via era smarrita""".lower().split()

index = defaultdict(list)
```
for w in words:
    init_letter = w[0]
    index[init_letter].append(w)

Printing index results in the following output:

defaultdict(<type 'list'>, {'c': ['cammin', 'che'], 'e': ['era'],
    'd': ['del', 'di', 'diritta'], 'm': ['mezzo', 'mi'],
    'l': ['la'], 'o': ['oscura'], 'n': ['nel', 'nostra'],
    'p': ['per'], 's': ['selva', 'smarrita'],
    'r': ['ritrovai'], 'u': ['una'], 'v': ['vita', 'via']}

(Contributed by Guido van Rossum.)

- The deque double-ended queue type supplied by the collections module now has a \texttt{remove(value)} method that removes the first occurrence of \texttt{value} in the queue, raising \texttt{ValueError} if the value isn't found. (Contributed by Raymond Hettinger.)

- New module: The \texttt{contextlib} module contains helper functions for use with the new \texttt{with} statement. See section \texttt{8.2} for more about this module.

- New module: The \texttt{cProfile} module is a C implementation of the existing \texttt{profile} module that has much lower overhead. The module's interface is the same as \texttt{profile}: you run \texttt{cProfile.run('main()')} to profile a function, can save profile data to a file, etc. It's not yet known if the Hotshot profiler, which is also written in C but doesn't match the \texttt{profile} module's interface, will continue to be maintained in future versions of Python. (Contributed by Armin Rigo.)

    Also, the \texttt{pstats} module for analyzing the data measured by the profiler now supports directing the output to any file object by supplying a \texttt{stream} argument to the \texttt{Stats} constructor. (Contributed by Skip Montanaro.)

- The \texttt{csv} module, which parses files in comma-separated value format, received several enhancements and a number of bugfixes. You can now set the maximum size in bytes of a field by calling the \texttt{csv.field_size_limit(new_limit)} function; omitting the \texttt{new_limit} argument will return the currently-set limit. The \texttt{reader} class now has a \texttt{line_num} attribute that counts the number of physical lines read from the
source; records can span multiple physical lines, so line_num is not the same as the number of records read.

The CSV parser is now stricter about multi-line quoted fields. Previously, if a line ended within a quoted field without a terminating newline character, a newline would be inserted into the returned field. This behavior caused problems when reading files that contained carriage return characters within fields, so the code was changed to return the field without inserting newlines. As a consequence, if newlines embedded within fields are important, the input should be split into lines in a manner that preserves the newline characters.

(Contributed by Skip Montanaro and Andrew McNamara.)

- The **datetime** class in the **datetime** module now has a `strptime(string, format)` method for parsing date strings, contributed by Josh Spoerri. It uses the same format characters as `time.strptime()` and `time.strftime()`:

  ```python
  from datetime import datetime
  ts = datetime.strptime('10:13:15 2006-03-07', '%H:%M:%S %Y-%m-%d')
  ```

- The **SequenceMatcher.get_matching_blocks()** method in the **difflib** module now guarantees to return a minimal list of blocks describing matching subsequences. Previously, the algorithm would occasionally break a block of matching elements into two list entries. (Enhancement by Tim Peters.)

- The **doctest** module gained a **SKIP** option that keeps an example from being executed at all. This is intended for code snippets that are usage examples intended for the reader and aren't actually test cases.

  An **encoding** parameter was added to the **testfile()** function and the **DocFileSuite** class to specify the file's encoding. This makes it easier to use non-ASCII characters in tests contained within a docstring. (Contributed by Bjorn Tillenius.)

- The **email** package has been updated to version 4.0. (Contributed by
Barry Warsaw.)

- The fileinput module was made more flexible. Unicode filenames are now supported, and a mode parameter that defaults to "r" was added to the input() function to allow opening files in binary or universal-newline mode. Another new parameter, openhook, lets you use a function other than open() to open the input files. Once you're iterating over the set of files, the FileInput object's new fileno() returns the file descriptor for the currently opened file. (Contributed by Georg Brandl.)

- In the gc module, the new get_count() function returns a 3-tuple containing the current collection counts for the three GC generations. This is accounting information for the garbage collector; when these counts reach a specified threshold, a garbage collection sweep will be made. The existing gc.collect() function now takes an optional generation argument of 0, 1, or 2 to specify which generation to collect. (Contributed by Barry Warsaw.)

- The nsmallest() and nlargest() functions in the heapq module now support a key keyword parameter similar to the one provided by the min()/max() functions and the sort() methods. For example:

  ```python
  >>> import heapq
  >>> L = ['short', 'medium', 'longest', 'longer still']
  >>> heapq.nsmallest(2, L)  # Return two lowest elements, lexicographically
  ['longer still', 'longest']
  >>> heapq.nsmallest(2, L, key=len)  # Return two shortest elements
  ['short', 'medium']
  ```
  (Contributed by Raymond Hettinger.)

- The itertools.islice() function now accepts None for the start and step arguments. This makes it more compatible with the attributes of slice objects, so that you can now write the following:

  ```python
  s = slice(5)   # Create slice object
  itertools.islice(iterable, s.start, s.stop, s.step)
  ```
  (Contributed by Raymond Hettinger.)

- The format() function in the locale module has been modified and
two new functions were added, `format_string()` and `currency()`.

The `format()` function's `val` parameter could previously be a string as long as no more than one `%` char specifier appeared; now the parameter must be exactly one `%` char specifier with no surrounding text. An optional `monetary` parameter was also added which, if `True`, will use the locale's rules for formatting currency in placing a separator between groups of three digits.

To format strings with multiple `%` char specifiers, use the new `format_string()` function that works like `format()` but also supports mixing `%` char specifiers with arbitrary text.

A new `currency()` function was also added that formats a number according to the current locale's settings.

(Contributed by Georg Brandl.)

- The `mailbox` module underwent a massive rewrite to add the capability to modify mailboxes in addition to reading them. A new set of classes that include `mbox`, `MH`, and `Maildir` are used to read mailboxes, and have an `add(message)` method to add messages, `remove(key)` to remove messages, and `lock() / unlock()` to lock/unlock the mailbox. The following example converts a maildir-format mailbox into an mbox-format one:

  ```python
  import mailbox
  
  src = mailbox.Maildir('maildir', factory=None)
  dest = mailbox.mbox('/tmp/mbox')
  
  for msg in src:
      dest.add(msg)
  ```

  (Contributed by Gregory K. Johnson. Funding was provided by Google's 2005 Summer of Code.)

- New module: the `msilib` module allows creating Microsoft Installer `.msi` files and CAB files. Some support for reading the `.msi` database is also included. (Contributed by Martin von Löwis.)
The nis module now supports accessing domains other than the system default domain by supplying a domain argument to the nis.match() and nis.maps() functions. (Contributed by Ben Bell.)

The operator module's itemgetter() and attrgetter() functions now support multiple fields. A call such as operator.attrgetter('a', 'b') will return a function that retrieves the a and b attributes. Combining this new feature with the sort() method's key parameter lets you easily sort lists using multiple fields. (Contributed by Raymond Hettinger.)

The optparse module was updated to version 1.5.1 of the Optik library. The OptionParser class gained an epilog attribute, a string that will be printed after the help message, and a destroy() method to break reference cycles created by the object. (Contributed by Greg Ward.)

The os module underwent several changes. The stat_float_times variable now defaults to true, meaning that os.stat() will now return time values as floats. (This doesn't necessarily mean that os.stat() will return times that are precise to fractions of a second; not all systems support such precision.)

Constants named os.SEEK_SET, os.SEEK_CUR, and os.SEEK_END have been added; these are the parameters to the os.lseek() function. Two new constants for locking are os.O_SHLOCK and os.O_EXLOCK.

Two new functions, wait3() and wait4(), were added. They're similar the waitpid() function which waits for a child process to exit and returns a tuple of the process ID and its exit status, but wait3() and wait4() return additional information. wait3() doesn't take a process ID as input, so it waits for any child process to exit and returns a 3-tuple of process-id, exit-status, resource-usage as returned from the resource.getrusage() function. wait4(pid) does take a process ID. (Contributed by Chad J. Schroeder.)

On FreeBSD, the os.stat() function now returns times with nanosecond resolution, and the returned object now has st_gen and st_birthtime. The st_flags member is also available, if the
The Python debugger provided by the pdb module can now store lists of commands to execute when a breakpoint is reached and execution stops. Once breakpoint #1 has been created, enter "commands 1" and enter a series of commands to be executed, finishing the list with "end". The command list can include commands that resume execution, such as "continue" or "next". (Contributed by Grégoire Dooms.)

The pickle and cPickle modules no longer accept a return value of None from the __reduce__() method; the method must return a tuple of arguments instead. The ability to return None was deprecated in Python 2.4, so this completes the removal of the feature.

The pkgutil module, containing various utility functions for finding packages, was enhanced to support PEP 302's import hooks and now also works for packages stored in ZIP-format archives. (Contributed by Phillip J. Eby.)

The pybench benchmark suite by Marc-André Lemburg is now included in the Tools/pybench directory. The pybench suite is an improvement on the commonly used pystone.py program because pybench provides a more detailed measurement of the interpreter's speed. It times particular operations such as function calls, tuple slicing, method lookups, and numeric operations, instead of performing many different operations and reducing the result to a single number as pystone.py does.

The pyexpat module now uses version 2.0 of the Expat parser. (Contributed by Trent Mick.)

The old regex and regsub modules, which have been deprecated ever since Python 2.0, have finally been deleted. Other deleted modules: statcache, tzparse, whrandom.

Also deleted: the lib-old directory, which includes ancient modules such as dircmp and ni, was removed. lib-old wasn't on the default sys.path, so unless your programs explicitly added the directory to sys.path, this removal shouldn't affect your code.
• The rlcompleter module is no longer dependent on importing the readline module and therefore now works on non-Unix platforms. (Patch from Robert Kiendl.)

• The SimpleXMLRPCServer and DocXMLRPCServer classes now have a rpc_paths attribute that constrains XML-RPC operations to a limited set of URL paths; the default is to allow only '/ ' and '/RPC2'. Setting rpc_paths to None or an empty tuple disables this path checking.

• The socket module now supports AF_NETLINK sockets on Linux, thanks to a patch from Philippe Biondi. Netlink sockets are a Linux-specific mechanism for communications between a user-space process and kernel code; an introductory article about them is at http://www.linuxjournal.com/article/7356. In Python code, netlink addresses are represented as a tuple of 2 integers, (pid, group_mask).

Two new methods on socket objects, recv_buf(buffer) and recvfrom_buf(buffer), store the received data in an object that supports the buffer protocol instead of returning the data as a string. This means you can put the data directly into an array or a memory-mapped file.

Socket objects also gained getfamily(), gettype(), and getproto() accessor methods to retrieve the family, type, and protocol values for the socket.

• New module: the spwd module provides functions for accessing the shadow password database on systems that support shadow passwords.

• The struct is now faster because it compiles format strings into Struct objects with pack() and unpack() methods. This is similar to how the re module lets you create compiled regular expression objects. You can still use the module-level pack() and unpack() functions; they'll create Struct objects and cache them. Or you can use Struct instances directly:

```python
s = struct.Struct('ih3s')
data = s.pack(1972, 187, 'abc')
year, number, name = s.unpack(data)
```
You can also pack and unpack data to and from buffer objects directly using the `pack_into(buffer, offset, v1, v2, ...)` and `unpack_from(buffer, offset)` methods. This lets you store data directly into an array or a memory-mapped file.

(Struct objects were implemented by Bob Ippolito at the NeedForSpeed sprint. Support for buffer objects was added by Martin Blais, also at the NeedForSpeed sprint.)

- The Python developers switched from CVS to Subversion during the 2.5 development process. Information about the exact build version is available as the `sys.subversion` variable, a 3-tuple of `(interpreter-name, branch-name, revision-range)`. For example, at the time of writing my copy of 2.5 was reporting ('CPython', 'trunk', '45313:45315').

This information is also available to C extensions via the `Py_GetBuildInfo()` function that returns a string of build information like this: "trunk:45355:45356M, Apr 13 2006, 07:42:19". (Contributed by Barry Warsaw.)

- Another new function, `sys._current_frames()`, returns the current stack frames for all running threads as a dictionary mapping thread identifiers to the topmost stack frame currently active in that thread at the time the function is called. (Contributed by Tim Peters.)

- The `TarFile` class in the `tarfile` module now has an `extractall()` method that extracts all members from the archive into the current working directory. It's also possible to set a different directory as the extraction target, and to unpack only a subset of the archive's members.

The compression used for a tarfile opened in stream mode can now be autodetected using the mode 'r|*'. (Contributed by Lars Gustäbel.)

- The `threading` module now lets you set the stack size used when new threads are created. The `stack_size([size])` function returns the currently configured stack size, and supplying the optional `size` parameter sets a new value. Not all platforms support changing the stack size, but Windows, POSIX threading, and OS/2 all do. (Contributed by Andrew
MacIntyre.)

- The **unicodedata** module has been updated to use version 4.1.0 of the Unicode character database. Version 3.2.0 is required by some specifications, so it's still available as **unicodedata.ucd_3_2_0**.

- New module: the **uuid** module generates universally unique identifiers (UUIDs) according to [RFC 4122](https://tools.ietf.org/html/rfc4122). The RFC defines several different UUID versions that are generated from a starting string, from system properties, or purely randomly. This module contains a UUID class and functions named *uuid1()* , *uuid3()* , *uuid4()* , and *uuid5()* to generate different versions of UUID. (Version 2 UUIDs are not specified in [RFC 4122](https://tools.ietf.org/html/rfc4122) and are not supported by this module.)

```python
>>> import uuid
>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')
```

(Contributed by Ka-Ping Yee.)

- The **weakref** module's **WeakKeyDictionary** and **WeakValueDictionary** types gained new methods for iterating over the weak references contained in the dictionary. *iterkeyrefs()* and *keyrefs()* methods were added to **WeakKeyDictionary**, and *itervaluerefs()* and *valuerefs()* were added to **WeakValueDictionary**. (Contributed by Fred L. Drake, Jr.)

- The **webbrowser** module received a number of enhancements. It's now usable as a script with **python -m webbrowser**, taking a URL as the argument; there are a number of switches to control the behaviour (for example, **-n** for a...
new browser window, -t for a new tab). New module-level functions, `open_new()` and `open_new_tab()`, were added to support this. The module's `open()` function supports an additional feature, an `autoraise` parameter that signals whether to raise the open window when possible. A number of additional browsers were added to the supported list such as Firefox, Opera, Konqueror, and elinks. (Contributed by Oleg Broytmann and Georg Brandl.)

- The `xmlrpclib` module now supports returning `datetime` objects for the XML-RPC date type. Supply `use_datetime=True` to the `loads()` function or the `Unmarshaller` class to enable this feature. (Contributed by Skip Montanaro.)

- The `zipfile` module now supports the ZIP64 version of the format, meaning that a .zip archive can now be larger than 4 GiB and can contain individual files larger than 4 GiB. (Contributed by Ronald Oussoren.)

- The `zlib` module's `Compress` and `Decompress` objects now support a `copy()` method that makes a copy of the object's internal state and returns a new `Compress` or `Decompress` object. (Contributed by Chris AtLee.)
13.1 The ctypes package

The ctypes package, written by Thomas Heller, has been added to the standard library. ctypes lets you call arbitrary functions in shared libraries or DLLs. Long-time users may remember the dl module, which provides functions for loading shared libraries and calling functions in them. The ctypes package is much fancier.

To load a shared library or DLL, you must create an instance of the CDLL class and provide the name or path of the shared library or DLL. Once that's done, you can call arbitrary functions by accessing them as attributes of the CDLL object.

```python
import ctypes
libc = ctypes.CDLL('libc.so.6')
result = libc.printf("Line of output\n")
```

Type constructors for the various C types are provided: c_int, c_float, c_double, c_char_p (equivalent to char *), and so forth. Unlike Python's types, the C versions are all mutable; you can assign to their value attribute to change the wrapped value. Python integers and strings will be automatically converted to the corresponding C types, but for other types you must call the correct type constructor. (And I mean must; getting it wrong will often result in the interpreter crashing with a segmentation fault.)

You shouldn't use c_char_p with a Python string when the C function will be modifying the memory area, because Python strings are supposed to be immutable; breaking this rule will cause puzzling bugs. When you need a modifiable memory area, use create_string_buffer():

```python
s = "this is a string"
buf = ctypes.create_string_buffer(s)
libc.strfry(buf)
```

C functions are assumed to return integers, but you can set the restype attribute of the function object to change this:

```python
>>> libc.atof('2.71828')
-1783957616
>>> libc.atof.restype = ctypes.c_double
```
ctypes also provides a wrapper for Python's C API as the \texttt{ctypes.pythonapi} object. This object does \textit{not} release the global interpreter lock before calling a function, because the lock must be held when calling into the interpreter's code. There's a \texttt{py\_object()} type constructor that will create a \texttt{PyObject *} pointer. A simple usage:

```python
import ctypes

d = {}
ctypes.pythonapi.PyObject_SetItem(ctypes.py_object(d),
            ctypes.py_object("abc"),  ctypes.py_object(1))
# d is now {"abc", 1}.
```

Don't forget to use \texttt{py\_object()}; if it's omitted you end up with a segmentation fault.

\texttt{ctypes} has been around for a while, but people still write and distribution hand-coded extension modules because you can't rely on \texttt{ctypes} being present. Perhaps developers will begin to write Python wrappers atop a library accessed through \texttt{ctypes} instead of extension modules, now that \texttt{ctypes} is included with core Python.

\begin{quote}
\textbf{See Also:}
\end{quote}

\begin{itemize}
\item \url{http://starship.python.net/crew/theller/ctypes/}
\hspace{2em} The ctypes web page, with a tutorial, reference, and FAQ.
\item \url{../lib/module-ctypes.html}
\hspace{2em} The documentation for the \texttt{ctypes} module.
\end{itemize}
13.2 The ElementTree package

A subset of Fredrik Lundh's ElementTree library for processing XML has been added to the standard library as xml.etree. The available modules are ElementTree, ElementPath, and ElementInclude from ElementTree 1.2.6. The cElementTree accelerator module is also included.

The rest of this section will provide a brief overview of using ElementTree. Full documentation for ElementTree is available at http://effbot.org/zone/element-index.htm.

ElementTree represents an XML document as a tree of element nodes. The text content of the document is stored as the .text and .tail attributes of (This is one of the major differences between ElementTree and the Document Object Model; in the DOM there are many different types of node, includingTextNode.)

The most commonly used parsing function is parse(), that takes either a string (assumed to contain a filename) or a file-like object and returns an ElementTree instance:

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

tree = ET.parse('ex-1.xml')

feed = urllib.urlopen('http://planet.python.org/rss10.xml')
tree = ET.parse(feed)
```

Once you have an ElementTree instance, you can call its getroot() method to get the root Element node.

There's also an XML() function that takes a string literal and returns an Element node (not an ElementTree). This function provides a tidy way to incorporate XML fragments, approaching the convenience of an XML literal:

```python
svg = ET.XML("""<svg width="10px" version="1.0">
    </svg>""")
svg.set('height', '320px')
svg.append(elem1)
```
Each XML element supports some dictionary-like and some list-like access methods. Dictionary-like operations are used to access attribute values, and list-like operations are used to access child nodes.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>elem[n]</td>
<td>Returns n'th child element.</td>
</tr>
<tr>
<td>elem[m:n]</td>
<td>Returns list of m'th through n'th child elements.</td>
</tr>
<tr>
<td>len(elem)</td>
<td>Returns number of child elements.</td>
</tr>
<tr>
<td>list(elem)</td>
<td>Returns list of child elements.</td>
</tr>
<tr>
<td>elem.append(elem2)</td>
<td>Adds elem2 as a child.</td>
</tr>
<tr>
<td>elem.insert(index, elem2)</td>
<td>Inserts elem2 at the specified location.</td>
</tr>
<tr>
<td>del elem[n]</td>
<td>Deletes n'th child element.</td>
</tr>
<tr>
<td>elem.keys()</td>
<td>Returns list of attribute names.</td>
</tr>
<tr>
<td>elem.get(name)</td>
<td>Returns value of attribute name.</td>
</tr>
<tr>
<td>elem.set(name, value)</td>
<td>Sets new value for attribute name.</td>
</tr>
<tr>
<td>elem.attrib</td>
<td>Retrieves the dictionary containing attributes.</td>
</tr>
<tr>
<td>del elem.attrib[name]</td>
<td>Deletes attribute name.</td>
</tr>
</tbody>
</table>

Comments and processing instructions are also represented as `Element` nodes. To check if a node is a comment or processing instructions:

```python
if elem.tag is ET.Comment:
    ...
elif elem.tag is ET.ProcessingInstruction:
    ...
```

To generate XML output, you should call the `ElementTree.write()` method. Like `parse()`, it can take either a string or a file-like object:

```python
# Encoding is US-ASCII
tree.write('output.xml')

# Encoding is UTF-8
f = open('output.xml', 'w')
tree.write(f, encoding='utf-8')
```

(Caution: the default encoding used for output is ASCII. For general XML work,
where an element's name may contain arbitrary Unicode characters, ASCII isn't a very useful encoding because it will raise an exception if an element's name contains any characters with values greater than 127. Therefore, it's best to specify a different encoding such as UTF-8 that can handle any Unicode character.)

This section is only a partial description of the ElementTree interfaces. Please read the package's official documentation for more details.

See Also:

http://effbot.org/zone/element-index.htm
Official documentation for ElementTree.
13.3 The hashlib package

A new `hashlib` module, written by Gregory P. Smith, has been added to replace the `md5` and `sha` modules. `hashlib` adds support for additional secure hashes (SHA-224, SHA-256, SHA-384, and SHA-512). When available, the module uses OpenSSL for fast platform optimized implementations of algorithms.

The old `md5` and `sha` modules still exist as wrappers around `hashlib` to preserve backwards compatibility. The new module's interface is very close to that of the old modules, but not identical. The most significant difference is that the constructor functions for creating new hashing objects are named differently.

```python
# Old versions
h = md5.md5()
h = md5.new()

# New version
h = hashlib.md5()

# Old versions
h = sha.sha()
h = sha.new()

# New version
h = hashlib.sha1()

# Hash that weren't previously available
h = hashlib.sha224()
h = hashlib.sha256()
h = hashlib.sha384()
h = hashlib.sha512()

# Alternative form
h = hashlib.new('md5')  # Provide algorithm as a string
```

Once a hash object has been created, its methods are the same as before: 
`update(string)` hashes the specified string into the current digest state, 
`digest()` and `hexdigest()` return the digest value as a binary string or a string of hex digits, and `copy()` returns a new hashing object with the same digest state.
See Also:

../lib/module-hashlib.html

The documentation for the hashlib module.
13.4 The sqlite3 package

The pysqlite module (http://www.pysqlite.org), a wrapper for the SQLite embedded database, has been added to the standard library under the package name sqlite3.

SQLite is a C library that provides a SQL-language database that stores data in disk files without requiring a separate server process. pysqlite was written by Gerhard Häring and provides a SQL interface compliant with the DB-API 2.0 specification described by PEP 249. This means that it should be possible to write the first version of your applications using SQLite for data storage. If switching to a larger database such as PostgreSQL or Oracle is later necessary, the switch should be relatively easy.

If you're compiling the Python source yourself, note that the source tree doesn't include the SQLite code, only the wrapper module. You'll need to have the SQLite libraries and headers installed before compiling Python, and the build process will compile the module when the necessary headers are available.

To use the module, you must first create a Connection object that represents the database. Here the data will be stored in the /tmp/example file:

    conn = sqlite3.connect('/tmp/example')

You can also supply the special name ":memory:" to create a database in RAM.

Once you have a Connection, you can create a Cursor object and call its execute() method to perform SQL commands:

    c = conn.cursor()

    # Create table
    c.execute('''create table stocks
                (date timestamp, trans varchar, symbol varchar,
                 qty decimal, price decimal)''')

    # Insert a row of data
    c.execute('''insert into stocks
                   values ('2006-01-05','BUY','RHAT',100,35.14)''')
Usually your SQL operations will need to use values from Python variables. You shouldn't assemble your query using Python's string operations because doing so is insecure; it makes your program vulnerable to an SQL injection attack.

Instead, use the DB-API's parameter substitution. Put "?" as a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor's execute() method. (Other database modules may use a different placeholder, such as "%s" or ":1".) For example:

```python
# Never do this -- insecure!
symbol = 'IBM'
c.execute("... where symbol = '%s'" % symbol)

# Do this instead
t = (symbol,)
c.execute('select * from stocks where symbol=?', t)

# Larger example
for t in (('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
          ('2006-04-05', 'BUY', 'MSOFT', 1000, 72.00),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
):
    c.execute('insert into stocks values (?, ?, ?, ?, ?)', t)
```

To retrieve data after executing a SELECT statement, you can either treat the cursor as an iterator, call the cursor's fetchone() method to retrieve a single matching row, or call fetchall() to get a list of the matching rows.

This example uses the iterator form:

```python
>>> c = conn.cursor()
>>> c.execute('select * from stocks order by price')
>>> for row in c:
...     print row
...
(u'2006-01-05', u'BUY', u'RHAT', 100, 35.140000000000001)
(u'2006-03-28', u'BUY', u'IBM', 1000, 45.0)
(u'2006-04-06', u'SELL', u'IBM', 500, 53.0)
(u'2006-04-05', u'BUY', u'MSOFT', 1000, 72.0)
```

For more information about the SQL dialect supported by SQLite, see [http://www.sqlite.org](http://www.sqlite.org).
See Also:

http://www.pysqlite.org
The pysqlite web page.

http://www.sqlite.org
The SQLite web page; the documentation describes the syntax and the available data types for the supported SQL dialect.

../lib/module-sqlite3.html
The documentation for the sqlite3 module.

PEP 249, Database API Specification 2.0
PEP written by Marc-André Lemburg.
13.5 The wsgiref package

The Web Server Gateway Interface (WSGI) v1.0 defines a standard interface between web servers and Python web applications and is described in PEP 333. The wsgiref package is a reference implementation of the WSGI specification.

The package includes a basic HTTP server that will run a WSGI application; this server is useful for debugging but isn't intended for production use. Setting up a server takes only a few lines of code:

```python
from wsgiref import simple_server

wsgi_app = ...

host = ''
port = 8000
httpd = simple_server.make_server(host, port, wsgi_app)
httpd.serve_forever()
```

See Also:

- [http://www.wsgi.org](http://www.wsgi.org)
  A central web site for WSGI-related resources.

- [PEP 333, Python Web Server Gateway Interface v1.0](http://www.python.org/doc/pep-0333/)
  PEP written by Phillip J. Eby.

---

Release 1.0.

See [About this document...](http://docs.python.org/about) for information on suggesting changes.
Subsections

- 14.1 Port-Specific Changes
14 Build and C API Changes

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

- The Python source tree was converted from CVS to Subversion, in a complex migration procedure that was supervised and flawlessly carried out by Martin von Löwis. The procedure was developed as PEP 347.

- Coverity, a company that markets a source code analysis tool called Prevent, provided the results of their examination of the Python source code. The analysis found about 60 bugs that were quickly fixed. Many of the bugs were refcounting problems, often occurring in error-handling code. See http://scan.coverity.com for the statistics.

- The largest change to the C API came from PEP 353, which modifies the interpreter to use a Py_ssize_t type definition instead of int. See the earlier section 10 for a discussion of this change.

- The design of the bytecode compiler has changed a great deal, no longer generating bytecode by traversing the parse tree. Instead the parse tree is converted to an abstract syntax tree (or AST), and it is the abstract syntax tree that's traversed to produce the bytecode.

It's possible for Python code to obtain AST objects by using the compile() built-in and specifying _ast.PyCF_ONLY_AST as the value of the flags parameter:

```python
from _ast import PyCF_ONLY_AST
ast = compile("""a=0
for i in range(10):
    a += i
""", "<string>", 'exec', PyCF_ONLY_AST)

assignment = ast.body[0]
for_loop = ast.body[1]
```

No official documentation has been written for the AST code yet, but PEP 339 discusses the design. To start learning about the code, read the definition of the various AST nodes in Parser/Python.asdl. A Python
script reads this file and generates a set of C structure definitions in Include/Python-ast.h. The PyParser_ASTFromString() and PyParser_ASTFromFile(), defined in Include/pythonrun.h, take Python source as input and return the root of an AST representing the contents. This AST can then be turned into a code object by PyAST_Compile(). For more information, read the source code, and then ask questions on python-dev.

The AST code was developed under Jeremy Hylton's management, and implemented by (in alphabetical order) Brett Cannon, Nick Coghlan, Grant Edwards, John Ehresman, Kurt Kaiser, Neal Norwitz, Tim Peters, Armin Rigo, and Neil Schemenauer, plus the participants in a number of AST sprints at conferences such as PyCon.

- Evan Jones's patch to obmalloc, first described in a talk at PyCon DC 2005, was applied. Python 2.4 allocated small objects in 256K-sized arenas, but never freed arenas. With this patch, Python will free arenas when they're empty. The net effect is that on some platforms, when you allocate many objects, Python's memory usage may actually drop when you delete them and the memory may be returned to the operating system. (Implemented by Evan Jones, and reworked by Tim Peters.)

Note that this change means extension modules must be more careful when allocating memory. Python's API has many different functions for allocating memory that are grouped into families. For example, PyMem_Malloc(), PyMem_Realloc(), and PyMem_Free() are one family that allocates raw memory, while PyObject_Malloc(), PyObject_Realloc(), and PyObject_Free() are another family that's supposed to be used for creating Python objects.

Previously these different families all reduced to the platform's malloc() and free() functions. This meant it didn't matter if you got things wrong and allocated memory with the PyMem function but freed it with the PyObject function. With 2.5's changes to obmalloc, these families now do different things and mismatches will probably result in a segfault. You should carefully test your C extension modules with Python 2.5.

- The built-in set types now have an official C API. Call PySet_New() and PyFrozenSet_New() to create a new set, PySet_Add() and
PySet_Discard() to add and remove elements, and
PySet_Contains and PySet_Size to examine the set's state.
(Contributed by Raymond Hettinger.)

- C code can now obtain information about the exact revision of the Python
  interpreter by calling the Py_GetBuildInfo() function that returns a
  string of build information like this: "trunk:45355:45356M, Apr
  13 2006, 07:42:19". (Contributed by Barry Warsaw.)

- Two new macros can be used to indicate C functions that are local to the
  current file so that a faster calling convention can be used.
  Py_LOCAL(type) declares the function as returning a value of the
  specified type and uses a fast-calling qualifier.
  Py_LOCAL_INLINE(type) does the same thing and also requests the
  function be inlined. If PY_LOCAL_AGGRESSIVE is defined before
  python.h is included, a set of more aggressive optimizations are enabled
  for the module; you should benchmark the results to find out if these
  optimizations actually make the code faster. (Contributed by Fredrik Lundh
  at the NeedForSpeed sprint.)

- PyErr_NewException(name, base, dict) can now accept a tuple of
  base classes as its base argument. (Contributed by Georg Brandl.)

- The PyErr.Warn() function for issuing warnings is now deprecated in
  favour of PyErr.WarnEx(category, message, stacklevel) which lets you
  specify the number of stack frames separating this function
  and the caller. A stacklevel of 1 is the function calling PyErr.WarnEx(),
  2 is the function above that, and so forth. (Added by Neal Norwitz.)

- The CPython interpreter is still written in C, but the code can now be
  compiled with a C++ compiler without errors. (Implemented by Anthony
  Baxter, Martin von Löwis, Skip Montanaro.)

- The PyRange_New() function was removed. It was never documented,
  never used in the core code, and had dangerously lax error checking. In the
  unlikely case that your extensions were using it, you can replace it by
  something like the following:

      range = PyObject_CallFunction((PyObject*) &PyRange_Type, "lll",
                                      start, stop, step);
14.1 Port-Specific Changes

- MacOS X (10.3 and higher): dynamic loading of modules now uses the `dlopen()` function instead of MacOS-specific functions.

- MacOS X: a `--enable-universalsdk` switch was added to the `configure` script that compiles the interpreter as a universal binary able to run on both PowerPC and Intel processors. (Contributed by Ronald Oussoren.)

- Windows: .dll is no longer supported as a filename extension for extension modules. .pyd is now the only filename extension that will be searched for.

---

Release 1.0.
See About this document... for information on suggesting changes.
15 Porting to Python 2.5

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

- ASCII is now the default encoding for modules. It's now a syntax error if a module contains string literals with 8-bit characters but doesn't have an encoding declaration. In Python 2.4 this triggered a warning, not a syntax error.

- Previously, the `gi_frame` attribute of a generator was always a frame object. Because of the PEP 342 changes described in section 7, it's now possible for `gi_frame` to be `None`.

- A new warning, `UnicodeWarning`, is triggered when you attempt to compare a Unicode string and an 8-bit string that can't be converted to Unicode using the default ASCII encoding. Previously such comparisons would raise a `UnicodeDecodeError` exception.

- Library: the `csv` module is now stricter about multi-line quoted fields. If your files contain newlines embedded within fields, the input should be split into lines in a manner which preserves the newline characters.

- Library: the `locale` module's `format()` function's would previously accept any string as long as no more than one `%char` specifier appeared. In Python 2.5, the argument must be exactly one `%char` specifier with no surrounding text.

- Library: The `pickle` and `cPickle` modules no longer accept a return value of `None` from the `__reduce__()` method; the method must return a tuple of arguments instead. The modules also no longer accept the deprecated `bin` keyword parameter.

- Library: The `SimpleXMLRPCServer` and `DocXMLRPCServer` classes now have a `rpc_paths` attribute that constrains XML-RPC operations to a limited set of URL paths; the default is to allow only `'/` and `'/RPC2'`. Setting `rpc_paths` to `None` or an empty tuple disables this path.
checking.

- C API: Many functions now use `Py_ssize_t` instead of `int` to allow processing more data on 64-bit machines. Extension code may need to make the same change to avoid warnings and to support 64-bit machines. See the earlier section 10 for a discussion of this change.

- C API: The obmalloc changes mean that you must be careful to not mix usage of the PyMem_*() and PyObject_*() families of functions. Memory allocated with one family's *_Malloc() must be freed with the corresponding family's *_Free() function.

Release 1.0.
See About this document... for information on suggesting changes.
Acknowledgements

These people have contributed in some way to the Python documentation. This list is probably not complete -- if you feel that you or anyone else should be on this list, please let us know (send email to docs@python.org), and we will be glad to correct the problem.

It is only with the input and contributions of the Python community that Python has such wonderful documentation -- Thank You!

Aahz
Michael Abbott
Steve Alexander
Jim Ahlstrom
Fred Allen
A. Amoroso
Pehr Anderson
Oliver Andrich
JesĂ¨s Cea AviĂ¨n
Daniel Barclay
Chris Barker
Don Bashford
Anthony Baxter
Bennett Benson
Jonathan Black
Robin Boerdijk
Jonathan Giddy
Shelley Gooch
Nathaniel Gray
Grant Griffin
Thomas Guettler
Anders Hammarquist
Mark Hammond
Harald Hanche-Olsen
Manus Hand
Gerhard Häring
Travis B. Hartwell
Janko Hauser
Bernhard Herzog
Magnus L. Hetland
Konrad Hinsen
Stefan Hoffmeister
Marc-AndrĂ© Lemburg
Ulf A. Lindgren
Everett Lipman
Mirko Liss
Martin von LĂ¶wis
Fredrik Lundh
Joakim Sernbrant
Jeff MacDonald
John Machin
Andrew MacIntyre
Vladimir Marangozov
Vincent Marchetti
Laura Matson
Daniel May
Doug Mennella
Paolo Milani
Skip Montanaro
Hugh Sasse
Bob Savage
Scott Schram
Neil Schemenauer
Barry Scott
Justin Sheehy
Michael Simcich
Ionel Simionescu
Gregory P. Smith
Roy Smith
Clay Spence
Nicholas Spies
Tage Stabell-Kulo
Frank Stajano
Anthony Starks
<table>
<thead>
<tr>
<th>Lele Gaifax</th>
<th>Erno Kuusela</th>
<th>Donald Wallace Rouse II</th>
<th>Cheng Zhang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matthew Gallagher</td>
<td>Detlef Lannert</td>
<td>Nick Russo</td>
<td></td>
</tr>
<tr>
<td>Ben Gertzfield</td>
<td>Piers Lauder</td>
<td>Chris Ryland</td>
<td></td>
</tr>
<tr>
<td>Nadim Ghaznavi</td>
<td>Glyph Lefkowitz</td>
<td>Constantina S.</td>
<td></td>
</tr>
</tbody>
</table>
Python Tutorial
Guido van Rossum

Python Software Foundation  Email: docs@python.org

Fred L. Drake, Jr., editor

Release 2.5
19th September, 2006

- Front Matter
- Contents
- 1. Whetting Your Appetite
- 2. Using the Python Interpreter
  - 2.1 Invoking the Interpreter
    - 2.1.1 Argument Passing
    - 2.1.2 Interactive Mode
  - 2.2 The Interpreter and Its Environment
    - 2.2.1 Error Handling
    - 2.2.2 Executable Python Scripts
    - 2.2.3 Source Code Encoding
    - 2.2.4 The Interactive Startup File
- 3. An Informal Introduction to Python
  - 3.1 Using Python as a Calculator
    - 3.1.1 Numbers
    - 3.1.2 Strings
    - 3.1.3 Unicode Strings
    - 3.1.4 Lists
  - 3.2 First Steps Towards Programming
- 4. More Control Flow Tools
  - 4.1 if Statements
  - 4.2 for Statements
  - 4.3 The range() Function
- 4.4 break and continue Statements, and else Clauses on Loops
- 4.5 pass Statements
- 4.6 Defining Functions
- 4.7 More on Defining Functions
  - 4.7.1 Default Argument Values
  - 4.7.2 Keyword Arguments
  - 4.7.3 Arbitrary Argument Lists
  - 4.7.4 Unpacking Argument Lists
  - 4.7.5 Lambda Forms
  - 4.7.6 Documentation Strings
- 5. Data Structures
  - 5.1 More on Lists
    - 5.1.1 Using Lists as Stacks
    - 5.1.2 Using Lists as Queues
    - 5.1.3 Functional Programming Tools
    - 5.1.4 List Comprehensions
  - 5.2 The del statement
  - 5.3 Tuples and Sequences
  - 5.4 Sets
  - 5.5 Dictionaries
  - 5.6 Looping Techniques
  - 5.7 More on Conditions
  - 5.8 Comparing Sequences and Other Types
- 6. Modules
  - 6.1 More on Modules
    - 6.1.1 The Module Search Path
    - 6.1.2 "Compiled" Python files
  - 6.2 Standard Modules
  - 6.3 The dir() Function
  - 6.4 Packages
    - 6.4.1 Importing * From a Package
    - 6.4.2 Intra-package References
    - 6.4.3 Packages in Multiple Directories
- 7. Input and Output
  - 7.1 Fancier Output Formatting
  - 7.2 Reading and Writing Files
    - 7.2.1 Methods of File Objects
    - 7.2.2 The pickle Module
8. Errors and Exceptions
   - 8.1 Syntax Errors
   - 8.2 Exceptions
   - 8.3 Handling Exceptions
   - 8.4 Raising Exceptions
   - 8.5 User-defined Exceptions
   - 8.6 Defining Clean-up Actions
   - 8.7 Predefined Clean-up Actions

9. Classes
   - 9.1 A Word About Terminology
   - 9.2 Python Scopes and Name Spaces
   - 9.3 A First Look at Classes
     - 9.3.1 Class Definition Syntax
     - 9.3.2 Class Objects
     - 9.3.3 Instance Objects
     - 9.3.4 Method Objects
   - 9.4 Random Remarks
   - 9.5 Inheritance
     - 9.5.1 Multiple Inheritance
   - 9.6 Private Variables
   - 9.7 Odds and Ends
   - 9.8 Exceptions Are Classes Too
   - 9.9 Iterators
   - 9.10 Generators
   - 9.11 Generator Expressions

10. Brief Tour of the Standard Library
    - 10.1 Operating System Interface
    - 10.2 File Wildcards
    - 10.3 Command Line Arguments
    - 10.4 Error Output Redirection and Program Termination
    - 10.5 String Pattern Matching
    - 10.6 Mathematics
    - 10.7 Internet Access
    - 10.8 Dates and Times
    - 10.9 Data Compression
    - 10.10 Performance Measurement
    - 10.11 Quality Control
    - 10.12 Batteries Included

11. Brief Tour of the Standard Library - Part II
• 11.1 Output Formatting
• 11.2 Templating
• 11.3 Working with Binary Data Record Layouts
• 11.4 Multi-threading
• 11.5 Logging
• 11.6 Weak References
• 11.7 Tools for Working with Lists
• 11.8 Decimal Floating Point Arithmetic

• 12. What Now?
• A. Interactive Input Editing and History Substitution
  • A.1 Line Editing
  • A.2 History Substitution
  • A.3 Key Bindings
  • A.4 Commentary
• B. Floating Point Arithmetic: Issues and Limitations
  • B.1 Representation Error
• C. History and License
  • C.1 History of the software
  • C.2 Terms and conditions for accessing or otherwise using Python
  • C.3 Licenses and Acknowledgements for Incorporated Software
    • C.3.1 Mersenne Twister
    • C.3.2 Sockets
    • C.3.3 Floating point exception control
    • C.3.4 MD5 message digest algorithm
    • C.3.5 Asynchronous socket services
    • C.3.6 Cookie management
    • C.3.7 Profiling
    • C.3.8 Execution tracing
    • C.3.9 UUencode and UUdecode functions
    • C.3.10 XML Remote Procedure Calls

• D. Glossary
• Index
• About this document ...

Python Tutorial

Up: Python Documentation Index  Next: Front Matter

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1. Whetting Your Appetite

If you do much work on computers, eventually you find that there's some task you'd like to automate. For example, you may wish to perform a search-and-replace over a large number of text files, or rename and rearrange a bunch of photo files in a complicated way. Perhaps you'd like to write a small custom database, or a specialized GUI application, or a simple game.

If you're a professional software developer, you may have to work with several C/C++/Java libraries but find the usual write/compile/test/re-compile cycle is too slow. Perhaps you're writing a test suite for such a library and find writing the testing code a tedious task. Or maybe you've written a program that could use an extension language, and you don't want to design and implement a whole new language for your application.

Python is just the language for you.

You could write a UNIX shell script or Windows batch files for some of these tasks, but shell scripts are best at moving around files and changing text data, not well-suited for GUI applications or games. You could write a C/C++/Java program, but it can take a lot of development time to get even a first-draft program. Python is simpler to use, available on Windows, MacOS X, and UNIX operating systems, and will help you get the job done more quickly.

Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than shell scripts or batch files can offer. On the other hand, Python also offers much more error checking than C, and, being a very-high-level language, it has high-level data types built in, such as flexible arrays and dictionaries. Because of its more general data types Python is applicable to a much larger problem domain than Awk or even Perl, yet many things are at least as easy in Python as in those languages.

Python allows you to split your program into modules that can be reused in other Python programs. It comes with a large collection of standard modules that you can use as the basis of your programs -- or as examples to start learning to program in Python. Some of these modules provide things like file I/O, system calls, sockets, and even interfaces to graphical user interface toolkits like Tk.
Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write throw-away programs, or to test functions during bottom-up program development. It is also a handy desk calculator.

Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:

- the high-level data types allow you to express complex operations in a single statement;
- statement grouping is done by indentation instead of beginning and ending brackets;
- no variable or argument declarations are necessary.

Python is extensible: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library). Once you are really hooked, you can link the Python interpreter into an application written in C and use it as an extension or command language for that application.

By the way, the language is named after the BBC show "Monty Python's Flying Circus" and has nothing to do with nasty reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged!

Now that you are all excited about Python, you'll want to examine it in some more detail. Since the best way to learn a language is to use it, the tutorial invites you to play with the Python interpreter as you read.

In the next chapter, the mechanics of using the interpreter are explained. This is rather mundane information, but essential for trying out the examples shown later.

The rest of the tutorial introduces various features of the Python language and system through examples, beginning with simple expressions, statements and data types, through functions and modules, and finally touching upon advanced concepts like exceptions and user-defined classes.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
Subsections

- **2.1 Invoking the Interpreter**
  - 2.1.1 Argument Passing
  - 2.1.2 Interactive Mode
- **2.2 The Interpreter and Its Environment**
  - 2.2.1 Error Handling
  - 2.2.2 Executable Python Scripts
  - 2.2.3 Source Code Encoding
  - 2.2.4 The Interactive Startup File
2. Using the Python Interpreter
2.1 Invoking the Interpreter

The Python interpreter is usually installed as /usr/local/bin/python on those machines where it is available; putting /usr/local/bin in your UNIX shell's search path makes it possible to start it by typing the command

```
python
```

to the shell. Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., /usr/local/python is a popular alternative location.)

On Windows machines, the Python installation is usually placed in C:\Python24, though you can change this when you're running the installer. To add this directory to your path, you can type the following command into the command prompt in a DOS box:

```
set path=%path%;C:\python24
```

Typing an end-of-file character (Control-D on UNIX, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following commands:

```
import sys; sys.exit()
```

The interpreter's line-editing features usually aren't very sophisticated. On UNIX, whoever installed the interpreter may have enabled support for the GNU readline library, which adds more elaborate interactive editing and history features. Perhaps the quickest check to see whether command line editing is supported is typing Control-P to the first Python prompt you get. If it beeps, you have command line editing; see Appendix A for an introduction to the keys. If nothing appears to happen, or if P is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

The interpreter operates somewhat like the UNIX shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a script from that file.
A second way of starting the interpreter is "\texttt{python -c command [arg]} . . .", which executes the statement(s) in \textit{command}, analogous to the shell's \texttt{-c} option. Since Python statements often contain spaces or other characters that are special to the shell, it is best to quote \textit{command} in its entirety with double quotes.

Some Python modules are also useful as scripts. These can be invoked using "\texttt{python -m module [arg]} . . .", which executes the source file for \textit{module} as if you had spelled out its full name on the command line.

Note that there is a difference between "\texttt{python file}" and "\texttt{python <file}". In the latter case, input requests from the program, such as calls to \texttt{input()} and \texttt{raw_input()}, are satisfied from \texttt{file}. Since this file has already been read until the end by the parser before the program starts executing, the program will encounter end-of-file immediately. In the former case (which is usually what you want) they are satisfied from whatever file or device is connected to standard input of the Python interpreter.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing \texttt{-i} before the script. (This does not work if the script is read from standard input, for the same reason as explained in the previous paragraph.)
2.1.1 Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are passed to the script in the variable `sys.argv`, which is a list of strings. Its length is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string. When the script name is given as `'- '` (meaning standard input), `sys.argv[0]` is set to `'- '`.

When `-c command` is used, `sys.argv[0]` is set to `'-c'`. When `-m module` is used, `sys.argv[0]` is set to the full name of the located module. Options found after `-c command` or `-m module` are not consumed by the Python interpreter's option processing but left in `sys.argv` for the command or module to handle.
2.1.2 Interactive Mode

When commands are read from a tty, the interpreter is said to be in interactive mode. In this mode it prompts for the next command with the primary prompt, usually three greater-than signs (">>> "); for continuation lines it prompts with the secondary prompt, by default three dots ("... "). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06) [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
```

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this `if` statement:

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
... Be careful not to fall off!
```
2.2 The Interpreter and Its Environment
2.2.1 Error Handling

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an except clause in a try statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from executed commands is written to standard output.

Typing the interrupt character (usually Control-C or DEL) to the primary or secondary prompt cancels the input and returns to the primary prompt. Typing an interrupt while a command is executing raises the KeyboardInterrupt exception, which may be handled by a try statement.
2.2.2 Executable Python Scripts

On BSD'ish UNIX systems, Python scripts can be made directly executable, like shell scripts, by putting the line

```bash
#!/usr/bin/env python
```

(assuming that the interpreter is on the user's PATH) at the beginning of the script and giving the file an executable mode. The "#!" must be the first two characters of the file. On some platforms, this first line must end with a UNIX-style line ending ("\n"), not a Mac OS ("\r") or Windows ("\r\n") line ending. Note that the hash, or pound, character, "#", is used to start a comment in Python.

The script can be given an executable mode, or permission, using the chmod command:

```bash
$ chmod +x myscript.py
```
2.2.3 Source Code Encoding

It is possible to use encodings different than ASCII in Python source files. The best way to do it is to put one more special comment line right after the `#!` line to define the source file encoding:

```
# -*- coding: encoding -*-
```

With that declaration, all characters in the source file will be treated as having the encoding `encoding`, and it will be possible to directly write Unicode string literals in the selected encoding. The list of possible encodings can be found in the Python Library Reference, in the section on codecs.

For example, to write Unicode literals including the Euro currency symbol, the ISO-8859-15 encoding can be used, with the Euro symbol having the ordinal value 164. This script will print the value 8364 (the Unicode codepoint corresponding to the Euro symbol) and then exit:

```
# -*- coding: iso-8859-15 -*-

currency = u"€"
print ord(currency)
```

If your editor supports saving files as UTF-8 with a UTF-8 byte order mark (aka BOM), you can use that instead of an encoding declaration. IDLE supports this capability if Options/General/Default Source Encoding/UTF-8 is set. Notice that this signature is not understood in older Python releases (2.2 and earlier), and also not understood by the operating system for script files with `#!` lines (only used on UNIX systems).

By using UTF-8 (either through the signature or an encoding declaration), characters of most languages in the world can be used simultaneously in string literals and comments. Using non-ASCII characters in identifiers is not supported. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.
2.2.4 The Interactive Startup File

When you use Python interactively, it is frequently handy to have some standard commands executed every time the interpreter is started. You can do this by setting an environment variable named PYTHONSTARTUP to the name of a file containing your start-up commands. This is similar to the .profile feature of the UNIX shells.

This file is only read in interactive sessions, not when Python reads commands from a script, and not when /dev/tty is given as the explicit source of commands (which otherwise behaves like an interactive session). It is executed in the same namespace where interactive commands are executed, so that objects that it defines or imports can be used without qualification in the interactive session. You can also change the prompts sys.ps1 and sys.ps2 in this file.

If you want to read an additional start-up file from the current directory, you can program this in the global start-up file using code like "if os.path.isfile('.pythonrc.py'):
execfile('.pythonrc.py')". If you want to use the startup file in a script, you must do this explicitly in the script:

```python
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    execfile(filename)
```

---

Footnotes

... prompt.\(^{2.1}\) A problem with the GNU Readline package may prevent this.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
Subsections

- 3.1 Using Python as a Calculator
  - 3.1.1 Numbers
  - 3.1.2 Strings
  - 3.1.3 Unicode Strings
  - 3.1.4 Lists
- 3.2 First Steps Towards Programming
3. An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (">>> " and "... "): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, "#", and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character.

Some examples:

```
# this is the first comment
SPAM = 1                  # and this is the second comment
                        # ... and now a third!
STRING = "# This is not a comment."
```
3.1 Using Python as a Calculator

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, ">>> ". (It shouldn't take long.)
3.1.1 Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, *, and / work just like in most other languages (for example, Pascal or C); parentheses can be used for grouping. For example:

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2 # and a comment on the same line as code
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
... 7/3
2
>>> 7/-3
-3
```

The equal sign ("=") is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

A value can be assigned to several variables simultaneously:

```
>>> x = y = z = 0 # Zero x, y and z
>>> x
0
>>> y
0
>>> z
0
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 3 * 3.75 / 1.5
7.5
```
Complex numbers are also supported; imaginary numbers are written with a suffix of "j" or "J". Complex numbers with a nonzero real component are written as "(real+imagj)", or can be created with the "complex(real, imag)" function.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Complex numbers are always represented as two floating point numbers, the real and imaginary part. To extract these parts from a complex number z, use z.real and z.imag.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

The conversion functions to floating point and integer (float(), int() and long()) don't work for complex numbers -- there is no one correct way to convert a complex number to a real number. Use abs(z) to get its magnitude (as a float) or z.real to get its real part.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
```
In interactive mode, the last printed expression is assigned to the variable __. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```python
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + __
113.0625
>>> round(__, 2)
113.06
``` 

This variable should be treated as read-only by the user. Don't explicitly assign a value to it -- you would create an independent local variable with the same name masking the built-in variable with its magic behavior.
3.1.2 Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes or double quotes:

```python
>>> 'spam eggs'
'spam eggs'
>>> 'doesn''t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> "'Yes,"
""he said."
>>> ""Yes," he said."
>>> ""Yes,\" he said."
>>> ""Isn\'t," she said."
""Isn't," she said."
```

String literals can span multiple lines in several ways. Continuation lines can be used, with a backslash as the last character on the line indicating that the next line is a logical continuation of the line:

```python
hello = "This is a rather long string containing
several lines of text just as you would do in C.\n
    Note that whitespace at the beginning of the line is\n    significant."

print hello
```

Note that newlines still need to be embedded in the string using \n; the newline following the trailing backslash is discarded. This example would print the following:

```
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant
```

If we make the string literal a ``raw'' string, however, the \n sequences are not converted to newlines, but the backslash at the end of the line, and the newline character in the source, are both included in the string as data. Thus, the example:

```python
hello = r"This is a rather long string containing\nseveral lines of text much as you would do in C."
```
print hello

would print:

This is a rather long string containing
several lines of text much as you would do in C.

Or, strings can be surrounded in a pair of matching triple-quotes: """" or ' '''. End of lines do not need to be escaped when using triple-quotes, but they will be included in the string.

print ""
Usage: thingy [OPTIONS]
-h Display this usage message
-H hostname Hostname to connect to
""

produces the following output:

Usage: thingy [OPTIONS]
-h Display this usage message
-H hostname Hostname to connect to

The interpreter prints the result of string operations in the same way as they are typed for input: inside quotes, and with quotes and other funny characters escaped by backslashes, to show the precise value. The string is enclosed in double quotes if the string contains a single quote and no double quotes, else it's enclosed in single quotes. (The print statement, described later, can be used to write strings without quotes or escapes.)

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```python
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>
Strings can be subscripted (indexed); like in C, the first character of a string has subscript (index) 0. There is no separate character type; a character is simply a string of size one. Like in Icon, substrings can be specified with the slice notation: two indices separated by a colon.

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

Unlike a C string, Python strings cannot be changed. Assigning to an indexed position in the string results in an error:

However, creating a new string with the combined content is easy and efficient:
Here's a useful invariant of slice operations: $s[:i] + s[i:]$ equals $s$.

```python
>>> word[2:] + word[2:]
'HelpA'
```

>>> word[:3] + word[3:]
'HelpA'

Degenerate slice indices are handled gracefully: an index that is too large is replaced by the string size, an upper bound smaller than the lower bound returns an empty string.

```python
>>> word[1:100]
'elpA'
```

```python
>>> word[10:]
''
```

```python
>>> word[2:1]
''
```

Indices may be negative numbers, to start counting from the right. For example:

```python
>>> word[-1]    # The last character
'A'
```

```python
>>> word[-2]    # The last-but-one character
'p'
```

```python
>>> word[-2:]   # The last two characters
'pA'
```

```python
>>> word[-2:]   # Everything except the last two characters
'Hel'
```

But note that -0 is really the same as 0, so it does not count from the right!

```python
>>> word[-0]    # (since -0 equals 0)
'H'
```

Out-of-range negative slice indices are truncated, but don't try this for single-element (non-slice) indices:

```python
>>> word[-100:]
'HelpA'
```

```python
>>> word[-10]   # error
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

The best way to remember how slices work is to think of the indices as pointing \textit{between} characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of $n$ characters has index $n$, for
example:

+--------+--------+--------+--------+--------+
| H | e | l | p | A |
+--------+--------+--------+--------+--------+
0   1   2   3   4   5
-5  -4  -3  -2  -1

The first row of numbers gives the position of the indices 0...5 in the string; the second row gives the corresponding negative indices. The slice from \( i \) to \( j \) consists of all characters between the edges labeled \( i \) and \( j \), respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of \( \text{word}[1:3] \) is 2.

The built-in function `len()` returns the length of a string:

```python
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

### See Also:

**Sequence Types**

Strings, and the Unicode strings described in the next section, are examples of *sequence types*, and support the common operations supported by such types.

**String Methods**

Both strings and Unicode strings support a large number of methods for basic transformations and searching.

**String Formatting Operations**

The formatting operations invoked when strings and Unicode strings are the left operand of the % operator are described in more detail here.
### 3.1.3 Unicode Strings

Starting with Python 2.0 a new data type for storing text data is available to the programmer: the Unicode object. It can be used to store and manipulate Unicode data (see [http://www.unicode.org/](http://www.unicode.org/)) and integrates well with the existing string objects, providing auto-conversions where necessary.

Unicode has the advantage of providing one ordinal for every character in every script used in modern and ancient texts. Previously, there were only 256 possible ordinals for script characters. Texts were typically bound to a code page which mapped the ordinals to script characters. This lead to very much confusion especially with respect to internationalization (usually written as "i18n" -- "i" + 18 characters + "n") of software. Unicode solves these problems by defining one code page for all scripts.

Creating Unicode strings in Python is just as simple as creating normal strings:

```python
>>> u'Hello World !'
'u'Hello World !'
```

The small "u" in front of the quote indicates that a Unicode string is supposed to be created. If you want to include special characters in the string, you can do so by using the Python *Unicode-Escape* encoding. The following example shows how:

```python
>>> u'Hello\u0020World !'
'u'Hello World !'
```

The escape sequence `\u0020` indicates to insert the Unicode character with the ordinal value 0x0020 (the space character) at the given position.

Other characters are interpreted by using their respective ordinal values directly as Unicode ordinals. If you have literal strings in the standard Latin-1 encoding that is used in many Western countries, you will find it convenient that the lower 256 characters of Unicode are the same as the 256 characters of Latin-1.

For experts, there is also a raw mode just like the one for normal strings. You have to prefix the opening quote with 'ur' to have Python use the *Raw-Unicode-Escape* encoding. It will only apply the above `\uXXXX` conversion if there is an
uneven number of backslashes in front of the small 'u'.

```
>>> ur'Hello\u0020World !'
u'Hello  World !'
```

The raw mode is most useful when you have to enter lots of backslashes, as can be necessary in regular expressions.

Apart from these standard encodings, Python provides a whole set of other ways of creating Unicode strings on the basis of a known encoding.

The built-in function `unicode()` provides access to all registered Unicode codecs (COders and DEcoders). Some of the more well known encodings which these codecs can convert are `Latin-1`, `ASCII`, `UTF-8`, and `UTF-16`. The latter two are variable-length encodings that store each Unicode character in one or more bytes. The default encoding is normally set to ASCII, which passes through characters in the range 0 to 127 and rejects any other characters with an error. When a Unicode string is printed, written to a file, or converted with `str()`, conversion takes place using this default encoding.

```
>>> u"abc"
u'abc'
```

```
>>> str(u"abc")
'abc'
```

```
>>> u"äöü"
u'\xe4\xf6\xfc'
```

```
>>> str(u"äöüß")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2: ordinal not in range(128)
```

To convert a Unicode string into an 8-bit string using a specific encoding, Unicode objects provide an `encode()` method that takes one argument, the name of the encoding. Lowercase names for encodings are preferred.

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

If you have data in a specific encoding and want to produce a corresponding Unicode string from it, you can use the `unicode()` function with the encoding name as the second argument.
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
3.1.4 Lists

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

```python
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on:

```python
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:1]
['eggs', 100]
>>> a[2:] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo
```

Unlike strings, which are immutable, it is possible to change individual elements of a list:

```python
>>> a
['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 123, 1234]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```python
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
```
The built-in function \texttt{len()} also applies to lists:

```
>>> len(a)
8
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')  # See section 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Note that in the last example, \texttt{p[1]} and \texttt{q} really refer to the same object! We'll come back to \textit{object semantics} later.
3.2 First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the *Fibonacci* series as follows:

```python
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

This example introduces several new features.

- The first line contains a *multiple assignment*: the variables `a` and `b` simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.

- The `while` loop executes as long as the condition (here: `b < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: `<` (less than), `>` (greater than), `==` (equal to), `<=` (less than or equal to), `>=` (greater than or equal to) and `!=` (not equal to).

- The *body* of the loop is *indented*: indentation is Python's way of grouping statements. Python does not (yet!) provide an intelligent input line editing
facility, so you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; most text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.

- The print statement writes the value of the expression(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple expressions and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```python
>>> i = 256*256
>>> print 'The value of i is', i
The value of i is 65536
```

A trailing comma avoids the newline after the output:

```python
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Note that the interpreter inserts a newline before it prints the next prompt if the last line was not completed.
Subsections

- 4.1 if Statements
- 4.2 for Statements
- 4.3 The range() Function
- 4.4 break and continue Statements, and else Clauses on Loops
- 4.5 pass Statements
- 4.6 Defining Functions
- 4.7 More on Defining Functions
  - 4.7.1 Default Argument Values
  - 4.7.2 Keyword Arguments
  - 4.7.3 Arbitrary Argument Lists
  - 4.7.4 Unpacking Argument Lists
  - 4.7.5 Lambda Forms
  - 4.7.6 Documentation Strings
4. More Control Flow Tools

Besides the `while` statement just introduced, Python knows the usual control flow statements known from other languages, with some twists.
4.1 if Statements

Perhaps the most well-known statement type is the if statement. For example:

```python
>>> x = int(input("Please enter an integer: "))
>>> if x < 0:
...    x = 0
...    print 'Negative changed to zero'
... elif x == 0:
...    print 'Zero'
... elif x == 1:
...    print 'Single'
... else:
...    print 'More'
...```

There can be zero or more elif parts, and the else part is optional. The keyword `elif` is short for `else if`, and is useful to avoid excessive indentation. An if ... elif ... elif ... sequence is a substitute for the switch or case statements found in other languages.
4.2 for Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```python
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
... cat 3
window 6
defenestrate 12
```

It is not safe to modify the sequence being iterated over in the loop (this can only happen for mutable sequence types, such as lists). If you need to modify the list you are iterating over (for example, to duplicate selected items) you must iterate over a copy. The slice notation makes this particularly convenient:

```python
>>> for x in a[:]: # make a slice copy of the entire list
...     if len(x) > 6: a.insert(0, x)
... >>> a
['defenestrate', 'cat', 'window', 'defenestrate']
```
4.3 The range() Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates lists containing arithmetic progressions:

```python
types.range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The given end point is never part of the generated list; `range(10)` generates a list of 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the `step'):

```python
types.range(5, 10)
[5, 6, 7, 8, 9]
types.range(0, 10, 3)
[0, 3, 6, 9]
types.range(-10, -100, -30)
[-10, -40, -70]
```

To iterate over the indices of a sequence, combine `range()` and `len()` as follows:

```python
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```
4.4 break and continue
Statements, and else Clauses on Loops

The break statement, like in C, breaks out of the smallest enclosing for or while loop.

The continue statement, also borrowed from C, continues with the next iteration of the loop.

Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the list (with for) or when the condition becomes false (with while), but not when the loop is terminated by a break statement. This is exemplified by the following loop, which searches for prime numbers:

```python
>>> for n in range(2, 10):
...    for x in range(2, n):
...        if n % x == 0:
...            print n, 'equals', x, '*', n/x
...            break
...        else:
...            # loop fell through without finding a factor
...            print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```
4.5 pass Statements

The **pass** statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```python
>>> while True:
    ...
    pass  # Busy-wait for keyboard interrupt
    ...
```
4.6 Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):  # write Fibonacci series up to n
...     '''Print a Fibonacci series up to n.'''
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
... >>> # Now call the function we just defined:
...     fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented. The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring.

There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so try to make a habit of it.

The execution of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the global symbol table, and then in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a `global` statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using call by value (where the value is always an object reference, not the value of the object). When a function calls another function, a new local symbol table is created for that call.
A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

```python
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

You might object that `fib` is not a function but a procedure. In Python, like in C, procedures are just functions that don't return a value. In fact, technically speaking, procedures do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to:

```python
>>> print fib(0)
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```python
>>> def fib2(n):
...     '''Return a list containing the Fibonacci series up to n.'''
...     result = []
...     a, b = 0, 1
...     while b < n:
...         result.append(b)  # see below
...         a, b = b, a+b
...     return result
...>>> f100 = fib2(100)  # call it
>>> f100                        # write the result
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a procedure also returns `None`.

- The statement `result.append(b)` calls a `method` of the list object
result. A method is a function that `belongs' to an object and is named `obj.methodname', where `obj' is some object (this may be an expression), and `methodname' is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using `classes', as discussed later in this tutorial.) The method `append()' shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to "result = result + [b]'", but more efficient.
4.7 More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.
4.7.1 Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```python
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    ok = raw_input(prompt)
    if ok in ('y', 'ye', 'yes'): return True
    if ok in ('n', 'no', 'nop', 'nope'): return False
    retries = retries - 1
    if retries < 0: raise IOError, 'refusenik user'
    print complaint
```

This function can be called either like this: `ask_ok('Do you really want to quit?')` or like this: `ask_ok('OK to overwrite the file?', 2).

This example also introduces the `in` keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the defining scope, so that

```python
i = 5
def f(arg=i):
    print arg
i = 6
f()
```

will print 5.

**Important warning:** The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```python
def f(a, L=[]):
    L.append(a)
    return L
```
print f(1)
print f(2)
print f(3)

This will print

[1]
[1, 2]
[1, 2, 3]

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
4.7.2 Keyword Arguments

Functions can also be called using keyword arguments of the form "keyword = value". For instance, the following function:

```python
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!
```

could be called in any of the following ways:

```python
parrot(1000)
parrot(action = 'V O O O O O M', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

but the following calls would all be invalid:

```python
parrot()  # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument following keyword
parrot(110, voltage=220)  # duplicate value for argument
parrot(actor='John Cleese')  # unknown keyword
```

In general, an argument list must have any positional arguments followed by any keyword arguments, where the keywords must be chosen from the formal parameter names. It's not important whether a formal parameter has a default value or not. No argument may receive a value more than once -- formal parameter names corresponding to positional arguments cannot be used as keywords in the same calls. Here's an example that fails due to this restriction:

```python
>>> def function(a):
...     pass
...  
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

When a final formal parameter of the form **name is present, it receives a dictionary containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form
*name* (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (*name* must occur before **name.*) For example, if we define a function like this:

```python
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, '?'
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments: print arg
    print '=-'*40
    keys = keywords.keys()
    keys.sort()
    for kw in keys: print kw, ':', keywords[kw]
```

It could be called like this:

```python
cheeseshop('Limburger', "It's very runny, sir.",
        "It's really very, VERY runny, sir.",
        client='John Cleese',
        shopkeeper='Michael Palin',
        sketch='Cheese Shop Sketch')
```

and of course it would print:

```
-- Do you have any Limburger?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Note that the `sort()` method of the list of keyword argument names is called before printing the contents of the `keywords` dictionary; if this is not done, the order in which the arguments are printed is undefined.
4.7.3 Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur.

```python
def fprintf(file, format, *args):
    file.write(format % args)
```
4.7.4 Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate `start` and `stop` arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

```python
>>> range(3, 6)  # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)  # call with arguments unpacked from a list
[3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments with the `**`-operator:

```python
>>> def parrot(voltage, state='a stiff', action='VOOM',):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it.",
...     print "E's", state, "!
...     
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM",}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through i
```
4.7.5 Lambda Forms

By popular demand, a few features commonly found in functional programming languages like Lisp have been added to Python. With the `lambda` keyword, small anonymous functions can be created. Here's a function that returns the sum of its two arguments: "\( \text{lambda } a, b: a+b \)". Lambda forms can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda forms can reference variables from the containing scope:

```python
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```
4.7.6 Documentation Strings

There are emerging conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line after the first line of the string determines the amount of indentation for the entire documentation string. (We can't use the first line since it is generally adjacent to the string's opening quotes so its indentation is not apparent in the string literal.) Whitespace ```equivalent`` to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a multi-line docstring:

```python
>>> def my_function():
    ...
    """Do nothing, but document it.
    ...
    ...
    No, really, it doesn't do anything.
    ...
    ""
    ...
    pass
    ...
>>> print my_function.__doc__
Do nothing, but document it.
    No, really, it doesn't do anything.
```
Footnotes

... object).  

Actually, *call by object reference* would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it (items inserted into a list).
Subsections

- 5.1 More on Lists
  - 5.1.1 Using Lists as Stacks
  - 5.1.2 Using Lists as Queues
  - 5.1.3 Functional Programming Tools
  - 5.1.4 List Comprehensions
- 5.2 The del statement
- 5.3 Tuples and Sequences
- 5.4 Sets
- 5.5 Dictionaries
- 5.6 Looping Techniques
- 5.7 More on Conditions
- 5.8 Comparing Sequences and Other Types
5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.
5.1 More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

- **append**(\(x\))
  Add an item to the end of the list; equivalent to \(a[\text{len}(a):] = [x]\).

- **extend**(\(L\))
  Extend the list by appending all the items in the given list; equivalent to \(a[\text{len}(a):] = L\).

- **insert**(\(i, x\))
  Insert an item at a given position. The first argument is the index of the element before which to insert, so \(a\text{.insert}(0, x)\) inserts at the front of the list, and \(a\text{.insert}(\text{len}(a), x)\) is equivalent to \(a\text{.append}(x)\).

- **remove**(\(x\))
  Remove the first item from the list whose value is \(x\). It is an error if there is no such item.

- **pop**(\([i]\))
  Remove the item at the given position in the list, and return it. If no index is specified, \(a\text{.pop}()\) removes and returns the last item in the list. (The square brackets around the \(i\) in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the *Python Library Reference*.)

- **index**(\(x\))
  Return the index in the list of the first item whose value is \(x\). It is an error if there is no such item.

- **count**(\(x\))
Return the number of times \( x \) appears in the list.

**sort()**

Sort the items of the list, in place.

**reverse()**

Reverse the elements of the list, in place.

An example that uses most of the list methods:

```python
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```
5.1.1 Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```python
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```
5.1.2 Using Lists as Queues

You can also use a list conveniently as a queue, where the first element added is the first element retrieved (``first-in, first-out''). To add an item to the back of the queue, use `append()`. To retrieve an item from the front of the queue, use `pop()` with 0 as the index. For example:

```python
>>> queue = ['Eric', 'John', 'Michael']
>>> queue.append('Terry')  # Terry arrives
>>> queue.append('Graham')  # Graham arrives
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```
5.1.3 Functional Programming Tools

There are three built-in functions that are very useful when used with lists: filter(), map(), and reduce().

"filter(function, sequence)" returns a sequence consisting of those items from the sequence for which function(item) is true. If sequence is a string or tuple, the result will be of the same type; otherwise, it is always a list. For example, to compute some primes:

```python
>>> def f(x): return x % 2 != 0 and x % 3 != 0
... >>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

"map(function, sequence)" calls function(item) for each of the sequence's items and returns a list of the return values. For example, to compute some cubes:

```python
>>> def cube(x): return x*x*x
... >>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

More than one sequence may be passed; the function must then have as many arguments as there are sequences and is called with the corresponding item from each sequence (or None if some sequence is shorter than another). For example:

```python
>>> seq = range(8)
>>> def add(x, y): return x+y
... >>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

"reduce(function, sequence)" returns a single value constructed by calling the binary function function on the first two items of the sequence, then on the result and the next item, and so on. For example, to compute the sum of the numbers 1 through 10:

```python
>>> def add(x, y): return x+y
... >>> reduce(add, range(1, 11))
55
```
If there's only one item in the sequence, its value is returned; if the sequence is empty, an exception is raised.

A third argument can be passed to indicate the starting value. In this case the starting value is returned for an empty sequence, and the function is first applied to the starting value and the first sequence item, then to the result and the next item, and so on. For example,

```python
def sum(seq):
    def add(x, y):
        return x + y
    return reduce(add, seq, 0)

>>> sum(range(1, 11))
55
>>> sum([])
0
```

Don't use this example's definition of `sum()`: since summing numbers is such a common need, a built-in function `sum(sequence)` is already provided, and works exactly like this. New in version 2.3.
5.1.4 List Comprehensions

List comprehensions provide a concise way to create lists without resorting to use of `map()`, `filter()` and/or `lambda`. The resulting list definition tends often to be clearer than lists built using those constructs. Each list comprehension consists of an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. If the expression would evaluate to a tuple, it must be parenthesized.

```python
>>> freshfruit = ['banana', 'loganberry', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit ']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [(x, x**2) for x in vec]
# error - parens required for tuples
File "<stdin>", line 1, in ?
    [x, x**2 for x in vec]
^ SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

List comprehensions are much more flexible than `map()` and can be applied to complex expressions and nested functions:

```python
>>> ['str(round(355/113.0, i)) for i in range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```
5.2 The `del` statement

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```python
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` can also be used to delete entire variables:

```python
>>> del a
```

Referencing the name `a` hereafter is an error (at least until another value is assigned to it). We'll find other uses for `del` later.
5.3 Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of sequence data types. Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the tuple.

A tuple consists of a number of values separated by commas, for instance:

```python
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression).

Tuples have many uses. For example: (x, y) coordinate pairs, employee records from a database, etc. Tuples, like strings, are immutable: it is not possible to assign to the individual items of a tuple (you can simulate much of the same effect with slicing and concatenation, though). It is also possible to create tuples which contain mutable objects, such as lists.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```python
>>> empty = ()
>>> singleton = 'hello',  # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
```
The statement `t = 12345, 54321, 'hello!'` is an example of tuple packing: the values 12345, 54321 and 'hello!' are packed together in a tuple. The reverse operation is also possible:

```python
>>> x, y, z = t
```

This is called, appropriately enough, sequence unpacking. Sequence unpacking requires the list of variables on the left to have the same number of elements as the length of the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking!

There is a small bit of asymmetry here: packing multiple values always creates a tuple, and unpacking works for any sequence.
5.4 Sets

Python also includes a data type for sets. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Here is a brief demonstration:

```python
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)  # create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit  # fast membership testing
True
>>> 'crabgrass' in fruit
False

>>> # Demonstrate set operations on unique letters from two words...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a  # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b  # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b  # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b  # letters in both a and b
set(['a', 'c'])
>>> a ^ b  # letters in a or b but not b
set(['r', 'd', 'b', 'm', 'z', 'l'])
```
5.5 Dictionaries

Another useful data type built into Python is the dictionary. Dictionaries are sometimes found in other languages as "associative memories" or "associative arrays". Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().

It is best to think of a dictionary as an unordered set of key: value pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

The keys( ) method of a dictionary object returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just apply the sort( ) method to the list of keys). To check whether a single key is in the dictionary, either use the dictionary's has_key( ) method or the in keyword.

Here is a small example using a dictionary:

```python
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
```
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
True
>>> 'guido' in tel
True

The `dict()` constructor builds dictionaries directly from lists of key-value pairs stored as tuples. When the pairs form a pattern, list comprehensions can compactly specify the key-value list.

    >>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
    {'sape': 4139, 'jack': 4098, 'guido': 4127}
    >>> dict([(x, x**2) for x in (2, 4, 6)])  # use a list comprehension
    {2: 4, 4: 16, 6: 36}

Later in the tutorial, we will learn about Generator Expressions which are even better suited for the task of supplying key-values pairs to the `dict()` constructor.

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

    >>> dict(sape=4139, guido=4127, jack=4098)
    {'sape': 4139, 'jack': 4098, 'guido': 4127}
5.6 Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `iteritems()` method.

```python
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function.

```python
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function.

```python
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your %s? It is %s.' % (q, a)
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the `reversed()` function.

```python
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
5
3
```
To loop over a sequence in sorted order, use the `sorted()` function which returns a new sorted list while leaving the source unaltered.

```python
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print f
...     apple
...     banana
...     orange
...     pear
```
5.7 More on Conditions

The conditions used in while and if statements can contain any operators, not just comparisons.

The comparison operators in and not in check whether a value occurs (does not occur) in a sequence. The operators is and is not compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, a < b == c tests whether a is less than b and moreover b equals c.

Comparisons may be combined using the Boolean operators and and or, and the outcome of a comparison (or of any other Boolean expression) may be negated with not. These have lower priorities than comparison operators; between them, not has the highest priority and or the lowest, so that A and not B or C is equivalent to (A and (not B)) or C. As always, parentheses can be used to express the desired composition.

The Boolean operators and and or are so-called short-circuit operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if A and C are true but B is false, A and B and C does not evaluate the expression C. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example,

```python
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Note that in Python, unlike C, assignment cannot occur inside expressions. C programmers may grumble about this, but it avoids a common class of problems.
encountered in C programs: typing = in an expression when == was intended.
5.8 Comparing Sequences and Other Types

Sequence objects may be compared to other objects with the same sequence type. The comparison uses lexicographical ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the ASCII ordering for individual characters. Some examples of comparisons between sequences of the same type:

\[
\begin{align*}
(1, 2, 3) & < (1, 2, 4) \\
[1, 2, 3] & < [1, 2, 4] \\
'ABC' & < 'C' < 'Pascal' < 'Python' \\
(1, 2, 3, 4) & < (1, 2, 4) \\
(1, 2) & < (1, 2, -1) \\
(1, 2, 3) & == (1.0, 2.0, 3.0) \\
(1, 2, ('aa', 'ab')) & < (1, 2, ('abc', 'a'), 4)
\end{align*}
\]

Note that comparing objects of different types is legal. The outcome is deterministic but arbitrary: the types are ordered by their name. Thus, a list is always smaller than a string, a string is always smaller than a tuple, etc. \footnote{5.1} Mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc.

---

**Footnotes**

\footnote{5.1} The rules for comparing objects of different types should not be relied upon; they may change in a future version of the language.
Release 2.5, documentation updated on 19th September, 2006.  
See About this document... for information on suggesting changes.
Subsections

- **6.1 More on Modules**
  - 6.1.1 The Module Search Path
  - 6.1.2 ``Compiled" Python files
- **6.2 Standard Modules**
- **6.3 The `dir()` Function**
- **6.4 Packages**
  - 6.4.1 Importing `*` From a Package
  - 6.4.2 Intra-package References
  - 6.4.3 Packages in Multiple Directories
6. Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a script. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable __name__. For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents:

```python
# Fibonacci numbers module

def fib(n):  # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n):  # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:
>>> import fibo

This does not enter the names of the functions defined in `fibo` directly in the current symbol table; it only enters the module name `fibo` there. Using the module name you can access the functions:

```python
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```
```python
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```
```python
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```python
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```
6.1 More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module is imported somewhere. 6.1

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, `modname.itemname`.

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

There is a variant of the `import` statement that imports names from a module directly into the importing module's symbol table. For example:

```python
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```python
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (_).
6.1.1 The Module Search Path

When a module named spam is imported, the interpreter searches for a file named spam.py in the current directory, and then in the list of directories specified by the environment variable PYTHONPATH. This has the same syntax as the shell variable PATH, that is, a list of directory names. When PYTHONPATH is not set, or when the file is not found there, the search continues in an installation-dependent default path; on UNIX, this is usually .:/usr/local/lib/python.

Actually, modules are searched in the list of directories given by the variable sys.path which is initialized from the directory containing the input script (or the current directory), PYTHONPATH and the installation-dependent default. This allows Python programs that know what they're doing to modify or replace the module search path. Note that because the directory containing the script being run is on the search path, it is important that the script not have the same name as a standard module, or Python will attempt to load the script as a module when that module is imported. This will generally be an error. See section 6.2, ``Standard Modules,'' for more information.
6.1.2 "Compiled" Python files

As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called `spam.pyc` exists in the directory where `spam.py` is found, this is assumed to contain an already-``byte-compiled" version of the module `spam`. The modification time of the version of `spam.py` used to create `spam.pyc` is recorded in `spam.py`, and the `.pyc` file is ignored if these don't match.

Normally, you don't need to do anything to create the `spam.pyc` file. Whenever `spam.py` is successfully compiled, an attempt is made to write the compiled version to `spam.pyc`. It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting `spam.pyc` file will be recognized as invalid and thus ignored later. The contents of the `spam.pyc` file are platform independent, so a Python module directory can be shared by machines of different architectures.

Some tips for experts:

- When the Python interpreter is invoked with the `-O` flag, optimized code is generated and stored in `.pyo` files. The optimizer currently doesn't help much; it only removes `assert` statements. When `-O` is used, all bytecode is optimized; `.pyc` files are ignored and `.py` files are compiled to optimized bytecode.

- Passing two `-O` flags to the Python interpreter (`-OO`) will cause the bytecode compiler to perform optimizations that could in some rare cases result in malfunctioning programs. Currently only `__doc__` strings are removed from the bytecode, resulting in more compact `.pyo` files. Since some programs may rely on having these available, you should only use this option if you know what you're doing.

- A program doesn't run any faster when it is read from a `.pyc` or `.pyo` file than when it is read from a `.py` file; the only thing that's faster about `.pyc` or `.pyo` files is the speed with which they are loaded.

- When a script is run by giving its name on the command line, the bytecode for the script is never written to a `.pyc` or `.pyo` file. Thus, the startup time of
a script may be reduced by moving most of its code to a module and having a small bootstrap script that imports that module. It is also possible to name a .pyc or .pyo file directly on the command line.

- It is possible to have a file called spam.pyc (or spam.pyo when -O is used) without a file spam.py for the same module. This can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer.

- The module compileall can create .pyc files (or .pyo files when -O is used) for all modules in a directory.
6.2 Standard Modules

Python comes with a library of standard modules, described in a separate document, the *Python Library Reference* ("Library Reference" hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the *amoeba* module is only provided on systems that somehow support Amoeba primitives. One particular module deserves some attention: *sys*, which is built into every Python interpreter. The variables *sys.ps1* and *sys.ps2* define the strings used as primary and secondary prompts:

```python
>>> import sys
>>> sys.ps1
'>>>
>>> sys.ps2
'...
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable *sys.path* is a list of strings that determines the interpreter's search path for modules. It is initialized to a default path taken from the environment variable PYTHONPATH, or from a built-in default if PYTHONPATH is not set. You can modify it using standard list operations:

```python
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```
6.3 The dir() Function

The built-in function dir() is used to find out which names a module defines. It returns a sorted list of strings:

```python
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
'__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
'builtin_module_names', 'byteorder', 'callstats', 'copyright',
'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
'exec_prefix', 'modules', 'path', 'path_hooks', 'path_importer_cache' 
'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', ' 
'version', 'version_info', 'warnoptions']
```

Without arguments, dir() lists the names you have defined currently:

```python
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'fib', 'fibo',
'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

dir() does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module __builtin__:

```python
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'Deprecation 
'ERROR', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
'FloatingPointError', 'FutureWarning', 'IOError', 'ImportError',
'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented 
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
```
6.4 Packages

Packages are a way of structuring Python's module namespace by using ``dotted module names''. For example, the module name A.B designates a submodule named "B" in a package named "A". Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other's module names.

Suppose you want to design a collection of modules (a "package") for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: .wav, .aiff, .au), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

```
Sound/
    __init__.py
    Formats/
        __init__.py
        wavread.py
        wavwrite.py
        aiffread.py
        aiffwrite.py
        auread.py
        auwrite.py
        ...
    Effects/
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
    Filters/
        __init__.py
        equalizer.py
        vocoder.py
```

Top-level package
Initialize the sound package
Subpackage for file format conversion
Subpackage for sound effects
Subpackage for filters
When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as "string", from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```python
import Sound.Effects.echo
```

This loads the submodule `Sound.Effects.echo`. It must be referenced with its full name.

```python
Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

An alternative way of importing the submodule is:

```python
from Sound.Effects import echo
```

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

```python
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```python
from Sound.Effects.echo import echofilter
```

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

```python
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from package import item`, the item can be either a
submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The import statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an ImportError exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.
6.4.1 Importing * From a Package

Now what happens when the user writes \texttt{from Sound.Effects import *}? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. Unfortunately, this operation does not work very well on Mac and Windows platforms, where the filesystem does not always have accurate information about the case of a filename! On these platforms, there is no guaranteed way to know whether a file \texttt{ECHO.PY} should be imported as a module \texttt{echo}, \texttt{Echo} or \texttt{ECHO}. (For example, Windows 95 has the annoying practice of showing all filenames with a capitalized first letter.) The DOS 8+3 filename restriction adds another interesting problem for long module names.

The only solution is for the package author to provide an explicit index of the package. The import statement uses the following convention: if a package's \texttt{\_init\_\_\_\_.py} code defines a list named \texttt{\_all\_}, it is taken to be the list of module names that should be imported when \texttt{from package import *} is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing * from their package. For example, the file \\
\texttt{Sounds/Effects/\_\_init\_\_\_.py} could contain the following code:

\begin{verbatim}
\_all\_ = ["echo", "surround", "reverse"]
\end{verbatim}

This would mean that \texttt{from Sound.Effects import *} would import the three named submodules of the Sound package.

If \texttt{\_all\_} is not defined, the statement \texttt{from Sound.Effects import *} does not import all submodules from the package \texttt{Sound.Effects} into the current namespace; it only ensures that the package \texttt{Sound.Effects} has been imported (possibly running any initialization code in \texttt{\_init\_\_\_.py}) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by \texttt{\_init\_\_\_.py}. It also includes any submodules of the package that were explicitly loaded by previous import statements. Consider this code:

\begin{verbatim}
import Sound.Effects.echo
import Sound.Effects.surround
\end{verbatim}
from Sound.Effects import *

In this example, the echo and surround modules are imported in the current namespace because they are defined in the Sound.Effects package when the from...import statement is executed. (This also works when __all__ is defined.)

Note that in general the practice of importing * from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions, and certain modules are designed to export only names that follow certain patterns.

Remember, there is nothing wrong with using from Package import specific_submodule! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.
6.4.2 Intra-package References

The submodules often need to refer to each other. For example, the `surround` module might use the `echo` module. In fact, such references are so common that the `import` statement first looks in the containing package before looking in the standard module search path. Thus, the `surround` module can simply use `import echo` or `from echo import echofilter`. If the imported module is not found in the current package (the package of which the current module is a submodule), the `import` statement looks for a top-level module with the given name.

When packages are structured into subpackages (as with the `Sound` package in the example), there's no shortcut to refer to submodules of sibling packages - the full name of the subpackage must be used. For example, if the module `Sound.Filters.vocoder` needs to use the `echo` module in the `Sound.Effects` package, it can use `from Sound.Effects import echo`.

Starting with Python 2.5, in addition to the implicit relative imports described above, you can write explicit relative imports with the `from module import name` form of import statement. These explicit relative imports use leading dots to indicate the current and parent packages involved in the relative import. From the `surround` module for example, you might use:

```python
from . import echo
from .. import Formats
from ..Filters import equalizer
```

Note that both explicit and implicit relative imports are based on the name of the current module. Since the name of the main module is always "__main__", modules intended for use as the main module of a Python application should always use absolute imports.
6.4.3 Packages in Multiple Directories

Packages support one more special attribute, __path__. This is initialized to be a list containing the name of the directory holding the package's __init__.py before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

While this feature is not often needed, it can be used to extend the set of modules found in a package.

Footnotes

... somewhere.\footnote{6.1}
In fact function definitions are also `statements' that are `executed'; the execution enters the function name in the module's global symbol table.
Subsections

- [7.1 Fancier Output Formatting](#)
- [7.2 Reading and Writing Files](#)
  - [7.2.1 Methods of File Objects](#)
  - [7.2.2 The pickle Module](#)
7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.
7.1 Fancier Output Formatting

So far we've encountered two ways of writing values: *expression statements* and the *print* statement. (A third way is using the *write()* method of file objects; the standard output file can be referenced as *sys.stdout*. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are two ways to format your output; the first way is to do all the string handling yourself; using string slicing and concatenation operations you can create any layout you can imagine. The standard module *string* contains some useful operations for padding strings to a given column width; these will be discussed shortly. The second way is to use the % operator with a string as the left argument. The % operator interprets the left argument much like a *sprintf()*-style format string to be applied to the right argument, and returns the string resulting from this formatting operation.

One question remains, of course: how do you convert values to strings? Luckily, Python has ways to convert any value to a string: pass it to the *repr()* or *str()* functions. Reverse quotes (``) are equivalent to *repr()* , but they are no longer used in modern Python code and will likely not be in future versions of the language.

The *str()* function is meant to return representations of values which are fairly human-readable, while *repr()* is meant to generate representations which can be read by the interpreter (or will force a *SyntaxError* if there is not equivalent syntax). For objects which don't have a particular representation for human consumption, *str()* will return the same value as *repr()* . Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings and floating point numbers, in particular, have two distinct representations.

Some examples:

```python
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
'"Hello, world."'
```
"'Hello, world.'"

>>> str(0.1)
'0.1'

>>> repr(0.1)
'0.10000000000000001'

>>> x = 10 * 3.25
>>> y = 200 * 200

>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '
>>> print s
The value of x is 32.5, and y is 40000...

>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'

>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'

>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"

>>> # reverse quotes are convenient in interactive sessions:
... `x, y, ('spam', 'eggs')`
"(32.5, 40000, ('spam', 'eggs'))"

Here are two ways to write a table of squares and cubes:

```python
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Note trailing comma on previous line
...     print repr(x*x*x).rjust(4)
...
   1    1    1
   2    4    8
   3    9   27
   4   16   64
   5   25  125
   6   36  216
   7   49  343
   8   64  512
   9   81  729
  10 100 1000
... for x in range(1,11):
...     print '%2d %3d %4d' % (x, x*x, x*x*x)
...
   1    1    1
   2    4    8
   3    9   27
   4   16   64
   5   25  125
   6   36  216
   7   49  343
   8   64  512
   9   81  729
  10 100 1000
```
(Note that one space between each column was added by the way print works: it always adds spaces between its arguments.)

This example demonstrates the \texttt{rjust()} method of string objects, which right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods \texttt{ljust()} and \texttt{center()}. These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in "x.ljust(n)[:n].")

There is another method, \texttt{zfill()}, which pads a numeric string on the left with zeros. It understands about plus and minus signs:

\begin{verbatim}
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
\end{verbatim}

Using the \% operator looks like this:

\begin{verbatim}
>>> import math
>>> print 'The value of PI is approximately %5.3f.' % math.pi
The value of PI is approximately 3.142.
\end{verbatim}

If there is more than one format in the string, you need to pass a tuple as right operand, as in this example:

\begin{verbatim}
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...         print '%-10s ==> %10d' % (name, phone)
... Jack ==> 4098
Dcab ==> 7678
Sjoerd ==> 4127
\end{verbatim}

Most formats work exactly as in C and require that you pass the proper type; however, if you don't you get an exception, not a core dump. The \texttt{\%s} format is more relaxed: if the corresponding argument is not a string object, it is converted to string using the \texttt{str()} built-in function. Using * to pass the width or
precision in as a separate (integer) argument is supported. The C formats %n and %p are not supported.

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by using form %%(name)format, as shown here:

```python
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' % table
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the new built-in `vars()` function, which returns a dictionary containing all local variables.
7.2 Reading and Writing Files

`open()` returns a file object, and is most commonly used with two arguments: "`open(filename, mode)`".

```python
>>> f=open('/tmp/workfile', 'w')
>>> print(f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. `mode` can be `r` when the file will only be read, `w` for only writing (an existing file with the same name will be erased), and `a` opens the file for appending; any data written to the file is automatically added to the end. `r+` opens the file for both reading and writing. The `mode` argument is optional; `r` will be assumed if it's omitted.

On Windows and the Macintosh, 'b' appended to the mode opens the file in binary mode, so there are also modes like `rb`, `wb`, and `r+b`. Windows makes a distinction between text and binary files; the end-of-line characters in text files are automatically altered slightly when data is read or written. This behind-the-scenes modification to file data is fine for ASCII text files, but it'll corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files.
7.2.1 Methods of File Objects

The rest of the examples in this section will assume that a file object called \texttt{f} has already been created.

To read a file's contents, call \texttt{f.read(size)}, which reads some quantity of data and returns it as a string. \texttt{size} is an optional numeric argument. When \texttt{size} is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most \texttt{size} bytes are read and returned. If the end of the file has been reached, \texttt{f.read()} will return an empty string (""").

\begin{verbatim}
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
\end{verbatim}

\texttt{f.readline()} reads a single line from the file; a newline character (\texttt{\textbackslash n}) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if \texttt{f.readline()} returns an empty string, the end of the file has been reached, while a blank line is represented by ' \texttt{\textbackslash n} ', a string containing only a single newline.

\begin{verbatim}
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
\end{verbatim}

\texttt{f.readlines()} returns a list containing all the lines of data in the file. If given an optional parameter \texttt{sizehint}, it reads that many bytes from the file and enough more to complete a line, and returns the lines from that. This is often used to allow efficient reading of a large file by lines, but without having to load the entire file in memory. Only complete lines will be returned.

\begin{verbatim}
>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the file\n'
\end{verbatim}

An alternate approach to reading lines is to loop over the file object. This is
memory efficient, fast, and leads to simpler code:

```python
>>> for line in f:
    print line,
This is the first line of the file.
Second line of the file
```

The alternative approach is simpler but does not provide as fine-grained control. Since the two approaches manage line buffering differently, they should not be mixed.

`f.write(string)` writes the contents of `string` to the file, returning `None`.

```python
>>> f.write('This is a test
')
```

To write something other than a string, it needs to be converted to a string first:

```python
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

`f.tell()` returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file. To change the file object's position, use "f.seek(offset, from_what)". The position is computed from adding `offset` to a reference point; the reference point is selected by the `from_what` argument. A `from_what` value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. `from_what` can be omitted and defaults to 0, using the beginning of the file as the reference point.

```python
>>> f = open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)  # Go to the 6th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

When you're done with a file, call `f.close()` to close it and free up any system resources taken up by the open file. After calling `f.close()`, attempts to use the file object will automatically fail.
File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.
7.2.2 The pickle Module

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like '123' and returns its numeric value 123. However, when you want to save more complex data types like lists, dictionaries, or class instances, things get a lot more complicated.

Rather than have users be constantly writing and debugging code to save complicated data types, Python provides a standard module called `pickle`. This is an amazing module that can take almost any Python object (even some forms of Python code!), and convert it to a string representation; this process is called `pickling`. Reconstructing the object from the string representation is called `unpickling`. Between pickling and unpickling, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

If you have an object `x`, and a file object `f` that's been opened for writing, the simplest way to pickle the object takes only one line of code:

```
pickle.dump(x, f)
```

To unpickle the object again, if `f` is a file object which has been opened for reading:

```
x = pickle.load(f)
```

(There are other variants of this, used when pickling many objects or when you don't want to write the pickled data to a file; consult the complete documentation for `pickle` in the [Python Library Reference](https://docs.python.org/3/library/pickle.html).)

`pickle` is the standard way to make Python objects which can be stored and reused by other programs or by a future invocation of the same program; the technical term for this is a *persistent* object. Because `pickle` is so widely used, many authors who write Python extensions take care to ensure that new data types such as matrices can be properly pickled and unpickled.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
Subsections

- 8.1 Syntax Errors
- 8.2 Exceptions
- 8.3 Handling Exceptions
- 8.4 Raising Exceptions
- 8.5 User-defined Exceptions
- 8.6 Defining Clean-up Actions
- 8.7 Predefined Clean-up Actions
8. Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: syntax errors and exceptions.
8.1 Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```python
>>> while True print 'Hello world'
File "<stdin>", line 1, in ?
    while True print 'Hello world'
^  
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little `arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token preceding the arrow: in the example, the error is detected at the keyword `print', since a colon (":") is missing before it. File name and line number are printed so you know where to look in case the input came from a script.
8.2 Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```python
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in 
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in 
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in 
TypeError: cannot concatenate 'str' and 'int' objects
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError` and `TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

The [Python Library Reference](https://docs.python.org/3/library/exceptions.html) lists the built-in exceptions and their meanings.
8.3 Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-C or whatever the operating system supports); note that a user-generated interruption is signalled by raising the `KeyboardInterrupt` exception.

```python
>>> while True:
    ...     try:
    ...         x = int(input("Please enter a number: "))
    ...         break
    ...     except ValueError:
    ...         print("Oops! That was no valid number. Try again...")
    ...
```

The `try` statement works as follows.

- First, the `try clause` (the statement(s) between the `try` and `except` keywords) is executed.

- If no exception occurs, the `except clause` is skipped and execution of the `try` statement is finished.

- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the `except` keyword, the except clause is executed, and then execution continues after the `try` statement.

- If an exception occurs which does not match the exception named in the `except clause`, it is passed on to outer `try` statements; if no handler is found, it is an `unhandled exception` and execution stops with a message as shown above.

A `try` statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same `try` statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:
... except (RuntimeError, TypeError, NameError):
    pass

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

    import sys
    try:
        f = open('myfile.txt')
        s = f.readline()
        i = int(s.strip())
    except IOError, (errno, strerror):
        print "I/O error(%s): %s" % (errno, strerror)
    except ValueError:
        print "Could not convert data to an integer."
    except:
        print "Unexpected error:", sys.exc_info()[0]
        raise

The try ... except statement has an optional else clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

    for arg in sys.argv[1:]:
        try:
            f = open(arg, 'r')
        except IOError:
            print 'cannot open', arg
        else:
            print arg, 'has', len(f.readlines()), 'lines'
            f.close()

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try ... except statement.

When an exception occurs, it may have an associated value, also known as the exception's argument. The presence and type of the argument depend on the exception type.

The except clause may specify a variable after the exception name (or tuple). The variable is bound to an exception instance with the arguments stored in
instance.args. For convenience, the exception instance defines
__getitem__ and __str__ so the arguments can be accessed or printed
directly without having to reference .args.

But use of .args is discouraged. Instead, the preferred use is to pass a single
argument to an exception (which can be a tuple if multiple arguments are
needed) and have it bound to the message attribute. One my also instantiate an
exception first before raising it and add any attributes to it as desired.

```python
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception, inst:
...     print type(inst)        # the exception instance
...     print inst.args         # arguments stored in .args
...     print inst              # __str__ allows args to printed directl
...     x, y = inst             # __getitem__ allows args to be unpacked
...     print 'x =', x
...     print 'y =', y
... <type 'instance'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

If an exception has an argument, it is printed as the last part (`detail`) of the
message for unhandled exceptions.

Exception handlers don't just handle exceptions if they occur immediately in the
try clause, but also if they occur inside functions that are called (even indirectly)
in the try clause. For example:

```python
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError, detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo by zero
```
8.4 Raising Exceptions

The raise statement allows the programmer to force a specified exception to occur. For example:

```python
>>> raise NameError, 'HiThere'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

The first argument to raise names the exception to be raised. The optional second argument specifies the exception's argument. Alternatively, the above could be written as `raise NameError('HiThere')`. Either form works fine, but there seems to be a growing stylistic preference for the latter.

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

```python
>>> try:
...    raise NameError, 'HiThere'
... except NameError:
...    print 'An exception flew by!'
...    raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```
8.5 User-defined Exceptions

Programs may name their own exceptions by creating a new exception class. Exceptions should typically be derived from the Exception class, either directly or indirectly. For example:

```python
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError, e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError, 'oops!'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

In this example, the default `__init__` of `Exception` has been overridden. The new behavior simply creates the `value` attribute. This replaces the default behavior of creating the `args` attribute.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

```python
class Error(Exception):
    """Base class for exceptions in this module.""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

Attributes:
    expression -- input expression in which the error occurred
    message -- explanation of the error
```
def __init__(self, expression, message):
    self.expression = expression
    self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not allowed."

    Attributes:
    previous -- state at beginning of transition
    next -- attempted new state
    message -- explanation of why the specific transition is not allowed
    ""
    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message

Most exceptions are defined with names that end in ``Error,'' similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define. More information on classes is presented in chapter 9, ``Classes.''

8.6 Defining Clean-up Actions

The try statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```python
>>> try:
...    raise KeyboardInterrupt
... finally:
...    print 'Goodbye, world!'
...
Goodbye, world!
```

A finally clause is always executed before leaving the try statement, whether an exception has occurred or not. When an exception has occurred in the try clause and has not been handled by an except clause (or it has occurred in a except or else clause), it is re-raised after the finally clause has been executed. The finally clause is also executed "on the way out" when any other clause of the try statement is left via a break, continue or return statement. A more complicated example:

```python
>>> def divide(x, y):
...    try:
...        result = x / y
...    except ZeroDivisionError:
...        print "division by zero!"
...    else:
...        print "result is", result
... finally:
...    print "executing finally clause"
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```
As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clauses has been executed.

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.
8.7 Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```python
for line in open("myfile.txt"):
    print line
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```python
with open("myfile.txt") as f:
    for line in f:
        print line
```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Other objects which provide predefined clean-up actions will indicate this in their documentation.
Subsections

- 9.1 A Word About Terminology
- 9.2 Python Scopes and Name Spaces
- 9.3 A First Look at Classes
  - 9.3.1 Class Definition Syntax
  - 9.3.2 Class Objects
  - 9.3.3 Instance Objects
  - 9.3.4 Method Objects
- 9.4 Random Remarks
- 9.5 Inheritance
  - 9.5.1 Multiple Inheritance
- 9.6 Private Variables
- 9.7 Odds and Ends
- 9.8 Exceptions Are Classes Too
- 9.9 Iterators
- 9.10 Generators
- 9.11 Generator Expressions
9. Classes

Python's class mechanism adds classes to the language with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. As is true for modules, classes in Python do not put an absolute barrier between definition and user, but rather rely on the politeness of the user not to ``break into the definition.'' The most important features of classes are retained with full power, however: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain an arbitrary amount of private data.

In C++ terminology, all class members (including the data members) are public, and all member functions are virtual. There are no special constructors or destructors. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects, albeit in the wider sense of the word: in Python, all data types are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++ but unlike in Modula-3, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.
9.1 A Word About Terminology

Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. (I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has an (intended!) effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most types representing entities outside the program (files, windows, etc.). This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change -- this eliminates the need for two different argument passing mechanisms as in Pascal.
9.2 Python Scopes and Name Spaces

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A namespace is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (functions such as abs( ), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function ``maximize'' without confusion -- users of the modules must prefix it with the module name.

By the way, I use the word attribute for any name following a dot -- for example, in the expression z.real, real is an attribute of the object z. Strictly speaking, references to names in modules are attribute references: in the expression modname.funcname, modname is a module object and funcname is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! 9.1

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write "modname.the_answer = 42". Writable attributes may also be deleted with the del statement. For example, "del modname.the_answer" will remove the attribute the_answer from the object named by modname.
Name spaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called __main__, so they have their own global namespace. (The built-in names actually also live in a module; this is called __builtin__.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A scope is a textual region of a Python program where a namespace is directly accessible. "Directly accessible" here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespaces are directly accessible: the innermost scope, which is searched first, contains the local names; the namespaces of any enclosing functions, which are searched starting with the nearest enclosing scope; the middle scope, searched next, contains the current module's global names; and the outermost scope (searched last) is the namespace containing built-in names.

If a name is declared global, then all references and assignments go directly to the middle scope containing the module's global names. Otherwise, all variables found outside of the innermost scope are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of
a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time -- however, the language definition is evolving towards static name resolution, at ``compile" time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that assignments always go into the innermost scope. Assignments do not copy data -- they just bind names to objects. The same is true for deletions: the statement "\texttt{del } x\texttt{"} removes the binding of \texttt{x} from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, import statements and function definitions bind the module or function name in the local scope. (The \texttt{global} statement can be used to indicate that particular variables live in the global scope.)
9.3 A First Look at Classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.
9.3.1 Class Definition Syntax

The simplest form of class definition looks like this:

```python
class ClassName:
    <statement-1>
    ...
    <statement-N>
```

Class definitions, like function definitions (def statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an if statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful -- we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods -- again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope -- thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a class object is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (ClassName in the example).
9.3.2 Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```python
class MyClass:
    "A simple example class"
    i = 12345
    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

Class instantiation uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```python
x = MyClass()
```

creates a new instance of the class and assigns this object to the local variable `x`.

The instantiation operation (``calling'' a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```python
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance. So in this example,
a new, initialized instance can be obtained by:

    x = MyClass()

Of course, the __init__() method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to __init__(). For example,

    >>> class Complex:
    ...     def __init__(self, realpart, imagpart):
    ...         self.r = realpart
    ...         self.i = imagpart
    ...
    >>> x = Complex(3.0, -4.5)
    >>> x.r, x.i
    (3.0, -4.5)
9.3.3 Instance Objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

*data attributes* correspond to "instance variables" in Smalltalk, and to "data members" in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if \( x \) is the instance of \( \text{MyClass} \) created above, the following piece of code will print the value 16, without leaving a trace:

```python
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

The other kind of instance attribute reference is a *method*. A method is a function that "belongs to" an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called append, insert, remove, sort, and so on. However, in the following discussion, we'll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, \( x.f \) is a valid method reference, since \( \text{MyClass}.f \) is a function, but \( x.i \) is not, since \( \text{MyClass}.i \) is not. But \( x.f \) is not the same thing as \( \text{MyClass}.f \) -- it is a *method object*, not a function object.
9.3.4 Method Objects

Usually, a method is called right after it is bound:

   x.f()

In the MyClass example, this will return the string 'hello world'. However, it is not necessary to call a method right away: x.f is a method object, and can be stored away and called at a later time. For example:

   xf = x.f
   while True:
     print xf()

will continue to print "hello world" until the end of time.

What exactly happens when a method is called? You may have noticed that x.f() was called without an argument above, even though the function definition for f specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any -- even if the argument isn't actually used...

Actually, you may have guessed the answer: the special thing about methods is that the object is passed as the first argument of the function. In our example, the call x.f() is exactly equivalent to MyClass.f(x). In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

If you still don't understand how methods work, a look at the implementation can perhaps clarify matters. When an instance attribute is referenced that isn't a data attribute, its class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, it is unpacked again, a new argument list is constructed from the instance object and the original argument list, and the function object is called with this new argument list.
9.4 Random Remarks

Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

Data attributes may be referenced by methods as well as by ordinary users (``clients'') of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding -- it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care -- clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided -- again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called self. This is nothing more than a convention: the name self has absolutely no special meaning to Python. (Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a class browser program might be written that relies upon such a convention.)

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also
ok. For example:

```python
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

Now f, g and h are all attributes of class C that refer to function objects, and consequently they are all methods of instances of C -- h being exactly equivalent to g. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the `self` argument:

```python
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing the class definition. (The class itself is never used as a global scope!) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class!
9.5 Inheritance

Of course, a language feature would not be worthy of the name "class" without supporting inheritance. The syntax for a derived class definition looks like this:

```python
class DerivedClassName(BaseClassName):
    <statement-1>  
    ...
    ...
    <statement-N>
```

The name `BaseClassName` must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```python
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively `virtual`.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to
call the base class method directly: just call
"BaseClassName.methodname(self, arguments)". This is
occasionally useful to clients as well. (Note that this only works if the base class
is defined or imported directly in the global scope.)
9.5.1 Multiple Inheritance

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```python
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

The only rule necessary to explain the semantics is the resolution rule used for class attribute references. This is depth-first, left-to-right. Thus, if an attribute is not found in `DerivedClassName`, it is searched in `Base1`, then (recursively) in the base classes of `Base1`, and only if it is not found there, it is searched in `Base2`, and so on.

(To some people breadth first -- searching `Base2` and `Base3` before the base classes of `Base1` -- looks more natural. However, this would require you to know whether a particular attribute of `Base1` is actually defined in `Base1` or in one of its base classes before you can figure out the consequences of a name conflict with an attribute of `Base2`. The depth-first rule makes no differences between direct and inherited attributes of `Base1`.)

It is clear that indiscriminate use of multiple inheritance is a maintenance nightmare, given the reliance in Python on conventions to avoid accidental name conflicts. A well-known problem with multiple inheritance is a class derived from two classes that happen to have a common base class. While it is easy enough to figure out what happens in this case (the instance will have a single copy of "instance variables" or data attributes used by the common base class), it is not clear that these semantics are in any way useful.
9.6 Private Variables

There is limited support for class-private identifiers. Any identifier of the form __spam (at least two leading underscores, at most one trailing underscore) is textually replaced with __classname__spam, where classname is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, so it can be used to define class-private instance and class variables, methods, variables stored in globals, and even variables stored in instances. private to this class on instances of other classes. Truncation may occur when the mangled name would be longer than 255 characters. Outside classes, or when the class name consists of only underscores, no mangling occurs.

Name mangling is intended to give classes an easy way to define ``private'' instance variables and methods, without having to worry about instance variables defined by derived classes, or mucking with instance variables by code outside the class. Note that the mangling rules are designed mostly to avoid accidents; it still is possible for a determined soul to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger, and that's one reason why this loophole is not closed. (Buglet: derivation of a class with the same name as the base class makes use of private variables of the base class possible.)

Notice that code passed to exec, eval() or execfile() does not consider the classname of the invoking class to be the current class; this is similar to the effect of the global statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to getattr(), setattr() and delattr(), as well as when referencing __dict__ directly.
9.7 Odds and Ends

Sometimes it is useful to have a data type similar to the Pascal ``record'' or C ``struct'', bundling together a few named data items. An empty class definition will do nicely:

```python
class Employee:
    pass

john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.im_self` is the instance object with the method `m`, and `m.im_func` is the function object corresponding to the method.
9.8 Exceptions Are Classes Too

User-defined exceptions are identified by classes as well. Using this mechanism it is possible to create extensible hierarchies of exceptions.

There are two new valid (semantic) forms for the raise statement:

\[
\text{raise Class, instance} \\
\text{raise instance}
\]

In the first form, \textit{instance} must be an instance of \texttt{Class} or of a class derived from it. The second form is a shorthand for:

\[
\text{raise instance.__class__, instance}
\]

A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around -- an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```python
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Note that if the except clauses were reversed (with "\texttt{except B}" first), it would have printed B, B, B -- the first matching except clause is triggered.

When an error message is printed for an unhandled exception, the exception's class name is printed, then a colon and a space, and finally the instance
converted to a string using the built-in function `str()`.
By now you have probably noticed that most container objects can be looped over using a `for` statement:

```python
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `next()` which accesses elements in the container one at a time. When there are no more elements, `next()` raises a `StopIteration` exception which tells the `for` loop to terminate. This example shows how it all works:

```python
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
    it.next()
StopIteration
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define a `__iter__()` method which returns an object with a `next()` method. If the class defines `next()`, then
__iter__() can just return self:

class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> for char in Reverse('spam'):
...    print char
...
ma
p
s
9.10 Generators

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called, the generator resumes where it left-off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

```python
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> for char in reverse('golf'):
    ...    print(char
    ...
f
  l
  o
  g
```

Anything that can be done with generators can also be done with class based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `next()` methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function.
### 9.11 Generator Expressions

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Examples:

```python
>>> sum(i*i for i in range(10))  # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))  # dot product
260

>>> from math import pi, sin
>>> sine_table = dict((x, sin(x*pi/180)) for x in range(0, 91))

>>> unique_words = set(word for line in page for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1,-1,-1))
['f', 'l', 'o', 'g']
```

---

**Footnotes**

... namespace!\[^{9.1}\] Except for one thing. Module objects have a secret read-only attribute called \_\_dict\_ which returns the dictionary used to implement the module's namespace; the name \_\_dict\_ is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem
debuggers.
Subsections

- 10.1 Operating System Interface
- 10.2 File Wildcards
- 10.3 Command Line Arguments
- 10.4 Error Output Redirection and Program Termination
- 10.5 String Pattern Matching
- 10.6 Mathematics
- 10.7 Internet Access
- 10.8 Dates and Times
- 10.9 Data Compression
- 10.10 Performance Measurement
- 10.11 Quality Control
- 10.12 Batteries Included
10. Brief Tour of the Standard Library
10.1 Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```python
>>> import os
>>> os.system('time 0:02')
0
>>> os.getcwd()    # Return the current working directory
'C:\Python24'
>>> os.chdir('/server/accesslogs')
```

Be sure to use the "import os" style instead of "from os import *". This will keep `os.open()` from shadowing the builtin `open()` function which operates much differently.

The builtin `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```python
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrin

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```python
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```
10.2 File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```python
>>> import glob
>>> glob.glob("*.py")
['primes.py', 'random.py', 'quote.py']
```
10.3 Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance the following output results from running "python demo.py one two three" at the command line:

```python
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

The `getopt` module processes `sys.argv` using the conventions of the UNIX `getopt()` function. More powerful and flexible command line processing is provided by the `optparse` module.
10.4 Error Output Redirection and Program Termination

The `sys` module also has attributes for `stdin`, `stdout`, and `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected:

```python
>>> sys.stderr.write('Warning, log file not found starting a new one
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use "sys.exit()".
10.5 String Pattern Matching

The `re` module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```python
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'\b[a-z]+\1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```python
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```
10.6 Mathematics

The \texttt{math} module gives access to the underlying C library functions for floating point math:

```python
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

The \texttt{random} module provides tools for making random selections:

```python
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)  # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()  # random float
0.17970987693706186
>>> random.randrange(6)  # random integer chosen from range(6)
4
```
10.7 Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are `urllib2` for retrieving data from urls and `smtplib` for sending mail:

```python
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/... if 'EST' in line or 'EDT' in line:  # look for Eastern Time...
     print line

<BR>Nov. 25, 09:43:32 PM EST
```

```python
>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
      "To: jcaesar@example.org
      From: soothsayer@example.org
      Beware the Ides of March.
      ")
>>> server.quit()
```
10.8 Dates and Times

The `datetime` module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```python
# dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

# dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```
10.9 Data Compression

Common data archiving and compression formats are directly supported by modules including: zlib, gzip, bz2, zipfile, and tarfile.

```python
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```
10.10 Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The `timeit` module quickly demonstrates a modest performance advantage:

```python
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
    0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
    0.54962537085770791
```

In contrast to `timeit`'s fine level of granularity, the `profile` and `pstats` modules provide tools for identifying time critical sections in larger blocks of code.
10.11 Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The doctest module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```python
def average(values):
    """Computes the arithmetic mean of a list of numbers."

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

import doctest
doctest.testmod()  # automatically validate the embedded tests
```

The unittest module is not as effortless as the doctest module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```python
import unittest
class TestStatisticalFunctions(unittest.TestCase):
    def test_average(self):
        self.assertAlmostEqual(average([20, 30, 70]), 40.0)
        self.assertAlmostEqual(round(average([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main()  # Calling from the command line invokes all tests
```
10.12 Batteries Included

Python has a "batteries included" philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The `xmlrpclib` and `SimpleXMLRPCServer` modules make implementing remote procedure calls into an almost trivial task. Despite the modules names, no direct knowledge or handling of XML is needed.
- The `email` package is a library for managing email messages, including MIME and other RFC 2822-based message documents. Unlike `smtplib` and `poplib` which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The `xml.dom` and `xml.sax` packages provide robust support for parsing this popular data interchange format. Likewise, the `csv` module supports direct reads and writes in a common database format. Together, these modules and packages greatly simplify data interchange between python applications and other tools.
- Internationalization is supported by a number of modules including `gettext`, `locale`, and the `codecs` package.
Subsections

- 11.1 Output Formatting
- 11.2 Templating
- 11.3 Working with Binary Data Record Layouts
- 11.4 Multi-threading
- 11.5 Logging
- 11.6 Weak References
- 11.7 Tools for Working with Lists
- 11.8 Decimal Floating Point Arithmetic
11. Brief Tour of the Standard Library - Part II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.
11.1 Output Formatting

The `repr` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```
>>> import repr
>>> repr.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

The `pprint` module offers more sophisticated control over printing both built-in and user defined objects in a way that is readable by the interpreter. When the result is longer than one line, the "pretty printer" adds line breaks and indentation to more clearly reveal data structure:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta', 'yellow'], 'blue']]]
... >>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
  'white',
  ['green', 'red']],
  [['magenta', 'yellow'],
    'blue']]]
```

The `textwrap` module formats paragraphs of text to fit a given screen width:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns a list of strings instead of one big string with newlines to separate the wrapped lines."""
... >>> print textwrap.fill(doc, width=40)
The wrap() method is just like fill() except that it returns a list of strings instead of one big string with newlines to separate the wrapped lines.
```

The `locale` module accesses a database of culture specific data formats. The grouping attribute of locale's format function provides a direct way of formatting numbers with group separators:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
```
'English_United States.1252'
>>> conv = locale.localeconv()  # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format("%s.*f", (conv['currency_symbol'],
... conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
11.2 Templating

The **string** module includes a versatile **Template** class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by "$" with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing "$$" creates a single escaped "$":

```python
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

The **substitute** method raises a **KeyError** when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user supplied data may be incomplete and the **safe_substitute** method may be more appropriate -- it will leave placeholders unchanged if data is missing:

```python
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...  
    KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

```python
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...   delimiter = '%'
>>> fmt = raw_input('Enter rename style (%d-date %n-seqnum %f-format')
Enter rename style (%d-date %n-seqnum %f-format): Ashley_%n%f
```
Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.
11.3 Working with Binary Data Record Layouts

The `struct` module provides `pack()` and `unpack()` functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file (with pack codes "H" and "L" representing two and four byte unsigned numbers respectively):

```python
import struct

data = open('myfile.zip', 'rb').read()
start = 0
for i in range(3):  # show the first 3 file
    start += 14
    fields = struct.unpack('LLLHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields
    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print filename, hex(crc32), comp_size, uncomp_size
    start += extra_size + comp_size  # skip to the next heade
```
11.4 Multi-threading

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background. A related use case is running I/O in parallel with computations in another thread.

The following code shows how the high level `threading` module can run tasks in background while the main program continues to run:

```python
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print 'Finished background zip of: ', self.infile

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print 'The main program continues to run in foreground.'

background.join()   # Wait for the background task to finish
print 'Main program waited until background was done.'
```

The principal challenge of multi-threaded applications is coordinating threads that share data or other resources. To that end, the threading module provides a number of synchronization primitives including locks, events, condition variables, and semaphores.

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the `Queue` module to feed that thread with requests from other threads. Applications using `Queue` objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.
11.5 Logging

The `logging` module offers a full featured and flexible logging system. At its simplest, log messages are sent to a file or to `sys.stderr`:

```python
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces the following output:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

By default, informational and debugging messages are suppressed and the output is sent to standard error. Other output options include routing messages through email, datagrams, sockets, or to an HTTP Server. New filters can select different routing based on message priority: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`.

The logging system can be configured directly from Python or can be loaded from a user editable configuration file for customized logging without altering the application.
11.6 Weak References

Python does automatic memory management (reference counting for most objects and garbage collection to eliminate cycles). The memory is freed shortly after the last reference to it has been eliminated.

This approach works fine for most applications but occasionally there is a need to track objects only as long as they are being used by something else. Unfortunately, just tracking them creates a reference that makes them permanent. The `weakref` module provides tools for tracking objects without creating a reference. When the object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

```python
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)  # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a  # does not create a reference
>>> d['primary']  # fetch the object if it is still
10
>>> del a  # remove the one reference
>>> gc.collect()  # run garbage collection right away
0
>>> d['primary']  # entry was automatically remove
Traceback (most recent call last):
  File "<pyshell#108>", line 1, in -toplevel-
    d['primary']  # entry was automatically remove
File "C:/PY24/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```
11.7 Tools for Working with Lists

Many data structure needs can be met with the built-in list type. However, sometimes there is a need for alternative implementations with different performance trade-offs.

The `array` module provides an `array()` object that is like a list that stores only homogenous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode "H") rather than the usual 16 bytes per entry for regular lists of python int objects:

```python
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

The `collections` module provides a `deque()` object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

```python
>>> from collections import deque
>>> d = deque(['task1', 'task2', 'task3'])
>>> d.append('task4')
>>> print "Handling", d.popleft()
Handling task1

unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

In addition to alternative list implementations, the library also offers other tools such as the `bisect` module with functions for manipulating sorted lists:

```python
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
```
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500,
The `heapq` module provides functions for implementing heaps based on regular lists. The lowest valued entry is always kept at position zero. This is useful for applications which repeatedly access the smallest element but do not want to run a full list sort:

```python
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)  # rearrange the list into
>>> heappush(data, -5)  # add a new entry
>>> [heappop(data) for i in range(3)]  # fetch the three smallest
[-5, 0, 1]
```
11.8 Decimal Floating Point Arithmetic

The `decimal` module offers a `Decimal` datatype for decimal floating point arithmetic. Compared to the built-in `float` implementation of binary floating point, the new class is especially helpful for financial applications and other uses which require exact decimal representation, control over precision, control over rounding to meet legal or regulatory requirements, tracking of significant decimal places, or for applications where the user expects the results to match calculations done by hand.

For example, calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point. The difference becomes significant if the results are rounded to the nearest cent:

```python
>>> from decimal import *
>>> Decimal('0.70') * Decimal('1.05')
Decimal("0.7350")
>>> .70 * 1.05
0.73499999999999999
```

The `Decimal` result keeps a trailing zero, automatically inferring four place significance from multiplicands with two place significance. Decimal reproduces mathematics as done by hand and avoids issues that can arise when binary floating point cannot exactly represent decimal quantities.

Exact representation enables the `Decimal` class to perform modulo calculations and equality tests that are unsuitable for binary floating point:

```python
>>> Decimal('1.00') % Decimal('.10')
Decimal("0.00")
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')] * 10) == Decimal('1.0')
True
>>> sum([0.1] * 10) == 1.0
False
```

The `decimal` module provides arithmetic with as much precision as needed:
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal("0.142857142857142857142857142857142857")
12. What Now?

Reading this tutorial has probably reinforced your interest in using Python -- you should be eager to apply Python to solving your real-world problems. Where should you go to learn more?

This tutorial is part of Python's documentation set. Some other documents in the set are:

- **Python Library Reference**: You should browse through this manual, which gives complete (though terse) reference material about types, functions, and the modules in the standard library. The standard Python distribution includes a lot of additional code. There are modules to read UNIX mailboxes, retrieve documents via HTTP, generate random numbers, parse command-line options, write CGI programs, compress data, and many other tasks. Skimming through the Library Reference will give you an idea of what's available.

- **Installing Python Modules** explains how to install external modules written by other Python users.

- **Language Reference**: A detailed explanation of Python's syntax and semantics. It's heavy reading, but is useful as a complete guide to the language itself.

More Python resources:

- **http://www.python.org**: The major Python Web site. It contains code, documentation, and pointers to Python-related pages around the Web. This Web site is mirrored in various places around the world, such as Europe, Japan, and Australia; a mirror may be faster than the main site, depending on your geographical location.

- **http://docs.python.org**: Fast access to Python's documentation.

- **http://cheeseshop.python.org**: The Python Package Index, nicknamed
the Cheese Shop, is an index of user-created Python modules that are available for download. Once you begin releasing code, you can register it here so that others can find it.

- **http://aspn.activestate.com/ASPN/Python/Cookbook**: The Python Cookbook is a sizable collection of code examples, larger modules, and useful scripts. Particularly notable contributions are collected in a book also titled *Python Cookbook* (O'Reilly & Associates, ISBN 0-596-00797-3.)

For Python-related questions and problem reports, you can post to the newsgroup comp.lang.python, or send them to the mailing list at python-list@python.org. The newsgroup and mailing list are gatewayed, so messages posted to one will automatically be forwarded to the other. There are around 120 postings a day (with peaks up to several hundred), asking (and answering) questions, suggesting new features, and announcing new modules. Before posting, be sure to check the list of Frequently Asked Questions (also called the FAQ), or look for it in the Misc/ directory of the Python source distribution. Mailing list archives are available at http://mail.python.org/pipermail/. The FAQ answers many of the questions that come up again and again, and may already contain the solution for your problem.
Subsections

- A.1 Line Editing
- A.2 History Substitution
- A.3 Key Bindings
- A.4 Commentary
A. Interactive Input Editing and History Substitution

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the GNU Readline library, which supports Emacs-style and vi-style editing. This library has its own documentation which I won't duplicate here; however, the basics are easily explained. The interactive editing and history described here are optionally available in the UNIX and Cygwin versions of the interpreter.

This chapter does not document the editing facilities of Mark Hammond's PythonWin package or the Tk-based environment, IDLE, distributed with Python. The command line history recall which operates within DOS boxes on NT and some other DOS and Windows flavors is yet another beast.
A.1 Line Editing

If supported, input line editing is active whenever the interpreter prints a primary or secondary prompt. The current line can be edited using the conventional Emacs control characters. The most important of these are: C-A (Control-A) moves the cursor to the beginning of the line, C-E to the end, C-B moves it one position to the left, C-F to the right. Backspace erases the character to the left of the cursor, C-D the character to its right. C-K kills (erases) the rest of the line to the right of the cursor, C-Y yanks back the last killed string. C-underscore undoes the last change you made; it can be repeated for cumulative effect.
A.2 History Substitution

History substitution works as follows. All non-empty input lines issued are saved in a history buffer, and when a new prompt is given you are positioned on a new line at the bottom of this buffer. C-P moves one line up (back) in the history buffer, C-N moves one down. Any line in the history buffer can be edited; an asterisk appears in front of the prompt to mark a line as modified. Pressing the Return key passes the current line to the interpreter. C-R starts an incremental reverse search; C-S starts a forward search.
A.3 Key Bindings

The key bindings and some other parameters of the Readline library can be customized by placing commands in an initialization file called ~/.inputrc. Key bindings have the form

```
key-name: function-name
```

or

```
"string": function-name
```

and options can be set with

```
set option-name value
```

For example:

```
# I prefer vi-style editing:
set editing-mode vi

# Edit using a single line:
set horizontal-scroll-mode On

# Rebind some keys:
Meta-h: backward-kill-word
"C-u": universal-argument
"C-x\C-r": re-read-init-file
```

Note that the default binding for Tab in Python is to insert a Tab character instead of Readline's default filename completion function. If you insist, you can override this by putting

```
Tab: complete
```

in your ~/.inputrc. (Of course, this makes it harder to type indented continuation lines if you're accustomed to using Tab for that purpose.)

Automatic completion of variable and module names is optionally available. To enable it in the interpreter's interactive mode, add the following to your startup file:A.1
This binds the Tab key to the completion function, so hitting the Tab key twice suggests completions; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `"." and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `__getattr__()` method is part of the expression.

A more capable startup file might look like this example. Note that this deletes the names it creates once they are no longer needed; this is done since the startup file is executed in the same namespace as the interactive commands, and removing the names avoids creating side effects in the interactive environment. You may find it convenient to keep some of the imported modules, such as `os`, which turn out to be needed in most sessions with the interpreter.

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')

# Add auto-completion and a stored history file of commands to your
# interactive interpreter. Requires Python 2.0+, readline. Autocomp
# bound to the Esc key by default (you can change it - see readline
#
# Store the file in ~/.pystartup, and set an environment variable to
# to it: "export PYTHONSTARTUP=/max/home/itamar/.pystartup" in bash
#
# Note that PYTHONSTARTUP does *not* expand "~", so you have to put
# full path to your home directory.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)

del os, atexit, readline, rlcompleter, save_history, historyPath
```
A.4 Commentary

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an indent token is required next). The completion mechanism might use the interpreter's symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

Footnotes

... file:A.1
Python will execute the contents of a file identified by the PYTHONSTARTUP environment variable when you start an interactive interpreter.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
Subsections

- **B.1 Representation Error**
B. Floating Point Arithmetic: Issues and Limitations

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the decimal fraction

0.125

has value $1/10 + 2/100 + 5/1000$, and in the same way the binary fraction

0.001

has value $0/2 + 0/4 + 1/8$. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at first in base 10. Consider the fraction $1/3$. You can approximate that as a base 10 fraction:

0.3

or, better,

0.33

or, better,

0.333

and so on. No matter how many digits you're willing to write down, the result will never be exactly $1/3$, but will be an increasingly better approximation of $1/3$.

In the same way, no matter how many base 2 digits you're willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2,
1/10 is the infinitely repeating fraction

\[0.0001100110011001100110011001100110011001100110011\ldots\]

Stop at any finite number of bits, and you get an approximation. This is why you see things like:

```python
>>> 0.1
0.10000000000000001
```

On most machines today, that is what you'll see if you enter 0.1 at a Python prompt. You may not, though, because the number of bits used by the hardware to store floating-point values can vary across machines, and Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display

```python
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

instead! The Python prompt uses the builtin `repr()` function to obtain a string version of everything it displays. For floats, `repr(float)` rounds the true decimal value to 17 significant digits, giving

```
0.10000000000000001
```

`repr(float)` produces 17 significant digits because it turns out that's enough (on most machines) so that `eval(repr(x)) == x` exactly for all finite floats `x`, but rounding to 16 digits is not enough to make that true.

Note that this is in the very nature of binary floating-point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not `display` the difference by default, or in all output modes).

Python's builtin `str()` function produces only 12 significant digits, and you may wish to use that instead. It's unusual for `eval(str(x))` to reproduce `x`, but the output may be more pleasant to look at:

```python
>>> print str(0.1)
```
0.1

It's important to realize that this is, in a real sense, an illusion: the value in the machine is not exactly 1/10, you're simply rounding the display of the true machine value.

Other surprises follow from this one. For example, after seeing

```python
>>> 0.1
0.10000000000000001
```

you may be tempted to use the `round()` function to chop it back to the single digit you expect. But that makes no difference:

```python
>>> round(0.1, 1)
0.10000000000000001
```

The problem is that the binary floating-point value stored for "0.1" was already the best possible binary approximation to 1/10, so trying to round it again can't make it better: it was already as good as it gets.

Another consequence is that since 0.1 is not exactly 1/10, summing ten values of 0.1 may not yield exactly 1.0, either:

```python
>>> sum = 0.0
>>> for i in range(10):
...    sum += 0.1
...
>>> sum
0.9999999999999989
```

Binary floating-point arithmetic holds many surprises like this. The problem with "0.1" is explained in precise detail below, in the "Representation Error" section. See *The Perils of Floating Point* for a more complete account of other common surprises.

As that says near the end, ``there are no easy answers." Still, don't be unduly wary of floating-point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in \(2^{**53}\) per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic, and that every float operation can suffer a new rounding error.
While pathological cases do exist, for most casual use of floating-point arithmetic you'll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. `str()` usually suffices, and for finer control see the discussion of Python's `%` format operator: the `%g`, `%f` and `%e` format codes supply flexible and easy ways to round float results for display.
B.1 Representation Error

This section explains the ``0.1'' example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

*Representation error* refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won't display the exact decimal number you expect:

```
>>> 0.1
0.10000000000000001
```

Why is that? 1/10 is not exactly representable as a binary fraction. Almost all machines today (November 2000) use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 "double precision". 754 doubles contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form \( J/2^N \) where \( J \) is an integer containing exactly 53 bits. Rewriting

\[
1 / 10 \sim J / (2^N)
\]
as

\[
J \sim 2^N / 10
\]

and recalling that \( J \) has exactly 53 bits (is \( \geq 2^{52} \) but \( < 2^{53} \)), the best value for \( N \) is 56:

```
>>> 2**52
4503599627370496L
>>> 2**53
9007199254740992L
>>> 2**56/10
7205759403792793L
```

That is, 56 is the only value for \( N \) that leaves \( J \) with exactly 53 bits. The best possible value for \( J \) is then that quotient rounded:

```
>>> q, r = divmod(2**56, 10)
```
Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

```python
>>> q + 1
7205759403792794L
```

Therefore the best possible approximation to 1/10 in 754 double precision is that over 2**56, or

```latex
\frac{7205759403792794}{2^{56}}
```

Note that since we rounded up, this is actually a little bit larger than 1/10; if we had not rounded up, the quotient would have been a little bit smaller than 1/10. But in no case can it be exactly 1/10!

So the computer never ``sees'' 1/10: what it sees is the exact fraction given above, the best 754 double approximation it can get:

```python
>>> .1 * 2**56
7205759403792794.0
```

If we multiply that fraction by 10**30, we can see the (truncated) value of its 30 most significant decimal digits:

```python
>>> 7205759403792794 * 10**30 / 2**56
100000000000000005551115123125L
```

meaning that the exact number stored in the computer is approximately equal to the decimal value 0.10000000000000000005551115123125. Rounding that to 17 significant digits gives the 0.10000000000000000000001 that Python displays (well, will display on any 754-conforming platform that does best-possible input and output conversions in its C library -- yours may not!).
Subsections

- C.1 History of the software
- C.2 Terms and conditions for accessing or otherwise using Python
- C.3 Licenses and Acknowledgements for Incorporated Software
  - C.3.1 Mersenne Twister
  - C.3.2 Sockets
  - C.3.3 Floating point exception control
  - C.3.4 MD5 message digest algorithm
  - C.3.5 Asynchronous socket services
  - C.3.6 Cookie management
  - C.3.7 Profiling
  - C.3.8 Execution tracing
  - C.3.9 UUencode and UUdecode functions
  - C.3.10 XML Remote Procedure Calls
C. History and License

C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

<table>
<thead>
<tr>
<th>Release</th>
<th>Derived from</th>
<th>Year</th>
<th>Owner</th>
<th>GPL compatible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.0 thru 1.2</td>
<td>n/a</td>
<td>1991-1995</td>
<td>CWI</td>
<td>yes</td>
</tr>
<tr>
<td>1.3 thru 1.5.2</td>
<td>1.2</td>
<td>1995-1999</td>
<td>CNRI</td>
<td>yes</td>
</tr>
<tr>
<td>1.6</td>
<td>1.5.2</td>
<td>2000</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.0</td>
<td>1.6</td>
<td>2000</td>
<td>BeOpen.com</td>
<td>no</td>
</tr>
<tr>
<td>1.6.1</td>
<td>1.6</td>
<td>2001</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>no</td>
</tr>
<tr>
<td>Version</td>
<td>Previous Version</td>
<td>Year</td>
<td>License</td>
<td>GPL Compatible</td>
</tr>
<tr>
<td>---------</td>
<td>----------------</td>
<td>--------</td>
<td>---------</td>
<td>----------------</td>
</tr>
<tr>
<td>2.0.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.1</td>
<td>2.1+2.0.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2</td>
<td>2.1.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.2</td>
<td>2.1.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.3</td>
<td>2.1.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.1</td>
<td>2.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.2</td>
<td>2.2.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.3</td>
<td>2.2.2</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3</td>
<td>2.2.2</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.1</td>
<td>2.3</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.2</td>
<td>2.3.1</td>
<td>2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.3</td>
<td>2.3.2</td>
<td>2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.4</td>
<td>2.3.3</td>
<td>2004</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.5</td>
<td>2.3.4</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4</td>
<td>2.3</td>
<td>2004</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.1</td>
<td>2.4</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.2</td>
<td>2.4.1</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.3</td>
<td>2.4.2</td>
<td>2006</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.5</td>
<td>2.4</td>
<td>2006</td>
<td>PSF</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Note:** GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.
C.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.5

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.5 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.5 alone or in any derivative version, provided, however, that PSF’s License Agreement and PSF’s notice of copyright, i.e., "Copyright © 2001-2006 Python Software Foundation; All Rights Reserved" are retained in Python 2.5 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.5 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.5.

4. PSF is making Python 2.5 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.5 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS
OF PYTHON 2.5 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.5, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.5, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com (``BeOpen''), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (``Licensee'') accessing and otherwise using this software in source or binary form and its associated documentation (``the Software '').

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an ``AS IS'' basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT
LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the ``BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (``CNRI''), and the Individual or Organization (``Licensee'') accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to
reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., ``Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): ``Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual
property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR
ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.
C.3.1 Mersenne Twister

The _random module includes code based on a download from http://www.math.keio.ac.jp/~matumoto/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26. Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed) or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura, All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.
http://www.math.keio.ac.jp/matumoto/emt.html
email: matumoto@math.keio.ac.jp
C.3.2 Sockets

The socket module uses the functions, getaddrinfo, and getnameinfo, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/about/index.html.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
C.3.3 Floating point exception control

The source for the fpectl1 module includes the following notice:

```
/                                  Copyright (c) 1996.
|                                  The Regents of the University of California.
|                                  All rights reserved.
|
| Permission to use, copy, modify, and distribute this software
| any purpose without fee is hereby granted, provided that this
| tire notice is included in all copies of any software which i
| includes a copy or modification of this software and in
| copies of the supporting documentation for such software.
|
| This work was produced at the University of California, Lawr
| Livermore National Laboratory under contract no. W-7405-EN
| between the U.S. Department of Energy and The Regents of
| University of California for the operation of UC LLNL.
|
| DISCLAIMER
| This software was prepared as an account of work sponsored b
| agency of the United States Government. Neither the United St
| Government nor the University of California nor any of their
| ployees, makes any warranty, express or implied, or assumes
| liability or responsibility for the accuracy, completeness
| usefulness of any information, apparatus, product, or pro
| disclosed, or represents that its use would not infr
| privately-owned rights. Reference herein to any specific com-
| cial products, process, or service by trade name, tradem
| manufacturer, or otherwise, does not necessarily constitute
| imply its endorsement, recommendation, or favoring by the Un
| States Government or the University of California. The views
| opinions of authors expressed herein do not necessarily stat
| reflect those of the United States Government or the Univer
| of California, and shall not be used for advertising or pro
| endorsement purposes.
```

C.3.4 MD5 message digest algorithm

The source code for the md5 module contains the following notice:

Copyright (C) 1999, 2002 Aladdin Enterprises. All rights reserved

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damage arising from the use of this software.

Permission is granted to anyone to use this software for any purpose including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose text is available at
http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite (section A.5) but excluding the rest of Appendix A. It does not include any code or documentation that is identified in the RFC as being copyrighted.

The original and principal author of md5.h is L. Peter Deutsch <ghost@aladdin.com>. Other authors are noted in the change history that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed references to Ghostscript; clarified derivation from RFC 132
now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extracti
1999-10-18 lpd Fixed typo in header comment (ansis2knr rather than added conditionalization for C++ compilation from Martin Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
C.3.5 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
C.3.6 Cookie management

The Cookie module contains the following notice:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Timothy O'Malley not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
C.3.7 Profiling

The profile and pstats modules contain the following notice:

Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose (subject to the restriction in the following sentence) without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in all supporting documentation, and that the name of InfoSeek not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. This permission is explicitly restricted to the copying and modification of the software to remain in Python, compiled Python, or other languages (such as C wherein the modified or derived code is exclusively imported into a Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
C.3.8 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rig
err... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights res
```

Permission to use, copy, modify, and distribute this Python softwar
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all co
and that both that copyright notice and this permission notice appe
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertai
distribution of the software without specific, written prior permis
C.3.9 UUencode and UUdecode functions

The uu module contains the following notice:

Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. Th
  version is still 5 times faster, though.
- Arguments more compliant with python standard
C.3.10 XML Remote Procedure Calls

The xmlrpclib module contains the following notice:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicit pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
D. Glossary

>>> The typical Python prompt of the interactive shell. Often seen for code examples that can be tried right away in the interpreter.

... The typical Python prompt of the interactive shell when entering code for an indented code block.

BDFL
Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.

byte code
The internal representation of a Python program in the interpreter. The byte code is also cached in .pyc and .pyo files so that executing the same file is faster the second time (recompilation from source to byte code can be avoided). This ``intermediate language'' is said to run on a ``virtual machine'' that calls the subroutines corresponding to each bytecode.

classic class
Any class which does not inherit from object. See new-style class.

coefficient
The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, int(3.15) converts the floating point number to the integer 3, but in 3+4.5, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a TypeError. Coercion between two operands can be performed with the coerce builtin function; thus, 3+4.5 is equivalent to calling operator.add(*coerce(3, 4.5)) and results in operator.add(3.0, 4.5). Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., float(3)+4.5 rather than just 3+4.5.

complex number
An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of \(-1\)), often written \(i\) in mathematics or \(j\) in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a \(j\) suffix, e.g., \(3+1j\). To get access to complex equivalents of the \texttt{math} module, use \texttt{cmath}. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

**descriptor**

Any *new-style* object that defines the methods \texttt{__get__()}, \texttt{__set__()}, or \texttt{__delete__()}. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, writing \(a.b\) looks up the object \(b\) in the class dictionary for \(a\), but if \(b\) is a descriptor, the defined method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

**dictionary**

An associative array, where arbitrary keys are mapped to values. The use of \texttt{dict} much resembles that for \texttt{list}, but the keys can be any object with a \texttt{__hash__()} function, not just integers starting from zero. Called a hash in Perl.

**duck-typing**

Pythonic programming style that determines an object's type by inspection of its method or attribute signature rather than by explicit relationship to some type object ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using \texttt{type()} or \texttt{isinstance()}. Instead, it typically employs \texttt{hasattr()} tests or \texttt{EAFP} programming.

**EAFP**

Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is
characterized by the presence of many try and except statements. The technique contrasts with the LBYL style that is common in many other languages such as C.

__future__
A pseudo module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression 11/4 currently evaluates to 2. If the module in which it is executed had enabled true division by executing:

```python
from __future__ import division
```
the expression 11/4 would evaluate to 2.75. By importing the __future__ module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

```python
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

generator
A function that returns an iterator. It looks like a normal function except that values are returned to the caller using a yield statement instead of a return statement. Generator functions often contain one or more for or while loops that yield elements back to the caller. The function execution is stopped at the yield keyword (returning the result) and is resumed there when the next element is requested by calling the next() method of the returned iterator.

generator expression
An expression that returns a generator. It looks like a normal expression followed by a for expression defining a loop variable, range, and an optional if expression. The combined expression generates values for an enclosing function:

```python
>>> sum(i*i for i in range(10))   # sum of squares 0, 1, 4, 9, ...
285
```

GIL
See global interpreter lock.
**global interpreter lock**

The lock used by Python threads to assure that only one thread can be run at a time. This simplifies Python by assuring that no two processes can access the same memory at the same time. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of some parallelism on multi-processor machines. Efforts have been made in the past to create a "free-threaded" interpreter (one which locks shared data at a much finer granularity), but performance suffered in the common single-processor case.

**IDLE**

An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment that ships with the standard distribution of Python. Good for beginners, it also serves as clear example code for those wanting to implement a moderately sophisticated, multi-platform GUI application.

**immutable**

An object with fixed value. Immutable objects are numbers, strings or tuples (and more). Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

**integer division**

Mathematical division discarding any remainder. For example, the expression 11/4 currently evaluates to 2 in contrast to the 2.75 returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a float), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the // operator instead of the / operator. See also *__future__*.

**interactive**

Python has an interactive interpreter which means that you can try out things and immediately see their results. Just launch python with no arguments (possibly by selecting it from your computer's main menu). It is
a very powerful way to test out new ideas or inspect modules and packages (remember help(x)).

interpreted
Python is an interpreted language, as opposed to a compiled one. This means that the source files can be run directly without first creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also interactive.

iterable
A container object capable of returning its members one at a time. Examples of iterables include all sequence types (such as list, str, and tuple) and some non-sequence types like dict and file and objects of any classes you define with an __iter__() or __getitem__() method. Iterables can be used in a for loop and in many other places where a sequence is needed (zip(), map(), ...). When an iterable object is passed as an argument to the builtin function iter(), it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call iter() or deal with iterator objects yourself. The for statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also iterator, sequence, and generator.

iterator
An object representing a stream of data. Repeated calls to the iterator's next() method return successive items in the stream. When no more data is available a StopIteration exception is raised instead. At this point, the iterator object is exhausted and any further calls to its next() method just raise StopIteration again. Iterators are required to have an __iter__() method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code that attempts multiple iteration passes. A container object (such as a list) produces a fresh new iterator each time you pass it to the iter() function or use it in a for loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty
container.

**LBYL**
Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the *EAFP* approach and is characterized by the presence of many *if* statements.

**list comprehension**
A compact way to process all or a subset of elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing hex numbers (0x..) that are even and in the range from 0 to 255. The *if* clause is optional. If omitted, all elements in `range(256)` are processed.

**mapping**
A container object (such as `dict`) that supports arbitrary key lookups using the special method `__getitem__()`. 

**metaclass**
The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

**mutable**
Mutable objects can change their value but keep their `id()` . See also *immutable*.

**namespace**
The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and builtin namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their
namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules respectively.

**nested scope**
The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

**new-style class**
Any class that inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()` (as well as `__getattr__()` and `__setattr__()`), class methods, and static methods.

**Python3000**
A mythical python release, not required to be backward compatible, with telepathic interface.

**__slots__**
A declaration inside a new-style class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

**sequence**
An iterable which supports efficient element access using integer indices via the `__getitem__()` and `__len__()` special methods. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary immutable keys rather than integers.
Zen of Python
Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing `import this` at the interactive prompt.
Index

Symbols | a | b | c | d | e | f | g | h | i | l | m | n | o | p | r | s | u | z
Symbols
_(underscore)_

all
__builtin__ (built-in module)
__future__
__slots__
A

append() (list method)
BDFL  byte code
classic class coercion compileall (standard module) complex number count() (list method)
D

descriptor
dictionary
docstrings, [Link]
documentation strings, [Link]
duck-typing
**EAEP**

environment variables

PATH, [Link]
PYTHONPATH, [Link], [Link], [Link], [Link]

environment variables (continued)

PYTHONSTARTUP, [Link]
extend() (list method)
file object for statement, [Link]
<table>
<thead>
<tr>
<th>generator</th>
<th>GIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>generator expression</td>
<td>global interpreter lock</td>
</tr>
</tbody>
</table>
help() (built-in function)
IDLE
immutable
index() (list method)
insert() (list method)
integer division

interactive
interpreted
iterable
iterator
<table>
<thead>
<tr>
<th>mapping</th>
<th>module</th>
</tr>
</thead>
<tbody>
<tr>
<td>metaclass</td>
<td>search path</td>
</tr>
<tr>
<td>method object</td>
<td>mutable</td>
</tr>
</tbody>
</table>
N

namespace
nested scope

new-style class
object

    file
    method

open() (built-in function)
path

module search
PATH (environment variable), [Link]
pickle (standard module)
pop() (list method)

Python3000
PYTHONPATH (environment variable), [Link], [Link], [Link], [Link], [Link]
PYTHONSTARTUP (environment variable), [Link]
<table>
<thead>
<tr>
<th>readline (built-in module)</th>
<th>reverse() (list method)</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove() (list method)</td>
<td>rlcompleter (standard module)</td>
</tr>
</tbody>
</table>
search

- path, module
- sequence
- sort() (list method)

statement
- for, [Link]

string (standard module)
strings, documentation, [Link]
sys (standard module)
**unicode() (built-in function)**
Z

Zen of Python

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
• Front Matter
• Contents
• 1. Introduction
• 2. Built-in Objects
  • 2.1 Built-in Functions
  • 2.2 Non-essential Built-in Functions
  • 2.3 Built-in Exceptions
  • 2.4 Built-in Constants
• 3. Built-in Types
  • 3.1 Truth Value Testing
  • 3.2 Boolean Operations -- and, or, not
  • 3.3 Comparisons
  • 3.4 Numeric Types -- int, float, long, complex
    • 3.4.1 Bit-string Operations on Integer Types
  • 3.5 Iterator Types
  • 3.6 Sequence Types -- str, unicode, list, tuple, buffer, xrange
    • 3.6.1 String Methods
    • 3.6.2 String Formatting Operations
    • 3.6.3 XRange Type
    • 3.6.4 Mutable Sequence Types
  • 3.7 Set Types -- set, frozenset
  • 3.8 Mapping Types -- dict
- 3.9 File Objects
- 3.10 Context Manager Types
- 3.11 Other Built-in Types
  - 3.11.1 Modules
  - 3.11.2 Classes and Class Instances
  - 3.11.3 Functions
  - 3.11.4 Methods
  - 3.11.5 Code Objects
  - 3.11.6 Type Objects
  - 3.11.7 The Null Object
  - 3.11.8 The Ellipsis Object
  - 3.11.9 Boolean Values
  - 3.11.10 Internal Objects
- 3.12 Special Attributes

- 4. String Services
  - 4.1 string -- Common string operations
    - 4.1.1 String constants
    - 4.1.2 Template strings
    - 4.1.3 String functions
    - 4.1.4 Deprecated string functions
  - 4.2 re -- Regular expression operations
    - 4.2.1 Regular Expression Syntax
    - 4.2.2 Matching vs Searching
    - 4.2.3 Module Contents
    - 4.2.4 Regular Expression Objects
    - 4.2.5 Match Objects
    - 4.2.6 Examples
  - 4.3 struct -- Interpret strings as packed binary data
  - 4.4 difflib -- Helpers for computing deltas
    - 4.4.1 SequenceMatcher Objects
    - 4.4.2 SequenceMatcher Examples
    - 4.4.3 Differ Objects
    - 4.4.4 Differ Example
  - 4.5 StringIO -- Read and write strings as files
  - 4.6 cStringIO -- Faster version of StringIO
  - 4.7 textwrap -- Text wrapping and filling
  - 4.8 codecs -- Codec registry and base classes
    - 4.8.1 Codec Base Classes
- 4.8.2 Encodings and Unicode
- 4.8.3 Standard Encodings
- 4.8.4 encodings.idna -- Internationalized Domain Names in Applications
- 4.8.5 encodings.utf_8_sig -- UTF-8 codec with BOM signature

- 4.9 unicodedata -- Unicode Database
- 4.10 stringprep -- Internet String Preparation
- 4.11 fpformat -- Floating point conversions

- 5. Data Types
  - 5.1 datetime -- Basic date and time types
    - 5.1.1 Available Types
    - 5.1.2 timedelta Objects
    - 5.1.3 date Objects
    - 5.1.4 datetime Objects
    - 5.1.5 time Objects
    - 5.1.6 tzinfo Objects
    - 5.1.7 strftime() Behavior
    - 5.1.8 Examples
  - 5.2 calendar -- General calendar-related functions
  - 5.3 collections -- High-performance container datatypes
    - 5.3.1 deque objects
    - 5.3.2 defaultdict objects
  - 5.4 heapq -- Heap queue algorithm
    - 5.4.1 Theory
  - 5.5 bisect -- Array bisection algorithm
    - 5.5.1 Examples
  - 5.6 array -- Efficient arrays of numeric values
  - 5.7 sets -- Unordered collections of unique elements
    - 5.7.1 Set Objects
    - 5.7.2 Example
    - 5.7.3 Protocol for automatic conversion to immutable
    - 5.7.4 Comparison to the built-in set types
  - 5.8 sched -- Event scheduler
    - 5.8.1 Scheduler Objects
  - 5.9 mutex -- Mutual exclusion support
    - 5.9.1 Mutex Objects
  - 5.10 Queue -- A synchronized queue class
5.10.1 Queue Objects

5.11 weakref -- Weak references
  5.11.1 Weak Reference Objects
  5.11.2 Example

5.12 UserDict -- Class wrapper for dictionary objects
5.13 UserList -- Class wrapper for list objects
5.14 UserString -- Class wrapper for string objects
5.15 types -- Names for built-in types
5.16 new -- Creation of runtime internal objects
5.17 copy -- Shallow and deep copy operations
5.18 pprint -- Data pretty printer
  5.18.1 PrettyPrinter Objects

5.19 repr -- Alternate repr() implementation
  5.19.1 Repr Objects
  5.19.2 Subclassing Repr Objects

6. Numeric and Mathematical Modules

6.1 math -- Mathematical functions
6.2 cmath -- Mathematical functions for complex numbers
6.3 decimal -- Decimal floating point arithmetic
  6.3.1 Quick-start Tutorial
  6.3.2 Decimal objects
  6.3.3 Context objects
  6.3.4 Signals
  6.3.5 Floating Point Notes
  6.3.6 Working with threads
  6.3.7 Recipes
  6.3.8 Decimal FAQ

6.4 random -- Generate pseudo-random numbers
6.5 itertools -- Functions creating iterators for efficient looping
  6.5.1 Itertool functions
  6.5.2 Examples
  6.5.3 Recipes

6.6 functools -- Higher order functions and operations on callable objects
  6.6.1 partial Objects

6.7 operator -- Standard operators as functions.
  6.7.1 Mapping Operators to Functions

7. Internet Data Handling
7.1 email -- An email and MIME handling package
   - 7.1.1 Representing an email message
   - 7.1.2 Parsing email messages
   - 7.1.3 Generating MIME documents
   - 7.1.4 Creating email and MIME objects from scratch
   - 7.1.5 Internationalized headers
   - 7.1.6 Representing character sets
   - 7.1.7 Encoders
   - 7.1.8 Exception and Defect classes
   - 7.1.9 Miscellaneous utilities
   - 7.1.10 Iterators
   - 7.1.11 Package History
   - 7.1.12 Differences from mimelib
   - 7.1.13 Examples

7.2 mailcap -- Mailcap file handling.

7.3 mailbox -- Manipulate mailboxes in various formats
   - 7.3.1 Mailbox objects
   - 7.3.2 Message objects
   - 7.3.3 Exceptions
   - 7.3.4 Deprecated classes and methods
   - 7.3.5 Examples

7.4 mhlib -- Access to MH mailboxes
   - 7.4.1 MH Objects
   - 7.4.2 Folder Objects
   - 7.4.3 Message Objects

7.5 mimetools -- Tools for parsing MIME messages
   - 7.5.1 Additional Methods of Message Objects

7.6 mimetypes -- Map filenames to MIME types
   - 7.6.1 MimeTypes Objects

7.7 MimeWriter -- Generic MIME file writer
   - 7.7.1 MimeWriter Objects

7.8 mimify -- MIME processing of mail messages

7.9 multifile -- Support for files containing distinct parts
   - 7.9.1 MultiFile Objects
   - 7.9.2 MultiFile Example

7.10 rfc822 -- Parse RFC 2822 mail headers
   - 7.10.1 Message Objects
   - 7.10.2 AddressList Objects
7.11 base64 -- RFC 3548: Base16, Base32, Base64 Data Encodings
7.12 binhex -- Encode and decode binhex4 files
   7.12.1 Notes
7.13 binascii -- Convert between binary and ASCII
7.14 quopri -- Encode and decode MIME quoted-printable data
7.15 uu -- Encode and decode uuencode files

8. Structured Markup Processing Tools
   8.1 HTMLParser -- Simple HTML and XHTML parser
      8.1.1 Example HTML Parser Application
   8.2 sgmllib -- Simple SGML parser
   8.3 htmllib -- A parser for HTML documents
      8.3.1 HTMLParser Objects
   8.4 htmlentitydefs -- Definitions of HTML general entities
   8.5 xml.parsers.expat -- Fast XML parsing using Expat
      8.5.1 XMLParser Objects
      8.5.2 ExpatError Exceptions
      8.5.3 Example
      8.5.4 Content Model Descriptions
      8.5.5 Expat error constants
   8.6 xml.dom -- The Document Object Model API
      8.6.1 Module Contents
      8.6.2 Objects in the DOM
      8.6.3 Conformance
   8.7 xml.dom.minidom -- Lightweight DOM implementation
      8.7.1 DOM Objects
      8.7.2 DOM Example
      8.7.3 minidom and the DOM standard
   8.8 xml.dom.pulldom -- Support for building partial DOM trees
      8.8.1 DOMEventStream Objects
   8.9 xml.sax -- Support for SAX2 parsers
      8.9.1 SAXException Objects
   8.10 xml.sax.handler -- Base classes for SAX handlers
      8.10.1 ContentHandler Objects
      8.10.2 DTDHandler Objects
      8.10.3 EntityResolver Objects
      8.10.4 ErrorHandler Objects
   8.11 xml.sax.saxutils -- SAX Utilities
   8.12 xml.sax.xmlreader -- Interface for XML parsers
8.12.1 XMLReader Objects
8.12.2 IncrementalParser Objects
8.12.3 Locator Objects
8.12.4 InputSource Objects
8.12.5 The Attributes Interface
8.12.6 The AttributesNS Interface

8.13 xml.etree.ElementTree -- The ElementTree XML API
8.13.1 Functions
8.13.2 ElementTree Objects
8.13.3 QName Objects
8.13.4 TreeBuilder Objects
8.13.5 XMLTreeBuilder Objects

9. File Formats
9.1 csv -- CSV File Reading and Writing
9.1.1 Module Contents
9.1.2 Dialects and Formatting Parameters
9.1.3 Reader Objects
9.1.4 Writer Objects
9.1.5 Examples
9.2 ConfigParser -- Configuration file parser
9.2.1 RawConfigParser Objects
9.2.2 ConfigParser Objects
9.2.3 SafeConfigParser Objects
9.3 robotparser -- Parser for robots.txt
9.4 netrc -- netrc file processing
9.4.1 netrc Objects
9.5 xdrlib -- Encode and decode XDR data
9.5.1 Packer Objects
9.5.2 Unpacker Objects
9.5.3 Exceptions

10. Cryptographic Services
10.1 hashlib -- Secure hashes and message digests
10.2 hmac -- Keyed-Hashing for Message Authentication
10.3 md5 -- MD5 message digest algorithm
10.4 sha -- SHA-1 message digest algorithm

11. File and Directory Access
11.1 os.path -- Common pathname manipulations
11.2 fileinput -- Iterate over lines from multiple input streams
11.3 stat -- Interpreting stat() results
11.4 statvfs -- Constants used with os.statvfs()
11.5 filecmp -- File and Directory Comparisons
   11.5.1 The dircmp class
11.6 tempfile -- Generate temporary files and directories
11.7 glob -- UNIX style pathname pattern expansion
11.8 fnmatch -- UNIX filename pattern matching
11.9 linecache -- Random access to text lines
11.10 shutil -- High-level file operations
   11.10.1 Example
11.11 dircache -- Cached directory listings

12. Data Compression and Archiving
   12.1 zlib -- Compression compatible with gzip
   12.2 gzip -- Support for gzip files
   12.3 bz2 -- Compression compatible with bzip2
      12.3.1 (De)compression of files
      12.3.2 Sequential (de)compression
      12.3.3 One-shot (de)compression
   12.4 zipfile -- Work with ZIP archives
      12.4.1 ZipFile Objects
      12.4.2 PyZipFile Objects
      12.4.3 ZipInfo Objects
   12.5 tarfile -- Read and write tar archive files
      12.5.1 TarFile Objects
      12.5.2 TarInfo Objects
      12.5.3 Examples

13. Data Persistence
   13.1 pickle -- Python object serialization
      13.1.1 Relationship to other Python modules
      13.1.2 Data stream format
      13.1.3 Usage
      13.1.4 What can be pickled and unpickled?
      13.1.5 The pickle protocol
      13.1.6 Subclassing Unpicklers
      13.1.7 Example
   13.2 cPickle -- A faster pickle
   13.3 copy_reg -- Register pickle support functions
   13.4 shelve -- Python object persistence
- 13.4.1 Restrictions
- 13.4.2 Example
- 13.5 marshal -- Internal Python object serialization
- 13.6 anydbm -- Generic access to DBM-style databases
- 13.7 whichdb -- Guess which DBM module created a database
- 13.8 dbm -- Simple "database" interface
- 13.9 gdbm -- GNU's reinterpretation of dbm
- 13.10 dbhash -- DBM-style interface to the BSD database library
  - 13.10.1 Database Objects
- 13.11 bsddb -- Interface to Berkeley DB library
  - 13.11.1 Hash, BTREE and Record Objects
- 13.12 dumbdbm -- Portable DBM implementation
  - 13.12.1 Dumbdbm Objects
- 13.13 sqlite3 -- DB-API 2.0 interface for SQLite databases
  - 13.13.1 Module functions and constants
  - 13.13.2 Connection Objects
  - 13.13.3 Cursor Objects
  - 13.13.4 SQLite and Python types
  - 13.13.5 Controlling Transactions
  - 13.13.6 Using pysqlite efficiently
- 14. Generic Operating System Services
- 14.1 os -- Miscellaneous operating system interfaces
  - 14.1.1 Process Parameters
  - 14.1.2 File Object Creation
  - 14.1.3 File Descriptor Operations
  - 14.1.4 Files and Directories
  - 14.1.5 Process Management
  - 14.1.6 Miscellaneous System Information
  - 14.1.7 Miscellaneous Functions
- 14.2 time -- Time access and conversions
- 14.3 optparse -- More powerful command line option parser
  - 14.3.1 Background
  - 14.3.2 Tutorial
  - 14.3.3 Reference Guide
  - 14.3.4 Option Callbacks
  - 14.3.5 Extending optparse
- 14.4 getopt -- Parser for command line options
- 14.5 logging -- Logging facility for Python
14.5.1 Logger Objects
14.5.2 Basic example
14.5.3 Logging to multiple destinations
14.5.4 Sending and receiving logging events across a network
14.5.5 Handler Objects
14.5.6 Formatter Objects
14.5.7 Filter Objects
14.5.8 LogRecord Objects
14.5.9 Thread Safety
14.5.10 Configuration

14.6 getpass -- Portable password input

14.7 curses -- Terminal handling for character-cell displays
   14.7.1 Functions
   14.7.2 Window Objects
   14.7.3 Constants

14.8 curses.textpad -- Text input widget for curses programs
   14.8.1 Textbox objects

14.9 curses.wrapper -- Terminal handler for curses programs

14.10 curses.ascii -- Utilities for ASCII characters

14.11 curses.panel -- A panel stack extension for curses.
   14.11.1 Functions
   14.11.2 Panel Objects

14.12 platform -- Access to underlying platform's identifying data.
   14.12.1 Cross Platform
   14.12.2 Java Platform
   14.12.3 Windows Platform
   14.12.4 Mac OS Platform
   14.12.5 UNIX Platforms

14.13 errno -- Standard errno system symbols

14.14 ctypes -- A foreign function library for Python.
   14.14.1 ctypes tutorial
   14.14.2 ctypes reference

15. Optional Operating System Services
   15.1 select -- Waiting for I/O completion
      15.1.1 Polling Objects
   15.2 thread -- Multiple threads of control
   15.3 threading -- Higher-level threading interface
      15.3.1 Lock Objects
15.3.2 RLock Objects
15.3.3 Condition Objects
15.3.4 Semaphore Objects
15.3.5 Event Objects
15.3.6 Thread Objects
15.3.7 Timer Objects
15.3.8 Using locks, conditions, and semaphores in the with statement

15.4 dummy_thread -- Drop-in replacement for the thread module
15.5 dummy_threading -- Drop-in replacement for the threading module
15.6 mmap -- Memory-mapped file support
15.7 readline -- GNU readline interface
   15.7.1 Example
15.8 rlcompleter -- Completion function for GNU readline
   15.8.1 Completer Objects

16. Unix Specific Services
16.1 posix -- The most common POSIX system calls
   16.1.1 Large File Support
   16.1.2 Module Contents
16.2 pwd -- The password database
16.3 spwd -- The shadow password database
16.4 grp -- The group database
16.5 crypt -- Function to check UNIX passwords
16.6 dl -- Call C functions in shared objects
   16.6.1 DL Objects
16.7 termios -- POSIX style tty control
   16.7.1 Example
16.8 tty -- Terminal control functions
16.9 pty -- Pseudo-terminal utilities
16.10 fcntl -- The fcntl() and ioctl() system calls
16.11 pipes -- Interface to shell pipelines
   16.11.1 Template Objects
16.12posixfile -- File-like objects with locking support
16.13 resource -- Resource usage information
   16.13.1 Resource Limits
   16.13.2 Resource Usage
16.14 nis -- Interface to Sun's NIS (Yellow Pages)
16.15 syslog -- UNIX syslog library routines
16.16 commands -- Utilities for running commands

17. Interprocess Communication and Networking
17.1 subprocess -- Subprocess management
  17.1.1 Using the subprocess Module
  17.1.2 Popen Objects
  17.1.3 Replacing Older Functions with the subprocess Module
17.2 socket -- Low-level networking interface
  17.2.1 Socket Objects
  17.2.2 SSL Objects
  17.2.3 Example
17.3 signal -- Set handlers for asynchronous events
  17.3.1 Example
17.4 popen2 -- Subprocesses with accessible I/O streams
  17.4.1 Popen3 and Popen4 Objects
  17.4.2 Flow Control Issues
17.5 asyncore -- Asynchronous socket handler
  17.5.1 asyncore Example basic HTTP client
17.6 asynchat -- Asynchronous socket command/response handler
  17.6.1 asynchat - Auxiliary Classes and Functions
  17.6.2 asynchat Example

18. Internet Protocols and Support
18.1 webbrowser -- Convenient Web-browser controller
  18.1.1 Browser Controller Objects
18.2 cgi -- Common Gateway Interface support.
  18.2.1 Introduction
  18.2.2 Using the cgi module
  18.2.3 Higher Level Interface
  18.2.4 Old classes
  18.2.5 Functions
  18.2.6 Caring about security
  18.2.7 Installing your CGI script on a UNIX system
  18.2.8 Testing your CGI script
  18.2.9 Debugging CGI scripts
  18.2.10 Common problems and solutions
18.3 cgithb -- Traceback manager for CGI scripts
18.4 wsgiref -- WSGI Utilities and Reference Implementation
- 18.4.1 wsgiref.util - WSGI environment utilities
- 18.4.2 wsgiref.headers - WSGI response header tools
- 18.4.3 wsgiref.simple_server - a simple WSGI HTTP server
- 18.4.4 wsgiref.validate - WSGI conformance checker
- 18.4.5 wsgiref.handlers - server/gateway base classes

- 18.5 urllib -- Open arbitrary resources by URL
  - 18.5.1 URLopener Objects
  - 18.5.2 Examples

- 18.6 urllib2 -- extensible library for opening URLs
  - 18.6.1 Request Objects
  - 18.6.2 OpenerDirector Objects
  - 18.6.3 BaseHandler Objects
  - 18.6.4 HTTPRedirectHandler Objects
  - 18.6.5 HTTPCookieProcessor Objects
  - 18.6.6 ProxyHandler Objects
  - 18.6.7 HTTPPasswordMgr Objects
  - 18.6.8 AbstractBasicAuthHandler Objects
  - 18.6.9 HTTPBasicAuthHandler Objects
  - 18.6.10 ProxyBasicAuthHandler Objects
  - 18.6.11 AbstractDigestAuthHandler Objects
  - 18.6.12 HTTPDigestAuthHandler Objects
  - 18.6.13 ProxyDigestAuthHandler Objects
  - 18.6.14 HTTPHandler Objects
  - 18.6.15 HTTPSHandler Objects
  - 18.6.16 FileHandler Objects
  - 18.6.17 FTPHandler Objects
  - 18.6.18 CacheFTPHandler Objects
  - 18.6.19 GopherHandler Objects
  - 18.6.20 UnknownHandler Objects
  - 18.6.21 HTTPErrorProcessor Objects
  - 18.6.22 Examples

- 18.7 httpclient -- HTTP client protocol
  - 18.7.1 HTTPConnection Objects
  - 18.7.2 HTTPResponse Objects
  - 18.7.3 Examples

- 18.8 ftplib -- FTP protocol client
  - 18.8.1 FTP Objects
- **18.9 gopherlib** -- Gopher protocol client
  - **18.10 poplib** -- POP3 protocol client
    - **18.10.1** POP3 Objects
    - **18.10.2** POP3 Example
  - **18.11 imaplib** -- IMAP4 protocol client
    - **18.11.1** IMAP4 Objects
    - **18.11.2** IMAP4 Example
  - **18.12 nntplib** -- NNTP protocol client
    - **18.12.1** NNTP Objects
  - **18.13 smtplib** -- SMTP protocol client
    - **18.13.1** SMTP Objects
    - **18.13.2** SMTP Example
  - **18.14 smtpd** -- SMTP Server
    - **18.14.1** SMTPServer Objects
    - **18.14.2** DebuggingServer Objects
    - **18.14.3** PureProxy Objects
    - **18.14.4** MailmanProxy Objects
  - **18.15 telnetlib** -- Telnet client
    - **18.15.1** Telnet Objects
    - **18.15.2** Telnet Example
  - **18.16 uuid** -- UUID objects according to RFC 4122
    - **18.16.1** Example
  - **18.17 urlparse** -- Parse URLs into components
    - **18.17.1** Results of `urlparse()` and `urlsplit()`
  - **18.18 SocketServer** -- A framework for network servers
    - **18.18.1** Server Creation Notes
    - **18.18.2** Server Objects
    - **18.18.3** RequestHandler Objects
  - **18.19 BaseHTTPServer** -- Basic HTTP server
  - **18.20 SimpleHTTPServer** -- Simple HTTP request handler
  - **18.21 CGIHTTPServer** -- CGI-capable HTTP request handler
  - **18.22 cookielib** -- Cookie handling for HTTP clients
    - **18.22.1** CookieJar and FileCookieJar Objects
    - **18.22.2** FileCookieJar subclasses and co-operation with web browsers
    - **18.22.3** CookiePolicy Objects
    - **18.22.4** DefaultCookiePolicy Objects
    - **18.22.5** Cookie Objects
- 18.22.6 Examples
- 18.23 Cookie -- HTTP state management
  - 18.23.1 Cookie Objects
  - 18.23.2 Morsel Objects
  - 18.23.3 Example
- 18.24 xmlrpclib -- XML-RPC client access
  - 18.24.1 ServerProxy Objects
  - 18.24.2 Boolean Objects
  - 18.24.3 DateTime Objects
  - 18.24.4 Binary Objects
  - 18.24.5 Fault Objects
  - 18.24.6 ProtocolError Objects
  - 18.24.7 MultiCall Objects
  - 18.24.8 Convenience Functions
  - 18.24.9 Example of Client Usage
- 18.25 SimpleXMLRPCServer -- Basic XML-RPC server
  - 18.25.1 SimpleXMLRPCServer Objects
  - 18.25.2 CGIHTTPRequestHandler
- 18.26 DocXMLRPCServer -- Self-documenting XML-RPC server
  - 18.26.1 DocXMLRPCServer Objects
  - 18.26.2 DocCGIHTTPRequestHandler
- 19. Multimedia Services
  - 19.1 audioop -- Manipulate raw audio data
  - 19.2 imageop -- Manipulate raw image data
  - 19.3 aifc -- Read and write AIFF and AIFC files
  - 19.4 sunau -- Read and write Sun AU files
    - 19.4.1 AU_read Objects
    - 19.4.2 AU_write Objects
  - 19.5 wave -- Read and write WAV files
    - 19.5.1 Wave_read Objects
    - 19.5.2 Wave_write Objects
  - 19.6 chunk -- Read IFF chunked data
  - 19.7 colorsys -- Conversions between color systems
  - 19.8 rgbimg -- Read and write ``SGI RGB'' files
  - 19.9 imghdr -- Determine the type of an image
  - 19.10 sndhdr -- Determine type of sound file
  - 19.11 ossaudiodev -- Access to OSS-compatible audio devices
    - 19.11.1 Audio Device Objects
20. Graphical User Interfaces with Tk
   20.1 Tkinter -- Python interface to Tcl/Tk
      20.1.1 Tkinter Modules
      20.1.2 Tkinter Life Preserver
      20.1.3 A (Very) Quick Look at Tcl/Tk
      20.1.4 Mapping Basic Tk into Tkinter
      20.1.5 How Tk and Tkinter are Related
      20.1.6 Handy Reference
   20.2 Tix -- Extension widgets for Tk
      20.2.1 Using Tix
      20.2.2 Tix Widgets
      20.2.3 Tix Commands
   20.3 ScrolledText -- Scrolled Text Widget
   20.4 turtle -- Turtle graphics for Tk
      20.4.1 Turtle, Pen and RawPen Objects
   20.5 Idle
      20.5.1 Menus
      20.5.2 Basic editing and navigation
      20.5.3 Syntax colors
   20.6 Other Graphical User Interface Packages
21. Internationalization
   21.1 gettext -- Multilingual internationalization services
      21.1.1 GNU gettext API
      21.1.2 Class-based API
      21.1.3 Internationalizing your programs and modules
      21.1.4 Acknowledgements
   21.2 locale -- Internationalization services
      21.2.1 Background, details, hints, tips and caveats
      21.2.2 For extension writers and programs that embed Python
      21.2.3 Access to message catalogs
22. Program Frameworks
   22.1 cmd -- Support for line-oriented command interpreters
      22.1.1 Cmd Objects
   22.2 shlex -- Simple lexical analysis
      22.2.1 shlex Objects
      22.2.2 Parsing Rules
23. Development Tools
23.1 pydoc -- Documentation generator and online help system
23.2 doctest -- Test interactive Python examples
   - 23.2.1 Simple Usage: Checking Examples in Docstrings
   - 23.2.2 Simple Usage: Checking Examples in a Text File
   - 23.2.3 How It Works
   - 23.2.4 Basic API
   - 23.2.5 Unittest API
   - 23.2.6 Advanced API
   - 23.2.7 Debugging
   - 23.2.8 Soapbox
23.3 unittest -- Unit testing framework
   - 23.3.1 Basic example
   - 23.3.2 Organizing test code
   - 23.3.3 Re-using old test code
   - 23.3.4 Classes and functions
   - 23.3.5 TestCase Objects
   - 23.3.6 TestSuite Objects
   - 23.3.7 TestResult Objects
   - 23.3.8 TestLoader Objects
23.4 test -- Regression tests package for Python
   - 23.4.1 Writing Unit Tests for the test package
   - 23.4.2 Running tests using test.regrtest
23.5 test.test_support -- Utility functions for tests
24. The Python Debugger
   - 24.1 Debugger Commands
   - 24.2 How It Works
25. The Python Profilers
   - 25.1 Introduction to the profilers
   - 25.2 Instant User's Manual
   - 25.3 What Is Deterministic Profiling?
   - 25.4 Reference Manual - profile and cProfile
     - 25.4.1 The Stats Class
   - 25.5 Limitations
   - 25.6 Calibration
   - 25.7 Extensions -- Deriving Better Profilers
   - 25.8 hotshot -- High performance logging profiler
     - 25.8.1 Profile Objects
     - 25.8.2 Using hotshot data
- 25.8.3 Example Usage
  - 25.9 timeit -- Measure execution time of small code snippets
    - 25.9.1 Command Line Interface
    - 25.9.2 Examples
  - 25.10 trace -- Trace or track Python statement execution
    - 25.10.1 Command Line Usage
    - 25.10.2 Programming Interface

- 26. Python Runtime Services
  - 26.1 sys -- System-specific parameters and functions
  - 26.2 __builtin__ -- Built-in objects
  - 26.3 __main__ -- Top-level script environment
  - 26.4 warnings -- Warning control
    - 26.4.1 Warning Categories
    - 26.4.2 The Warnings Filter
    - 26.4.3 Available Functions
  - 26.5 contextlib -- Utilities for with-statement contexts.
  - 26.6 atexit -- Exit handlers
    - 26.6.1 atexit Example
  - 26.7 traceback -- Print or retrieve a stack traceback
    - 26.7.1 Traceback Example
  - 26.8 __future__ -- Future statement definitions
  - 26.9 gc -- Garbage Collector interface
  - 26.10 inspect -- Inspect live objects
    - 26.10.1 Types and members
    - 26.10.2 Retrieving source code
    - 26.10.3 Classes and functions
    - 26.10.4 The interpreter stack
  - 26.11 site -- Site-specific configuration hook
  - 26.12 user -- User-specific configuration hook
  - 26.13 fpectl -- Floating point exception control
    - 26.13.1 Example
    - 26.13.2 Limitations and other considerations

- 27. Custom Python Interpreters
  - 27.1 code -- Interpreter base classes
    - 27.1.1 Interactive Interpreter Objects
    - 27.1.2 Interactive Console Objects
  - 27.2 codeop -- Compile Python code

- 28. Restricted Execution
28.1 rexec -- Restricted execution framework
   28.1.1 RExec Objects
   28.1.2 Defining restricted environments
   28.1.3 An example
28.2 Bastion -- Restricting access to objects

29. Importing Modules
   29.1 imp -- Access the import internals
     29.1.1 Examples
   29.2 zipimport -- Import modules from Zip archives
     29.2.1 zipimporter Objects
     29.2.2 Examples
   29.3 pkgutil -- Package extension utility
   29.4 modulefinder -- Find modules used by a script
   29.5 runpy -- Locating and executing Python modules

30. Python Language Services
   30.1 parser -- Access Python parse trees
     30.1.1 Creating AST Objects
     30.1.2 Converting AST Objects
     30.1.3 Queries on AST Objects
     30.1.4 Exceptions and Error Handling
     30.1.5 AST Objects
     30.1.6 Examples
   30.2 symbol -- Constants used with Python parse trees
   30.3 token -- Constants used with Python parse trees
   30.4 keyword -- Testing for Python keywords
   30.5 tokenize -- Tokenizer for Python source
   30.6 tabnanny -- Detection of ambiguous indentation
   30.7 pycparser -- Python class browser support
     30.7.1 Class Descriptor Objects
     30.7.2 Function Descriptor Objects
   30.8 py_compile -- Compile Python source files
   30.9 compileall -- Byte-compile Python libraries
   30.10 dis -- Disassembler for Python byte code
     30.10.1 Python Byte Code Instructions
   30.11 pickletools -- Tools for pickle developers
   30.12 distutils -- Building and installing Python modules

31. Python compiler package
   31.1 The basic interface
- 31.2 Limitations
- 31.3 Python Abstract Syntax
  - 31.3.1 AST Nodes
  - 31.3.2 Assignment nodes
  - 31.3.3 Examples
- 31.4 Using Visitors to Walk ASTs
- 31.5 Bytecode Generation

- 32. Abstract Syntax Trees
- 32.1 Abstract Grammar

- 33. Miscellaneous Services
  - 33.1 formatter -- Generic output formatting
    - 33.1.1 The Formatter Interface
    - 33.1.2 Formatter Implementations
    - 33.1.3 The Writer Interface
    - 33.1.4 Writer Implementations
- 34. SGI IRIX Specific Services
  - 34.1 al -- Audio functions on the SGI
    - 34.1.1 Configuration Objects
    - 34.1.2 Port Objects
  - 34.2 AL -- Constants used with the al module
  - 34.3 cd -- CD-ROM access on SGI systems
    - 34.3.1 Player Objects
    - 34.3.2 Parser Objects
  - 34.4 fl -- FORMS library for graphical user interfaces
    - 34.4.1 Functions Defined in Module fl
    - 34.4.2 Form Objects
    - 34.4.3 FORMS Objects
  - 34.5 FL -- Constants used with the fl module
  - 34.6 flp -- Functions for loading stored FORMS designs
  - 34.7 fm -- Font Manager interface
  - 34.8 gl -- Graphics Library interface
  - 34.9 DEVICE -- Constants used with the gl module
  - 34.10 GL -- Constants used with the gl module
  - 34.11 imgfile -- Support for SGI imglib files
  - 34.12 jpeg -- Read and write JPEG files

- 35. SunOS Specific Services
  - 35.1 sunaudiodev -- Access to Sun audio hardware
    - 35.1.1 Audio Device Objects
- **35.2 SUNAUDIODEV -- Constants used with sunaudiodev**
- **36. MS Windows Specific Services**
  - **36.1 msilib -- Read and write Microsoft Installer files**
    - 36.1.1 Database Objects
    - 36.1.2 View Objects
    - 36.1.3 Summary Information Objects
    - 36.1.4 Record Objects
    - 36.1.5 Errors
    - 36.1.6 CAB Objects
    - 36.1.7 Directory Objects
    - 36.1.8 Features
    - 36.1.9 GUI classes
    - 36.1.10 Precomputed tables
  - **36.2 msvcrt - Useful routines from the MS VC++ runtime**
    - 36.2.1 File Operations
    - 36.2.2 Console I/O
    - 36.2.3 Other Functions
  - **36.3 _winreg - Windows registry access**
    - 36.3.1 Registry Handle Objects
  - **36.4 winsound -- Sound-playing interface for Windows**
- **A. Undocumented Modules**
  - A.1 Frameworks
  - A.2 Miscellaneous useful utilities
  - A.3 Platform specific modules
  - A.4 Multimedia
  - A.5 Obsolete
  - A.6 SGI-specific Extension modules
- **B. Reporting Bugs**
- **C. History and License**
  - C.1 History of the software
  - C.2 Terms and conditions for accessing or otherwise using Python
  - C.3 Licenses and Acknowledgements for Incorporated Software
    - C.3.1 Mersenne Twister
    - C.3.2 Sockets
    - C.3.3 Floating point exception control
    - C.3.4 MD5 message digest algorithm
    - C.3.5 Asynchronous socket services
    - C.3.6 Cookie management
- C.3.7 Profiling
- C.3.8 Execution tracing
- C.3.9 UUencode and UUdecode functions
- C.3.10 XML Remote Procedure Calls

- Module Index
- Index
- About this document ...

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1. Introduction

The ``Python library'' contains several different kinds of components.

It contains data types that would normally be considered part of the ``core'' of a
language, such as numbers and lists. For these types, the Python language core
defines the form of literals and places some constraints on their semantics, but
does not fully define the semantics. (On the other hand, the language core does
define syntactic properties like the spelling and priorities of operators.)

The library also contains built-in functions and exceptions -- objects that can be
used by all Python code without the need of an import statement. Some of
these are defined by the core language, but many are not essential for the core
semantics and are only described here.

The bulk of the library, however, consists of a collection of modules. There are
many ways to dissect this collection. Some modules are written in C and built in
to the Python interpreter; others are written in Python and imported in source
form. Some modules provide interfaces that are highly specific to Python, like
printing a stack trace; some provide interfaces that are specific to particular
operating systems, such as access to specific hardware; others provide interfaces
that are specific to a particular application domain, like the World Wide Web.
Some modules are available in all versions and ports of Python; others are only
available when the underlying system supports or requires them; yet others are
available only when a particular configuration option was chosen at the time
when Python was compiled and installed.

This manual is organized ``from the inside out:'' it first describes the built-in data
types, then the built-in functions and exceptions, and finally the modules,
组长 in chapters of related modules. The ordering of the chapters as well as
the ordering of the modules within each chapter is roughly from most relevant to
least important.

This means that if you start reading this manual from the start, and skip to the
next chapter when you get bored, you will get a reasonable overview of the
available modules and application areas that are supported by the Python library.
Of course, you don't have to read it like a novel -- you can also browse the table
of contents (in front of the manual), or look for a specific function, module or term in the index (in the back). And finally, if you enjoy learning about random subjects, you choose a random page number (see module random) and read a section or two. Regardless of the order in which you read the sections of this manual, it helps to start with chapter 2, "Built-in Types, Exceptions and Functions," as the remainder of the manual assumes familiarity with this material.

Let the show begin!
2. Built-in Objects

Names for built-in exceptions and functions and a number of constants are found in a separate symbol table. This table is searched last when the interpreter looks up the meaning of a name, so local and global user-defined names can override built-in names. Built-in types are described together here for easy reference. 2.1

The tables in this chapter document the priorities of operators by listing them in order of ascending priority (within a table) and grouping operators that have the same priority in the same box. Binary operators of the same priority group from left to right. (Unary operators group from right to left, but there you have no real choice.) See chapter 5 of the *Python Reference Manual* for the complete picture on operator priorities.

---

Footnotes

... reference 2.1

Most descriptions sorely lack explanations of the exceptions that may be raised -- this will be fixed in a future version of this manual.

---

Subsections

- 2.1 Built-in Functions
- 2.2 Non-essential Built-in Functions
- 2.3 Built-in Exceptions
- 2.4 Built-in Constants

---

*Release 2.5, documentation updated on 19th September, 2006.*
See About this document... for information on suggesting changes.
2.1 Built-in Functions

The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order.

```python
__import__( name[ , globals[ , locals[ , fromlist[ , level]]]] )
```

This function is invoked by the `import` statement. It mainly exists so that you can replace it with another function that has a compatible interface, in order to change the semantics of the `import` statement. For examples of why and how you would do this, see the standard library modules `ihooks` and `reexec`. See also the built-in module `imp`, which defines some useful operations out of which you can build your own `__import__()` function.

For example, the statement "import spam" results in the following call: `__import__('spam', globals(), locals(), [], -1)`; the statement "from spam.ham import eggs" results in `__import__('spam.ham', globals(), locals(), ['eggs'], -1)`. Note that even though `locals()` and `['eggs']` are passed in as arguments, the `__import__()` function does not set the local variable named `eggs`; this is done by subsequent code that is generated for the import statement. (In fact, the standard implementation does not use its `locals` argument at all, and uses its `globals` only to determine the package context of the `import` statement.)

When the `name` variable is of the form `package.module`, normally, the top-level package (the name up till the first dot) is returned, not the module named by `name`. However, when a non-empty `fromlist` argument is given, the module named by `name` is returned. This is done for compatibility with the bytecode generated for the different kinds of import statement; when using "import spam.ham.eggs", the top-level package `spam` must be placed in the importing namespace, but when using "from spam.ham import eggs", the `spam.ham` subpackage must be used to find the `eggs` variable. As a workaround for this behavior, use `getattr()` to extract the desired components. For example, you could define the following helper:
def my_import(name):
    mod = __import__(name)
    components = name.split('.
    for comp in components[1:]:
        mod = getattr(mod, comp)
    return mod

level specifies whether to use absolute or relative imports. The default is -1 which indicates both absolute and relative imports will be attempted. 0 means only perform absolute imports. Positive values for level indicate the number of parent directories to search relative to the directory of the module calling __import__. Changed in version 2.5: The level parameter was added. Changed in version 2.5: Keyword support for parameters was added.

abs(x)
Return the absolute value of a number. The argument may be a plain or long integer or a floating point number. If the argument is a complex number, its magnitude is returned.

all(iterable)
Return True if all elements of the iterable are true. Equivalent to:

def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True

New in version 2.5.

any(iterable)
Return True if any element of the iterable is true. Equivalent to:

def any(iterable):
    for element in iterable:
        if element:
            return True
    return False

New in version 2.5.

basestring()
This abstract type is the superclass for \texttt{str} and \texttt{unicode}. It cannot be called or instantiated, but it can be used to test whether an object is an instance of \texttt{str} or \texttt{unicode}. \texttt{isinstance(obj, basestring)} is equivalent to \texttt{isinstance(obj, (str, unicode))}. New in version 2.3.

\textbf{bool}([\textit{x}])

Convert a value to a Boolean, using the standard truth testing procedure. If \textit{x} is false or omitted, this returns \texttt{False}; otherwise it returns \texttt{True}. \texttt{bool} is also a class, which is a subclass of \texttt{int}. Class \texttt{bool} cannot be subclassed further. Its only instances are \texttt{False} and \texttt{True}.

New in version 2.2.1. Changed in version 2.3: If no argument is given, this function returns \texttt{False}.

\textbf{callable}( \texttt{object} )

Return true if the \texttt{object} argument appears callable, false if not. If this returns true, it is still possible that a call fails, but if it is false, calling \texttt{object} will never succeed. Note that classes are callable (calling a class returns a new instance); class instances are callable if they have a \texttt{__call__()} method.

\textbf{chr}( \textit{i} )

Return a string of one character whose ASCII code is the integer \textit{i}. For example, \texttt{chr(97)} returns the string \texttt{'a'}. This is the inverse of \texttt{ord()}. The argument must be in the range \([0..255]\), inclusive; \texttt{ValueError} will be raised if \textit{i} is outside that range.

\textbf{classmethod}( \texttt{function} )

Return a class method for \texttt{function}.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

\begin{verbatim}
class C:
    @classmethod
    def f(cls, arg1, arg2, ...): ...
\end{verbatim}
The `@classmethod` form is a function decorator - see the description of function definitions in chapter 7 of the *Python Reference Manual* for details.

It can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see `staticmethod()` in this section.

For more information on class methods, consult the documentation on the standard type hierarchy in chapter 3 of the *Python Reference Manual* (at the bottom). New in version 2.2. Changed in version 2.4: Function decorator syntax added.

**cmp(x, y)**

Compare the two objects `x` and `y` and return an integer according to the outcome. The return value is negative if `x < y`, zero if `x == y` and strictly positive if `x > y`.

**compile(string, filename, kind[, flags[, dont_inherit]])**

Compile the `string` into a code object. Code objects can be executed by an `exec` statement or evaluated by a call to `eval()`. The `filename` argument should give the file from which the code was read; pass some recognizable value if it wasn't read from a file ('<string>' is commonly used). The `kind` argument specifies what kind of code must be compiled: it can be 'exec' if `string` consists of a sequence of statements, 'eval' if it consists of a single expression, or 'single' if it consists of a single interactive statement (in the latter case, expression statements that evaluate to something else than `None` will be printed).

When compiling multi-line statements, two caveats apply: line endings must be represented by a single newline character ('\n'), and the input must be terminated by at least one newline character. If line endings are represented by '\r\n', use the string `replace()` method to change them into '\n'.
The optional arguments flags and dont_inherit (which are new in Python 2.2) control which future statements (see PEP 236) affect the compilation of string. If neither is present (or both are zero) the code is compiled with those future statements that are in effect in the code that is calling compile. If the flags argument is given and dont_inherit is not (or is zero) then the future statements specified by the flags argument are used in addition to those that would be used anyway. If dont_inherit is a non-zero integer then the flags argument is it - the future statements in effect around the call to compile are ignored.

Future statements are specified by bits which can be bitwise or-ed together to specify multiple statements. The bitfield required to specify a given feature can be found as the compiler_flag attribute on the _Feature instance in the __future__ module.

**complex([real[, imag]])**
Create a complex number with the value real + imag*j or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If imag is omitted, it defaults to zero and the function serves as a numeric conversion function like int(), long() and float(). If both arguments are omitted, returns 0j.

**delattr(object, name)**
This is a relative of setattr(). The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, delattr(x, 'foobar') is equivalent to del x.foobar.

**dict([mapping-or-sequence])**
Return a new dictionary initialized from an optional positional argument or from a set of keyword arguments. If no arguments are given, return a new empty dictionary. If the positional argument is a mapping object, return a dictionary mapping the same keys to the same values as does the mapping object. Otherwise the positional argument must be a sequence, a container that supports iteration, or an iterator object. The elements of the argument
must each also be of one of those kinds, and each must in turn contain exactly two objects. The first is used as a key in the new dictionary, and the second as the key's value. If a given key is seen more than once, the last value associated with it is retained in the new dictionary.

If keyword arguments are given, the keywords themselves with their associated values are added as items to the dictionary. If a key is specified both in the positional argument and as a keyword argument, the value associated with the keyword is retained in the dictionary. For example, these all return a dictionary equal to 

\{"one": 2, "two": 3\}:

- \texttt{dict\{\'one\': 2, \'two\': 3\}}
- \texttt{dict\{\'one\': 2, \'two\': 3\}.items()}
- \texttt{dict\{\'one\': 2, \'two\': 3\}.iteritems()}
- \texttt{dict(zip((\'one\', \'two\'), (2, 3))}
- \texttt{dict([[\'two\', 3], [\'one\', 2]])}
- \texttt{dict(one=2, two=3)}
- \texttt{dict([[\'one\', \'two\'][i-2], i) for i in (2, 3)]}

New in version 2.2. Changed in version 2.3: Support for building a dictionary from keyword arguments added.

dir\([\text{object}]\)

Without arguments, return the list of names in the current local symbol table. With an argument, attempts to return a list of valid attributes for that object. This information is gleaned from the object's \_\_dict\_\_ attribute, if defined, and from the class or type object. The list is not necessarily complete. If the object is a module object, the list contains the names of the module's attributes. If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases. Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes. The resulting list is sorted alphabetically. For example:

```python
>>> import struct
>>> dir()
['__builtins__', '__doc__', '__name__', 'struct']
>>> dir(struct)
['__doc__', '__name__', 'calcsize', 'error', 'pack', 'unpack']
```
**Note:** Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases.

**`divmod(a, b)`**

Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using long division. With mixed operand types, the rules for binary arithmetic operators apply. For plain and long integers, the result is the same as `(a // b, a % b)`. For floating point numbers the result is `(q, a % b)`, where `q` is usually `math.floor(a / b)` but may be 1 less than that. In any case `q * b + a % b` is very close to `a`, if `a % b` is non-zero it has the same sign as `b`, and `0 <= abs(a % b) < abs(b)`.

Changed in version 2.3: Using `divmod()` with complex numbers is deprecated.

**`enumerate(iterable)`**

Return an enumerate object. `iterable` must be a sequence, an iterator, or some other object which supports iteration. The `next()` method of the iterator returned by `enumerate()` returns a tuple containing a count (from zero) and the corresponding value obtained from iterating over `iterable`. `enumerate()` is useful for obtaining an indexed series: `(0, seq[0]), (1, seq[1]), (2, seq[2]), .... New in version 2.3.

**`eval(expression[, globals[, locals]]))`**

The arguments are a string and optional globals and locals. If provided, `globals` must be a dictionary. If provided, `locals` can be any mapping object. Changed in version 2.4: formerly `locals` was required to be a dictionary.

The `expression` argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the `globals` and `locals` dictionaries as global and local name space. If the `globals` dictionary is present and lacks `'__builtins__'`, the current globals are copied into `globals` before `expression` is parsed. This means that `expression` normally has full access to the standard `__builtin__` module and restricted environments are propagated. If the `locals` dictionary is omitted it defaults to the `globals`
dictionary. If both dictionaries are omitted, the expression is executed in the
environment where `eval` is called. The return value is the result of the
evaluated expression. Syntax errors are reported as exceptions. Example:

```python
>>> x = 1
>>> print eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as
those created by `compile()`). In this case pass a code object instead of a
string. The code object must have been compiled passing 'eval' as the
`kind` argument.

Hints: dynamic execution of statements is supported by the `exec`
statement. Execution of statements from a file is supported by the
`execfile()` function. The `globals()` and `locals()` functions
returns the current global and local dictionary, respectively, which may be
useful to pass around for use by `eval()` or `execfile()`.

``execfile(filename[, globals[, locals]])``

This function is similar to the `exec` statement, but parses a file instead of a
string. It is different from the `import` statement in that it does not use the
module administration -- it reads the file unconditionally and does not
create a new module.22

The arguments are a file name and two optional dictionaries. The file is
parsed and evaluated as a sequence of Python statements (similarly to a
module) using the `globals` and `locals` dictionaries as global and local
namespace. If provided, `locals` can be any mapping object. Changed in
version 2.4: formerly `locals` was required to be a dictionary. If the `locals`
dictionary is omitted it defaults to the `globals` dictionary. If both
dictionaries are omitted, the expression is executed in the environment
where `execfile()` is called. The return value is `None`.

**Warning:** The default `locals` act as described for function `locals()`
below: modifications to the default `locals` dictionary should not be
attempted. Pass an explicit `locals` dictionary if you need to see effects of the
code on `locals` after function `execfile()` returns. `execfile()` cannot
be used reliably to modify a function's `locals`. 
**file**( *filename*[ , *mode*[ , *bufsize]*)

Constructor function for the file type, described further in section 3.9, `File Objects`. The constructor's arguments are the same as those of the open() built-in function described below.

When opening a file, it's preferable to use open() instead of invoking this constructor directly. file is more suited to type testing (for example, writing "isinstance(f, file)").

New in version 2.2.

**filter**( *function*, *list*)

Construct a list from those elements of list for which function returns true. list may be either a sequence, a container which supports iteration, or an iterator, If list is a string or a tuple, the result also has that type; otherwise it is always a list. If function is None, the identity function is assumed, that is, all elements of list that are false are removed.

Note that filter(function, list) is equivalent to [item for item in list if function(item)] if function is not None and [item for item in list if item] if function is None.

**float**( *[x]*)

Convert a string or a number to floating point. If the argument is a string, it must contain a possibly signed decimal or floating point number, possibly embedded in whitespace. Otherwise, the argument may be a plain or long integer or a floating point number, and a floating point number with the same value (within Python's floating point precision) is returned. If no argument is given, returns 0.0.

**Note:** When passing in a string, values for NaN and Infinity may be returned, depending on the underlying C library. The specific set of strings accepted which cause these values to be returned depends entirely on the C library and is known to vary.

**frozenset**( *[iterable]*)

Return a frozenset object whose elements are taken from iterable. Frozen sets are sets that have no update methods but can be hashed and used
as members of other sets or as dictionary keys. The elements of a frozenset must be immutable themselves. To represent sets of sets, the inner sets should also be frozenset objects. If iterable is not specified, returns a new empty set, frozenset([]). New in version 2.4.

`getattr(object, name[, default])`
Return the value of the named attributed of `object`. `name` must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribute does not exist, `default` is returned if provided, otherwise `AttributeError` is raised.

`globals()`
Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

`hasattr(object, name)`
The arguments are an object and a string. The result is `True` if the string is the name of one of the object's attributes, `False` if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an exception or not.)

`hash(object)`
Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

`help([object])`
Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated.
New in version 2.2.

**hex**\((x)\)

Convert an integer number (of any size) to a hexadecimal string. The result is a valid Python expression. Changed in version 2.4: Formerly only returned an unsigned literal.

**id**\((\text{object})\)

Return the ```identity``` of an object. This is an integer (or long integer) which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same id() value. (Implementation note: this is the address of the object.)

**input**\([\text{prompt}]\)

Equivalent to `eval`\((\text{raw_input}(\text{prompt}))\). **Warning:** This function is not safe from user errors! It expects a valid Python expression as input; if the input is not syntactically valid, a SyntaxError will be raised. Other exceptions may be raised if there is an error during evaluation. (On the other hand, sometimes this is exactly what you need when writing a quick script for expert use.)

If the **readline** module was loaded, then `input()` will use it to provide elaborate line editing and history features.

Consider using the `raw_input()` function for general input from users.

**int**\([x[, \text{radix}]]\)

Convert a string or number to a plain integer. If the argument is a string, it must contain a possibly signed decimal number representable as a Python integer, possibly embedded in whitespace. The `radix` parameter gives the base for the conversion and may be any integer in the range \([2, 36]\), or zero. If `radix` is zero, the proper radix is guessed based on the contents of string; the interpretation is the same as for integer literals. If `radix` is specified and \(x\) is not a string, TypeError is raised. Otherwise, the argument may be a plain or long integer or a floating point number. Conversion of floating point numbers to integers truncates (towards zero). If the argument is outside the integer range a long object will be returned instead. If no arguments are given, returns \(0\).
**isinstance** *(object, classinfo)*
Return true if the *object* argument is an instance of the *classinfo* argument, or of a (direct or indirect) subclass thereof. Also return true if *classinfo* is a type object and *object* is an object of that type. If *object* is not a class instance or an object of the given type, the function always returns false. If *classinfo* is neither a class object nor a type object, it may be a tuple of class or type objects, or may recursively contain other such tuples (other sequence types are not accepted). If *classinfo* is not a class, type, or tuple of classes, types, and such tuples, a *TypeError* exception is raised. Changed in version 2.2: Support for a tuple of type information was added.

**issubclass** *(class, classinfo)*
Return true if *class* is a subclass (direct or indirect) of *classinfo*. A class is considered a subclass of itself. *classinfo* may be a tuple of class objects, in which case every entry in *classinfo* will be checked. In any other case, a *TypeError* exception is raised. Changed in version 2.3: Support for a tuple of type information was added.

**iter** *(o[, sentinel])*
Return an iterator object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, *o* must be a collection object which supports the iteration protocol (the `__iter__()` method), or it must support the sequence protocol (the `__getitem__()` method with integer arguments starting at 0). If it does not support either of those protocols, *TypeError* is raised. If the second argument, *sentinel*, is given, then *o* must be a callable object. The iterator created in this case will call *o* with no arguments for each call to its `next()` method; if the value returned is equal to *sentinel*, *StopIteration* will be raised, otherwise the value will be returned. New in version 2.2.

**len** *(s)*
Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).

**list** *[sequence]*
Return a list whose items are the same and in the same order as *sequence*'s
items. sequence may be either a sequence, a container that supports iteration, or an iterator object. If sequence is already a list, a copy is made and returned, similar to sequence[:]. For instance, list('abc') returns ['a', 'b', 'c'] and list((1, 2, 3)) returns [1, 2, 3]. If no argument is given, returns a new empty list, [].

locals()
Update and return a dictionary representing the current local symbol table. **Warning:** The contents of this dictionary should not be modified; changes may not affect the values of local variables used by the interpreter.

long([x[, radix]])
Convert a string or number to a long integer. If the argument is a string, it must contain a possibly signed number of arbitrary size, possibly embedded in whitespace. The radix argument is interpreted in the same way as for int(), and may only be given when x is a string. Otherwise, the argument may be a plain or long integer or a floating point number, and a long integer with the same value is returned. Conversion of floating point numbers to integers truncates (towards zero). If no arguments are given, returns 0L.

map(function, list, ...)
Apply function to every item of list and return a list of the results. If additional list arguments are passed, function must take that many arguments and is applied to the items of all lists in parallel; if a list is shorter than another it is assumed to be extended with None items. If function is None, the identity function is assumed; if there are multiple list arguments, map() returns a list consisting of tuples containing the corresponding items from all lists (a kind of transpose operation). The list arguments may be any kind of sequence; the result is always a list.

max(s[, args...][key])
With a single argument s, return the largest item of a non-empty sequence (such as a string, tuple or list). With more than one argument, return the largest of the arguments.

The optional key argument specifies a one-argument ordering function like that used for list.sort(). The key argument, if supplied, must be in
keyword form (for example, "\texttt{max(a, b, c, key=func)}"). Changed in version 2.5: Added support for the optional \texttt{key} argument.

\begin{verbatim}
\texttt{min(s[, args...][key])}
With a single argument \texttt{s}, return the smallest item of a non-empty sequence (such as a string, tuple or list). With more than one argument, return the smallest of the arguments.

The optional \texttt{key} argument specifies a one-argument ordering function like that used for \texttt{list.sort()}. The \texttt{key} argument, if supplied, must be in keyword form (for example, "\texttt{min(a, b, c, key=func)}"). Changed in version 2.5: Added support for the optional \texttt{key} argument.
\end{verbatim}

\begin{verbatim}
\texttt{object()}
Return a new featureless object. \texttt{object} is a base for all new style classes. It has the methods that are common to all instances of new style classes. New in version 2.2.

Changed in version 2.3: This function does not accept any arguments. Formerly, it accepted arguments but ignored them.
\end{verbatim}

\begin{verbatim}
\texttt{oct(~x)}
Convert an integer number (of any size) to an octal string. The result is a valid Python expression. Changed in version 2.4: Formerly only returned an unsigned literal.
\end{verbatim}

\begin{verbatim}
\texttt{open(filename[, mode[, bufsize]]})
Open a file, returning an object of the \texttt{file} type described in section 3.9, \texttt{``File Objects''}. If the file cannot be opened, \texttt{IOError} is raised. When opening a file, it's preferable to use \texttt{open()} instead of invoking the \texttt{file} constructor directly.

The first two arguments are the same as for \texttt{stdio's fopen()}: \texttt{filename} is the file name to be opened, and \texttt{mode} is a string indicating how the file is to be opened.

The most commonly-used values of \texttt{mode} are 'r' for reading, 'w' for writing (truncating the file if it already exists), and 'a' for appending
(which on some UNIX systems means that all writes append to the end of the file regardless of the current seek position). If \textit{mode} is omitted, it defaults to 'r'. When opening a binary file, you should append 'b' to the \textit{mode} value to open the file in binary mode, which will improve portability. (Appending 'b' is useful even on systems that don't treat binary and text files differently, where it serves as documentation.) See below for more possible values of \textit{mode}.

The optional \textit{bufsize} argument specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size. A negative \textit{bufsize} means to use the system default, which is usually line buffered for tty devices and fully buffered for other files. If omitted, the system default is used.\footnote{In some systems, 0 is interpreted as line buffering, not unbuffering.}

Modes 'r+', 'w+' and 'a+' open the file for updating (note that 'w+' truncates the file). Append 'b' to the mode to open the file in binary mode, on systems that differentiate between binary and text files; on systems that don't have this distinction, adding the 'b' has no effect.

In addition to the standard \textit{fopen()} values \textit{mode} may be 'U' or 'rU'. Python is usually built with universal newline support; supplying 'U' opens the file as a text file, but lines may be terminated by any of the following: the UNIX end-of-line convention '
', the Macintosh convention '', or the Windows convention '
'. All of these external representations are seen as '
' by the Python program. If Python is built without universal newline support a \textit{mode} with 'U' is the same as normal text mode. Note that file objects so opened also have an attribute called \texttt{newlines} which has a value of \texttt{None} (if no newlines have yet been seen), '
', '', '
', or a tuple containing all the newline types seen.

Python enforces that the mode, after stripping 'U', begins with 'r', 'w' or 'a'.

Changed in version 2.5: Restriction on first letter of mode string introduced.

\texttt{ord(c)}

Given a string of length one, return an integer representing the Unicode
code point of the character when the argument is a unicode object, or the value of the byte when the argument is an 8-bit string. For example, \texttt{ord('a')} returns the integer 97, \texttt{ord(u'\u2020')} returns 8224. This is the inverse of \texttt{chr()} for 8-bit strings and of \texttt{unichr()} for unicode objects. If a unicode argument is given and Python was built with UCS2 Unicode, then the character's code point must be in the range \([0..65535]\) inclusive; otherwise the string length is two, and a \texttt{TypeError} will be raised.

\begin{verbatim}
property([fget[, fset[, fdel[, doc]]]])
\end{verbatim}

Return a property attribute for new-style classes (classes that derive from \texttt{object}).

\texttt{fget} is a function for getting an attribute value, likewise \texttt{fset} is a function for setting, and \texttt{fdel} a function for del'ing, an attribute. Typical use is to define a managed attribute \texttt{x}:

\begin{verbatim}
class C(object):
    def __init__(self): self.__x = None
    def getx(self): return self.__x
\end{verbatim}
If given, `doc` will be the docstring of the property attribute. Otherwise, the property will copy `fget`'s docstring (if it exists). This makes it possible to create read-only properties easily using `property()` as a decorator:

```python
class Parrot(object):
    def __init__(self):
        self._voltage = 100000

@property
def voltage(self):
    """Get the current voltage."""
    return self._voltage
```

turns the `voltage()` method into a `getter` for a read-only attribute with the same name.

New in version 2.2. Changed in version 2.5: Use `fget`'s docstring if no `doc` given.

`range([start,] stop[, step])`
This is a versatile function to create lists containing arithmetic progressions. It is most often used in `for` loops. The arguments must be plain integers. If the `step` argument is omitted, it defaults to 1. If the `start` argument is omitted, it defaults to 0. The full form returns a list of plain integers `[start, start + step, start + 2 * step, ...]`. If `step` is positive, the last element is the largest `start + i * step` less than `stop`; if `step` is negative, the last element is the smallest `start + i * step` greater than `stop`. `step` must not be zero (or else `ValueError` is raised). Example:

```python
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
```
raw_input([prompt])

If the prompt argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, EOFError is raised. Example:

```python
>>> s = raw_input('--> ') --> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the readline module was loaded, then raw_input() will use it to provide elaborate line editing and history features.

reduce(function, sequence[, initializer])

Apply function of two arguments cumulatively to the items of sequence, from left to right, so as to reduce the sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). The left argument, x, is the accumulated value and the right argument, y, is the update value from the sequence. If the optional initializer is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If initializer is not given and sequence contains only one item, the first item is returned.

reload(module)

Reload a previously imported module. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the module argument).

When reload(module) is executed:

- Python modules' code is recompiled and the module-level code reexecuted, defining a new set of objects which are bound to names in the module's dictionary. The init function of extension modules is
not called a second time.

- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.

- The names in the module namespace are updated to point to any new or changed objects.

- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

If a module is syntactically correct but its initialization fails, the first import statement for it does not bind its name locally, but does store a (partially initialized) module object in sys.modules. To reload the module you must first import it again (this will bind the name to the partially initialized module object) before you can reload() it.

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects -- with a try statement it can test for the table's presence and skip its initialization if desired:

```python
try:
    cache
except NameError:
    cache = {}
```

It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for `sys`, `__main__` and `__builtin__`. In many cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using from ... import ..., calling reload() for the other module does not redefine the objects
imported from it -- one way around this is to re-execute the \texttt{from}
statement, another is to use \texttt{import} and qualified names (\texttt{module.name})
instead.

If a module instantiates instances of a class, reloading the module that
defines the class does not affect the method definitions of the instances --
they continue to use the old class definition. The same is true for derived
classes.

\texttt{repr}(\texttt{object})

Return a string containing a printable representation of an object. This is the
same value yielded by conversions (reverse quotes). It is sometimes useful
to be able to access this operation as an ordinary function. For many types,
this function makes an attempt to return a string that would yield an object
with the same value when passed to \texttt{eval()}.

\texttt{reversed}(\texttt{seq})

Return a reverse iterator. \texttt{seq} must be an object which supports the sequence
protocol (the \texttt{__len__()} method and the \texttt{__getitem__()} method with
integer arguments starting at 0). New in version 2.4.

\texttt{round}(\texttt{x}[n])

Return the floating point value \texttt{x} rounded to \texttt{n} digits after the decimal point.
If \texttt{n} is omitted, it defaults to zero. The result is a floating point number.
Values are rounded to the closest multiple of 10 to the power minus \texttt{n}; if
two multiples are equally close, rounding is done away from 0 (so, for
example, \texttt{round(0.5)} is 1.0 and \texttt{round(-0.5)} is -1.0).

\texttt{set}([\texttt{iterable}])

Return a set whose elements are taken from \texttt{iterable}. The elements must be
immutable. To represent sets of sets, the inner sets should be \texttt{frozenset}
objects. If \texttt{iterable} is not specified, returns a new empty set, \texttt{set([])}.
New in version 2.4.

\texttt{setattr}(\texttt{object}, \texttt{name}, \texttt{value})

This is the counterpart of \texttt{getattr()}. The arguments are an object, a
string and an arbitrary value. The string may name an existing attribute or a
new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

**slice**([`start`, `stop`, `step`])

Return a slice object representing the set of indices specified by `range(start, stop, step)`. The `start` and `step` arguments default to `None`. Slice objects have read-only data attributes `start`, `stop` and `step` which merely return the argument values (or their default). They have no other explicit functionality; however they are used by Numerical Python and other third party extensions. Slice objects are also generated when extended indexing syntax is used. For example: "a[start:stop:step]" or "a[start:stop, i]".

**sorted**([`iterable`, `cmp`, `key`, `reverse`])

Return a new sorted list from the items in `iterable`.

The optional arguments `cmp`, `key`, and `reverse` have the same meaning as those for the `list.sort()` method (described in section 3.6.4).

`cmp` specifies a custom comparison function of two arguments (iterable elements) which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument: "`cmp=lambda x,y: cmp(x.lower(), y.lower())""

`key` specifies a function of one argument that is used to extract a comparison key from each list element: "`key=str.lower""

`reverse` is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

In general, the `key` and `reverse` conversion processes are much faster than specifying an equivalent `cmp` function. This is because `cmp` is called multiple times for each list element while `key` and `reverse` touch each element only once.

New in version 2.4.
**staticmethod(function)**

Return a static method for *function*.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```python
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...
```

The @staticmethod form is a function decorator - see the description of function definitions in chapter 7 of the *Python Reference Manual* for details.

It can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++. For a more advanced concept, see `classmethod()` in this section.

For more information on static methods, consult the documentation on the standard type hierarchy in chapter 3 of the *Python Reference Manual* (at the bottom). New in version 2.2. Changed in version 2.4: Function decorator syntax added.

**str([object])**

Return a string containing a nicely printable representation of an object. For strings, this returns the string itself. The difference with `repr(object)` is that `str(object)` does not always attempt to return a string that is acceptable to `eval()`; its goal is to return a printable string. If no argument is given, returns the empty string, `'.'`.

**sum(sequence[, start])**

Sums *start* and the items of a `sequence`, from left to right, and returns the total. *start* defaults to 0. The `sequence`'s items are normally numbers, and are not allowed to be strings. The fast, correct way to concatenate sequence of strings is by calling `''.join(sequence)`.

Note that `sum(range(n), m)` is equivalent to `reduce(operator.add, range(n), m)` New in version 2.3.
**super** *(type[, object-or-type])*

Return the superclass of *type*. If the second argument is omitted the super object returned is unbound. If the second argument is an object, *isinstance(obj, type)* must be true. If the second argument is a type, *issubclass(type2, type)* must be true. *super()* only works for new-style classes.

A typical use for calling a cooperative superclass method is:

```python
class C(B):
    def meth(self, arg):
        super(C, self).meth(arg)
```

Note that *super* is implemented as part of the binding process for explicit dotted attribute lookups such as "super(C, self).__getitem__(name)". Accordingly, *super* is undefined for implicit lookups using statements or operators such as "super(C, self)[name]". New in version 2.2.

**tuple** *[sequence]*

Return a tuple whose items are the same and in the same order as *sequence*'s items. *sequence* may be a sequence, a container that supports iteration, or an iterator object. If *sequence* is already a tuple, it is returned unchanged. For instance, *tuple('abc')* returns ('a', 'b', 'c') and *tuple([1, 2, 3])* returns (1, 2, 3). If no argument is given, returns a new empty tuple, ()

**type** *(object)*

Return the type of an *object*. The return value is a type object. The *isinstance()* built-in function is recommended for testing the type of an object.

With three arguments, *type* functions as a constructor as detailed below.

**type** *(name, bases, dict)*

Return a new type object. This is essentially a dynamic form of the *class* statement. The *name* string is the class name and becomes the **name** attribute; the *bases* tuple itemizes the base classes and becomes the
__bases__ attribute; and the dict dictionary is the namespace containing definitions for class body and becomes the __dict__ attribute. For example, the following two statements create identical type objects:

```python
globals()['X'] = type('X', (object,), dict(a=1))
```

New in version 2.2.

unichr( i)

Return the Unicode string of one character whose Unicode code is the integer i. For example, unichr(97) returns the string u'a'. This is the inverse of ord() for Unicode strings. The valid range for the argument depends how Python was configured - it may be either UCS2 [0..0xFFFF] or UCS4 [0..0x10FFFF]. ValueError is raised otherwise. New in version 2.0.

unicode([ object[, encoding [, errors]]]])

Return the Unicode string version of object using one of the following modes:

If encoding and/or errors are given, unicode() will decode the object which can either be an 8-bit string or a character buffer using the codec for encoding. The encoding parameter is a string giving the name of an encoding; if the encoding is not known, LookupError is raised. Error handling is done according to errors; this specifies the treatment of characters which are invalid in the input encoding. If errors is 'strict' (the default), a ValueError is raised on errors, while a value of 'ignore' causes errors to be silently ignored, and a value of 'replace' causes the official Unicode replacement character, U+FFFD, to be used to replace input characters which cannot be decoded. See also the codecs module.

If no optional parameters are given, unicode() will mimic the behaviour of str() except that it returns Unicode strings instead of 8-bit strings.
More precisely, if object is a Unicode string or subclass it will return that Unicode string without any additional decoding applied.
For objects which provide a `__unicode__()` method, it will call this method without arguments to create a Unicode string. For all other objects, the 8-bit string version or representation is requested and then converted to a Unicode string using the codec for the default encoding in 'strict' mode.

New in version 2.0. Changed in version 2.2: Support for `__unicode__()` added.

**vars([object])**

Without arguments, return a dictionary corresponding to the current local symbol table. With a module, class or class instance object as argument (or anything else that has a `__dict__` attribute), returns a dictionary corresponding to the object's symbol table. The returned dictionary should not be modified: the effects on the corresponding symbol table are undefined.\[\textup{2.4}\]

**xrange([start,] stop[, step])**

This function is very similar to `range()`, but returns an "xrange object" instead of a list. This is an opaque sequence type which yields the same values as the corresponding list, without actually storing them all simultaneously. The advantage of `xrange()` over `range()` is minimal (since `xrange()` still has to create the values when asked for them) except when a very large range is used on a memory-starved machine or when all of the range's elements are never used (such as when the loop is usually terminated with `break`).

**Note:** `xrange()` is intended to be simple and fast. Implementations may impose restrictions to achieve this. The C implementation of Python restricts all arguments to native C longs ("short" Python integers), and also requires that the number of elements fit in a native C long.

**zip([iterable, ...])**

This function returns a list of tuples, where the $i$-th tuple contains the $i$-th element from each of the argument sequences or iterables. The returned list is truncated in length to the length of the shortest argument sequence. When there are multiple arguments which are all of the same length, `zip()` is
similar to `map()` with an initial argument of `None`. With a single sequence argument, it returns a list of 1-tuples. With no arguments, it returns an empty list. New in version 2.0.

Changed in version 2.4: Formerly, `zip()` required at least one argument and `zip()` raised a `TypeError` instead of returning an empty list.

---

**Footnotes**

... module.\footnote{2.2}
It is used relatively rarely so does not warrant being made into a statement.

... used.\footnote{2.3}
Specifying a buffer size currently has no effect on systems that don't have `setvbuf()`. The interface to specify the buffer size is not done using a method that calls `setvbuf()`, because that may dump core when called after any I/O has been performed, and there's no reliable way to determine whether this is the case.

... undefined.\footnote{2.4}
In the current implementation, local variable bindings cannot normally be affected this way, but variables retrieved from other scopes (such as modules) can be. This may change.
2.2 Non-essential Built-in Functions

There are several built-in functions that are no longer essential to learn, know or use in modern Python programming. They have been kept here to maintain backwards compatibility with programs written for older versions of Python.

Python programmers, trainers, students and bookwriters should feel free to bypass these functions without concerns about missing something important.

**apply**(function, args[], keywords)

The function argument must be a callable object (a user-defined or built-in function or method, or a class object) and the args argument must be a sequence. The function is called with args as the argument list; the number of arguments is the length of the tuple. If the optional keywords argument is present, it must be a dictionary whose keys are strings. It specifies keyword arguments to be added to the end of the argument list. Calling apply() is different from just calling function(args), since in that case there is always exactly one argument. The use of apply() is equivalent to function(*args, **keywords). Use of apply() is not necessary since the "extended call syntax," as used in the last example, is completely equivalent.

**Deprecated since release 2.3.** Use the extended call syntax instead, as described above.

**buffer**(object[, offset[, size]])

The object argument must be an object that supports the buffer call interface (such as strings, arrays, and buffers). A new buffer object will be created which references the object argument. The buffer object will be a slice from the beginning of object (or from the specified offset). The slice will extend to the end of object (or will have a length given by the size argument).

**coerce**(x, y)

Return a tuple consisting of the two numeric arguments converted to a common type, using the same rules as used by arithmetic operations. If coercion is not possible, raise TypeError.
`intern(string)`

Enter `string` in the table of "interned" strings and return the interned string - which is `string` itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup - if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys. Changed in version 2.3: Interned strings are not immortal (like they used to be in Python 2.2 and before); you must keep a reference to the return value of `intern()` around to benefit from it.
2.3 Built-in Exceptions

Exceptions should be class objects. The exceptions are defined in the module `exceptions`. This module never needs to be imported explicitly: the exceptions are provided in the built-in namespace as well as the `exceptions` module.

**Note:** In past versions of Python string exceptions were supported. In Python 1.5 and newer versions, all standard exceptions have been converted to class objects and users are encouraged to do the same. String exceptions will raise a `DeprecationWarning` in Python 2.5 and newer. In future versions, support for string exceptions will be removed.

Two distinct string objects with the same value are considered different exceptions. This is done to force programmers to use exception names rather than their string value when specifying exception handlers. The string value of all built-in exceptions is their name, but this is not a requirement for user-defined exceptions or exceptions defined by library modules.

For class exceptions, in a `try` statement with an `except` clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which it is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an ``associated value" indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code). The associated value is the second argument to the `raise` statement. For string exceptions, the associated value itself will be stored in the variable named as the second argument of the `except` clause (if any). For class
exceptions, that variable receives the exception instance. If the exception class is derived from the standard root class BaseException, the associated value is present as the exception instance's args attribute. If there is a single argument (as is preferred), it is bound to the message attribute.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition "just like" the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be sub-classed to define new exceptions; programmers are encouraged to at least derive new exceptions from the Exception class and not BaseException. More information on defining exceptions is available in the Python Tutorial under the heading "User-defined Exceptions."

The following exceptions are only used as base classes for other exceptions.

**Exception BaseException**

The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that use Exception). If str() or unicode() is called on an instance of this class, the representation of the argument(s) to the instance are returned or the empty string when there were no arguments. If only a single argument is passed in, it is stored in the message attribute. If more than one argument is passed in, message is set to the empty string. These semantics are meant to reflect the fact that message is to store a text message explaining why the exception had been raised. If more data needs to be attached to the exception, attach it through arbitrary attributes on the instance. All arguments are also stored in args as a tuple, but it will eventually be deprecated and thus its use is discouraged. New in version 2.5.

**Exception Exception**

All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class. Changed in version 2.5: Changed to inherit from BaseException.

**Exception StandardError**

The base class for all built-in exceptions except StopIteration,
GeneratorExit, KeyboardInterrupt and SystemExit. StandardError itself is derived from Exception.

**exception ArithmeticError**
The base class for those built-in exceptions that are raised for various arithmetic errors: OverflowError, ZeroDivisionError, FloatingPointError.

**exception LookupError**
The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: IndexError, KeyError. This can be raised directly by sys.setdefaultencoding().

**exception EnvironmentError**
The base class for exceptions that can occur outside the Python system: IOError, OSError. When exceptions of this type are created with a 2-tuple, the first item is available on the instance's errno attribute (it is assumed to be an error number), and the second item is available on the strerror attribute (it is usually the associated error message). The tuple itself is also available on the args attribute. New in version 1.5.2.

When an EnvironmentError exception is instantiated with a 3-tuple, the first two items are available as above, while the third item is available on the filename attribute. However, for backwards compatibility, the args attribute contains only a 2-tuple of the first two constructor arguments.

The filename attribute is None when this exception is created with other than 3 arguments. The errno and strerror attributes are also None when the instance was created with other than 2 or 3 arguments. In this last case, args contains the verbatim constructor arguments as a tuple.

The following exceptions are the exceptions that are actually raised.

**exception AssertionError**
Raised when an assert statement fails.

**exception AttributeError**
Raised when an attribute reference or assignment fails. (When an object
does not support attribute references or attribute assignments at all, 
\texttt{TypeError} is raised.)

\textbf{exception \texttt{EOFError}}
Raised when one of the built-in functions (\texttt{input() or raw_input()})
hits an end-of-file condition (EOF) without reading any data. (N.B.: the 
\texttt{read()} and \texttt{readline()} methods of file objects return an empty string 
when they hit EOF.)

\textbf{exception \texttt{FloatingPointError}}
Raised when a floating point operation fails. This exception is always 
defined, but can only be raised when Python is configured with the \texttt{--with-
fpsectl} option, or the \texttt{WANT\_SIGFPE\_HANDLER} symbol is defined in the 
\texttt{pyconfig.h} file.

\textbf{exception \texttt{GeneratorExit}}
Raise when a generator's \texttt{close()} method is called. It directly inherits 
from \texttt{Exception} instead of \texttt{StandardError} since it is technically not 
an error. New in version 2.5.

\textbf{exception \texttt{IOError}}
Raised when an I/O operation (such as a \texttt{print} statement, the built-in 
\texttt{open()} function or a method of a file object) fails for an I/O-related 
reason, e.g., ``file not found'' or ``disk full''.

This class is derived from \texttt{EnvironmentError}. See the discussion 
above for more information on exception instance attributes.

\textbf{exception \texttt{ImportError}}
Raised when an \texttt{import} statement fails to find the module definition or 
when a \texttt{from ... import} fails to find a name that is to be imported.

\textbf{exception \texttt{IndexError}}
Raised when a sequence subscript is out of range. (Slice indices are silently 
truncated to fall in the allowed range; if an index is not a plain integer, 
\texttt{TypeError} is raised.)

\textbf{exception \texttt{KeyError}}
Raised when a mapping (dictionary) key is not found in the set of existing
exception KeyboardInterrupt
Raised when the user hits the interrupt key (normally control-c or delete). During execution, a check for interrupts is made regularly. Interrupts typed when a built-in function input() or raw_input() is waiting for input also raise this exception. The exception inherits from BaseException so as to not be accidentally caught by code that catches Exception and thus prevent the interpreter from exiting. Changed in version 2.5: Changed to inherit from BaseException.

exception MemoryError
Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C’s malloc() function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

exception NameError
Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

exception NotImplementedError
This exception is derived from RuntimeError. In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method. New in version 1.5.2.

exception OSError
This class is derived from EnvironmentError and is used primarily as the os module's os.error exception. See EnvironmentError above for a description of the possible associated values. New in version 1.5.2.

exception OverflowError
Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for long integers (which would rather raise MemoryError than give up). Because of the lack of standardization of
floating point exception handling in C, most floating point operations also aren't checked. For plain integers, all operations that can overflow are checked except left shift, where typical applications prefer to drop bits than raise an exception.

**exception ReferenceError**

This exception is raised when a weak reference proxy, created by the `weakref.proxy()` function, is used to access an attribute of the referent after it has been garbage collected. For more information on weak references, see the `weakref` module. New in version 2.2: Previously known as the `weakref`.ReferenceError exception.

**exception RuntimeError**

Raised when an error is detected that doesn't fall in any of the other categories. The associated value is a string indicating what precisely went wrong. (This exception is mostly a relic from a previous version of the interpreter; it is not used very much any more.)

**exception StopIteration**

Raised by an iterator's `next()` method to signal that there are no further values. This is derived from `Exception` rather than `StandardError`, since this is not considered an error in its normal application. New in version 2.2.

**exception SyntaxError**

Raised when the parser encounters a syntax error. This may occur in an `import` statement, in an `exec` statement, in a call to the built-in function `eval()` or `input()`, or when reading the initial script or standard input (also interactively).

Instances of this class have attributes `filename`, `lineno`, `offset` and `text` for easier access to the details. `str()` of the exception instance returns only the message.

**exception SystemError**

Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms).
You should report this to the author or maintainer of your Python interpreter. Be sure to report the version of the Python interpreter (`sys.version`; it is also printed at the start of an interactive Python session), the exact error message (the exception's associated value) and if possible the source of the program that triggered the error.

**exception SystemExit**

This exception is raised by the `sys.exit()` function. When it is not handled, the Python interpreter exits; no stack traceback is printed. If the associated value is a plain integer, it specifies the system exit status (passed to C's `exit()` function); if it is `None`, the exit status is zero; if it has another type (such as a string), the object's value is printed and the exit status is one.

Instances have an attribute `code` which is set to the proposed exit status or error message (defaulting to `None`). Also, this exception derives directly from `BaseException` and not `StandardError`, since it is not technically an error.

A call to `sys.exit()` is translated into an exception so that clean-up handlers (`finally` clauses of `try` statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The `os._exit()` function can be used if it is absolutely positively necessary to exit immediately (for example, in the child process after a call to `fork()`).

The exception inherits from `BaseException` instead of `StandardError` or `Exception` so that it is not accidentally caught by code that catches `Exception`. This allows the exception to properly propagate up and cause the interpreter to exit. Changed in version 2.5: Changed to inherit from `BaseException`.

**exception TypeError**

Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

**exception UnboundLocalError**

Raised when a reference is made to a local variable in a function or method,
but no value has been bound to that variable. This is a subclass of NameError. New in version 2.0.

**exception UnicodeError**
Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of ValueError. New in version 2.0.

**exception UnicodeEncodeError**
Raised when a Unicode-related error occurs during encoding. It is a subclass of UnicodeError. New in version 2.3.

**exception UnicodeDecodeError**
Raised when a Unicode-related error occurs during decoding. It is a subclass of UnicodeError. New in version 2.3.

**exception UnicodeTranslateError**
Raised when a Unicode-related error occurs during translating. It is a subclass of UnicodeError. New in version 2.3.

**exception ValueError**
Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as IndexError.

**exception WindowsError**
Raised when a Windows-specific error occurs or when the error number does not correspond to an errno value. The winerror and strerror values are created from the return values of the GetLastError() and FormatMessage() functions from the Windows Platform API. The errno value maps the winerror value to corresponding errno.h values. This is a subclass of OSError. New in version 2.0. Changed in version 2.5: Previous versions put the GetLastError() codes into errno.

**exception ZeroDivisionError**
Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.
The following exceptions are used as warning categories; see the `warnings` module for more information.

**exception Warning**
Base class for warning categories.

**exception UserWarning**
Base class for warnings generated by user code.

**exception DeprecationWarning**
Base class for warnings about deprecated features.

**exception PendingDeprecationWarning**
Base class for warnings about features which will be deprecated in the future.

**exception SyntaxWarning**
Base class for warnings about dubious syntax

**exception RuntimeWarning**
Base class for warnings about dubious runtime behavior.

**exception FutureWarning**
Base class for warnings about constructs that will change semantically in the future.

**exception ImportWarning**
Base class for warnings about probable mistakes in module imports. New in version 2.5.

**exception UnicodeWarning**
Base class for warnings related to Unicode. New in version 2.5.

The class hierarchy for built-in exceptions is:

```
BaseException
  +-- SystemExit
  +-- KeyboardInterrupt
  +-- Exception
    +-- GeneratorExit
    +-- StopIteration
```
Download as text (original file name: ../../Lib/test/exception_hierarchy.txt).
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
2.4 Built-in Constants

A small number of constants live in the built-in namespace. They are:

**False**

The false value of the bool type. New in version 2.3.

**True**

The true value of the bool type. New in version 2.3.

**None**

The sole value of types.NoneType. None is frequently used to represent the absence of a value, as when default arguments are not passed to a function.

**NotImplemented**

Special value which can be returned by the ``rich comparison'' special methods (__eq__(), __lt__( ), and friends), to indicate that the comparison is not implemented with respect to the other type.

**Ellipsis**

Special value used in conjunction with extended slicing syntax.
3. Built-in Types

The following sections describe the standard types that are built into the interpreter. **Note:** Historically (until release 2.2), Python's built-in types have differed from user-defined types because it was not possible to use the built-in types as the basis for object-oriented inheritance. This limitation does not exist any longer.

The principal built-in types are numerics, sequences, mappings, files, classes, instances and exceptions.

Some operations are supported by several object types; in particular, practically all objects can be compared, tested for truth value, and converted to a string (with the `repr()` function or the slightly different `str()` function). The latter function is implicitly used when an object is written by the `print` statement. (Information on the `print` statement and other language statements can be found in the *Python Reference Manual* and the *Python Tutorial*.)

---

**Subsections**

- **3.1 Truth Value Testing**
- **3.2 Boolean Operations -- and, or, not**
- **3.3 Comparisons**
- **3.4 Numeric Types -- int, float, long, complex**
  - **3.4.1 Bit-string Operations on Integer Types**
- **3.5 Iterator Types**
- **3.6 Sequence Types -- str, unicode, list, tuple, buffer, xrange**
  - **3.6.1 String Methods**
  - **3.6.2 String Formatting Operations**
  - **3.6.3 XRange Type**
  - **3.6.4 Mutable Sequence Types**
- **3.7 Set Types -- set, frozenset**
- **3.8 Mapping Types -- dict**
- **3.9 File Objects**
3.10 Context Manager Types

3.11 Other Built-in Types
   - 3.11.1 Modules
   - 3.11.2 Classes and Class Instances
   - 3.11.3 Functions
   - 3.11.4 Methods
   - 3.11.5 Code Objects
   - 3.11.6 Type Objects
   - 3.11.7 TheNull Object
   - 3.11.8 The Ellipsis Object
   - 3.11.9 Boolean Values
   - 3.11.10 Internal Objects

3.12 Special Attributes

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
3.1 Truth Value Testing

Any object can be tested for truth value, for use in an if or while condition or as operand of the Boolean operations below. The following values are considered false:

- None
- False
- zero of any numeric type, for example, 0, 0L, 0.0, 0j.
- any empty sequence, for example, '', (), [].
- any empty mapping, for example, {}.
- instances of user-defined classes, if the class defines a __nonzero__() or __len__() method, when that method returns the integer zero or bool value False.\(^{3.1}\)

All other values are considered true -- so objects of many types are always true.

Operations and built-in functions that have a Boolean result always return 0 or False for false and 1 or True for true, unless otherwise stated. (Important exception: the Boolean operations "or" and "and" always return one of their operands.)

Footnotes

...False.\(^{3.1}\)

Additional information on these special methods may be found in the Python Reference Manual.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
# 3.2 Boolean Operations -- and, or, not

These are the Boolean operations, ordered by ascending priority:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \text{ or } y$</td>
<td>if $x$ is false, then $y$, else $x$</td>
<td>(1)</td>
</tr>
<tr>
<td>$x \text{ and } y$</td>
<td>if $x$ is false, then $x$, else $y$</td>
<td>(1)</td>
</tr>
<tr>
<td>not $x$</td>
<td>if $x$ is false, then True, else False</td>
<td>(2)</td>
</tr>
</tbody>
</table>

Notes:

(1) These only evaluate their second argument if needed for their outcome.

(2) "not" has a lower priority than non-Boolean operators, so not $a == b$ is interpreted as not ($a == b$), and $a == not b$ is a syntax error.
3.3 Comparisons

Comparison operations are supported by all objects. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, \( x < y \leq z \) is equivalent to \( x < y \) and \( y \leq z \), except that \( y \) is evaluated only once (but in both cases \( z \) is not evaluated at all when \( x < y \) is found to be false).

This table summarizes the comparison operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>strictly less than</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>strictly greater than</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td>(1)</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equal</td>
<td>(1)</td>
</tr>
<tr>
<td>is</td>
<td>object identity</td>
<td></td>
</tr>
<tr>
<td>is not</td>
<td>negated object identity</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1)  
\(<>\) and \(!=\) are alternate spellings for the same operator. \(!=\) is the preferred spelling; \(<>\) is obsolescent.

Objects of different types, except different numeric types and different string types, never compare equal; such objects are ordered consistently but arbitrarily (so that sorting a heterogeneous array yields a consistent result). Furthermore, some types (for example, file objects) support only a degenerate notion of comparison where any two objects of that type are unequal. Again, such objects are ordered arbitrarily but consistently. The \( <, <=, > \) and \( >= \) operators will raise a **TypeError** exception when any operand is a complex number.
Instances of a class normally compare as non-equal unless the class defines the `__cmp__()` method. Refer to the *Python Reference Manual* for information on the use of this method to effect object comparisons.

**Implementation note:** Objects of different types except numbers are ordered by their type names; objects of the same types that don't support proper comparison are ordered by their address.

Two more operations with the same syntactic priority, "in" and "not in", are supported only by sequence types (below).
3.4 Numeric Types -- int, float, long, complex

There are four distinct numeric types: plain integers, long integers, floating point numbers, and complex numbers. In addition, Booleans are a subtype of plain integers. Plain integers (also just called integers) are implemented using long in C, which gives them at least 32 bits of precision (sys.maxint is always set to the maximum plain integer value for the current platform, the minimum value is \(-sys.maxint - 1\)). Long integers have unlimited precision. Floating point numbers are implemented using double in C. All bets on their precision are off unless you happen to know the machine you are working with.

Complex numbers have a real and imaginary part, which are each implemented using double in C. To extract these parts from a complex number z, use z.real and z.imag.

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including hex and octal numbers) yield plain integers unless the value they denote is too large to be represented as a plain integer, in which case they yield a long integer. Integer literals with an "L" or "l" suffix yield long integers ("L" is preferred because "1l" looks too much like eleven!). Numeric literals containing a decimal point or an exponent sign yield floating point numbers. Appending "j" or "J" to a numeric literal yields a complex number with a zero real part. A complex numeric literal is the sum of a real and an imaginary part.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the `narrower` type is widened to that of the other, where plain integer is narrower than long integer is narrower than floating point is narrower than complex. Comparisons between numbers of mixed type use the same rule. The constructors int(), long(), float(), and complex() can be used to produce numbers of a specific type.

All numeric types (except complex) support the following operations, sorted by
ascending priority (operations in the same box have the same priority; all numeric operations have a higher priority than comparison operations):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + y$</td>
<td>sum of $x$ and $y$</td>
<td></td>
</tr>
<tr>
<td>$x - y$</td>
<td>difference of $x$ and $y$</td>
<td></td>
</tr>
<tr>
<td>$x * y$</td>
<td>product of $x$ and $y$</td>
<td></td>
</tr>
<tr>
<td>$x / y$</td>
<td>quotient of $x$ and $y$</td>
<td>(1)</td>
</tr>
<tr>
<td>$x // y$</td>
<td>(floored) quotient of $x$ and $y$</td>
<td>(5)</td>
</tr>
<tr>
<td>$x % y$</td>
<td>remainder of $x / y$</td>
<td>(4)</td>
</tr>
<tr>
<td>$-x$</td>
<td>$x$ negated</td>
<td></td>
</tr>
<tr>
<td>$+x$</td>
<td>$x$ unchanged</td>
<td></td>
</tr>
<tr>
<td>$\text{abs}(x)$</td>
<td>absolute value or magnitude of $x$</td>
<td></td>
</tr>
<tr>
<td>$\text{int}(x)$</td>
<td>$x$ converted to integer</td>
<td>(2)</td>
</tr>
<tr>
<td>$\text{long}(x)$</td>
<td>$x$ converted to long integer</td>
<td>(2)</td>
</tr>
<tr>
<td>$\text{float}(x)$</td>
<td>$x$ converted to floating point</td>
<td></td>
</tr>
<tr>
<td>$\text{complex}(re, im)$</td>
<td>a complex number with real part $re$, imaginary part $im$. $im$ defaults to zero.</td>
<td></td>
</tr>
<tr>
<td>$c.\text{conjugate}()$</td>
<td>conjugate of the complex number $c$</td>
<td></td>
</tr>
<tr>
<td>$\text{divmod}(x, y)$</td>
<td>the pair $(x // y, x % y)$</td>
<td>(3)(4)</td>
</tr>
<tr>
<td>$\text{pow}(x, y)$</td>
<td>$x$ to the power $y$</td>
<td></td>
</tr>
<tr>
<td>$x ** y$</td>
<td>$x$ to the power $y$</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1)

For (plain or long) integer division, the result is an integer. The result is always rounded towards minus infinity: $1/2$ is $0$, $(-1)/2$ is $-1$, $1/(-2)$ is $-1$, and $(-1)/(-2)$ is $0$. Note that the result is a long integer if either operand is a long integer, regardless of the numeric value.

(2)

Conversion from floating point to (long or plain) integer may round or truncate as in C; see functions $\text{floor}()$ and $\text{ceil}()$ in the $\text{math}$ module for well-defined conversions.
See section 2.1, ``Built-in Functions,'' for a full description.

Complex floor division operator, modulo operator, and \texttt{divmod}().

\textbf{Deprecated since release 2.3.} Instead convert to float using \texttt{abs()} if appropriate.

Also referred to as integer division. The resultant value is a whole integer, though the result's type is not necessarily \texttt{int}.

Footnotes

... rule.\textsuperscript{3.2}

As a consequence, the list $[1, 2]$ is considered equal to $[1.0, 2.0]$, and similarly for tuples.

Subsections

- 3.4.1 Bit-string Operations on Integer Types

Release 2.5, documentation updated on 19th September, 2006.
See \texttt{About this document...} for information on suggesting changes.
3.4.1 Bit-string Operations on Integer Types

Plain and long integer types support additional operations that make sense only for bit-strings. Negative numbers are treated as their 2's complement value (for long integers, this assumes a sufficiently large number of bits that no overflow occurs during the operation).

The priorities of the binary bit-wise operations are all lower than the numeric operations and higher than the comparisons; the unary operation "~" has the same priority as the other unary numeric operations ("+" and "-").

This table lists the bit-string operations sorted in ascending priority (operations in the same box have the same priority):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \mid y$</td>
<td>bitwise or of $x$ and $y$</td>
<td></td>
</tr>
<tr>
<td>$x \wedge y$</td>
<td>bitwise exclusive or of $x$ and $y$</td>
<td></td>
</tr>
<tr>
<td>$x &amp; y$</td>
<td>bitwise and of $x$ and $y$</td>
<td></td>
</tr>
<tr>
<td>$x \ll n$</td>
<td>$x$ shifted left by $n$ bits</td>
<td>(1), (2)</td>
</tr>
<tr>
<td>$x \gg n$</td>
<td>$x$ shifted right by $n$ bits</td>
<td>(1), (3)</td>
</tr>
<tr>
<td>$\neg x$</td>
<td>the bits of $x$ inverted</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1) Negative shift counts are illegal and cause a `ValueError` to be raised.

(2) A left shift by $n$ bits is equivalent to multiplication by $\text{pow}(2, n)$ without overflow check.

(3) A right shift by $n$ bits is equivalent to division by $\text{pow}(2, n)$ without overflow check.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
3.5 Iterator Types

New in version 2.2.

Python supports a concept of iteration over containers. This is implemented using two distinct methods; these are used to allow user-defined classes to support iteration. Sequences, described below in more detail, always support the iteration methods.

One method needs to be defined for container objects to provide iteration support:

```python
__iter__(self)
```

Return an iterator object. The object is required to support the iterator protocol described below. If a container supports different types of iteration, additional methods can be provided to specifically request iterators for those iteration types. (An example of an object supporting multiple forms of iteration would be a tree structure which supports both breadth-first and depth-first traversal.) This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

The iterator objects themselves are required to support the following two methods, which together form the iterator protocol:

```python
__iter__(self)
```

Return the iterator object itself. This is required to allow both containers and iterators to be used with the `for` and `in` statements. This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

```python
next(self)
```

Return the next item from the container. If there are no further items, raise the `StopIteration` exception. This method corresponds to the `tp_iter_next` slot of the type structure for Python objects in the Python/C API.
Python defines several iterator objects to support iteration over general and specific sequence types, dictionaries, and other more specialized forms. The specific types are not important beyond their implementation of the iterator protocol.

The intention of the protocol is that once an iterator's `next()` method raises `StopIteration`, it will continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken. (This constraint was added in Python 2.3; in Python 2.2, various iterators are broken according to this rule.)

Python's generators provide a convenient way to implement the iterator protocol. If a container object's `__iter__()` method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the `__iter__()` and `next()` methods.
3.6 Sequence Types -- str, unicode, list, tuple, buffer, xrange

There are six sequence types: strings, Unicode strings, lists, tuples, buffers, and xrange objects.

String literals are written in single or double quotes: 'xyzzy', "frobozz". See chapter 2 of the Python Reference Manual for more about string literals. Unicode strings are much like strings, but are specified in the syntax using a preceding "u" character: u'abc', u'def'. Lists are constructed with square brackets, separating items with commas: [a, b, c]. Tuples are constructed by the comma operator (not within square brackets), with or without enclosing parentheses, but an empty tuple must have the enclosing parentheses, such as a, b, c or (). A single item tuple must have a trailing comma, such as (d, ).

Buffer objects are not directly supported by Python syntax, but can be created by calling the built-in function buffer(). They don't support concatenation or repetition.

Xrange objects are similar to buffers in that there is no specific syntax to create them, but they are created using the xrange() function. They don't support slicing, concatenation or repetition, and using in, not in, min() or max() on them is inefficient.

Most sequence types support the following operations. The "in" and "not in" operations have the same priorities as the comparison operations. The "+" and "*" operations have the same priority as the corresponding numeric operations.3.3

This table lists the sequence operations sorted in ascending priority (operations in the same box have the same priority). In the table, s and t are sequences of the same type; n, i and j are integers:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x in s</td>
<td>True if an item of s is equal to x, else False</td>
<td>(1)</td>
</tr>
<tr>
<td>$x$ not in $s$</td>
<td>False if an item of $s$ is equal to $x$, else True</td>
<td>(1)</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>$s + t$</td>
<td>the concatenation of $s$ and $t$</td>
<td>(6)</td>
</tr>
<tr>
<td>$s * n$, $n * s$</td>
<td>$n$ shallow copies of $s$ concatenated</td>
<td>(2)</td>
</tr>
<tr>
<td>$s[i]$</td>
<td>$i$'th item of $s$, origin 0</td>
<td>(3)</td>
</tr>
<tr>
<td>$s[i:j]$</td>
<td>slice of $s$ from $i$ to $j$</td>
<td>(3), (4)</td>
</tr>
<tr>
<td>$s[i:j:k]$</td>
<td>slice of $s$ from $i$ to $j$ with step $k$</td>
<td>(3), (5)</td>
</tr>
<tr>
<td>len($s$)</td>
<td>length of $s$</td>
<td></td>
</tr>
<tr>
<td>min($s$)</td>
<td>smallest item of $s$</td>
<td></td>
</tr>
<tr>
<td>max($s$)</td>
<td>largest item of $s$</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1) When $s$ is a string or Unicode string object the `in` and `not in` operations act like a substring test. In Python versions before 2.3, $x$ had to be a string of length 1. In Python 2.3 and beyond, $x$ may be a string of any length.

(2) Values of $n$ less than 0 are treated as 0 (which yields an empty sequence of the same type as $s$). Note also that the copies are shallow; nested structures are not copied. This often haunts new Python programmers; consider:

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

What has happened is that `[]` is a one-element list containing an empty list, so all three elements of `[[ ]] * 3` are (pointers to) this single empty list. Modifying any of the elements of `lists` modifies this single list. You can create a list of different lists this way:

```
>>> lists = [[i for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```
If $i$ or $j$ is negative, the index is relative to the end of the string: $\text{len}(s) + i$ or $\text{len}(s) + j$ is substituted. But note that $-0$ is still $0$.

The slice of $s$ from $i$ to $j$ is defined as the sequence of items with index $k$ such that $i \leq k < j$. If $i$ or $j$ is greater than $\text{len}(s)$, use $\text{len}(s)$. If $i$ is omitted or $\text{None}$, use $0$. If $j$ is omitted or $\text{None}$, use $\text{len}(s)$. If $i$ is greater than or equal to $j$, the slice is empty.

The slice of $s$ from $i$ to $j$ with step $k$ is defined as the sequence of items with index $x = i + n*\frac{k}{k}$ such that $0 \leq n < \frac{i-j}{k}$. In other words, the indices are $i$, $i+k$, $i+2*k$, $i+3*k$ and so on, stopping when $j$ is reached (but never including $j$). If $i$ or $j$ is greater than $\text{len}(s)$, use $\text{len}(s)$. If $i$ or $j$ are omitted or $\text{None}$, they become `end` values (which end depends on the sign of $k$). Note, $k$ cannot be zero. If $k$ is $\text{None}$, it is treated like $1$.

If $s$ and $t$ are both strings, some Python implementations such as CPython can usually perform an in-place optimization for assignments of the form $s=s+t$ or $s+=t$. When applicable, this optimization makes quadratic run-time much less likely. This optimization is both version and implementation dependent. For performance sensitive code, it is preferable to use the $\text{str.join()}$ method which assures consistent linear concatenation performance across versions and implementations. Changed in version 2.4: Formerly, string concatenation never occurred in-place.

---

Footnotes

... operations.\footnote{3.3}

They must have since the parser can't tell the type of the operands.
3.6.1 String Methods

These are the string methods which both 8-bit strings and Unicode objects support:

**capitalize()**
Return a copy of the string with only its first character capitalized.

For 8-bit strings, this method is locale-dependent.

**center( width[, fillchar])**
Return centered in a string of length width. Padding is done using the specified fillchar (default is a space). Changed in version 2.4: Support for the fillchar argument.

**count( sub[, start[, end]])**
Return the number of occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

**decode( [encoding[, errors]])**
Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding. errors may be given to set a different error handling scheme. The default is 'strict', meaning that encoding errors raise UnicodeError. Other possible values are 'ignore', 'replace' and any other name registered via codecs.register_error, see section 4.8.1. New in version 2.2. Changed in version 2.3: Support for other error handling schemes added.

**encode([encoding[,errors]])**
Return an encoded version of the string. Default encoding is the current default string encoding. errors may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a UnicodeError. Other possible values are 'ignore','replace','xmlcharrefreplace', 'backslashreplace' and any other name registered via
codecs.register_error, see section 4.8.1. For a list of possible encodings, see section 4.8.3. New in version 2.0. Changed in version 2.3: Support for 'xmlcharrefreplace' and 'backslashreplace' and other error handling schemes added.

**endswith**(suffix[, start[, end]])
Return True if the string ends with the specified suffix, otherwise return False. suffix can also be a tuple of suffixes to look for. With optional start, test beginning at that position. With optional end, stop comparing at that position.

Changed in version 2.5: Accept tuples as suffix.

**expandtabs**(^[tabsize])
Return a copy of the string where all tab characters are expanded using spaces. If tabsize is not given, a tab size of 8 characters is assumed.

**find**(sub[, start[, end]])
Return the lowest index in the string where substring sub is found, such that sub is contained in the range [start, end]. Optional arguments start and end are interpreted as in slice notation. Return -1 if sub is not found.

**index**(sub[, start[, end]])
Like find(), but raise ValueError when the substring is not found.

**isalnum()**
Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

**isalpha()**
Return true if all characters in the string are alphabetic and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.
isdigit()
    Return true if all characters in the string are digits and there is at least one character, false otherwise.

    For 8-bit strings, this method is locale-dependent.

islower()
    Return true if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

    For 8-bit strings, this method is locale-dependent.

isspace()
    Return true if there are only whitespace characters in the string and there is at least one character, false otherwise.

    For 8-bit strings, this method is locale-dependent.

istitle()
    Return true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise.

    For 8-bit strings, this method is locale-dependent.

isupper()
    Return true if all cased characters in the string are uppercase and there is at least one cased character, false otherwise.

    For 8-bit strings, this method is locale-dependent.

join(seq)
    Return a string which is the concatenation of the strings in the sequence seq. The separator between elements is the string providing this method.

ljust(width[, fillchar])
    Return the string left justified in a string of length width. Padding is done
using the specified fillchar (default is a space). The original string is returned if width is less than len(s). Changed in version 2.4: Support for the fillchar argument.

lower()
Return a copy of the string converted to lowercase.

For 8-bit strings, this method is locale-dependent.

lstrip([chars])
Return a copy of the string with leading characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a prefix; rather, all combinations of its values are stripped:

```python
>>> '    spacious     '.lstrip()
'spacious'
>>> 'www.example.com'.lstrip('cmowz.')</n'example.com'
```

Changed in version 2.2.2: Support for the chars argument.

partition(sep)
Split the string at the first occurrence of sep, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings. New in version 2.5.

replace(old, new[, count])
Return a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

rfind(sub [,start [,end]])
Return the highest index in the string where substring sub is found, such that sub is contained within s[start,end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.
**rindex**(*sub*, *start*, *end*))

Like `rfind()` but raises `ValueError` when the substring `sub` is not found.

**rjust**(*width*, *fillchar*)

Return the string right justified in a string of length `width`. Padding is done using the specified `fillchar` (default is a space). The original string is returned if `width` is less than `len(s)`. Changed in version 2.4: Support for the `fillchar` argument.

**rpartition**(*sep*)

Split the string at the last occurrence of `sep`, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself. New in version 2.5.

**rsplit**(*sep*, *maxsplit*)

Return a list of the words in the string, using `sep` as the delimiter string. If `maxsplit` is given, at most `maxsplit` splits are done, the rightmost ones. If `sep` is not specified or `None`, any whitespace string is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below. New in version 2.4.

**rstrip**(*chars*)

Return a copy of the string with trailing characters removed. The `chars` argument is a string specifying the set of characters to be removed. If omitted or `None`, the `chars` argument defaults to removing whitespace. The `chars` argument is not a suffix; rather, all combinations of its values are stripped:

```python
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

Changed in version 2.2.2: Support for the `chars` argument.

**split**(*sep*, *maxsplit*)


Return a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. (thus, the list will have at most maxsplit+1 elements). If maxsplit is not specified, then there is no limit on the number of splits (all possible splits are made). Consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, "'1''2'.split('', '')" returns "['1', '', '2']"). The sep argument may consist of multiple characters (for example, "'1, 2, 3'.split(',' )" returns "['__1', '__2', '__3']"). Splitting an empty string with a specified separator returns "['__']"

If sep is not specified or is None, a different splitting algorithm is applied. First, whitespace characters (spaces, tabs, newlines, returns, and formfeeds) are stripped from both ends. Then, words are separated by arbitrary length strings of whitespace characters. Consecutive whitespace delimiters are treated as a single delimiter ("'1 2 3'.split()" returns "['__1', '__2', '__3']"). Splitting an empty string or a string consisting of just whitespace returns an empty list.

**splitlines([keepends])**

Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.

**startswith(prefix[, start[, end]])**

Return True if string starts with the prefix, otherwise return False. prefix can also be a tuple of suffixes to look for. With optional start, test string beginning at that position. With optional end, stop comparing string at that position.

Changed in version 2.5: Accept tuples as prefix.

**strip([chars])**

Return a copy of the string with the leading and trailing characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a prefix or suffix; rather, all combinations of its values are stripped:
\[\text{spacious}.\text{strip()}\]
\[\text{spacious}\]
\[\text{www.example.com}.\text{strip('cmowz.')}\]
\[\text{example}\]

Changed in version 2.2.2: Support for the \textit{chars} argument.

\texttt{swapcase()}  
Return a copy of the string with uppercase characters converted to lowercase and vice versa.

For 8-bit strings, this method is locale-dependent.

\texttt{title()}  
Return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase.

For 8-bit strings, this method is locale-dependent.

\texttt{translate}(\textit{table[, deletechars]}))  
Return a copy of the string where all characters occurring in the optional argument \textit{deletechars} are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256.

For Unicode objects, the \texttt{translate()} method does not accept the optional \textit{deletechars} argument. Instead, it returns a copy of the string where all characters have been mapped through the given translation table which must be a mapping of Unicode ordinals to Unicode ordinals, Unicode strings or \texttt{None}. Unmapped characters are left untouched. Characters mapped to \texttt{None} are deleted. Note, a more flexible approach is to create a custom character mapping codec using the \texttt{codecs} module (see \texttt{encodings.cp1251} for an example).

\texttt{upper()}  
Return a copy of the string converted to uppercase.

For 8-bit strings, this method is locale-dependent.
**zfill()**

Return the numeric string left filled with zeros in a string of length `width`. The original string is returned if `width` is less than `len(s)`. New in version 2.2.2.
3.6.2 String Formatting Operations

String and Unicode objects have one unique built-in operation: the % operator (modulo). This is also known as the string formatting or interpolation operator. Given `format % values` (where `format` is a string or Unicode object), % conversion specifications in `format` are replaced with zero or more elements of `values`. The effect is similar to the using `sprintf()` in the C language. If `format` is a Unicode object, or if any of the objects being converted using the `%s` conversion are Unicode objects, the result will also be a Unicode object.

If `format` requires a single argument, `values` may be a single non-tuple object. Otherwise, `values` must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The "%" character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, `(somename)`).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an "*" (asterisk), the actual width is read from the next element of the tuple in `values`, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a "." (dot) followed by the precision. If specified as "*" (an asterisk), the actual width is read from the next element of the tuple in `values`, and the value to convert comes after the precision.
7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the string must include a parenthesised mapping key into that dictionary inserted immediately after the "%" character. The mapping key selects the value to be formatted from the mapping. For example:

```python
>>> print '%(language)s has %(#)03d quote types.' %
    {'language': 'Python', '#': 2}
```
Python has 002 quote types.

In this case no * specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>The value conversion will use the <code>alternate form</code> (where defined below).</td>
</tr>
<tr>
<td>0</td>
<td>The conversion will be zero padded for numeric values.</td>
</tr>
<tr>
<td>-</td>
<td>The converted value is left adjusted (overrides the &quot;0&quot; conversion if both are given).</td>
</tr>
<tr>
<td></td>
<td>(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.</td>
</tr>
<tr>
<td>+</td>
<td>A sign character (&quot;+&quot; or &quot;-&quot;) will precede the conversion (overrides a &quot;space&quot; flag).</td>
</tr>
</tbody>
</table>

A length modifier (h, l, or L) may be present, but is ignored as it is not necessary for Python.

The conversion types are:

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Signed integer decimal.</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>Signed integer decimal.</td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>Unsigned octal.</td>
<td>(1)</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal.</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>Unsigned hexadecimal (lowercase).</td>
<td>(2)</td>
</tr>
<tr>
<td>X</td>
<td>Unsigned hexadecimal (uppercase).</td>
<td>(2)</td>
</tr>
<tr>
<td>e</td>
<td>Floating point exponential format (lowercase).</td>
<td>(3)</td>
</tr>
<tr>
<td>E</td>
<td>Floating point exponential format (uppercase).</td>
<td>(3)</td>
</tr>
<tr>
<td>f</td>
<td>Floating point decimal format.</td>
<td>(3)</td>
</tr>
<tr>
<td>F</td>
<td>Floating point decimal format.</td>
<td>(3)</td>
</tr>
<tr>
<td>g</td>
<td>Floating point format. Uses exponential format if exponent is greater than -4 or less than precision, decimal format otherwise.</td>
<td>(4)</td>
</tr>
<tr>
<td>G</td>
<td>Floating point format. Uses exponential format if exponent is greater than -4 or less than precision, decimal format otherwise. (4)</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>Single character (accepts integer or single character string). (c)</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>String (converts any python object using repr()). (5)</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>String (converts any python object using str()). (6)</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>No argument is converted, results in a &quot;%&quot; character in the result.</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1) The alternate form causes a leading zero ("0") to be inserted between left-hand padding and the formatting of the number if the leading character of the result is not already a zero.

(2) The alternate form causes a leading '0x' or '0X' (depending on whether the "x" or "X" format was used) to be inserted between left-hand padding and the formatting of the number if the leading character of the result is not already a zero.

(3) The alternate form causes the result to always contain a decimal point, even if no digits follow it.

The precision determines the number of digits after the decimal point and defaults to 6.

(4) The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

The precision determines the number of significant digits before and after the decimal point and defaults to 6.

(5) The %r conversion was added in Python 2.0.

The precision determines the maximal number of characters used.
If the object or format provided is a `unicode` string, the resulting string will also be `unicode`.

The precision determines the maximal number of characters used.

Since Python strings have an explicit length, `%s` conversions do not assume that ' \0 ' is the end of the string.

For safety reasons, floating point precisions are clipped to 50; `%f` conversions for numbers whose absolute value is over 1e25 are replaced by `%g` conversions.3.5 All other errors raise exceptions.

Additional string operations are defined in standard modules `string` and `re`.

---

**Footnotes**

... object.3.4

To format only a tuple you should therefore provide a singleton tuple whose only element is the tuple to be formatted.

... conversions.3.5

These numbers are fairly arbitrary. They are intended to avoid printing endless strings of meaningless digits without hampering correct use and without having to know the exact precision of floating point values on a particular machine.
3.6.3 XRange Type

The `xrange` type is an immutable sequence which is commonly used for looping. The advantage of the `xrange` type is that an `xrange` object will always take the same amount of memory, no matter the size of the range it represents. There are no consistent performance advantages.

XRange objects have very little behavior: they only support indexing, iteration, and the `len()` function.
3.6.4 Mutable Sequence Types

List objects support additional operations that allow in-place modification of the object. Other mutable sequence types (when added to the language) should also support these operations. Strings and tuples are immutable sequence types: such objects cannot be modified once created. The following operations are defined on mutable sequence types (where \( x \) is an arbitrary object):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s[i] = x )</td>
<td>item ( i ) of ( s ) is replaced by ( x )</td>
<td></td>
</tr>
<tr>
<td>( s[i:j] = t )</td>
<td>slice of ( s ) from ( i ) to ( j ) is replaced by the contents of the iterable ( t )</td>
<td></td>
</tr>
<tr>
<td>( \text{del } s[i:j] )</td>
<td>same as ( s[i:j] = [] )</td>
<td></td>
</tr>
<tr>
<td>( s[i:j:k] = t )</td>
<td>the elements of ( s[i:j:k] ) are replaced by those of ( t )</td>
<td>(1)</td>
</tr>
<tr>
<td>( \text{del } s[i:j:k] )</td>
<td>removes the elements of ( s[i:j:k] ) from the list</td>
<td></td>
</tr>
<tr>
<td>( s.append(x) )</td>
<td>same as ( s[len(s):len(s)] = [x] )</td>
<td>(2)</td>
</tr>
<tr>
<td>( s.extend(x) )</td>
<td>same as ( s[len(s):len(s)] = x )</td>
<td>(3)</td>
</tr>
<tr>
<td>( s.count(x) )</td>
<td>return number of ( i )'s for which ( s[i] == x )</td>
<td></td>
</tr>
<tr>
<td>( s.index(x[, i[, j]]) )</td>
<td>return smallest ( k ) such that ( s[k] == x ) and ( i &lt;= k &lt; j )</td>
<td>(4)</td>
</tr>
<tr>
<td>( s.insert(i, x) )</td>
<td>same as ( s[i:i] = [x] )</td>
<td>(5)</td>
</tr>
<tr>
<td>( s.pop([i]) )</td>
<td>same as ( x = s[i]; \text{del } s[i]; \text{return } x )</td>
<td>(6)</td>
</tr>
<tr>
<td>( s.remove(x) )</td>
<td>same as ( \text{del } s[s.index(x)] )</td>
<td>(4)</td>
</tr>
<tr>
<td>( s.reverse() )</td>
<td>reverses the items of ( s ) in place</td>
<td>(7)</td>
</tr>
<tr>
<td>( s.sort([cmp[, key[, reverse]]]]) )</td>
<td>sort the items of ( s ) in place</td>
<td>(7), (8), (9), (10)</td>
</tr>
</tbody>
</table>

Notes:

(1)
\( t \) must have the same length as the slice it is replacing.

(2) The C implementation of Python has historically accepted multiple parameters and implicitly joined them into a tuple; this no longer works in Python 2.0. Use of this misfeature has been deprecated since Python 1.4.

(3) \( x \) can be any iterable object.

(4) Raises \texttt{ValueError} when \( x \) is not found in \( s \). When a negative index is passed as the second or third parameter to the \texttt{index()} method, the list length is added, as for slice indices. If it is still negative, it is truncated to zero, as for slice indices. Changed in version 2.3: Previously, \texttt{index()} didn't have arguments for specifying start and stop positions.

(5) When a negative index is passed as the first parameter to the \texttt{insert()} method, the list length is added, as for slice indices. If it is still negative, it is truncated to zero, as for slice indices. Changed in version 2.3: Previously, all negative indices were truncated to zero.

(6) The \texttt{pop()} method is only supported by the list and array types. The optional argument \( i \) defaults to \(-1\), so that by default the last item is removed and returned.

(7) The \texttt{sort()} and \texttt{reverse()} methods modify the list in place for economy of space when sorting or reversing a large list. To remind you that they operate by side effect, they don't return the sorted or reversed list.

(8) The \texttt{sort()} method takes optional arguments for controlling the comparisons.

\texttt{cmp} specifies a custom comparison function of two arguments (list items) which should return a negative, zero or positive number depending on
whether the first argument is considered smaller than, equal to, or larger
than the second argument: "\texttt{cmp=\lambda x,y: \texttt{cmp}(x.lower(), y.lower())}"

\textit{key} specifies a function of one argument that is used to extract a comparison
key from each list element: "\texttt{key=\texttt{str}.lower}"

\textit{reverse} is a boolean value. If set to \texttt{True}, then the list elements are sorted
as if each comparison were reversed.

In general, the \textit{key} and \textit{reverse} conversion processes are much faster than
specifying an equivalent \texttt{cmp} function. This is because \texttt{cmp} is called
multiple times for each list element while \textit{key} and \textit{reverse} touch each
element only once.

Changed in version 2.3: Support for \texttt{None} as an equivalent to omitting \texttt{cmp}
was added.

Changed in version 2.4: Support for \textit{key} and \textit{reverse} was added.

(9)

Starting with Python 2.3, the \texttt{sort()} method is guaranteed to be stable. A
sort is stable if it guarantees not to change the relative order of elements
that compare equal -- this is helpful for sorting in multiple passes (for
example, sort by department, then by salary grade).

(10)

While a list is being sorted, the effect of attempting to mutate, or even
inspect, the list is undefined. The C implementation of Python 2.3 and
newer makes the list appear empty for the duration, and raises
\texttt{ValueError} if it can detect that the list has been mutated during a sort.
3.7 Set Types -- set, frozenset

A set object is an unordered collection of immutable values. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference. New in version 2.4.

Like other collections, sets support \( x \text{ in } \text{set} \), \( \text{len(set)} \), and \( \text{for } x \text{ in } \text{set} \). Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

There are currently two builtin set types, set and frozenset. The set type is mutable -- the contents can be changed using methods like \( \text{add()} \) and \( \text{remove()} \). Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. The frozenset type is immutable and hashable -- its contents cannot be altered after is created; however, it can be used as a dictionary key or as an element of another set.

Instances of set and frozenset provide the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Equivalent</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{len(s)} )</td>
<td></td>
<td>cardinality of set ( s )</td>
</tr>
<tr>
<td>( x \text{ in } s )</td>
<td></td>
<td>test ( x ) for membership in ( s )</td>
</tr>
<tr>
<td>( x \text{ not in } s )</td>
<td></td>
<td>test ( x ) for non-membership in ( s )</td>
</tr>
<tr>
<td>( s \text{.issubset}(t) )</td>
<td>( s &lt;= t )</td>
<td>test whether every element in ( s ) is in ( t )</td>
</tr>
<tr>
<td>( s \text{.issuperset}(t) )</td>
<td>( s &gt;= t )</td>
<td>test whether every element in ( t ) is in ( s )</td>
</tr>
<tr>
<td>( s \text{.union}(t) )</td>
<td>( s</td>
<td>t )</td>
</tr>
<tr>
<td>( s \text{.intersection}(t) )</td>
<td>( s &amp; t )</td>
<td>new set with elements common to ( s ) and ( t )</td>
</tr>
<tr>
<td>( s \text{.difference}(t) )</td>
<td>( s - t )</td>
<td>new set with elements in ( s ) but not in ( t )</td>
</tr>
<tr>
<td>Operation</td>
<td>Expression</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td><code>s.symmetric_difference(t)</code></td>
<td><code>s^t</code></td>
<td>new set with elements in either s or t but not both</td>
</tr>
<tr>
<td><code>s.copy()</code></td>
<td></td>
<td>new set with a shallow copy of s</td>
</tr>
</tbody>
</table>

Note, the non-operator versions of `union()`, `intersection()`, `difference()`, and `symmetric_difference()`, `issubset()`, and `issuperset()` methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like `set('abc') & 'cbs'` in favor of the more readable `set('abc').intersection('cbs')`.

Both `set` and `frozenset` support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

Instances of `set` are compared to instances of `frozenset` based on their members. For example, "set('abc') == frozenset('abc')" returns `True`.

The subset and equality comparisons do not generalize to a complete ordering function. For example, any two disjoint sets are not equal and are not subsets of each other, so all of the following return `False`: `a<b`, `a==b`, or `a>b`. Accordingly, sets do not implement the `__cmp__` method.

Since sets only define partial ordering (subset relationships), the output of the `list.sort()` method is undefined for lists of sets.

Set elements are like dictionary keys; they need to define both `__hash__` and `__eq__` methods.

Binary operations that mix `set` instances with `frozenset` return the type of the first operand. For example: "frozenset('ab') | set('bc')"
returns an instance of `frozenset`. 
The following table lists operations available for `set` that do not apply to immutable instances of `frozenset`:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Equivalent</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s.update(t)</code></td>
<td>`s</td>
<td>= t`</td>
</tr>
<tr>
<td><code>s.intersection_update(t)</code></td>
<td><code>s &amp;*= t</code></td>
<td>update set <code>s</code>, keeping only elements found in both <code>s</code> and <code>t</code></td>
</tr>
<tr>
<td><code>s.difference_update(t)</code></td>
<td><code>s -= t</code></td>
<td>update set <code>s</code>, removing elements found in <code>t</code></td>
</tr>
<tr>
<td><code>s.symmetric_difference_update(t)</code></td>
<td><code>s ^= t</code></td>
<td>update set <code>s</code>, keeping only elements found in either <code>s</code> or <code>t</code> but not in both</td>
</tr>
<tr>
<td><code>s.add(x)</code></td>
<td></td>
<td>add element <code>x</code> to set <code>s</code></td>
</tr>
<tr>
<td><code>s.remove(x)</code></td>
<td></td>
<td>remove <code>x</code> from set <code>s</code>; raises <code>KeyError</code> if not present</td>
</tr>
<tr>
<td><code>s.discard(x)</code></td>
<td></td>
<td>removes <code>x</code> from set <code>s</code> if present</td>
</tr>
<tr>
<td><code>s.pop()</code></td>
<td></td>
<td>remove and return an arbitrary element from <code>s</code>; raises <code>KeyError</code> if empty</td>
</tr>
</tbody>
</table>
s.clear() remove all elements from set s

Note, the non-operator versions of the update(), intersection_update(), difference_update(), and symmetric_difference_update() methods will accept any iterable as an argument.

The design of the set types was based on lessons learned from the sets module.

See Also:

Comparison to the built-in set types
Differences between the sets module and the built-in set types.
3.8 Mapping Types -- dict

A mapping object maps immutable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary. A dictionary's keys are almost arbitrary values. Only values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (such as 1 and 1.0) then they can be used interchangeably to index the same dictionary entry.

Dictionaries are created by placing a comma-separated list of key: value pairs within braces, for example: {'jack': 4098, 'sjoerd': 4127} or {4098: 'jack', 4127: 'sjoerd'}.

The following operations are defined on mappings (where a and b are mappings, k is a key, and v and x are arbitrary objects):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>len(a)</td>
<td>the number of items in a</td>
<td></td>
</tr>
<tr>
<td>a[k]</td>
<td>the item of a with key k</td>
<td>(1), (10)</td>
</tr>
<tr>
<td>a[k] = v</td>
<td>set a[k] to v</td>
<td></td>
</tr>
<tr>
<td>del a[k]</td>
<td>remove a[k] from a</td>
<td>(1)</td>
</tr>
<tr>
<td>a.clear()</td>
<td>remove all items from a</td>
<td></td>
</tr>
<tr>
<td>a.copy()</td>
<td>a (shallow) copy of a</td>
<td></td>
</tr>
<tr>
<td>k in a</td>
<td>True if a has a key k, else False</td>
<td>(2)</td>
</tr>
<tr>
<td>k not in a</td>
<td>Equivalent to not k in a</td>
<td>(2)</td>
</tr>
<tr>
<td>a.has_key(k)</td>
<td>Equivalent to k in a, use that form in new code</td>
<td></td>
</tr>
<tr>
<td>a.items()</td>
<td>a copy of a's list of (key, value) pairs</td>
<td>(3)</td>
</tr>
<tr>
<td>a.keys()</td>
<td>a copy of a's list of keys</td>
<td>(3)</td>
</tr>
<tr>
<td>a.update([b])</td>
<td>updates (and overwrites) key/value pairs from b</td>
<td>(9)</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td><code>a.fromkeys(seq[, value])</code></td>
<td>Creates a new dictionary with keys from <code>seq</code> and values set to <code>value</code></td>
<td>(7)</td>
</tr>
<tr>
<td><code>a.values()</code></td>
<td>a copy of a's list of values</td>
<td>(3)</td>
</tr>
<tr>
<td><code>a.get(k[, x])</code></td>
<td><code>a[k]</code> if <code>k</code> in <code>a</code>, else <code>x</code></td>
<td>(4)</td>
</tr>
<tr>
<td><code>a.setdefault(k[, x])</code></td>
<td><code>a[k]</code> if <code>k</code> in <code>a</code>, else <code>x</code> (also setting it)</td>
<td>(5)</td>
</tr>
<tr>
<td><code>a.pop(k[, x])</code></td>
<td><code>a[k]</code> if <code>k</code> in <code>a</code>, else <code>x</code> (and remove k)</td>
<td>(8)</td>
</tr>
<tr>
<td><code>a.popitem()</code></td>
<td>remove and return an arbitrary <code>(key, value)</code> pair</td>
<td>(6)</td>
</tr>
<tr>
<td><code>a.iteritems()</code></td>
<td>return an iterator over <code>(key, value)</code> pairs</td>
<td>(2), (3)</td>
</tr>
<tr>
<td><code>a.iterkeys()</code></td>
<td>return an iterator over the mapping's keys</td>
<td>(2), (3)</td>
</tr>
<tr>
<td><code>a.itervalues()</code></td>
<td>return an iterator over the mapping's values</td>
<td>(2), (3)</td>
</tr>
</tbody>
</table>

Notes:

(1) Raises a `KeyError` exception if `k` is not in the map.

(2) New in version 2.2.

(3) Keys and values are listed in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions. If `items()`, `keys()`, `values()`, `iteritems()`, `iterkeys()`, and `itervalues()` are called with no intervening modifications to the dictionary, the lists will directly correspond. This allows the creation of `(value, key)` pairs using `zip()`:

```
pairs = zip(a.values(), a.keys())
```

The same relationship holds for the `iterkeys()` and `itervalues()` methods:

```
pairs = zip(a.itervalues(), a.iterkeys())
```

This provides the same value for `pairs`. Another way to create the same list is:

```
pairs = [(v, k) for (k, v) in a.iteritems()]
```
Never raises an exception if $k$ is not in the map, instead it returns $x$. $x$ is optional; when $x$ is not provided and $k$ is not in the map, $\text{None}$ is returned.

`setdefault()` is like `get()`, except that if $k$ is missing, $x$ is both returned and inserted into the dictionary as the value of $k$. $x$ defaults to $\text{None}$.

`popitem()` is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling `popitem()` raises a `KeyError`.

`fromkeys()` is a class method that returns a new dictionary. `value` defaults to $\text{None}$. New in version 2.3.

`pop()` raises a `KeyError` when no default value is given and the key is not found. New in version 2.3.

`update()` accepts either another mapping object or an iterable of key/value pairs (as a tuple or other iterable of length two). If keyword arguments are specified, the mapping is then updated with those key/value pairs: "d.update(red=1, blue=2)". Changed in version 2.4: Allowed the argument to be an iterable of key/value pairs and allowed keyword arguments.

If a subclass of dict defines a method `__missing__`, if the key $k$ is not present, the $a[k]$ operation calls that method with the key $k$ as argument. The $a[k]$ operation then returns or raises whatever is returned or raised by the `__missing__(k)` call if the key is not present. No other operations or methods invoke `__missing__()`. If `__missing__` is not defined, `KeyError` is raised. `__missing__` must be a method; it cannot be an instance variable. For an example, see `collections.defaultdict`. New in version 2.5.
3.9 File Objects

File objects are implemented using C's `stdio` package and can be created with the built-in constructor `file()` described in section 2.1, "Built-in Functions." File objects are also returned by some other built-in functions and methods, such as `os.popen()` and `os.fdopen()` and the `makefile()` method of socket objects.

When a file operation fails for an I/O-related reason, the exception `IOError` is raised. This includes situations where the operation is not defined for some reason, like `seek()` on a tty device or writing a file opened for reading.

Files have the following methods:

**close()**

Close the file. A closed file cannot be read or written any more. Any operation which requires that the file be open will raise a `ValueError` after the file has been closed. Calling `close()` more than once is allowed.

As of Python 2.5, you can avoid having to call this method explicitly if you use the `with` statement. For example, the following code will automatically close `f` when the `with` block is exited:

```python
from __future__ import with_statement

with open("hello.txt") as f:
    for line in f:
        print line
```

In older versions of Python, you would have needed to do this to get the same effect:

```python
f = open("hello.txt")
try:
    for line in f:
        print line
finally:
    f.close()
```

**Note**: Not all "file-like" types in Python support use as a context manager
for the `with` statement. If your code is intended to work with any file-like object, you can use the `closing()` function in the `contextlib` module instead of using the object directly. See section 26.5 for details.

**flush()**
Flush the internal buffer, like `stdio's fflush()`. This may be a no-op on some file-like objects.

**fileno()**
Return the integer ```file descriptor``` that is used by the underlying implementation to request I/O operations from the operating system. This can be useful for other, lower level interfaces that use file descriptors, such as the `fcntl` module or `os.read() and friends`. **Note:** File-like objects which do not have a real file descriptor should not provide this method!

**isatty()**
Return `True` if the file is connected to a tty(-like) device, else `False`. **Note:** If a file-like object is not associated with a real file, this method should *not* be implemented.

**next()**
A file object is its own iterator, for example `iter(f)` returns `f` (unless `f` is closed). When a file is used as an iterator, typically in a `for` loop (for example, `for line in f: print line`), the `next()` method is called repeatedly. This method returns the next input line, or raises `StopIteration` when EOF is hit. In order to make a `for` loop the most efficient way of looping over the lines of a file (a very common operation), the `next()` method uses a hidden read-ahead buffer. As a consequence of using a read-ahead buffer, combining `next()` with other file methods (like `readline()`) does not work right. However, using `seek()` to reposition the file to an absolute position will flush the read-ahead buffer. New in version 2.3.

**read([size])**
Read at most `size` bytes from the file (less if the read hits EOF before obtaining `size` bytes). If the `size` argument is negative or omitted, read all data until EOF is reached. The bytes are returned as a string object. An
empty string is returned when EOF is encountered immediately. (For certain files, like ttys, it makes sense to continue reading after an EOF is hit.) Note that this method may call the underlying C function `fread()` more than once in an effort to acquire as close to `size` bytes as possible. Also note that when in non-blocking mode, less data than what was requested may be returned, even if no `size` parameter was given.

`readline([size])`

Read one entire line from the file. A trailing newline character is kept in the string (but may be absent when a file ends with an incomplete line). If the `size` argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned. An empty string is returned only when EOF is encountered immediately. **Note:** Unlike `stdio`'s `fgets()`, the returned string contains null characters (`'\0'`) if they occurred in the input.

`readlines([sizehint])`

Read until EOF using `readline()` and return a list containing the lines thus read. If the optional `sizehint` argument is present, instead of reading up to EOF, whole lines totalling approximately `sizehint` bytes (possibly after rounding up to an internal buffer size) are read. Objects implementing a file-like interface may choose to ignore `sizehint` if it cannot be implemented, or cannot be implemented efficiently.

`xreadlines()`

This method returns the same thing as `iter(f)`. New in version 2.1. **Deprecated since release 2.3.** Use "for line in file" instead.

`seek(offset[, whence])`

Set the file's current position, like `stdio`'s `fseek()`. The `whence` argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file's end). There is no return value. Note that if the file is opened for appending (mode 'a' or 'a+'), any `seek()` operations will be undone at the next write. If the file is only opened for writing in append mode (mode 'a'), this method is essentially a no-op, but it remains useful for files opened in append mode with reading enabled (mode 'a+'). If the file is
opened in text mode (without 'b'), only offsets returned by `tell()` are legal. Use of other offsets causes undefined behavior.

Note that not all file objects are seekable.

**tell()**
Return the file's current position, like `stdio's ftell()`.

*Note:* On Windows, `tell()` can return illegal values (after an `fgets()`) when reading files with UNIX-style line-endings. Use binary mode ('rb') to circumvent this problem.

**truncate([size])**
Truncate the file's size. If the optional size argument is present, the file is truncated to (at most) that size. The size defaults to the current position. The current file position is not changed. Note that if a specified size exceeds the file's current size, the result is platform-dependent: possibilities include that the file may remain unchanged, increase to the specified size as if zero-filled, or increase to the specified size with undefined new content. Availability: Windows, many UNIX variants.

**write(str)**
Write a string to the file. There is no return value. Due to buffering, the string may not actually show up in the file until the `flush()` or `close()` method is called.

**writelines(sequence)**
Write a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value. (The name is intended to match `readlines()`; `writelines()` does not add line separators.)

Files support the iterator protocol. Each iteration returns the same result as `file.readline()`, and iteration ends when the `readline()` method returns an empty string.

File objects also offer a number of other interesting attributes. These are not required for file-like objects, but should be implemented if they make sense for
the particular object.

**closed**
bool indicating the current state of the file object. This is a read-only attribute; the `close()` method changes the value. It may not be available on all file-like objects.

**encoding**
The encoding that this file uses. When Unicode strings are written to a file, they will be converted to byte strings using this encoding. In addition, when the file is connected to a terminal, the attribute gives the encoding that the terminal is likely to use (that information might be incorrect if the user has misconfigured the terminal). The attribute is read-only and may not be present on all file-like objects. It may also be `None`, in which case the file uses the system default encoding for converting Unicode strings.

New in version 2.3.

**mode**
The I/O mode for the file. If the file was created using the `open()` built-in function, this will be the value of the `mode` parameter. This is a read-only attribute and may not be present on all file-like objects.

**name**
If the file object was created using `open()`, the name of the file. Otherwise, some string that indicates the source of the file object, of the form "<...>". This is a read-only attribute and may not be present on all file-like objects.

**newlines**
If Python was built with the `--with-universal-newlines` option to `configure` (the default) this read-only attribute exists, and for files opened in universal newline read mode it keeps track of the types of newlines encountered while reading the file. The values it can take are '\r', '\n', '\r\n', `None` (unknown, no newlines read yet) or a tuple containing all the newline types seen, to indicate that multiple newline conventions were encountered. For files not opened in universal newline read mode the value of this attribute will be `None`. 
softspace

Boolean that indicates whether a space character needs to be printed before another value when using the print statement. Classes that are trying to simulate a file object should also have a writable softspace attribute, which should be initialized to zero. This will be automatic for most classes implemented in Python (care may be needed for objects that override attribute access); types implemented in C will have to provide a writable softspace attribute. Note: This attribute is not used to control the print statement, but to allow the implementation of print to keep track of its internal state.

Footnotes

... Functions."^3.6

file() is new in Python 2.2. The older built-in open() is an alias for file().

... line).^3.7

The advantage of leaving the newline on is that returning an empty string is then an unambiguous EOF indication. It is also possible (in cases where it might matter, for example, if you want to make an exact copy of a file while scanning its lines) to tell whether the last line of a file ended in a newline or not (yes this happens!).

Python Library Reference

Previous: 3.8 Mapping Types  Up: 3. Built-in Types  Next: 3.10 Context Manager Types

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
3.10 Context Manager Types

New in version 2.5.

Python's `with` statement supports the concept of a runtime context defined by a context manager. This is implemented using two separate methods that allow user-defined classes to define a runtime context that is entered before the statement body is executed and exited when the statement ends.

The context management protocol consists of a pair of methods that need to be provided for a context manager object to define a runtime context:

`__enter__()`

Enter the runtime context and return either this object or another object related to the runtime context. The value returned by this method is bound to the identifier in the `as` clause of `with` statements using this context manager.

An example of a context manager that returns itself is a file object. File objects return themselves from `__enter__()` to allow `open()` to be used as the context expression in a `with` statement.

An example of a context manager that returns a related object is the one returned by `decimal.Context.get_manager()`. These managers set the active decimal context to a copy of the original decimal context and then return the copy. This allows changes to be made to the current decimal context in the body of the `with` statement without affecting code outside the `with` statement.

`__exit__(exc_type, exc_val, exc_tb)`

Exit the runtime context and return a Boolean flag indicating if any exception that occurred should be suppressed. If an exception occurred while executing the body of the `with` statement, the arguments contain the exception type, value and traceback information. Otherwise, all three arguments are `None`. 

The context management protocol consists of a pair of methods that need to be provided for a context manager object to define a runtime context:
Returning a true value from this method will cause the `with` statement to suppress the exception and continue execution with the statement immediately following the `with` statement. Otherwise the exception continues propagating after this method has finished executing. Exceptions that occur during execution of this method will replace any exception that occurred in the body of the `with` statement.

The exception passed in should never be reraised explicitly - instead, this method should return a false value to indicate that the method completed successfully and does not want to suppress the raised exception. This allows context management code (such as `contextlib.nested`) to easily detect whether or not an `__exit__()` method has actually failed.

Python defines several context managers to support easy thread synchronisation, prompt closure of files or other objects, and simpler manipulation of the active decimal arithmetic context. The specific types are not treated specially beyond their implementation of the context management protocol.

Python's generators and the `contextlib.contextfactory` decorator provide a convenient way to implement these protocols. If a generator function is decorated with the `contextlib.contextfactory` decorator, it will return a context manager implementing the necessary `__enter__()` and `__exit__()` methods, rather than the iterator produced by an undecorated generator function.

Note that there is no specific slot for any of these methods in the type structure for Python objects in the Python/C API. Extension types wanting to define these methods must provide them as a normal Python accessible method. Compared to the overhead of setting up the runtime context, the overhead of a single class dictionary lookup is negligible.
3.11 Other Built-in Types

The interpreter supports several other kinds of objects. Most of these support only one or two operations.

---

Subsections

- 3.11.1 Modules
- 3.11.2 Classes and Class Instances
- 3.11.3 Functions
- 3.11.4 Methods
- 3.11.5 Code Objects
- 3.11.6 Type Objects
- 3.11.7 The Null Object
- 3.11.8 The Ellipsis Object
- 3.11.9 Boolean Values
- 3.11.10 Internal Objects

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
3.11.1 Modules

The only special operation on a module is attribute access: \texttt{m.name}, where \texttt{m} is a module and \texttt{name} accesses a name defined in \texttt{m}'s symbol table. Module attributes can be assigned to. (Note that the \texttt{import} statement is not, strictly speaking, an operation on a module object; \texttt{import foo} does not require a module object named \texttt{foo} to exist, rather it requires an (external) \textit{definition} for a module named \texttt{foo} somewhere.)

A special member of every module is \texttt{__dict__}. This is the dictionary containing the module's symbol table. Modifying this dictionary will actually change the module's symbol table, but direct assignment to the \texttt{__dict__} attribute is not possible (you can write \texttt{m.__dict__['a']} = 1, which defines \texttt{m.a} to be 1, but you can't write \texttt{m.__dict__ = {}}). Modifying \texttt{__dict__} directly is not recommended.

Modules built into the interpreter are written like this: \texttt{<module 'sys' (built-in)>}. If loaded from a file, they are written as \texttt{<module 'os' from '/usr/local/lib/python2.5/os.pyc'>}.

\begin{center}
\textbf{Python Library Reference}
\end{center}

\begin{tabular}{c}
\textbf{Previous:} 3.11 Other Built-in Types  \\
\textbf{Up:} 3.11 Other Built-in Types  \\
\textbf{Next:} 3.11.2 Classes and Class
\end{tabular}

\begin{center}
\textit{Release 2.5, documentation updated on 19th September, 2006.  \\
See \textit{About this document...} for information on suggesting changes.}
\end{center}
3.11.2 Classes and Class Instances

See chapters 3 and 7 of the Python Reference Manual for these.
3.11.3 Functions

Function objects are created by function definitions. The only operation on a function object is to call it: \texttt{func(argument-list)}.

There are really two flavors of function objects: built-in functions and user-defined functions. Both support the same operation (to call the function), but the implementation is different, hence the different object types.

See the \textit{Python Reference Manual} for more information.
3.11.4 Methods

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as `append()` on lists) and class instance methods. Built-in methods are described with the types that support them.

The implementation adds two special read-only attributes to class instance methods: `m.im_self` is the object on which the method operates, and `m.im_func` is the function implementing the method. Calling `m(arg-1, arg-2, ..., arg-n)` is completely equivalent to calling `m.im_func(m.im_self, arg-1, arg-2, ..., arg-n)`.

Class instance methods are either bound or unbound, referring to whether the method was accessed through an instance or a class, respectively. When a method is unbound, its `im_self` attribute will be `None` and if called, an explicit `self` object must be passed as the first argument. In this case, `self` must be an instance of the unbound method's class (or a subclass of that class), otherwise a `TypeError` is raised.

Like function objects, methods objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (`meth.im_func`), setting method attributes on either bound or unbound methods is disallowed. Attempting to set a method attribute results in a `TypeError` being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object:

```python
class C:
    def method(self):
        pass

c = C()
c.method.im_func.whoami = 'my name is c'
```

See the Python Reference Manual for more information.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
3.11.5 Code Objects

Code objects are used by the implementation to represent ``pseudo-compiled'' executable Python code such as a function body. They differ from function objects because they don't contain a reference to their global execution environment. Code objects are returned by the built-in compile() function and can be extracted from function objects through their func_code attribute.

A code object can be executed or evaluated by passing it (instead of a source string) to the exec statement or the built-in eval() function.

See the Python Reference Manual for more information.
3.11.6 Type Objects

Type objects represent the various object types. An object's type is accessed by the built-in function `type()`. There are no special operations on types. The standard module `types` defines names for all standard built-in types.

Types are written like this: `<type 'int'>`. 
3.11.7 The Null Object

This object is returned by functions that don't explicitly return a value. It supports no special operations. There is exactly one null object, named None (a built-in name).

It is written as None.
3.11.8 The Ellipsis Object

This object is used by extended slice notation (see the Python Reference Manual). It supports no special operations. There is exactly one ellipsis object, named Ellipsis (a built-in name).

It is written as Ellipsis.
3.11.9 Boolean Values

Boolean values are the two constant objects False and True. They are used to represent truth values (although other values can also be considered false or true). In numeric contexts (for example when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively. The built-in function bool() can be used to cast any value to a Boolean, if the value can be interpreted as a truth value (see section Truth Value Testing above).

They are written as False and True, respectively.
3.11.10 Internal Objects

See the *Python Reference Manual* for this information. It describes stack frame objects, traceback objects, and slice objects.
3.12 Special Attributes

The implementation adds a few special read-only attributes to several object types, where they are relevant. Some of these are not reported by the `dir()` built-in function.

___dict___
A dictionary or other mapping object used to store an object's (writable) attributes.

___methods___
**Deprecated since release 2.2.** Use the built-in function `dir()` to get a list of an object's attributes. This attribute is no longer available.

___members___
**Deprecated since release 2.2.** Use the built-in function `dir()` to get a list of an object's attributes. This attribute is no longer available.

___class___
The class to which a class instance belongs.

___bases___
The tuple of base classes of a class object. If there are no base classes, this will be an empty tuple.

___name___
The name of the class or type.
string
4. String Services

The modules described in this chapter provide a wide range of string manipulation operations. Here's an overview:

- **string**
  - Common string operations.
- **re**
  - Regular expression search and match operations with a Perl-style expression syntax.
- **struct**
  - Interpret strings as packed binary data.
- **difflib**
  - Helpers for computing differences between objects.
- **StringIO**
  - Read and write strings as if they were files.
- **cStringIO**
  - Faster version of **StringIO**, but not subclassable.
- **textwrap**
  - Text wrapping and filling
- **codecs**
  - Encode and decode data and streams.
- **encodings.idna**
  - Internationalized Domain Names implementation
- **encodings.utf-8-sig**
  - UTF-8 codec with BOM signature
- **unicodedata**
  - Access the Unicode Database.
- **stringprep**
  - String preparation, as per RFC 3453
- **fpformat**
  - General floating point formatting functions.

Information on the methods of string objects can be found in section 3.6.1, "String Methods."
4.1 string -- Common string operations

The string module contains a number of useful constants and classes, as well as some deprecated legacy functions that are also available as methods on strings. See the module `re` for string functions based on regular expressions.

Subsections

- 4.1.1 String constants
- 4.1.2 Template strings
- 4.1.3 String functions
- 4.1.4 Deprecated string functions

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
4.1.1 String constants

The constants defined in this module are:

**ascii_letters**
The concatenation of the `ascii_lowercase` and `ascii_uppercase` constants described below. This value is not locale-dependent.

**ascii_lowercase**
The lowercase letters `'abcdefghijklmnopqrstuvwxyz'`. This value is not locale-dependent and will not change.

**ascii_uppercase**
The uppercase letters `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. This value is not locale-dependent and will not change.

**digits**
The string `'0123456789'`.

**hexdigits**
The string `'0123456789abcdefABCDEF'`.

**letters**
The concatenation of the strings lowercase and uppercase described below. The specific value is locale-dependent, and will be updated when `locale.setlocale()` is called.

**lowercase**
A string containing all the characters that are considered lowercase letters. On most systems this is the string `'abcdefghijklmnopqrstuvwxyz'`. Do not change its definition -- the effect on the routines `upper()` and `swapcase()` is undefined. The specific value is locale-dependent, and will be updated when `locale.setlocale()` is called.

**octdigits**
The string `'01234567'`. 
**punctuation**
String of ASCII characters which are considered punctuation characters in the "C" locale.

**printable**
String of characters which are considered printable. This is a combination of digits, letters, punctuation, and whitespace.

**uppercase**
A string containing all the characters that are considered uppercase letters. On most systems this is the string 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. Do not change its definition -- the effect on the routines `lower()` and `swapcase()` is undefined. The specific value is locale-dependent, and will be updated when `locale.setlocale()` is called.

**whitespace**
A string containing all characters that are considered whitespace. On most systems this includes the characters space, tab, linefeed, return, formfeed, and vertical tab. Do not change its definition -- the effect on the routines `strip()` and `split()` is undefined.
4.1.2 Template strings

Templates provide simpler string substitutions as described in PEP 292. Instead of the normal "%"-based substitutions, Templates support "$"-based substitutions, using the following rules:

- "$$" is an escape; it is replaced with a single "$".

- "$identifier" names a substitution placeholder matching a mapping key of "identifier". By default, "identifier" must spell a Python identifier. The first non-identifier character after the "$" character terminates this placeholder specification.

- "${identifier}" is equivalent to "$identifier". It is required when valid identifier characters follow the placeholder but are not part of the placeholder, such as "${noun}ification".

Any other appearance of "$" in the string will result in a ValueError being raised.

New in version 2.4.

The string module provides a Template class that implements these rules. The methods of Template are:

class Template(template)

The constructor takes a single argument which is the template string.

substitute(mapping[, **kws])

Performs the template substitution, returning a new string. mapping is any dictionary-like object with keys that match the placeholders in the template. Alternatively, you can provide keyword arguments, where the keywords are the placeholders. When both mapping and kws are given and there are duplicates, the placeholders from kws take precedence.

safe_substitute(mapping[, **kws])

Like substitute(), except that if placeholders are missing from
mapping and kws, instead of raising a KeyError exception, the original placeholder will appear in the resulting string intact. Also, unlike with substitute(), any other appearances of the "$" will simply return "$" instead of raising ValueError.

While other exceptions may still occur, this method is called ``safe'' because substitutions always tries to return a usable string instead of raising an exception. In another sense, safe_substitute() may be anything other than safe, since it will silently ignore malformed templates containing dangling delimiters, unmatched braces, or placeholders that are not valid Python identifiers.

Template instances also provide one public data attribute:

**template**

This is the object passed to the constructor's template argument. In general, you shouldn't change it, but read-only access is not enforced.

Here is an example of how to use a Template:

```python
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
[...]
ValueError: Invalid placeholder in string: line 1, col 10
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
[...]
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

Advanced usage: you can derive subclasses of Template to customize the placeholder syntax, delimiter character, or the entire regular expression used to parse template strings. To do this, you can override these class attributes:

- **delimiter** - This is the literal string describing a placeholder introducing delimiter. The default value "$". Note that this should *not* be a regular expression, as the implementation will call re.escape() on this string.
as needed.

- **idpattern** - This is the regular expression describing the pattern for non-braced placeholders (the braces will be added automatically as appropriate). The default value is the regular expression "[\_a - z][\_a - z\0 - 9] *".

Alternatively, you can provide the entire regular expression pattern by overriding the class attribute `pattern`. If you do this, the value must be a regular expression object with four named capturing groups. The capturing groups correspond to the rules given above, along with the invalid placeholder rule:

- **escaped** - This group matches the escape sequence, e.g. "\$$", in the default pattern.
- **named** - This group matches the unbraced placeholder name; it should not include the delimiter in capturing group.
- **braced** - This group matches the brace enclosed placeholder name; it should not include either the delimiter or braces in the capturing group.
- **invalid** - This group matches any other delimiter pattern (usually a single delimiter), and it should appear last in the regular expression.
4.1.3 String functions

The following functions are available to operate on string and Unicode objects. They are not available as string methods.

**capwords(s)**
Split the argument into words using split(), capitalize each word using capitalize(), and join the capitalized words using join(). Note that this replaces runs of whitespace characters by a single space, and removes leading and trailing whitespace.

**maketrans(from, to)**
Return a translation table suitable for passing to translate() or regex.compile(), that will map each character in from into the character at the same position in to; from and to must have the same length.

**Warning:** Don't use strings derived from lowercase and uppercase as arguments; in some locales, these don't have the same length. For case conversions, always use lower() and upper().
4.1.4 Deprecated string functions

The following list of functions are also defined as methods of string and Unicode objects; see "String Methods" (section 3.6.1) for more information on those. You should consider these functions as deprecated, although they will not be removed until Python 3.0. The functions defined in this module are:

**atof(s)**

*Deprecated since release 2.0.* Use the `float()` built-in function.

Convert a string to a floating point number. The string must have the standard syntax for a floating point literal in Python, optionally preceded by a sign ("+" or "-"). Note that this behaves identical to the built-in function `float()` when passed a string.

**Note:** When passing in a string, values for NaN and Infinity may be returned, depending on the underlying C library. The specific set of strings accepted which cause these values to be returned depends entirely on the C library and is known to vary.

**atoi(s[, base])**

*Deprecated since release 2.0.* Use the `int()` built-in function.

Convert string `s` to an integer in the given `base`. The string must consist of one or more digits, optionally preceded by a sign ("+" or "-"). The `base` defaults to 10. If it is 0, a default base is chosen depending on the leading characters of the string (after stripping the sign): "0x" or "0X" means 16, "0" means 8, anything else means 10. If `base` is 16, a leading "0x" or "0X" is always accepted, though not required. This behaves identically to the built-in function `int()` when passed a string. (Also note: for a more flexible interpretation of numeric literals, use the built-in function `eval()`.)

**atol(s[, base])**

*Deprecated since release 2.0.* Use the `long()` built-in function.
Convert string \( s \) to a long integer in the given \( \textit{base} \). The string must consist of one or more digits, optionally preceded by a sign ("+" or "-"), The \( \textit{base} \) argument has the same meaning as for \( \texttt{atoi()} \). A trailing "l" or "L" is not allowed, except if the base is 0. Note that when invoked without \( \textit{base} \) or with \( \textit{base} \) set to 10, this behaves identical to the built-in function \( \texttt{long()} \) when passed a string.

**capitalize** (\( \textit{word} \))

Return a copy of \( \textit{word} \) with only its first character capitalized.

**expandtabs** (\( s[, \textit{tabsize}] \))

Expand tabs in a string replacing them by one or more spaces, depending on the current column and the given tab size. The column number is reset to zero after each newline occurring in the string. This doesn't understand other non-printing characters or escape sequences. The tab size defaults to 8.

**find** (\( s, \textit{sub}[,, \textit{start}][,\textit{end}] \))

Return the lowest index in \( s \) where the substring \( \textit{sub} \) is found such that \( \textit{sub} \) is wholly contained in \( s[\textit{start}:	extit{end}] \). Return -1 on failure. Defaults for \( \textit{start} \) and \( \textit{end} \) and interpretation of negative values is the same as for slices.

**rfind** (\( s, \textit{sub}[,, \textit{start}][,\textit{end}] \))

Like **find()** but find the highest index.

**index** (\( s, \textit{sub}[,, \textit{start}][,\textit{end}] \))

Like **find()** but raise **ValueError** when the substring is not found.

**rindex** (\( s, \textit{sub}[,, \textit{start}][,\textit{end}] \))

Like **rfind()** but raise **ValueError** when the substring is not found.

**count** (\( s, \textit{sub}[,, \textit{start}][,\textit{end}] \))

Return the number of (non-overlapping) occurrences of substring \( \textit{sub} \) in string \( s[\textit{start}:	extit{end}] \). Defaults for \( \textit{start} \) and \( \textit{end} \) and interpretation of negative values are the same as for slices.
lower(s)
Return a copy of s, but with upper case letters converted to lower case.

split(s[, sep[, maxsplit]])
Return a list of the words of the string s. If the optional second argument sep is absent or None, the words are separated by arbitrary strings of whitespace characters (space, tab, newline, return, formfeed). If the second argument sep is present and not None, it specifies a string to be used as the word separator. The returned list will then have one more item than the number of non-overlapping occurrences of the separator in the string. The optional third argument maxsplit defaults to 0. If it is nonzero, at most maxsplit number of splits occur, and the remainder of the string is returned as the final element of the list (thus, the list will have at most maxsplit+1 elements).

The behavior of split on an empty string depends on the value of sep. If sep is not specified, or specified as None, the result will be an empty list. If sep is specified as any string, the result will be a list containing one element which is an empty string.

rsplit(s[, sep[, maxsplit]])
Return a list of the words of the string s, scanning s from the end. To all intents and purposes, the resulting list of words is the same as returned by split(), except when the optional third argument maxsplit is explicitly specified and nonzero. When maxsplit is nonzero, at most maxsplit number of splits - the rightmost ones - occur, and the remainder of the string is returned as the first element of the list (thus, the list will have at most maxsplit+1 elements). New in version 2.4.

splitfields(s[, sep[, maxsplit]])
This function behaves identically to split(). (In the past, split() was only used with one argument, while splitfields() was only used with two arguments.)

join(words[, sep])
Concatenate a list or tuple of words with intervening occurrences of sep. The default value for sep is a single space character. It is always true that
"string.join(string.split(s, sep), sep)" equals s.

joinfields( words[, sep])
This function behaves identically to join(). (In the past, join() was only used with one argument, while joinfields() was only used with two arguments.) Note that there is no joinfields() method on string objects; use the join() method instead.

lstrip( s[, chars])
Return a copy of the string with leading characters removed. If chars is omitted or None, whitespace characters are removed. If given and not None, chars must be a string; the characters in the string will be stripped from the beginning of the string this method is called on. Changed in version 2.2.3: The chars parameter was added. The chars parameter cannot be passed in earlier 2.2 versions.

rstrip( s[, chars])
Return a copy of the string with trailing characters removed. If chars is omitted or None, whitespace characters are removed. If given and not None, chars must be a string; the characters in the string will be stripped from the end of the string this method is called on. Changed in version 2.2.3: The chars parameter was added. The chars parameter cannot be passed in earlier 2.2 versions.

strip( s[, chars])
Return a copy of the string with leading and trailing characters removed. If chars is omitted or None, whitespace characters are removed. If given and not None, chars must be a string; the characters in the string will be stripped from the both ends of the string this method is called on. Changed in version 2.2.3: The chars parameter was added. The chars parameter cannot be passed in earlier 2.2 versions.

swapcase( s)
Return a copy of s, but with lower case letters converted to upper case and vice versa.

translate( s, table[, deletechars])
Delete all characters from s that are in deletechars (if present), and then translate the characters using table, which must be a 256-character string giving the translation for each character value, indexed by its ordinal.

**upper**(*s*)

Return a copy of s, but with lower case letters converted to upper case.

**ljust**(*s*, *width*)
**rjust**(*s*, *width*)
**center**(*s*, *width*)

These functions respectively left-justify, right-justify and center a string in a field of given width. They return a string that is at least width characters wide, created by padding the string s with spaces until the given width on the right, left or both sides. The string is never truncated.

**zfill**(*s*, *width*)

Pad a numeric string on the left with zero digits until the given width is reached. Strings starting with a sign are handled correctly.

**replace** *(str, old, new[, maxreplace]*)

Return a copy of string str with all occurrences of substring old replaced by new. If the optional argument maxreplace is given, the first maxreplace occurrences are replaced.
This module provides regular expression matching operations similar to those found in Perl. Regular expression pattern strings may not contain null bytes, but can specify the null byte using the \number notation. Both patterns and strings to be searched can be Unicode strings as well as 8-bit strings. The re module is always available.

Regular expressions use the backslash character ("\") to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python's usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write '\\\' as the pattern string, because the regular expression must be "\\", and each backslash must be expressed as "\\" inside a regular Python string literal.

The solution is to use Python's raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with "r". So r"\n" is a two-character string containing "\" and "\n", while "\n" is a one-character string containing a newline. Usually patterns will be expressed in Python code using this raw string notation.

**See Also:**

*Mastering Regular Expressions*

Book on regular expressions by Jeffrey Friedl, published by O'Reilly. The second edition of the book no longer covers Python at all, but the first edition covered writing good regular expression patterns in great detail.

---

**Subsections**

- [4.2.1 Regular Expression Syntax](#)
4.2.2 Matching vs Searching
4.2.3 Module Contents
4.2.4 Regular Expression Objects
4.2.5 Match Objects
4.2.6 Examples

Python Library Reference

Previous: 4.1.4 Deprecated string functions  Up: 4. String Services  Next: 4.2.1

Regular Expression Syntax

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.2.1 Regular Expression Syntax

A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

Regular expressions can be concatenated to form new regular expressions; if A and B are both regular expressions, then AB is also a regular expression. In general, if a string p matches A and another string q matches B, the string pq will match AB. This holds unless A or B contain low precedence operations; boundary conditions between A and B; or have numbered group references. Thus, complex expressions can easily be constructed from simpler primitive expressions like the ones described here. For details of the theory and implementation of regular expressions, consult the Friedl book referenced above, or almost any textbook about compiler construction.

A brief explanation of the format of regular expressions follows. For further information and a gentler presentation, consult the Regular Expression HOWTO, accessible from http://www.python.org/doc/howto/.

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like "A", "a", or "0", are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so last matches the string 'last'. (In the rest of this section, we'll write RE's in this special style, usually without quotes, and strings to be matched 'in single quotes'.)

Some characters, like " |" or "(" , are special. Special characters either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted.

The special characters are:

"." (Dot.) In the default mode, this matches any character except a newline. If the DOTALL flag has been specified, this matches any character including a newline.
"^" (Caret.) Matches the start of the string, and in MULTILINE mode also matches immediately after each newline.

"$" Matches the end of the string or just before the newline at the end of the string, and in MULTILINE mode also matches before a newline. foo matches both 'foo' and 'foobar', while the regular expression foo$ matches only 'foo'. More interestingly, searching for foo.$ in 'foo1
foo2' matches 'foo2' normally, but 'foo1' in MULTILINE mode.

"*" Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. ab* will match 'a', 'ab', or 'a' followed by any number of 'b's.

"+" Causes the resulting RE to match 1 or more repetitions of the preceding RE. ab+ will match 'a' followed by any non-zero number of 'b's; it will not match just 'a'.

"?" Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. ab? will match either 'a' or 'ab'.

*?, +?, ?? The "*", "+", and "?" qualifiers are all greedy; they match as much text as possible. Sometimes this behaviour isn't desired; if the RE <.*> is matched against '<H1>title</H1>', it will match the entire string, and not just '<H1>'. Adding '?' after the qualifier makes it perform the match in non-greedy or minimal fashion; as few characters as possible will be matched. Using .*? in the previous expression will match only '<H1>'.

{m} Specifies that exactly m copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, a{6} will match exactly six "a" characters, but not five.

{m, n}
Causes the resulting RE to match from $m$ to $n$ repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, $a\{3, 5\}$ will match from 3 to 5 "a" characters. Omitting $m$ specifies a lower bound of zero, and omitting $n$ specifies an infinite upper bound. As an example, $a\{4, \}$b will match $aaaab$ or a thousand "a" characters followed by a b, but not $aaab$. The comma may not be omitted or the modifier would be confused with the previously described form.

$\{m, n\}$?
Causes the resulting RE to match from $m$ to $n$ repetitions of the preceding RE, attempting to match as few repetitions as possible. This is the non-greedy version of the previous qualifier. For example, on the 6-character string 'aaaaaa', $a\{3, 5\}$ will match 5 "a" characters, while $a\{3, 5\}?$ will only match 3 characters.

"\"
Either escapes special characters (permitting you to match characters like "*", "?", and so forth), or signals a special sequence; special sequences are discussed below.

If you're not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn't recognized by Python's parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This is complicated and hard to understand, so it's highly recommended that you use raw strings for all but the simplest expressions.

[]
Used to indicate a set of characters. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a " - ". Special characters are not active inside sets. For example, [akm$] will match any of the characters "a", "k", "m", or "$"; [a-z] will match any lowercase letter, and [a-zA-Z0-9] matches any letter or digit. Character classes such as \w or \S (defined below) are also acceptable inside a range. If you want to include a "]" or a "-" inside a set, precede it with a backslash, or place it as the first character. The pattern [[]] will match ' [] ', for example.
You can match the characters not within a range by complementing the set. This is indicated by including a "^" as the first character of the set; "^" elsewhere will simply match the "^" character. For example, [^5] will match any character except "5", and [^\^] will match any character except "^".

" | "
A | B, where A and B can be arbitrary REs, creates a regular expression that will match either A or B. An arbitrary number of REs can be separated by the " | " in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by " | " are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once A matches, B will not be tested further, even if it would produce a longer overall match. In other words, the " | " operator is never greedy. To match a literal " | ", use \ | , or enclose it inside a character class, as in [ | ].

(...) Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the \number special sequence, described below. To match the literals "(" or ")", use \( or \), or enclose them inside a character class: [[[ ]]].

(?....)
This is an extension notation (a "?" following a "(") is not meaningful otherwise). The first character after the "?" determines what the meaning and further syntax of the construct is. Extensions usually do not create a new group; (?P<name>. . .) is the only exception to this rule. Following are the currently supported extensions.

(?ilmsux)
(One or more letters from the set "i", "L", "m", "s", "u", "x".) The group matches the empty string; the letters set the corresponding flags (re.I, re.L, re.M, re.S, re.U, re.X) for the entire regular expression. This is useful if you wish to include the flags as part of the regular expression, instead of passing a flag argument to the compile() function.
Note that the (\?x) flag changes how the expression is parsed. It should be used first in the expression string, or after one or more whitespace characters. If there are non-whitespace characters before the flag, the results are undefined.

(\?:\ldots)
A non-grouping version of regular parentheses. Matches whatever regular expression is inside the parentheses, but the substring matched by the group cannot be retrieved after performing a match or referenced later in the pattern.

(\?P<name>\ldots)
Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name name. Group names must be valid Python identifiers, and each group name must be defined only once within a regular expression. A symbolic group is also a numbered group, just as if the group were not named. So the group named 'id' in the example above can also be referenced as the numbered group 1.

For example, if the pattern is (?P<id>[a-zA-Z_\w\*]), the group can be referenced by its name in arguments to methods of match objects, such as m.group('id') or m.end('id'), and also by name in pattern text (for example, (?P=id)) and replacement text (such as \g<id>).

(\?P=name)
Matches whatever text was matched by the earlier group named name.

(\?#\ldots)
A comment; the contents of the parentheses are simply ignored.

(\?=\ldots)
Matches if ... matches next, but doesn't consume any of the string. This is called a lookahead assertion. For example, Isaac (?=Asimov) will match 'Isaac ' only if it's followed by 'Asimov'.

(\?!\ldots)
Matches if ... doesn't match next. This is a negative lookahead assertion. For example, Isaac (?!Asimov) will match 'Isaac ' only if it's not followed by 'Asimov'.
(?:<=...)  
Matches if the current position in the string is preceded by a match for . . . that ends at the current position. This is called a positive lookbehind assertion. (?<=abc)def will find a match in "abcdef", since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that abc or a|b are allowed, but a* and a{3,4} are not. Note that patterns which start with positive lookbehind assertions will never match at the beginning of the string being searched; you will most likely want to use the search() function rather than the match() function:

```python
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

This example looks for a word following a hyphen:

```python
>>> m = re.search('(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

(?:<!...)  
Matches if the current position in the string is not preceded by a match for . . . This is called a negative lookbehind assertion. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length. Patterns which start with negative lookbehind assertions may match at the beginning of the string being searched.

(?:<id/name>yese-pattern|no-pattern)  
Will try to match with yes-pattern if the group with given id or name exists, and with no-pattern if it doesn't. |no-pattern is optional and can be omitted. For example, (<)?(\w+@\w+(?:\.:\w+)?)((1)>)
is a poor email matching pattern, which will match with '

The special sequences consist of "\" and a character from the list below. If the ordinary character is not on the list, then the resulting RE will match the second character. For example, \$ matches the character "$". 
\textbf{\texttt{number}}

Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, ( . + ) \texttt{\textbackslash 1} matches 'the the' or '55 55', but not 'the end' (note the space after the group). This special sequence can only be used to match one of the first 99 groups. If the first digit of \texttt{number} is 0, or \texttt{number} is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value \texttt{number}. Inside the "[ " and "] " of a character class, all numeric escapes are treated as characters.

\textbf{\texttt{\textbackslash A}}

Matches only at the start of the string.

\textbf{\texttt{\textbackslash b}}

Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of alphanumeric or underscore characters, so the end of a word is indicated by whitespace or a non-alphanumeric, non-underscore character. Note that \texttt{\textbackslash b} is defined as the boundary between \texttt{\textbackslash w} and \texttt{\textbackslash W}, so the precise set of characters deemed to be alphanumeric depends on the values of the UNICODE and LOCALE flags. Inside a character range, \texttt{\textbackslash b} represents the backspace character, for compatibility with Python's string literals.

\textbf{\texttt{\textbackslash B}}

Matches the empty string, but only when it is not at the beginning or end of a word. This is just the opposite of \texttt{\textbackslash b}, so is also subject to the settings of LOCALE and UNICODE.

\textbf{\texttt{\textbackslash d}}

When the UNICODE flag is not specified, matches any decimal digit; this is equivalent to the set \texttt{[0-9]}. With UNICODE, it will match whatever is classified as a digit in the Unicode character properties database.

\textbf{\texttt{\textbackslash D}}

When the UNICODE flag is not specified, matches any non-digit character; this is equivalent to the set \texttt{[^0-9]}. With UNICODE, it will match anything other than character marked as digits in the Unicode character properties database.
When the `LOCALE` and `UNICODE` flags are not specified, matches any whitespace character; this is equivalent to the set `[ \t\n\r\f\v]`. With `LOCALE`, it will match this set plus whatever characters are defined as space for the current locale. If `UNICODE` is set, this will match the characters `[ \t\n\r\f\v]` plus whatever is classified as space in the Unicode character properties database.

When the `LOCALE` and `UNICODE` flags are not specified, matches any non-whitespace character; this is equivalent to the set `[^[ \t\n\r\f\v]]`. With `LOCALE`, it will match any character not in this set, and not defined as space in the current locale. If `UNICODE` is set, this will match anything other than `[ \t\n\r\f\v]` and characters marked as space in the Unicode character properties database.

When the `LOCALE` and `UNICODE` flags are not specified, matches any alphanumeric character and the underscore; this is equivalent to the set `[a-zA-Z0-9_]`. With `LOCALE`, it will match the set `[θ-9_]` plus whatever characters are defined as alphanumeric for the current locale. If `UNICODE` is set, this will match the characters `[θ-9_]` plus whatever is classified as alphanumeric in the Unicode character properties database.

When the `LOCALE` and `UNICODE` flags are not specified, matches any non-alphanumeric character; this is equivalent to the set `[^[a-zA-Z0-9_]]. With `LOCALE`, it will match any character not in the set `[θ-9_]`, and not defined as alphanumeric for the current locale. If `UNICODE` is set, this will match anything other than `[θ-9_]` and characters marked as alphanumeric in the Unicode character properties database.

Matches only at the end of the string.

Most of the standard escapes supported by Python string literals are also accepted by the regular expression parser:

```
\a \b \f \n
```
Octal escapes are included in a limited form: If the first digit is a 0, or if there are three octal digits, it is considered an octal escape. Otherwise, it is a group reference. As for string literals, octal escapes are always at most three digits in length.
4.2.2 Matching vs Searching

Python offers two different primitive operations based on regular expressions: match and search. If you are accustomed to Perl's semantics, the search operation is what you're looking for. See the search() function and corresponding method of compiled regular expression objects.

Note that match may differ from search using a regular expression beginning with "^": "^" matches only at the start of the string, or in MULTILINE mode also immediately following a newline. The `match` operation succeeds only if the pattern matches at the start of the string regardless of mode, or at the starting position given by the optional pos argument regardless of whether a newline precedes it.

```
re.compile("a").match("ba", 1)                         # succeeds
re.compile("^a").search("ba", 1)                      # fails; 'a' not at start
re.compile("^a").search("\na", 1)                    # fails; 'a' not at start
re.compile("^a", re.M).search("\na", 1)              # succeeds
re.compile("^a", re.M).search("ba", 1)               # fails; no preceding 
```

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.2.3 Module Contents

The module defines several functions, constants, and an exception. Some of the functions are simplified versions of the full featured methods for compiled regular expressions. Most non-trivial applications always use the compiled form.

**compile**(pattern[, flags])

Compile a regular expression pattern into a regular expression object, which can be used for matching using its match() and search() methods, described below.

The expression's behaviour can be modified by specifying a flags value. Values can be any of the following variables, combined using bitwise OR (the | operator).

The sequence

```
prog = re.compile(pat)
result = prog.match(str)
```

is equivalent to

```
result = re.match(pat, str)
```

but the version using compile() is more efficient when the expression will be used several times in a single program.

**I**

**IGNORECASE**

Perform case-insensitive matching; expressions like [A-Z] will match lowercase letters, too. This is not affected by the current locale.

**L**

**LOCALSE**

Make \w, \W, \b, \B, \s and \S dependent on the current locale.

**M**

**MULTILINE**

When specified, the pattern character "^" matches at the beginning of the
string and at the beginning of each line (immediately following each newline); and the pattern character "$" matches at the end of the string and at the end of each line (immediately preceding each newline). By default, "^" matches only at the beginning of the string, and "$" only at the end of the string and immediately before the newline (if any) at the end of the string.

S
DOTALL
Make the "." special character match any character at all, including a newline; without this flag, "." will match anything except a newline.

U
UNICODE
Make \w, \W, \b, \B, \d, \D, \s and \S dependent on the Unicode character properties database. New in version 2.0.

X
VERBOSE
This flag allows you to write regular expressions that look nicer. Whitespace within the pattern is ignored, except when in a character class or preceded by an unescaped backslash, and, when a line contains a "#" neither in a character class or preceded by an unescaped backslash, all characters from the leftmost such "#" through the end of the line are ignored.

search(pattern, string[, flags])
Scan through string looking for a location where the regular expression pattern produces a match, and return a corresponding MatchObject instance. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

match(pattern, string[, flags])
If zero or more characters at the beginning of string match the regular expression pattern, return a corresponding MatchObject instance. Return None if the string does not match the pattern; note that this is different from a zero-length match.
**Note:** If you want to locate a match anywhere in `string`, use `search()` instead.

**split( pattern, string[, maxsplit = 0])**

Split `string` by the occurrences of `pattern`. If capturing parentheses are used in `pattern`, then the text of all groups in the pattern are also returned as part of the resulting list. If `maxsplit` is nonzero, at most `maxsplit` splits occur, and the remainder of the string is returned as the final element of the list. (Incompatibility note: in the original Python 1.5 release, `maxsplit` was ignored. This has been fixed in later releases.)

```python
>>> re.split('%W+', 'Words, words, words.')
['Words', 'words', 'words', '']
```

```python
>>> re.split('%(\W+)\%', 'Words, words, words.')
['Words', ',', ',', 'words', ',', ',', 'words', ',', ',', '']
```

```python
>>> re.split('%W+', 'Words, words, words.', 1)
['Words', 'words, words. ']
```

**findall( pattern, string[, flags])**

Return a list of all non-overlapping matches of `pattern` in `string`. If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group. Empty matches are included in the result unless they touch the beginning of another match. New in version 1.5.2. Changed in version 2.4: Added the optional flags argument.

**finditer( pattern, string[, flags])**

Return an iterator over all non-overlapping matches for the RE `pattern` in `string`. For each match, the iterator returns a match object. Empty matches are included in the result unless they touch the beginning of another match. New in version 2.2. Changed in version 2.4: Added the optional flags argument.

**sub( pattern, repl, string[, count])**

Return the string obtained by replacing the leftmost non-overlapping occurrences of `pattern` in `string` by the replacement `repl`. If the pattern isn't found, `string` is returned unchanged. `repl` can be a string or a function; if it is a string, any backslash escapes in it are processed. That is, "\n" is converted to a single newline character, "\r" is converted to a linefeed, and
so forth. Unknown escapes such as "\j" are left alone. Backreferences, such as "\6", are replaced with the substring matched by group 6 in the pattern. For example:

```python
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
    ...      r'static PyObject*\npy_\1(void)\n{',
    ...      'def myfunc():
    ...    'static PyObject*\npy_myfunc(void)\n{
```

If `repl` is a function, it is called for every non-overlapping occurrence of `pattern`. The function takes a single match object argument, and returns the replacement string. For example:

```python
>>> def dashrepl(matchobj):
    ...      if matchobj.group(0) == '-':
    ...         return ' '
    ...      else:
    ...         return '-'

>>> re.sub('-\{1,2}\', dashrepl, 'pro----gram-files')
'pro--gram files'
```

The pattern may be a string or an RE object; if you need to specify regular expression flags, you must use a RE object, or use embedded modifiers in a pattern; for example, "sub("(?i)b+", "x", "bbbb BBBBB")" returns 'x  x'.

The optional argument `count` is the maximum number of pattern occurrences to be replaced; `count` must be a non-negative integer. If omitted or zero, all occurrences will be replaced. Empty matches for the pattern are replaced only when not adjacent to a previous match, so "sub('x*', ' - ', 'abc')" returns '-a-b-c-'.

In addition to character escapes and backreferences as described above, "\g<name>" will use the substring matched by the group named "name", as defined by the (?P<name>...) syntax. "\g<number>" uses the corresponding group number; "\g<2>" is therefore equivalent to "\2", but isn't ambiguous in a replacement such as "\g<2>0". "\20" would be interpreted as a reference to group 20, not a reference to group 2 followed by the literal character "0". The backreference "\g<@>" substitutes in the entire substring matched by the RE.

```python
subn(pattern, repl, string[, count])
```
Perform the same operation as `sub()`, but return a tuple (new_string,
\( \text{number_of_subs_made} \).

**escape**\((\text{string})\)

Return \text{string} with all non-alphanumerics backslashed; this is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it.

**exception error**

Exception raised when a string passed to one of the functions here is not a valid regular expression (for example, it might contain unmatched parentheses) or when some other error occurs during compilation or matching. It is never an error if a string contains no match for a pattern.
**4.2.4 Regular Expression Objects**

Compiled regular expression objects support the following methods and attributes:

```python
match(string[, pos[, endpos]])
```

If zero or more characters at the beginning of `string` match this regular expression, return a corresponding `MatchObject` instance. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

**Note:** If you want to locate a match anywhere in `string`, use `search()` instead.

The optional second parameter `pos` gives an index in the string where the search is to start; it defaults to `0`. This is not completely equivalent to slicing the string; the `'\^'` pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.

The optional parameter `endpos` limits how far the string will be searched; it will be as if the string is `endpos` characters long, so only the characters from `pos` to `endpos` - 1 will be searched for a match. If `endpos` is less than `pos`, no match will be found, otherwise, if `rx` is a compiled regular expression object, `rx.match(string, 0, 50)` is equivalent to `rx.match(string[:50], 0)`.

```python
search(string[, pos[, endpos]])
```

Scan through `string` looking for a location where this regular expression produces a match, and return a corresponding `MatchObject` instance. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

The optional `pos` and `endpos` parameters have the same meaning as for the `match()` method.

```python
split()
```
Identical to the `split()` function, using the compiled pattern.

### `findall(string[, pos[, endpos]])`
Identical to the `findall()` function, using the compiled pattern.

### `finditer(string[, pos[, endpos]])`
Identical to the `finditer()` function, using the compiled pattern.

### `sub(repl, string[, count = 0])`
Identical to the `sub()` function, using the compiled pattern.

### `subn(repl, string[, count = 0])`
Identical to the `subn()` function, using the compiled pattern.

### `flags`
The flags argument used when the RE object was compiled, or 0 if no flags were provided.

### `groupindex`
A dictionary mapping any symbolic group names defined by `(?P<id>)` to group numbers. The dictionary is empty if no symbolic groups were used in the pattern.

### `pattern`
The pattern string from which the RE object was compiled.
4.2.5 Match Objects

MatchObject instances support the following methods and attributes:

**expand** *(template)*
Return the string obtained by doing backslash substitution on the template string *template*, as done by the `sub()` method. Escapes such as "\n" are converted to the appropriate characters, and numeric backreferences ("\1", "\2") and named backreferences ("\g<1>", "\g<name>") are replaced by the contents of the corresponding group.

**group** *( [group1, ...])*
Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, `group1` defaults to zero (the whole match is returned). If a `groupN` argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an `IndexError` exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is `None`. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

If the regular expression uses the (?P<name>...) syntax, the `groupN` arguments may also be strings identifying groups by their group name. If a string argument is not used as a group name in the pattern, an `IndexError` exception is raised.

A moderately complicated example:

```python
m = re.match(r"(?P<int>\d+).\d*", '3.14')
```

After performing this match, `m.group(1)` is '3', as is `m.group('int')`, and `m.group(2)` is '14'.

**groups** *( [default])*

None
Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The *default* argument is used for groups that did not participate in the match; it defaults to *None*.
(Incompatibility note: in the original Python 1.5 release, if the tuple was one element long, a string would be returned instead. In later versions (from 1.5.1 on), a singleton tuple is returned in such cases.)

**groupdict([default])**

Return a dictionary containing all the *named* subgroups of the match, keyed by the subgroup name. The *default* argument is used for groups that did not participate in the match; it defaults to *None*.

**start([group])**

**end([group])**

Return the indices of the start and end of the substring matched by *group*; *group* defaults to zero (meaning the whole matched substring). Return -1 if *group* exists but did not contribute to the match. For a match object *m*, and a group *g* that did contribute to the match, the substring matched by group *g* (equivalent to *m*.group(*g*)) is

\[
\text{m.string[m.start(g):m.end(g)]}
\]

Note that *m.start(group)* will equal *m.end(group)* if *group* matched a null string. For example, after *m* = `re.search('b(c?)', 'cba')`, *m.start(0)* is 1, *m.end(0)* is 2, *m.start(1)* and *m.end(1)* are both 2, and *m.start(2)* raises an *IndexError* exception.

**span([group])**

For *MatchObject m*, return the 2-tuple (*m.start(group)*, *m.end(group)*). Note that if *group* did not contribute to the match, this is (-1, -1). Again, *group* defaults to zero.

**pos**

The value of *pos* which was passed to the *search()* or *match()* method of the *RegexObject*. This is the index into the string at which the RE engine started looking for a match.
**endpos**
The value of *endpos* which was passed to the `search()` or `match()` method of the *RegexObject*. This is the index into the string beyond which the RE engine will not go.

**lastindex**
The integer index of the last matched capturing group, or *None* if no group was matched at all. For example, the expressions *(a)b*, *((a)(b))* , and *((ab))* will have `lastindex == 1` if applied to the string 'ab', while the expression *(a)(b)* will have `lastindex == 2`, if applied to the same string.

**lastgroup**
The name of the last matched capturing group, or *None* if the group didn't have a name, or if no group was matched at all.

**re**
The regular expression object whose `match()` or `search()` method produced this *MatchObject* instance.

**string**
The string passed to `match()` or `search()`.

---

*Release 2.5, documentation updated on 19th September, 2006.*

*See [About this document...](...) for information on suggesting changes.*
4.2.6 Examples

Simulating \texttt{scanf(\()}

Python does not currently have an equivalent to \texttt{scanf()}\texttt{.} Regular expressions are generally more powerful, though also more verbose, than \texttt{scanf()} format strings. The table below offers some more-or-less equivalent mappings between \texttt{scanf()} format tokens and regular expressions.

<table>
<thead>
<tr>
<th>\texttt{scanf()} Token</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>.</td>
</tr>
<tr>
<td>%5c</td>
<td>{5}</td>
</tr>
<tr>
<td>%d</td>
<td>[-+]?d+</td>
</tr>
<tr>
<td>%e, %E, %f, %g</td>
<td>[-+]?(d+(\d*)?</td>
</tr>
<tr>
<td>%i</td>
<td>[-+]?0[xX][dA-Fa-f]+</td>
</tr>
<tr>
<td>%o</td>
<td>0[0-7]</td>
</tr>
<tr>
<td>%s</td>
<td>\S+</td>
</tr>
<tr>
<td>%u</td>
<td>\d+</td>
</tr>
<tr>
<td>%x, %X</td>
<td>0[xX][dA-Fa-f]+</td>
</tr>
</tbody>
</table>

To extract the filename and numbers from a string like

\texttt{/usr/sbin/sendmail - 0 errors, 4 warnings}

you would use a \texttt{scanf()} format like

\texttt{\%s - \%d errors, \%d warnings}

The equivalent regular expression would be

\texttt{(S+\) - (d+) errors, (d+) warnings}

Avoiding recursion

If you create regular expressions that require the engine to perform a lot of recursion, you may encounter a \texttt{RuntimeError} exception with the message \texttt{maximum recursion limit} exceeded. For example,
>>> import re
>>> s = 'Begin ' + 1000*'a very long string ' + 'end'
>>> re.match('Begin (\w| )*? end', s).end()
Traceback (most recent call last):
  File "<stdin>", line 1, in 
  File "/usr/local/lib/python2.5/re.py", line 132, in match
    return _compile(pattern, flags).match(string)
RuntimeError: maximum recursion limit exceeded

You can often restructure your regular expression to avoid recursion.

Starting with Python 2.3, simple uses of the *? pattern are special-cased to avoid recursion. Thus, the above regular expression can avoid recursion by being recast as Begin [a-zA-Z0-9_ ]*?end. As a further benefit, such regular expressions will run faster than their recursive equivalents.
4.3 struct -- Interpret strings as packed binary data

This module performs conversions between Python values and C structs represented as Python strings. It uses format strings (explained below) as compact descriptions of the lay-out of the C structs and the intended conversion to/from Python values. This can be used in handling binary data stored in files or from network connections, among other sources.

The module defines the following exception and functions:

**exception error**

Exception raised on various occasions; argument is a string describing what is wrong.

**pack( fmt, v1, v2, ...)**

Return a string containing the values v1, v2, ... packed according to the given format. The arguments must match the values required by the format exactly.

**unpack( fmt, string)**

Unpack the string (presumably packed by pack(fmt, ...)) according to the given format. The result is a tuple even if it contains exactly one item. The string must contain exactly the amount of data required by the format (len(string) must equal calcsize(fmt)).

**calcsize( fmt)**

Return the size of the struct (and hence of the string) corresponding to the given format.

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types:

<table>
<thead>
<tr>
<th>Format</th>
<th>C Type</th>
<th>Python</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>pad byte</td>
<td>no value</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>char</td>
<td>string of length 1</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-----------------</td>
<td>--------------------</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>signed char</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>unsigned char</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>short</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>unsigned short</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>int</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>unsigned int</td>
<td>long</td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>long</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>unsigned long</td>
<td>long</td>
<td></td>
</tr>
<tr>
<td>q</td>
<td>long long</td>
<td>long</td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>unsigned long long</td>
<td>long</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>float</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>double</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>char[]</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>char[]</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>void *</td>
<td>integer</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1) The "q" and "Q" conversion codes are available in native mode only if the platform C compiler supports C long long, or, on Windows, __int64. They are always available in standard modes. New in version 2.2.

A format character may be preceded by an integral repeat count. For example, the format string '4h' means exactly the same as 'hhhh'.

Whitespace characters between formats are ignored; a count and its format must not contain whitespace though.

For the "s" format character, the count is interpreted as the size of the string, not a repeat count like for the other format characters; for example, '10s' means a single 10-byte string, while '10c' means 10 characters. For packing, the string is truncated or padded with null bytes as appropriate to make it fit. For unpacking, the resulting string always has exactly the specified number of bytes. As a special case, '0s' means a single, empty string (while '0c' means 0
The "p" format character encodes a "Pascal string", meaning a short variable-length string stored in a fixed number of bytes. The count is the total number of bytes stored. The first byte stored is the length of the string, or 255, whichever is smaller. The bytes of the string follow. If the string passed in to pack() is too long (longer than the count minus 1), only the leading count-1 bytes of the string are stored. If the string is shorter than count-1, it is padded with null bytes so that exactly count bytes in all are used. Note that for unpack(), the "p" format character consumes count bytes, but that the string returned can never contain more than 255 characters.

For the "I", "L", "q" and "Q" format characters, the return value is a Python long integer.

For the "P" format character, the return value is a Python integer or long integer, depending on the size needed to hold a pointer when it has been cast to an integer type. A NULL pointer will always be returned as the Python integer 0. When packing pointer-sized values, Python integer or long integer objects may be used. For example, the Alpha and Merced processors use 64-bit pointer values, meaning a Python long integer will be used to hold the pointer; other platforms use 32-bit pointers and will use a Python integer.

By default, C numbers are represented in the machine's native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler).

Alternatively, the first character of the format string can be used to indicate the byte order, size and alignment of the packed data, according to the following table:

<table>
<thead>
<tr>
<th>Character</th>
<th>Byte order</th>
<th>Size and alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>native</td>
<td>native</td>
</tr>
<tr>
<td>=</td>
<td>native</td>
<td>standard</td>
</tr>
<tr>
<td>&lt;</td>
<td>little-endian</td>
<td>standard</td>
</tr>
<tr>
<td>&gt;</td>
<td>big-endian</td>
<td>standard</td>
</tr>
<tr>
<td>!</td>
<td>network (= big-endian)</td>
<td>standard</td>
</tr>
</tbody>
</table>
If the first character is not one of these, "@" is assumed.

Native byte order is big-endian or little-endian, depending on the host system. For example, Motorola and Sun processors are big-endian; Intel and DEC processors are little-endian.

Native size and alignment are determined using the C compiler's `sizeof` expression. This is always combined with native byte order.

Standard size and alignment are as follows: no alignment is required for any type (so you have to use pad bytes); short is 2 bytes; int and long are 4 bytes; long long (__int64 on Windows) is 8 bytes; float and double are 32-bit and 64-bit IEEE floating point numbers, respectively.

Note the difference between "@" and "=": both use native byte order, but the size and alignment of the latter is standardized.

The form "!" is available for those poor souls who claim they can't remember whether network byte order is big-endian or little-endian.

There is no way to indicate non-native byte order (force byte-swapping); use the appropriate choice of "<" or ">".

The "P" format character is only available for the native byte ordering (selected as the default or with the "@" byte order character). The byte order character "=" chooses to use little- or big-endian ordering based on the host system. The struct module does not interpret this as native ordering, so the "P" format is not available.

Examples (all using native byte order, size and alignment, on a big-endian machine):

```python
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', '\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
8
```

Hint: to align the end of a structure to the alignment requirement of a particular
type, end the format with the code for that type with a repeat count of zero. For example, the format 'llh01' specifies two pad bytes at the end, assuming longs are aligned on 4-byte boundaries. This only works when native size and alignment are in effect; standard size and alignment does not enforce any alignment.

**See Also:**

Module **array**:  
Packed binary storage of homogeneous data.

Module **xdrlib**:  
Packing and unpacking of XDR data.
4.4 difflib -- Helpers for computing deltas

New in version 2.1.

class SequenceMatcher
This is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are hashable. The basic algorithm predates, and is a little fancier than, an algorithm published in the late 1980's by Ratcliff and Obershulp under the hyperbolic name ``gestalt pattern matching.'' The idea is to find the longest contiguous matching subsequence that contains no ``junk'' elements (the Ratcliff and Obershulp algorithm doesn't address junk). The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence. This does not yield minimal edit sequences, but does tend to yield matches that ``look right'' to people.

Timing: The basic Ratcliff-Obershulp algorithm is cubic time in the worst case and quadratic time in the expected case. SequenceMatcher is quadratic time for the worst case and has expected-case behavior dependent in a complicated way on how many elements the sequences have in common; best case time is linear.

class Differ
This is a class for comparing sequences of lines of text, and producing human-readable differences or deltas. Differ uses SequenceMatcher both to compare sequences of lines, and to compare sequences of characters within similar (near-matching) lines.

Each line of a Differ delta begins with a two-letter code:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-'</td>
<td>line unique to sequence 1</td>
</tr>
<tr>
<td>'+'</td>
<td>line unique to sequence 2</td>
</tr>
<tr>
<td></td>
<td>line common to both sequences</td>
</tr>
</tbody>
</table>
Lines beginning with `? ` attempt to guide the eye to intraline differences, and were not present in either input sequence. These lines can be confusing if the sequences contain tab characters.

**class HtmlDiff**

This class can be used to create an HTML table (or a complete HTML file containing the table) showing a side by side, line by line comparison of text with inter-line and intra-line change highlights. The table can be generated in either full or contextual difference mode.

The constructor for this class is:

```python
__init__([tabsize][, wrapcolumn][, linejunk][, charjunk])
```

Initializes instance of HtmlDiff.

*tabsize* is an optional keyword argument to specify tab stop spacing and defaults to 8.

*wrapcolumn* is an optional keyword to specify column number where lines are broken and wrapped, defaults to None where lines are not wrapped.

*linejunk* and *charjunk* are optional keyword arguments passed into ndiff() (used by HtmlDiff to generate the side by side HTML differences). See ndiff() documentation for argument default values and descriptions.

The following methods are public:

```python
make_file(fromlines, tolines [, fromdesc][, todesc][, context][, numlines])
```

Compares *fromlines* and *tolines* (lists of strings) and returns a string which is a complete HTML file containing a table showing line by line differences with inter-line and intra-line changes highlighted.
fromdesc and todesc are optional keyword arguments to specify from/to file column header strings (both default to an empty string).

context and numlines are both optional keyword arguments. Set context to True when contextual differences are to be shown, else the default is False to show the full files. numlines defaults to 5. When context is True numlines controls the number of context lines which surround the difference highlights. When context is False numlines controls the number of lines which are shown before a difference highlight when using the "next" hyperlinks (setting to zero would cause the "next" hyperlinks to place the next difference highlight at the top of the browser without any leading context).

make_table(fromlines, tolines, [fromdesc], [todesc], [context], [numlines])

Compares fromlines and tolines (lists of strings) and returns a string which is a complete HTML table showing line by line differences with inter-line and intra-line changes highlighted.

The arguments for this method are the same as those for the make_file() method.

Tools/scripts/diff.py is a command-line front-end to this class and contains a good example of its use.

New in version 2.4.

countext_diff(a, b, [fromfile], [tofile], [fromfiledate], [tofiledate], [n], [lineterm])

Compare a and b (lists of strings); return a delta (a generator generating the delta lines) in context diff format.

Context diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in a before/after style. The number of context lines is set by n which defaults to three.

By default, the diff control lines (those with *** or - - - ) are created with a trailing newline. This is helpful so that inputs created from
file.readlines() result in diffs that are suitable for use with file.writelines() since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the lineterm argument to "" so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for fromfile, tofile, fromfiledate, and tofiledate. The modification times are normally expressed in the format returned by time.ctime(). If not specified, the strings default to blanks.

Tools/scripts/diff.py is a command-line front-end for this function.

New in version 2.3.

def get_close_matches(word, possibilities[, n][, cutoff])
Return a list of the best "good enough" matches. word is a sequence for which close matches are desired (typically a string), and possibilities is a list of sequences against which to match word (typically a list of strings).

Optional argument n (default 3) is the maximum number of close matches to return; n must be greater than 0.

Optional argument cutoff (default 0.6) is a float in the range [0, 1]. Possibilities that don't score at least that similar to word are ignored.

The best (no more than n) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

```python
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy', 'apple', 'ape'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)[
['while']
>>> get_close_matches('apple', keyword.kwlist)[
[]
>>> get_close_matches('accept', keyword.kwlist)['except']
```
ndiff( a, b[, linejunk][, charjunk])

Compare a and b (lists of strings); return a Differ-style delta (a generator generating the delta lines).

Optional keyword parameters linejunk and charjunk are for filter functions (or None):

linejunk: A function that accepts a single string argument, and returns true if the string is junk, or false if not. The default is (None), starting with Python 2.3. Before then, the default was the module-level function IS_LINE_JUNK(), which filters out lines without visible characters, except for at most one pound character ("#"). As of Python 2.3, the underlying SequenceMatcher class does a dynamic analysis of which lines are so frequent as to constitute noise, and this usually works better than the pre-2.3 default.

charjunk: A function that accepts a character (a string of length 1), and returns if the character is junk, or false if not. The default is module-level function IS_CHARACTER_JUNK(), which filters out whitespace characters (a blank or tab; note: bad idea to include newline in this!).

Tools/scripts/ndiff.py is a command-line front-end to this function.

```python
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(1),
               'ore\ntree\nemu\n'.splitlines(1))
... >>> print ''.join(diff),
  - one
  ?  ^
+ ore
  ?  ^
  - two
  - three
  ?  -
+ tree
+ emu
```

restore (sequence, which)

Return one of the two sequences that generated a delta.

Given a sequence produced by Differ.compare() or ndiff(), extract lines originating from file 1 or 2 (parameter which), stripping off line prefixes.
Example:

```python
>>> diff = ndiff('one\ntwo
three
'.splitlines(1),
...   'ore\ntree\nemu
'.splitlines(1))
>>> diff = list(diff)  # materialize the generated delta into a :
>>> print ''.join(restore(diff, 1)),
one
two
three
>>> print ''.join(restore(diff, 2)),
ore
tree
emu
```

```python
unified_diff(a, b[, fromfile][, tofile][, fromfiledate][, tofiledate][, n][, lineterm])
```

Compare `a` and `b` (lists of strings); return a delta (a generator generating the delta lines) in unified diff format.

Unified diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in an inline style (instead of separate before/after blocks). The number of context lines is set by `n` which defaults to three.

By default, the diff control lines (those with ---, +++ or @@) are created with a trailing newline. This is helpful so that inputs created from `file.readlines()` result in diffs that are suitable for use with `file.writelines()` since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the `lineterm` argument to "" so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for `fromfile`, `tofile`, `fromfiledate`, and `tofiledate`. The modification times are normally expressed in the format returned by `time.ctime()`. If not specified, the strings default to blanks.

`Tools/scripts/diff.py` is a command-line front-end for this function.

New in version 2.3.
**IS_LINE_JUNK**( *line*)

Return true for ignorable lines. The line *line* is ignorable if *line* is blank or contains a single "#", otherwise it is not ignorable. Used as a default for parameter *linejunk* in ndiff() before Python 2.3.

**IS_CHARACTER_JUNK**( *ch*)

Return true for ignorable characters. The character *ch* is ignorable if *ch* is a space or tab, otherwise it is not ignorable. Used as a default for parameter *charjunk* in ndiff().

---

**See Also:**

*Pattern Matching: The Gestalt Approach*

Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzener. This was published in *Dr. Dobb's Journal* in July, 1988.

---

**Subsections**

- 4.4.1 SequenceMatcher Objects
- 4.4.2 SequenceMatcher Examples
- 4.4.3 Differ Objects
- 4.4.4 Differ Example

---

Release 2.5, documentation updated on 19th September, 2006.
See *About this document...* for information on suggesting changes.
4.4.1 SequenceMatcher Objects

The SequenceMatcher class has this constructor:

```python
class SequenceMatcher([isjunk[, a[, b]]])
```

Optional argument isjunk must be None (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is ``junk'' and should be ignored. Passing None for isjunk is equivalent to passing `lambda x: 0`; in other words, no elements are ignored. For example, pass:

```python
lambda x: x in " \	"
```

if you're comparing lines as sequences of characters, and don't want to synch up on blanks or hard tabs.

The optional arguments a and b are sequences to be compared; both default to empty strings. The elements of both sequences must be hashable.

SequenceMatcher objects have the following methods:

```python
set_seqs(a, b)
```

Set the two sequences to be compared.

SequenceMatcher computes and caches detailed information about the second sequence, so if you want to compare one sequence against many sequences, use `set_seq2()` to set the commonly used sequence once and call `set_seq1()` repeatedly, once for each of the other sequences.

```python
set_seq1(a)
```

Set the first sequence to be compared. The second sequence to be compared is not changed.

```python
set_seq2(b)
```

Set the second sequence to be compared. The first sequence to be compared is not changed.
**find_longest_match**( *alo, ahi, blo, bhi*)

Find longest matching block in `a[alo:ahi]` and `b[blo:bhi]`.

If `isjunk` was omitted or `None`, `get_longest_match()` returns `(i, j, k)` such that `a[i:i+k]` is equal to `b[j:j+k]`, where `alo <= i <= i+k <= ahi` and `blo <= j <= j+k <= bhi`. For all `(i', j', k')` meeting those conditions, the additional conditions `k >= k'`, `i <= i'`, and if `i == i'`, `j <= j'` are also met. In other words, of all maximal matching blocks, return one that starts earliest in `a`, and of all those maximal matching blocks that start earliest in `a`, return the one that starts earliest in `b`.

```python
>>> s = SequenceMatcher(None, "abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
(0, 4, 5)
```

If `isjunk` was provided, first the longest matching block is determined as above, but with the additional restriction that no junk element appears in the block. Then that block is extended as far as possible by matching (only) junk elements on both sides. So the resulting block never matches on junk except as identical junk happens to be adjacent to an interesting match.

Here's the same example as before, but considering blanks to be junk. That prevents `'abcd'` from matching the `'abcd'` at the tail end of the second sequence directly. Instead only the `'abcd'` can match, and matches the leftmost `'abcd'` in the second sequence:

```python
>>> s = SequenceMatcher(lambda x: x==" ", "abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
(1, 0, 4)
```

If no blocks match, this returns `(alo, blo, 0)`.

**get_matching_blocks()**

Return list of triples describing matching subsequences. Each triple is of the form `(i, j, n)`, and means that `a[i:i+n] == b[j:j+n]`. The triples are monotonically increasing in `i` and `j`.

The last triple is a dummy, and has the value `(len(a), len(b), 0)`. It is the only triple with `n == 0`. 
If \((i, j, n)\) and \((i', j', n')\) are adjacent triples in the list, and the second is not the last triple in the list, then \(i+n \neq i'\) or \(j+n \neq j'\); in other words, adjacent triples always describe non-adjacent equal blocks. Changed in version 2.5: The guarantee that adjacent triples always describe non-adjacent blocks was implemented.

```python
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[(0, 0, 2), (3, 2, 2), (5, 4, 0)]
```

**get_opcodes()**

Return list of 5-tuples describing how to turn \(a\) into \(b\). Each tuple is of the form \((tag, i1, i2, j1, j2)\). The first tuple has \(i1 == j1 == 0\), and remaining tuples have \(i1\) equal to the \(i2\) from the preceding tuple, and, likewise, \(j1\) equal to the previous \(j2\).

The \(tag\) values are strings, with these meanings:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'replace'</td>
<td>(a[i1:i2]) should be replaced by (b[j1:j2]).</td>
</tr>
<tr>
<td>'delete'</td>
<td>(a[i1:i2]) should be deleted. Note that (j1 == j2) in this case.</td>
</tr>
<tr>
<td>'insert'</td>
<td>(b[j1:j2]) should be inserted at (a[i1:i1]). Note that (i1 == i2) in this case.</td>
</tr>
<tr>
<td>'equal'</td>
<td>(a[i1:i2] == b[j1:j2]) (the sub-sequences are equal).</td>
</tr>
</tbody>
</table>

For example:

```python
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...    print("%7s a[%d:%d] (%s) b[%d:%d] (%s)" %
...            (tag, i1, i2, a[i1:i2], j1, j2, b[j1:j2]))
...    if tag == 'replace':
...        print("delete a[0:1] (q) b[0:0] ()")
...        print("equal a[1:3] (ab) b[0:2] (ab)
...        print("replace a[3:4] (x) b[2:3] (y)
...        print("equal a[4:6] (cd) b[3:5] (cd)

get_grouped_opcodes([n])
Return a generator of groups with up to $n$ lines of context.

Starting with the groups returned by `get_opcodes()`, this method splits out smaller change clusters and eliminates intervening ranges which have no changes.

The groups are returned in the same format as `get_opcodes()`. New in version 2.3.

**ratio()**

Return a measure of the sequences' similarity as a float in the range \([0, 1]\).

Where $T$ is the total number of elements in both sequences, and $M$ is the number of matches, this is $2.0 \times M / T$. Note that this is 1.0 if the sequences are identical, and 0.0 if they have nothing in common.

This is expensive to compute if `get_matching_blocks()` or `get_opcodes()` hasn't already been called, in which case you may want to try `quick_ratio()` or `real_quick_ratio()` first to get an upper bound.

**quick_ratio()**

Return an upper bound on `ratio()` relatively quickly.

This isn't defined beyond that it is an upper bound on `ratio()`, and is faster to compute.

**real_quick_ratio()**

Return an upper bound on `ratio()` very quickly.

This isn't defined beyond that it is an upper bound on `ratio()`, and is faster to compute than either `ratio()` or `quick_ratio()`.

The three methods that return the ratio of matching to total characters can give different results due to differing levels of approximation, although `quick_ratio()` and `real_quick_ratio()` are always at least as large as `ratio()`:

```python
>>> s = SequenceMatcher(None, "abcd", "bcde")
```
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0

---

Python Library Reference

Previous: 4.4 difflib  Up: 4.4 difflib  Next: 4.4.2 SequenceMatcher Examples

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.4.2 SequenceMatcher Examples

This example compares two strings, considering blanks to be `junk:

```python
>>> s = SequenceMatcher(lambda x: x == " ",
...                       "private Thread currentThread;",
...                       "private volatile Thread currentThread;"
)
```

`ratio()` returns a float in [0, 1], measuring the similarity of the sequences. As a rule of thumb, a `ratio()` value over 0.6 means the sequences are close matches:

```python
>>> print round(s.ratio(), 3)
0.866
```

If you're only interested in where the sequences match, `get_matching_blocks()` is handy:

```python
>>> for block in s.get_matching_blocks():
...     print "a[%d] and b[%d] match for %d elements" % block
...     a[0] and b[0] match for 8 elements
...     a[8] and b[17] match for 6 elements
...     a[14] and b[23] match for 15 elements
...     a[29] and b[38] match for 0 elements
```

Note that the last tuple returned by `get_matching_blocks()` is always a dummy, `(len(a), len(b), 0)`, and this is the only case in which the last tuple element (number of elements matched) is 0.

If you want to know how to change the first sequence into the second, use `get_opcodes()`:

```python
>>> for opcode in s.get_opcodes():
...     print "%6s a[%d:%d] b[%d:%d]" % opcode
...     equal a[0:8] b[0:8]
...     insert a[8:8] b[8:17]
...     equal a[8:14] b[17:23]
...     equal a[14:29] b[23:38]
```

See also the function `get_close_matches()` in this module, which shows how simple code building on `SequenceMatcher` can be used to do useful work.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
### 4.4.3 Differ Objects

Note that Differ-generated deltas make no claim to be **minimal** diffs. To the contrary, minimal diffs are often counter-intuitive, because they synch up anywhere possible, sometimes accidental matches 100 pages apart. Restricting synch points to contiguous matches preserves some notion of locality, at the occasional cost of producing a longer diff.

The **Differ** class has this constructor:

```python
class Differ([linejunk[, charjunk]])
```

Optional keyword parameters `linejunk` and `charjunk` are for filter functions (or None):

- **linejunk**: A function that accepts a single string argument, and returns true if the string is junk. The default is `None`, meaning that no line is considered junk.

  ```python
def linejunk(line):
    # Filter out junk lines
    return line
  ``

- **charjunk**: A function that accepts a single character argument (a string of length 1), and returns true if the character is junk. The default is `None`, meaning that no character is considered junk.

  ```python
def charjunk(char):
    # Filter out junk characters
    return char
  ``

Differ objects are used (deltas generated) via a single method:

```python
def compare(a, b)
```

Compare two sequences of lines, and generate the delta (a sequence of lines).

Each sequence must contain individual single-line strings ending with newlines. Such sequences can be obtained from the `readlines()` method of file-like objects. The delta generated also consists of newline-terminated strings, ready to be printed as-is via the `writelines()` method of a file-like object.
Example

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
This example compares two texts. First we set up the texts, sequences of individual single-line strings ending with newlines (such sequences can also be obtained from the \texttt{readlines()} method of file-like objects):

```python
>>> text1 = '''
1. Beautiful is better than ugly.
...
2. Explicit is better than implicit.
...
3. Simple is better than complex.
...
4. Complex is better than complicated.
'''.splitlines(1)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
```

Next we instantiate a Differ object:

```python
>>> d = Differ()
```

Note that when instantiating a \texttt{Differ} object we may pass functions to filter out line and character \texttt{``junk.''} See the \texttt{Differ()} constructor for details.

Finally, we compare the two:

```python
>>> result = list(d.compare(text1, text2))
```

\texttt{result} is a list of strings, so let's pretty-print it:

```python
>>> from pprint import pprint
>>> pprint(result)
['	1. Beautiful is better than ugly.\n', '-
2. Explicit is better than implicit.\n', '-
3. Simple is better than complex.\n', '+
3. Simple is better than complex.\n', '?
++
', '-
4. Complex is better than complicated.\n', '?
    +++ ^
', '+
4. Complicated is better than complex.\n', '?
    ^
']
```
' +  5. Flat is better than nested.
']

As a single multi-line string it looks like this:

```python
>>> import sys
>>> sys.stdout.writelines(result)
1. Beautiful is better than ugly.
- 2. Explicit is better than implicit.
- 3. Simple is better than complex.
+ 3. Simple is better than complex.
? ++
- 4. Complex is better than complicated.
? ^
- 4. Complicated is better than complex.
? +++ ^
+ 5. Flat is better than nested.
```
4.5 StringIO -- Read and write strings as files

This module implements a file-like class, `StringIO`, that reads and writes a string buffer (also known as memory files). See the description of file objects for operations (section 3.9).

```python
class StringIO([buffer])
```

When a `StringIO` object is created, it can be initialized to an existing string by passing the string to the constructor. If no string is given, the `StringIO` will start empty. In both cases, the initial file position starts at zero.

The `StringIO` object can accept either Unicode or 8-bit strings, but mixing the two may take some care. If both are used, 8-bit strings that cannot be interpreted as 7-bit ASCII (that use the 8th bit) will cause a `UnicodeError` to be raised when `getvalue()` is called.

The following methods of `StringIO` objects require special mention:

`getvalue()`

Retrieve the entire contents of the ``file'' at any time before the `StringIO` object's `close()` method is called. See the note above for information about mixing Unicode and 8-bit strings; such mixing can cause this method to raise `UnicodeError`.

`close()`

Free the memory buffer.

Example usage:

```python
import StringIO
output = StringIO.StringIO()
output.write('First line.
')
print >>output, 'Second line.'
```
# Retrieve file contents -- this will be
# 'First line.
Second line.
' contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
4.6 cStringIO -- Faster version of StringIO

The module cStringIO provides an interface similar to that of the StringIO module. Heavy use of StringIO.StringIO objects can be made more efficient by using the function StringIO() from this module instead.

Since this module provides a factory function which returns objects of built-in types, there's no way to build your own version using subclassing. Use the original StringIO module in that case.

Unlike the memory files implemented by the StringIO module, those provided by this module are not able to accept Unicode strings that cannot be encoded as plain ASCII strings.

Another difference from the StringIO module is that calling StringIO() with a string parameter creates a read-only object. Unlike an object created without a string parameter, it does not have write methods. These objects are not generally visible. They turn up in tracebacks as StringI and StringO.

The following data objects are provided as well:

**InputType**

The type object of the objects created by calling StringIO with a string parameter.

**OutputType**

The type object of the objects returned by calling StringIO with no parameters.

There is a C API to the module as well; refer to the module source for more information.

Example usage:

```python
import cStringIO
```
output = cStringIO.StringIO()
output.write('First line.\n')
print >>output, 'Second line.'

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n' contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
4.7 textwrap -- Text wrapping and filling

New in version 2.3.

The textwrap module provides two convenience functions, `wrap()` and `fill()`, as well as TextWrapper, the class that does all the work, and a utility function `dedent()`. If you're just wrapping or filling one or two text strings, the convenience functions should be good enough; otherwise, you should use an instance of TextWrapper for efficiency.

```python
wrap( text, width=70]
```
Wraps the single paragraph in `text` (a string) so every line is at most `width` characters long. Returns a list of output lines, without final newlines.

Optional keyword arguments correspond to the instance attributes of TextWrapper, documented below. `width` defaults to 70.

```python
fill( text, width=70]
```
Wraps the single paragraph in `text`, and returns a single string containing the wrapped paragraph. `fill()` is shorthand for

```
"\n"></\n\n.join(wrap(text, ...
```

In particular, `fill()` accepts exactly the same keyword arguments as `wrap()`.

Both `wrap()` and `fill()` work by creating a TextWrapper instance and calling a single method on it. That instance is not reused, so for applications that wrap/fill many text strings, it will be more efficient for you to create your own TextWrapper object.

An additional utility function, `dedent()`, is provided to remove indentation from strings that have unwanted whitespace to the left of the text.

```python
dedent( text)
```
Remove any common leading whitespace from every line in `text`.

This can be used to make triple-quoted strings line up with the left edge of the display, while still presenting them in the source code in indented form.

Note that tabs and spaces are both treated as whitespace, but they are not equal: the lines " hello" and "\hello" are considered to have no common leading whitespace. (This behaviour is new in Python 2.5; older versions of this module incorrectly expanded tabs before searching for common leading whitespace.)

For example:

```python
def test():
    # end first line with \ to avoid the empty line!
    s = '''
    hello
    world
    '''
    print repr(s)  # prints ' hello
    print repr(dedent(s))  # prints 'hello
    world
'''
```

```text
class TextWrapper(...)
    The TextWrapper constructor accepts a number of optional keyword arguments. Each argument corresponds to one instance attribute, so for example

    wrapper = TextWrapper(initial_indent="* ")

    is the same as

    wrapper = TextWrapper()
    wrapper.initial_indent = "* 

    You can re-use the same TextWrapper object many times, and you can change any of its options through direct assignment to instance attributes between uses.

    The TextWrapper instance attributes (and keyword arguments to the constructor) are as follows:

    width
        (default: 70) The maximum length of wrapped lines. As long as there are
no individual words in the input text longer than \texttt{width}, \texttt{TextWrapper} guarantees that no output line will be longer than \texttt{width} characters.

\texttt{expand\_tabs}
(default: \texttt{True}) If true, then all tab characters in \texttt{text} will be expanded to spaces using the \texttt{expandtabs()} method of \texttt{text}.

\texttt{replace\_whitespace}
(default: \texttt{True}) If true, each whitespace character (as defined by \texttt{string.whitespace}) remaining after tab expansion will be replaced by a single space. \textbf{Note:} If \texttt{expand\_tabs} is false and \texttt{replace\_whitespace} is true, each tab character will be replaced by a single space, which is \textit{not} the same as tab expansion.

\texttt{initial\_indent}
(default: \texttt{''}) String that will be prepended to the first line of wrapped output. Counts towards the length of the first line.

\texttt{subsequent\_indent}
(default: \texttt{''}) String that will be prepended to all lines of wrapped output except the first. Counts towards the length of each line except the first.

\texttt{fix\_sentence\_endings}
(default: \texttt{False}) If true, TextWrapper attempts to detect sentence endings and ensure that sentences are always separated by exactly two spaces. This is generally desired for text in a monospaced font. However, the sentence detection algorithm is imperfect: it assumes that a sentence ending consists of a lowercase letter followed by one of \texttt{"."}, \texttt{"!"}, or \texttt{"?"}, possibly followed by one of \texttt{""} or \texttt{"!'"}, followed by a space. One problem with this is algorithm is that it is unable to detect the difference between \texttt{``Dr.''} in

\begin{verbatim}
    [...] Dr. Frankenstein's monster [...] 
\end{verbatim}

and \texttt{``Spot.''} in

\begin{verbatim}
    [...] See Spot. See Spot run [...] 
\end{verbatim}

\texttt{fix\_sentence\_endings} is false by default.
Since the sentence detection algorithm relies on `string.lowercase` for the definition of "lowercase letter," and a convention of using two spaces after a period to separate sentences on the same line, it is specific to English-language texts.

**break_long_words**

(default: True) If true, then words longer than `width` will be broken in order to ensure that no lines are longer than `width`. If it is false, long words will not be broken, and some lines may be longer than `width`. (Long words will be put on a line by themselves, in order to minimize the amount by which `width` is exceeded.)

**TextWrapper** also provides two public methods, analogous to the module-level convenience functions:

**wrap**(text)

Wraps the single paragraph in `text` (a string) so every line is at most `width` characters long. All wrapping options are taken from instance attributes of the **TextWrapper** instance. Returns a list of output lines, without final newlines.

**fill**(text)

Wraps the single paragraph in `text`, and returns a single string containing the wrapped paragraph.
4.8 codecs -- Codec registry and base classes

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry which manages the codec and error handling lookup process.

It defines the following functions:

register( search_function)

Register a codec search function. Search functions are expected to take one argument, the encoding name in all lower case letters, and return a CodecInfo object having the following attributes:

- **name** The name of the encoding;
- **encoder** The stateless encoding function;
- **decoder** The stateless decoding function;
- **incrementalencoder** An incremental encoder class or factory function;
- **incrementaldecoder** An incremental decoder class or factory function;
- **streamwriter** A stream writer class or factory function;
- **streamreader** A stream reader class or factory function.

The various functions or classes take the following arguments:

*encoder* and *decoder*: These must be functions or methods which have the same interface as the **encode()**/**decode()** methods of Codec instances (see Codec Interface). The functions/methods are expected to work in a stateless mode.

*incrementalencoder* and *incrementalencoder*: These have to be factory functions providing the following interface:

factory( *errors*='strict')
The factory functions must return objects providing the interfaces defined by the base classes `IncrementalEncoder` and `IncrementalEncoder`, respectively. Incremental codecs can maintain state.

`streamreader` and `streamwriter`: These have to be factory functions providing the following interface:

```python
factory(stream, errors='strict')
```

The factory functions must return objects providing the interfaces defined by the base classes `StreamWriter` and `StreamReader`, respectively. Stream codecs can maintain state.

Possible values for errors are `'strict'` (raise an exception in case of an encoding error), `'replace'` (replace malformed data with a suitable replacement marker, such as `"?"`), `'ignore'` (ignore malformed data and continue without further notice), `'xmlcharrefreplace'` (replace with the appropriate XML character reference (for encoding only)) and `'backslashreplace'` (replace with backslashed escape sequences (for encoding only)) as well as any other error handling name defined via `register_error()`.

In case a search function cannot find a given encoding, it should return `None`.

`lookup(encoding)`

Looks up the codec info in the Python codec registry and returns a `CodecInfo` object as defined above.

Encodings are first looked up in the registry's cache. If not found, the list of registered search functions is scanned. If no `CodecInfo` object is found, a `LookupError` is raised. Otherwise, the `CodecInfo` object is stored in the cache and returned to the caller.

To simplify access to the various codecs, the module provides these additional functions which use `lookup()` for the codec lookup:

```python
getencoder(encoding)
```
Look up the codec for the given encoding and return its encoder function. Raises a LookupError in case the encoding cannot be found.

```python
getdecoder(encoding)
```
Look up the codec for the given encoding and return its decoder function.

Raises a LookupError in case the encoding cannot be found.

```python
getincrementalencoder(encoding)
```
Look up the codec for the given encoding and return its incremental encoder class or factory function.

Raises a LookupError in case the encoding cannot be found or the codec doesn't support an incremental encoder. New in version 2.5.

```python
getincrementaldecoder(encoding)
```
Look up the codec for the given encoding and return its incremental decoder class or factory function.

Raises a LookupError in case the encoding cannot be found or the codec doesn't support an incremental decoder. New in version 2.5.

```python
getreader(encoding)
```
Look up the codec for the given encoding and return its StreamReader class or factory function.

Raises a LookupError in case the encoding cannot be found.

```python
getwriter(encoding)
```
Look up the codec for the given encoding and return its StreamWriter class or factory function.

Raises a LookupError in case the encoding cannot be found.

```python
register_error(name, error_handler)
```
Register the error handling function `error_handler` under the name `name`. `error_handler` will be called during encoding and decoding in case of an
error, when name is specified as the errors parameter.

For encoding error_handler will be called with a UnicodeEncodeError instance, which contains information about the location of the error. The error handler must either raise this or a different exception or return a tuple with a replacement for the unencodable part of the input and a position where encoding should continue. The encoder will encode the replacement and continue encoding the original input at the specified position. Negative position values will be treated as being relative to the end of the input string. If the resulting position is out of bound an IndexError will be raised.

Decoding and translating works similar, except UnicodeDecodeError or UnicodeTranslateError will be passed to the handler and that the replacement from the error handler will be put into the output directly.

lookup_error(name)
    Return the error handler previously registered under the name name.

    Raises a LookupError in case the handler cannot be found.

strict_errors(exception)
    Implements the strict error handling.

replace_errors(exception)
    Implements the replace error handling.

ignore_errors(exception)
    Implements the ignore error handling.

xmlcharrefreplace_errors_errors(exception)
    Implements the xmlcharrefreplace error handling.

backslashreplace_errors_errors(exception)
    Implements the backslashreplace error handling.

To simplify working with encoded files or stream, the module also defines these utility functions:
**open**( *filename*, *mode*[ , *encoding*[ , *errors*[ , *buffering]*)]*))

Open an encoded file using the given *mode* and return a wrapped version providing transparent encoding/decoding.

**Note:** The wrapped version will only accept the object format defined by the codecs, i.e. Unicode objects for most built-in codecs. Output is also codec-dependent and will usually be Unicode as well.

*encoding* specifies the encoding which is to be used for the file.

*errors* may be given to define the error handling. It defaults to 'strict' which causes a `ValueError` to be raised in case an encoding error occurs.

*buffering* has the same meaning as for the built-in open() function. It defaults to line buffered.

**EncodedFile**( *file*, *input*[ , *output*[ , *errors]*)*)

Return a wrapped version of file which provides transparent encoding translation.

Strings written to the wrapped file are interpreted according to the given *input* encoding and then written to the original file as strings using the *output* encoding. The intermediate encoding will usually be Unicode but depends on the specified codecs.

If *output* is not given, it defaults to *input*.

*errors* may be given to define the error handling. It defaults to 'strict', which causes `ValueError` to be raised in case an encoding error occurs.

**iterencode**( *iterable*, *encoding*[ , *errors]*))

Uses an incremental encoder to iteratively encode the input provided by *iterable*. This function is a generator. *errors* (as well as any other keyword argument) is passed through to the incremental encoder. New in version 2.5.

**iterdecode**( *iterable*, *encoding*[ , *errors]*)

Uses an incremental decoder to iteratively decode the input provided by
iterable. This function is a generator. errors (as well as any other keyword argument) is passed through to the incremental encoder. New in version 2.5.

The module also provides the following constants which are useful for reading and writing to platform dependent files:

BOM
BOM_BE
BOM_LE
BOM_UTF8
BOM_UTF16
BOM_UTF16_BE
BOM_UTF16_LE
BOM_UTF32
BOM_UTF32_BE
BOM_UTF32_LE

These constants define various encodings of the Unicode byte order mark (BOM) used in UTF-16 and UTF-32 data streams to indicate the byte order used in the stream or file and in UTF-8 as a Unicode signature. BOM_UTF16 is either BOM_UTF16_BE or BOM_UTF16_LE depending on the platform's native byte order, BOM is an alias for BOM_UTF16, BOM_LE for BOM_UTF16_LE and BOM_BE for BOM_UTF16_BE. The others represent the BOM in UTF-8 and UTF-32 encodings.

Subsections

- 4.8.1 Codec Base Classes
  - 4.8.1.1 Codec Objects
  - 4.8.1.2 IncrementalEncoder Objects
  - 4.8.1.3 IncrementalDecoder Objects
  - 4.8.1.4 StreamWriter Objects
  - 4.8.1.5 StreamReader Objects
  - 4.8.1.6 StreamReaderWriter Objects
  - 4.8.1.7 StreamRecoder Objects
- 4.8.2 Encodings and Unicode
- 4.8.3 Standard Encodings
- 4.8.4 encodings.idna -- Internationalized Domain Names in Applications
4.8.5 encodings.utf_8_sig -- UTF-8 codec with BOM signature
4.8.1 Codec Base Classes

The `codecs` module defines a set of base classes which define the interface and can also be used to easily write your own codecs for use in Python.

Each codec has to define four interfaces to make it usable as codec in Python: stateless encoder, stateless decoder, stream reader and stream writer. The stream reader and writers typically reuse the stateless encoder/decoder to implement the file protocols.

The `Codec` class defines the interface for stateless encoders/decoders.

To simplify and standardize error handling, the `encode()` and `decode()` methods may implement different error handling schemes by providing the `errors` string argument. The following string values are defined and implemented by all standard Python codecs:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'strict'</td>
<td>Raise <code>UnicodeError</code> (or a subclass); this is the default.</td>
</tr>
<tr>
<td>'ignore'</td>
<td>Ignore the character and continue with the next.</td>
</tr>
<tr>
<td>'replace'</td>
<td>Replace with a suitable replacement character; Python will use the official U+FFFD REPLACEMENT CHARACTER for the built-in Unicode codecs on decoding and '?' on encoding.</td>
</tr>
<tr>
<td>'xmlcharrefreplace'</td>
<td>Replace with the appropriate XML character reference (only for encoding).</td>
</tr>
<tr>
<td>'backslashreplace'</td>
<td>Replace with backslashed escape sequences (only for encoding).</td>
</tr>
</tbody>
</table>

The set of allowed values can be extended via `register_error`. 

---

Subsections
4.8.1.1 Codec Objects
4.8.1.2 IncrementalEncoder Objects
4.8.1.3 IncrementalDecoder Objects
4.8.1.4 StreamWriter Objects
4.8.1.5 StreamReader Objects
4.8.1.6 StreamReaderWriter Objects
4.8.1.7 StreamRecoder Objects
4.8.2 Encodings and Unicode

Unicode strings are stored internally as sequences of codepoints (to be precise as Py_UNICODE arrays). Depending on the way Python is compiled (either via --enable-unicode=ucs2 or --enable-unicode=ucs4, with the former being the default) Py_UNICODE is either a 16-bit or 32-bit data type. Once a Unicode object is used outside of CPU and memory, CPU endianness and how these arrays are stored as bytes become an issue. Transforming a unicode object into a sequence of bytes is called encoding and recreating the unicode object from the sequence of bytes is known as decoding. There are many different methods for how this transformation can be done (these methods are also called encodings). The simplest method is to map the codepoints 0-255 to the bytes 0x0-0xff. This means that a unicode object that contains codepoints above U+00FF can't be encoded with this method (which is called 'latin-1' or 'iso-8859-1'). unicode.encode() will raise a UnicodeEncodeError that looks like this: "UnicodeEncodeError: 'latin-1' codec can't encode character u'\u1234' in position 3: ordinal not in range(256)".

There's another group of encodings (the so called charmap encodings) that choose a different subset of all unicode code points and how these codepoints are mapped to the bytes 0x0-0xff. To see how this is done simply open e.g. encodings/cp1252.py (which is an encoding that is used primarily on Windows). There's a string constant with 256 characters that shows you which character is mapped to which byte value.

All of these encodings can only encode 256 of the 65536 (or 1114111) codepoints defined in unicode. A simple and straightforward way that can store each Unicode code point, is to store each codepoint as two consecutive bytes. There are two possibilities: Store the bytes in big endian or in little endian order. These two encodings are called UTF-16-BE and UTF-16-LE respectively. Their disadvantage is that if e.g. you use UTF-16-BE on a little endian machine you will always have to swap bytes on encoding and decoding. UTF-16 avoids this problem: Bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a UTF-16 byte sequence, there's the so called BOM (the "Byte Order Mark"). This is the Unicode character U+FEFF. This
character will be prepended to every UTF-16 byte sequence. The byte swapped version of this character (\texttt{0xFFFE}) is an illegal character that may not appear in a Unicode text. So when the first character in an UTF-16 byte sequence appears to be a \texttt{U+FFE} the bytes have to be swapped on decoding. Unfortunately upto Unicode 4.0 the character \texttt{U+FEFF} had a second purpose as a "ZERO WIDTH NO-BREAK SPACE": A character that has no width and doesn't allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using \texttt{U+FEFF} as a "ZERO WIDTH NO-BREAK SPACE" has been deprecated (with \texttt{U+2060} ("WORD JOINER") assuming this role). Nevertheless Unicode software still must be able to handle \texttt{U+FEFF} in both roles: As a BOM it's a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a Unicode string; as a "ZERO WIDTH NO-BREAK SPACE" it's a normal character that will be decoded like any other.

There's another encoding that is able to encoding the full range of Unicode characters: UTF-8. UTF-8 is an 8-bit encoding, which means there are no issues with byte order in UTF-8. Each byte in a UTF-8 byte sequence consists of two parts: Marker bits (the most significant bits) and payload bits. The marker bits are a sequence of zero to six 1 bits followed by a 0 bit. Unicode characters are encoded like this (with \texttt{x} being payload bits, which when concatenated give the Unicode character):

<table>
<thead>
<tr>
<th>Range</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>U-00000000 ... U-0000007F</td>
<td>0xxxxxxx</td>
</tr>
<tr>
<td>U-00000080 ... U-000007FF</td>
<td>110xxxxx 10xxxxxx</td>
</tr>
<tr>
<td>U-00000080 ... U-000007FF</td>
<td>1110xxxx 10xxxxxx 10xxxxxx</td>
</tr>
<tr>
<td>U-00010000 ... U-0001FFFF</td>
<td>11110xx 10xxxxxx 10xxxxxx 10xxxxxx</td>
</tr>
<tr>
<td>U-00200000 ... U-03FFFFFF</td>
<td>111110xx 10xxxxxx 10xxxxxx 10xxxxxx</td>
</tr>
<tr>
<td>U-04000000 ... U-7FFFFFFF</td>
<td>1111110x 10xxxxxx 10xxxxxx 10xxxxxx</td>
</tr>
</tbody>
</table>

10xxxxxx
The least significant bit of the Unicode character is the rightmost x bit.

As UTF-8 is an 8-bit encoding no BOM is required and any U+FEFF character in the decoded Unicode string (even if it's the first character) is treated as a "ZERO WIDTH NO-BREAK SPACE".

Without external information it's impossible to reliably determine which encoding was used for encoding a Unicode string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte sequences have a structure that doesn't allow arbitrary byte sequence. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python 2.5 calls "utf-8-sig") for its Notepad program: Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence: 0xef, 0xbb, 0xbf) is written. As it's rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to

LATIN SMALL LETTER I WITH DIAERESIS
RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
INVERTED QUESTION MARK

in iso-8859-1), this increases the probability that a utf-8-sig encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the utf-8-sig codec will write 0xef, 0xbb, 0xbf as the first three bytes to the file. On decoding utf-8-sig will skip those three bytes if they appear as the first three bytes in the file.
4.8.3 Standard Encodings

Python comes with a number of codecs built-in, either implemented as C functions or with dictionaries as mapping tables. The following table lists the codecs by name, together with a few common aliases, and the languages for which the encoding is likely used. Neither the list of aliases nor the list of languages is meant to be exhaustive. Notice that spelling alternatives that only differ in case or use a hyphen instead of an underscore are also valid aliases.

Many of the character sets support the same languages. They vary in individual characters (e.g. whether the EURO SIGN is supported or not), and in the assignment of characters to code positions. For the European languages in particular, the following variants typically exist:

- an ISO 8859 codeset
- a Microsoft Windows code page, which is typically derived from a 8859 codeset, but replaces control characters with additional graphic characters
- an IBM EBCDIC code page
- an IBM PC code page, which is ASCII compatible

<table>
<thead>
<tr>
<th>Codec</th>
<th>Aliases</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii</td>
<td>646, us-ascii</td>
<td>English</td>
</tr>
<tr>
<td>big5</td>
<td>big5-tw, csbig5</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>big5hkscs</td>
<td>big5-hkscs, hkscs</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>cp037</td>
<td>IBM037, IBM039</td>
<td>English</td>
</tr>
<tr>
<td>cp424</td>
<td>EBCDIC-CP-HE, IBM424</td>
<td>Hebrew</td>
</tr>
<tr>
<td>cp437</td>
<td>437, IBM437</td>
<td>English</td>
</tr>
<tr>
<td>cp500</td>
<td>EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp737</td>
<td></td>
<td>Greek</td>
</tr>
<tr>
<td>cp775</td>
<td>IBM775</td>
<td>Baltic languages</td>
</tr>
<tr>
<td>cp850</td>
<td>850, IBM850</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp852</td>
<td>852, IBM852</td>
<td>Central and Eastern Europe</td>
</tr>
<tr>
<td>cp855</td>
<td>855, IBM855</td>
<td>Bulgarian, Byelorussian,</td>
</tr>
<tr>
<td>Code</td>
<td>Language(s)</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------</td>
<td></td>
</tr>
<tr>
<td>cp856</td>
<td>Hebrew</td>
<td></td>
</tr>
<tr>
<td>cp857</td>
<td>Turkish</td>
<td></td>
</tr>
<tr>
<td>cp860</td>
<td>Portuguese</td>
<td></td>
</tr>
<tr>
<td>cp861</td>
<td>Icelandic</td>
<td></td>
</tr>
<tr>
<td>cp862</td>
<td>Hebrew</td>
<td></td>
</tr>
<tr>
<td>cp863</td>
<td>Canadian</td>
<td></td>
</tr>
<tr>
<td>cp864</td>
<td>Arabic</td>
<td></td>
</tr>
<tr>
<td>cp865</td>
<td>Danish, Norwegian</td>
<td></td>
</tr>
<tr>
<td>cp866</td>
<td>Russian</td>
<td></td>
</tr>
<tr>
<td>cp869</td>
<td>Greek</td>
<td></td>
</tr>
<tr>
<td>cp869</td>
<td>Japanese</td>
<td></td>
</tr>
<tr>
<td>cp874</td>
<td>Thai</td>
<td></td>
</tr>
<tr>
<td>cp875</td>
<td>Greek</td>
<td></td>
</tr>
<tr>
<td>cp932</td>
<td>Japanese</td>
<td></td>
</tr>
<tr>
<td>cp949</td>
<td>Korean</td>
<td></td>
</tr>
<tr>
<td>cp950</td>
<td>Traditional Chinese</td>
<td></td>
</tr>
<tr>
<td>cp1006</td>
<td>Urdu</td>
<td></td>
</tr>
<tr>
<td>cp1026</td>
<td>Turkish</td>
<td></td>
</tr>
<tr>
<td>cp1140</td>
<td>Western Europe</td>
<td></td>
</tr>
<tr>
<td>cp1250</td>
<td>Central and Eastern Europe</td>
<td></td>
</tr>
<tr>
<td>cp1251</td>
<td>Bulgarian, Byelorussian, Macedonian, Russian, Serbian</td>
<td></td>
</tr>
<tr>
<td>cp1252</td>
<td>Western Europe</td>
<td></td>
</tr>
<tr>
<td>cp1253</td>
<td>Greek</td>
<td></td>
</tr>
<tr>
<td>cp1254</td>
<td>Turkish</td>
<td></td>
</tr>
<tr>
<td>cp1255</td>
<td>Hebrew</td>
<td></td>
</tr>
<tr>
<td>cp1256</td>
<td>Arabic</td>
<td></td>
</tr>
<tr>
<td>cp1257</td>
<td>Baltic languages</td>
<td></td>
</tr>
<tr>
<td>cp1258</td>
<td>Vietnamese</td>
<td></td>
</tr>
<tr>
<td>euc_jp</td>
<td>Japanese</td>
<td></td>
</tr>
<tr>
<td>euc_jis_2004</td>
<td>jisx0213, eucjis2004</td>
<td></td>
</tr>
<tr>
<td>Encoding</td>
<td>Description</td>
<td>Language(s)</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>euc_jisx0213</td>
<td>eucjisx0213</td>
<td>Japanese</td>
</tr>
<tr>
<td>euc_kr</td>
<td>euckr, korean, ksc5601, ks_c-5601, ks_c-561-1987, ksx1001, ks_x-1001</td>
<td>Korean</td>
</tr>
<tr>
<td>gb2312</td>
<td>chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58</td>
<td>Simplified Chinese</td>
</tr>
<tr>
<td>gbk</td>
<td>936, cp936, ms936</td>
<td>Unified Chinese</td>
</tr>
<tr>
<td>gb18030</td>
<td>gb18030-2000</td>
<td>Unified Chinese</td>
</tr>
<tr>
<td>hz</td>
<td>hzgb, hz-gb, hz-gb-2312</td>
<td>Simplified Chinese</td>
</tr>
<tr>
<td>iso2022 jp</td>
<td>csiso2022jp, iso2022jp, iso-2022-jp</td>
<td>Japanese</td>
</tr>
<tr>
<td>iso2022 jp_1</td>
<td>iso2022jp-1, iso-2022-jp-1</td>
<td>Japanese</td>
</tr>
<tr>
<td>iso2022 jp_2</td>
<td>iso2022jp-2, iso-2022-jp-2</td>
<td>Japanese, Korean, Simplified Chinese, Western Europe, Greek</td>
</tr>
<tr>
<td>iso2022 jp_3</td>
<td>iso2022jp-3, iso-2022-jp-3</td>
<td>Japanese</td>
</tr>
<tr>
<td>iso2022 jp_ext</td>
<td>iso2022jp-ext, iso-2022- jp-ext</td>
<td>Japanese</td>
</tr>
<tr>
<td>iso2022 kr</td>
<td>csiso2022kr, iso2022kr, iso-2022-kr</td>
<td>Korean</td>
</tr>
<tr>
<td>latin_1</td>
<td>iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1</td>
<td>West Europe</td>
</tr>
<tr>
<td>iso8859_2</td>
<td>iso-8859-2, latin2, L2</td>
<td>Central and Eastern Europe</td>
</tr>
<tr>
<td>iso8859_3</td>
<td>iso-8859-3, latin3, L3</td>
<td>Esperanto, Maltese</td>
</tr>
<tr>
<td>iso8859_4</td>
<td>iso-8859-4, latin4, L4</td>
<td>Baltic languages</td>
</tr>
<tr>
<td>iso8859_5</td>
<td>iso-8859-5, cyrillic</td>
<td>Bulgarian, Byelorussian, Macedonian, Russian, Serbian</td>
</tr>
<tr>
<td>iso8859_6</td>
<td>iso-8859-6, arabic</td>
<td>Arabic</td>
</tr>
<tr>
<td>iso8859_7</td>
<td>iso-8859-7, greek, greek8</td>
<td>Greek</td>
</tr>
<tr>
<td>iso8859_8</td>
<td>iso-8859-8, hebrew</td>
<td>Hebrew</td>
</tr>
<tr>
<td>Codec</td>
<td>Encoding</td>
<td>Languages</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>iso8859_9</td>
<td>iso-8859-9, latin5, L5</td>
<td>Turkish</td>
</tr>
<tr>
<td>iso8859_10</td>
<td>iso-8859-10, latin6, L6</td>
<td>Nordic languages</td>
</tr>
<tr>
<td>iso8859_13</td>
<td>iso-8859-13</td>
<td>Baltic languages</td>
</tr>
<tr>
<td>iso8859_14</td>
<td>iso-8859-14, latin8, L8</td>
<td>Celtic languages</td>
</tr>
<tr>
<td>iso8859_15</td>
<td>iso-8859-15</td>
<td>Western Europe</td>
</tr>
<tr>
<td>johab</td>
<td>cp1361, ms1361</td>
<td>Korean</td>
</tr>
<tr>
<td>koi8_r</td>
<td></td>
<td>Russian</td>
</tr>
<tr>
<td>koi8_u</td>
<td></td>
<td>Ukrainian</td>
</tr>
<tr>
<td>mac_cyrillic</td>
<td>maccyrillic</td>
<td>Bulgarian, Byelorussian, Macedonian, Russian, Serbian</td>
</tr>
<tr>
<td>mac_greek</td>
<td>macgreek</td>
<td>Greek</td>
</tr>
<tr>
<td>mac_iceland</td>
<td>maciceland</td>
<td>Icelandic</td>
</tr>
<tr>
<td>mac_latin2</td>
<td>maclatin2, maccentraleurope</td>
<td>Central and Eastern Europe</td>
</tr>
<tr>
<td>mac_roman</td>
<td>macroman</td>
<td>Western Europe</td>
</tr>
<tr>
<td>mac_turkish</td>
<td>macturkish</td>
<td>Turkish</td>
</tr>
<tr>
<td>ptcp154</td>
<td>csptcp154, pt154, cp154, cyrillic-asian</td>
<td>Kazakh</td>
</tr>
<tr>
<td>shift_jis</td>
<td>csshiftjis, shiftjis, sjis, s_jis</td>
<td>Japanese</td>
</tr>
<tr>
<td>shift_jisx0213</td>
<td>shiftjisx0213, sjisx0213, s_jisx0213</td>
<td>Japanese</td>
</tr>
<tr>
<td>utf_16</td>
<td>U16, utf16</td>
<td>all languages</td>
</tr>
<tr>
<td>utf_16_be</td>
<td>UTF-16BE</td>
<td>all languages (BMP only)</td>
</tr>
<tr>
<td>utf_16_le</td>
<td>UTF-16LE</td>
<td>all languages (BMP only)</td>
</tr>
<tr>
<td>utf_7</td>
<td>U7, unicode-1-1-utf-7</td>
<td>all languages</td>
</tr>
<tr>
<td>utf_8</td>
<td>U8, UTF, utf8</td>
<td>all languages</td>
</tr>
<tr>
<td>utf_8_sig</td>
<td></td>
<td>all languages</td>
</tr>
</tbody>
</table>

A number of codecs are specific to Python, so their codec names have no meaning outside Python. Some of them don't convert from Unicode strings to byte strings, but instead use the property of the Python codecs machinery that
any bijective function with one argument can be considered as an encoding.

For the codecs listed below, the result in the ``encoding`` direction is always a byte string. The result of the ``decoding`` direction is listed as operand type in the table.

<table>
<thead>
<tr>
<th>Codec</th>
<th>Aliases</th>
<th>Operand type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64_codec</td>
<td>base64, base-64</td>
<td>byte string</td>
<td>Convert operand to MIME base64</td>
</tr>
<tr>
<td>bz2_codec</td>
<td>bz2</td>
<td>byte string</td>
<td>Compress the operand using bz2</td>
</tr>
<tr>
<td>hex_codec</td>
<td>hex</td>
<td>byte string</td>
<td>Convert operand to hexadecimal representation, with two digits per byte</td>
</tr>
<tr>
<td>mbcsc</td>
<td>dbcs</td>
<td>Unicode string</td>
<td>Windows only: Encode operand according to the ANSI codepage (CP_ACP)</td>
</tr>
<tr>
<td>palmos</td>
<td></td>
<td>Unicode string</td>
<td>Encoding of PalmOS 3.5</td>
</tr>
<tr>
<td>quopri_codec</td>
<td>quopri, quoted-printable, quotedprintable</td>
<td>byte string</td>
<td>Convert operand to MIME quoted printable</td>
</tr>
<tr>
<td>raw_unicode_escape</td>
<td></td>
<td>Unicode string</td>
<td>Produce a string that is suitable as raw Unicode literal in Python source code</td>
</tr>
<tr>
<td>rot_13</td>
<td>rot13</td>
<td>Unicode string</td>
<td>Returns the Caesar-cypher encryption of the operand</td>
</tr>
<tr>
<td>Function</td>
<td>Argument</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>string_escape</td>
<td></td>
<td>byte string</td>
<td>Produce a string that is suitable as string literal in Python source code</td>
</tr>
<tr>
<td>undefined</td>
<td></td>
<td>any</td>
<td>Raise an exception for all conversions. Can be used as the system encoding if no automatic coercion between byte and Unicode strings is desired.</td>
</tr>
<tr>
<td>unicode_escape</td>
<td></td>
<td>Unicode string</td>
<td>Produce a string that is suitable as Unicode literal in Python source code</td>
</tr>
<tr>
<td>unicode_internal</td>
<td></td>
<td>Unicode string</td>
<td>Return the internal representation of the operand</td>
</tr>
<tr>
<td>uu_codec</td>
<td>uu</td>
<td>byte string</td>
<td>Convert the operand using uuencode</td>
</tr>
<tr>
<td>zlib_codec</td>
<td>zip, zlib</td>
<td>byte string</td>
<td>Compress the operand using gzip</td>
</tr>
</tbody>
</table>
4.8.4 encodings.idna -- Internationalized Domain Names in Applications

New in version 2.3.

This module implements RFC 3490 (Internationalized Domain Names in Applications) and RFC 3492 (Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)). It builds upon the punycode encoding and stringprep.

These RFCs together define a protocol to support non-ASCII characters in domain names. A domain name containing non-ASCII characters (such as `www.Alliancefrancaise.nu`) is converted into an ASCII-compatible encoding (ACE, such as `www.xn-alliancefranaise-npb.nu`). The ACE form of the domain name is then used in all places where arbitrary characters are not allowed by the protocol, such as DNS queries, HTTP Host: fields, and so on. This conversion is carried out in the application; if possible invisible to the user: The application should transparently convert Unicode domain labels to IDNA on the wire, and convert back ACE labels to Unicode before presenting them to the user.

Python supports this conversion in several ways: The idna codec allows to convert between Unicode and the ACE. Furthermore, the socket module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the socket module. On top of that, modules that have host names as function parameters, such as httplib and ftplib, accept Unicode host names (httplib then also transparently sends an IDNA hostname in the Host: field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed: Applications wishing to present such host names to the user should decode them to Unicode.

The module encodings.idna also implements the nameprep procedure, which performs certain normalizations on host names, to achieve case-insensitivity of international domain names, and to unify similar characters. The
nameprep functions can be used directly if desired.

**nameprep** *(label)*

Return the nameprepped version of *label*. The implementation currently assumes query strings, so AllowUnassigned is true.

**ToASCII** *(label)*

Convert a label to ASCII, as specified in [RFC 3490]. UseSTD3ASCIIRules is assumed to be false.

**ToUnicode** *(label)*

Convert a label to Unicode, as specified in [RFC 3490].
4.8.5 encodings.utf_8_sig -- UTF-8 codec with BOM signature

New in version 2.5.

This module implements a variant of the UTF-8 codec: On encoding a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). For decoding an optional UTF-8 encoded BOM at the start of the data will be skipped.
4.9 unicodedata -- Unicode Database

This module provides access to the Unicode Character Database which defines character properties for all Unicode characters. The data in this database is based on the UnicodeData.txt file version 4.1.0 which is publicly available from ftp://ftp.unicode.org/.

The module uses the same names and symbols as defined by the UnicodeData File Format 4.1.0 (see http://www.unicode.org/Public/4.1.0/ucd/UCD.html). It defines the following functions:

lookup(name)
Look up character by name. If a character with the given name is found, return the corresponding Unicode character. If not found, KeyError is raised.

name(unichr[, default])
Returns the name assigned to the Unicode character unichr as a string. If no name is defined, default is returned, or, if not given, ValueError is raised.

decimal(unichr[, default])
Returns the decimal value assigned to the Unicode character unichr as integer. If no such value is defined, default is returned, or, if not given, ValueError is raised.

digit(unichr[, default])
Returns the digit value assigned to the Unicode character unichr as integer. If no such value is defined, default is returned, or, if not given, ValueError is raised.

numeric(unichr[, default])
Returns the numeric value assigned to the Unicode character unichr as
float. If no such value is defined, *default* is returned, or, if not given, *ValueError* is raised.

**category**(*unichr*)

Returns the general category assigned to the Unicode character *unichr* as string.

**bidirectional**(*unichr*)

Returns the bidirectional category assigned to the Unicode character *unichr* as string. If no such value is defined, an empty string is returned.

**combining**(*unichr*)

Returns the canonical combining class assigned to the Unicode character *unichr* as integer. Returns 0 if no combining class is defined.

**east_asian_width**(*unichr*)

Returns the east asian width assigned to the Unicode character *unichr* as string. New in version 2.4.

**mirrored**(*unichr*)

Returns the mirrored property assigned to the Unicode character *unichr* as integer. Returns 1 if the character has been identified as a "mirrored" character in bidirectional text, 0 otherwise.

**decomposition**(*unichr*)

Returns the character decomposition mapping assigned to the Unicode character *unichr* as string. An empty string is returned in case no such mapping is defined.

**normalize**(*form, unistr*)

Return the normal form *form* for the Unicode string *unistr*. Valid values for *form* are 'NFC', 'NFKC', 'NFD', and 'NFKD'.

The Unicode standard defines various normalization forms of a Unicode string, based on the definition of canonical equivalence and compatibility equivalence. In Unicode, several characters can be expressed in various way. For example, the character U+00C7 (LATIN CAPITAL LETTER C
WITH CEDILLA) can also be expressed as the sequence U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA).

For each character, there are two normal forms: normal form C and normal form D. Normal form D (NFD) is also known as canonical decomposition, and translates each character into its decomposed form. Normal form C (NFC) first applies a canonical decomposition, then composes pre-combined characters again.

In addition to these two forms, there are two additional normal forms based on compatibility equivalence. In Unicode, certain characters are supported which normally would be unified with other characters. For example, U+2160 (ROMAN NUMERAL ONE) is really the same thing as U+0049 (LATIN CAPITAL LETTER I). However, it is supported in Unicode for compatibility with existing character sets (e.g. gb2312).

The normal form KD (NFKD) will apply the compatibility decomposition, i.e. replace all compatibility characters with their equivalents. The normal form KC (NFKC) first applies the compatibility decomposition, followed by the canonical composition.

New in version 2.3.

In addition, the module exposes the following constant:

```
unidata_version
The version of the Unicode database used in this module.

New in version 2.3.
```

```
ucd_3_2_0
This is an object that has the same methods as the entire module, but uses the Unicode database version 3.2 instead, for applications that require this specific version of the Unicode database (such as IDNA).

New in version 2.5.
```

Examples:
```
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'u'\'{
```
>>> unicodedata.name(u'/')
'SOLIDUS'
>>> unicodedata.decimal(u'9')
9
>>> unicodedata.decimal(u'a')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: not a decimal
>>> unicodedata.category(u'A') # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional(u'\u0660') # 'A'rabic, 'N'umber
'AN'

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.10 stringprep -- Internet String Preparation

New in version 2.3.

When identifying things (such as host names) in the internet, it is often necessary to compare such identifications for ``equality''. Exactly how this comparison is executed may depend on the application domain, e.g. whether it should be case-insensitive or not. It may be also necessary to restrict the possible identifications, to allow only identifications consisting of ``printable'' characters.

RFC 3454 defines a procedure for ``preparing'' Unicode strings in internet protocols. Before passing strings onto the wire, they are processed with the preparation procedure, after which they have a certain normalized form. The RFC defines a set of tables, which can be combined into profiles. Each profile must define which tables it uses, and what other optional parts of the stringprep procedure are part of the profile. One example of a stringprep profile is nameprep, which is used for internationalized domain names.

The module stringprep only exposes the tables from RFC 3454. As these tables would be very large to represent them as dictionaries or lists, the module uses the Unicode character database internally. The module source code itself was generated using the mkstringprep.py utility.

As a result, these tables are exposed as functions, not as data structures. There are two kinds of tables in the RFC: sets and mappings. For a set, stringprep provides the ``characteristic function'', i.e. a function that returns true if the parameter is part of the set. For mappings, it provides the mapping function: given the key, it returns the associated value. Below is a list of all functions available in the module.

**in_table_a1**(code)

Determine whether *code* is in tableA.1 (Unassigned code points in Unicode 3.2).
`in_table_b1(code)`
Determine whether `code` is in tableB.1 (Commonly mapped to nothing).

`map_table_b2(code)`
Return the mapped value for `code` according to tableB.2 (Mapping for case-folding used with NFKC).

`map_table_b3(code)`
Return the mapped value for `code` according to tableB.3 (Mapping for case-folding used with no normalization).

`in_table_c11(code)`
Determine whether `code` is in tableC.1.1 (ASCII space characters).

`in_table_c12(code)`
Determine whether `code` is in tableC.1.2 (Non-ASCII space characters).

`in_table_c11_c12(code)`
Determine whether `code` is in tableC.1 (Space characters, union of C.1.1 and C.1.2).

`in_table_c21(code)`
Determine whether `code` is in tableC.2.1 (ASCII control characters).

`in_table_c22(code)`
Determine whether `code` is in tableC.2.2 (Non-ASCII control characters).

`in_table_c21_c22(code)`
Determine whether `code` is in tableC.2 (Control characters, union of C.2.1 and C.2.2).

`in_table_c3(code)`
Determine whether `code` is in tableC.3 (Private use).

`in_table_c4(code)`
Determine whether `code` is in tableC.4 (Non-character code points).
**in_table_c5**(*code*)
Determine whether *code* is in table C.5 (Surrogate codes).

**in_table_c6**(*code*)
Determine whether *code* is in table C.6 (Inappropriate for plain text).

**in_table_c7**(*code*)
Determine whether *code* is in table C.7 (Inappropriate for canonical representation).

**in_table_c8**(*code*)
Determine whether *code* is in table C.8 (Change display properties or are deprecated).

**in_table_c9**(*code*)
Determine whether *code* is in table C.9 (Tagging characters).

**in_table_d1**(*code*)
Determine whether *code* is in table D.1 (Characters with bidirectional property ``R'' or ``AL '').

**in_table_d2**(*code*)
Determine whether *code* is in table D.2 (Characters with bidirectional property ``L '').
4.11 fpformat -- Floating point conversions

The fpformat module defines functions for dealing with floating point numbers representations in 100% pure Python. **Note:** This module is unneeded: everything here could be done via the % string interpolation operator.

The fpformat module defines the following functions and an exception:

**fix(x, digs)**
Format x as [-]ddd.ddd with digs digits after the point and at least one digit before. If digs <= 0, the decimal point is suppressed.

x can be either a number or a string that looks like one. digs is an integer.

Return value is a string.

**sci(x, digs)**
Format x as [-]d.dddE[+-]ddd with digs digits after the point and exactly one digit before. If digs <= 0, one digit is kept and the point is suppressed.

x can be either a real number, or a string that looks like one. digs is an integer.

Return value is a string.

**exception NotANumber**
Exception raised when a string passed to fix() or sci() as the x parameter does not look like a number. This is a subclass of ValueError when the standard exceptions are strings. The exception value is the improperly formatted string that caused the exception to be raised.

Example:

```python
>>> import fpformat
```
>>> fpformat.fix(1.23, 1)
'1.2'
5. Data Types

The modules described in this chapter provide a variety of specialized data types such as dates and times, fixed-type arrays, heap queues, synchronized queues, and sets.

The following modules are documented in this chapter:

- **datetime**: Basic date and time types.
- **calendar**: Functions for working with calendars, including some emulation of the UNIX `cal` program.
- **collections**: High-performance datatypes.
- **heapq**: Heap queue algorithm (a.k.a. priority queue).
- **bisect**: Array bisection algorithms for binary searching.
- **array**: Efficient arrays of uniformly typed numeric values.
- **sets**: Implementation of sets of unique elements.
- **sched**: General purpose event scheduler.
- **mutex**: Lock and queue for mutual exclusion.
- **Queue**: A synchronized queue class.
- **weakref**: Support for weak references and weak dictionaries.
- **UserDict**: Class wrapper for dictionary objects.
- **UserList**: Class wrapper for list objects.
- **UserString**: Class wrapper for string objects.
- **types**: Names for built-in types.
- **new**: Interface to the creation of runtime implementation objects.
- **copy**: Shallow and deep copy operations.
- **pprint**: Data pretty printer.
- **repr**: Alternate `repr()` implementation with size limits.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.1 datetime -- Basic date and time types

New in version 2.3.

The datetime module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation.

There are two kinds of date and time objects: "naive" and "aware". This distinction refers to whether the object has any notion of time zone, daylight saving time, or other kind of algorithmic or political time adjustment. Whether a naive datetime object represents Coordinated Universal Time (UTC), local time, or time in some other timezone is purely up to the program, just like it's up to the program whether a particular number represents metres, miles, or mass. Naive datetime objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

For applications requiring more, datetime and time objects have an optional time zone information member, tzinfo, that can contain an instance of a subclass of the abstract tzinfo class. These tzinfo objects capture information about the offset from UTC time, the time zone name, and whether Daylight Saving Time is in effect. Note that no concrete tzinfo classes are supplied by the datetime module. Supporting timezones at whatever level of detail is required is up to the application. The rules for time adjustment across the world are more political than rational, and there is no standard suitable for every application.

The datetime module exports the following constants:

MINYEAR
   The smallest year number allowed in a date or datetime object.
   MINYEAR is 1.

MAXYEAR
The largest year number allowed in a date or datetime object. MAXYEAR is 9999.

See Also:

Module calendar:
General calendar related functions.

Module time:
Time access and conversions.
5.1.1 Available Types

class date
An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: year, month, and day.

class time
An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds (there is no notion of "leap seconds" here). Attributes: hour, minute, second, microsecond, and tzinfo.

class datetime
A combination of a date and a time. Attributes: year, month, day, hour, minute, second, microsecond, and tzinfo.

class timedelta
A duration expressing the difference between two date, time, or datetime instances to microsecond resolution.

class tzinfo
An abstract base class for time zone information objects. These are used by the datetime and time classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

Objects of these types are immutable.

Objects of the date type are always naive.

An object \( d \) of type time or datetime may be naive or aware. \( d \) is aware if \( d.tzinfo \) is not None and \( d.tzinfo.utcoffset(d) \) does not return None. If \( d.tzinfo \) is None, or if \( d.tzinfo \) is not None but \( d.tzinfo.utcoffset(d) \) returns None, \( d \) is naive.

The distinction between naive and aware doesn't apply to timedelta objects.
Subclass relationships:

- object
- timedelta
- tzinfo
- time
- date
- datetime
5.1.2 timedelta Objects

A timedelta object represents a duration, the difference between two dates or times.

class timedelta([days[, seconds[, microseconds[, milliseconds[, minutes[, hours[, weeks]]]]]])

All arguments are optional and default to 0. Arguments may be ints, longs, or floats, and may be positive or negative.

Only days, seconds and microseconds are stored internally. Arguments are converted to those units:

- A millisecond is converted to 1000 microseconds.
- A minute is converted to 60 seconds.
- An hour is converted to 3600 seconds.
- A week is converted to 7 days.

and days, seconds and microseconds are then normalized so that the representation is unique, with

- $0 \leq \text{microseconds} < 1000000$
- $0 \leq \text{seconds} < 3600 \times 24$ (the number of seconds in one day)
- $-999999999 \leq \text{days} \leq 999999999$

If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond. If no argument is a float, the conversion and normalization processes are exact (no information is lost).

If the normalized value of days lies outside the indicated range, OverflowError is raised.

Note that normalization of negative values may be surprising at first. For example,

```python
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
```
Class attributes are:

**min**

The most negative `timedelta` object, `timedelta(-999999999)`.

**max**

The most positive `timedelta` object, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

**resolution**

The smallest possible difference between non-equal `timedelta` objects, `timedelta(microseconds=1)`.

Note that, because of normalization, `timedelta.max > -timedelta.min`. `-timedelta.max` is not representable as a `timedelta` object.

Instance attributes (read-only):

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>days</td>
<td>Between -999999999 and 999999999 inclusive</td>
</tr>
<tr>
<td>seconds</td>
<td>Between 0 and 86399 inclusive</td>
</tr>
<tr>
<td>microseconds</td>
<td>Between 0 and 999999 inclusive</td>
</tr>
</tbody>
</table>

Supported operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t1 = t2 + t3</code></td>
<td>Sum of <code>t2</code> and <code>t3</code>. Afterwards <code>t1-t2 == t3</code> and <code>t1-t3 == t2</code> are true. (1)</td>
</tr>
<tr>
<td><code>t1 = t2 - t3</code></td>
<td>Difference of <code>t2</code> and <code>t3</code>. Afterwards <code>t1 == t2 - t3</code> and <code>t2 == t1 + t3</code> are true. (1)</td>
</tr>
<tr>
<td><code>t1 = t2 * i</code> or <code>t1 = i * t2</code></td>
<td>Delta multiplied by an integer or long. Afterwards <code>t1 // i == t2</code> is true, provided <code>i != 0</code>.</td>
</tr>
<tr>
<td><code>t1 = t2 // i</code></td>
<td>In general, <code>t1 * i == t1 * (i-1) + t1</code> is true. (1)</td>
</tr>
<tr>
<td><code>t1 = t2 // i</code></td>
<td>The floor is computed and the remainder (if any) is thrown away. (3)</td>
</tr>
<tr>
<td>Operation</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>$+t_1$</td>
<td>Returns a <code>timedelta</code> object with the same value. (2)</td>
</tr>
<tr>
<td>$-t_1$</td>
<td>Equivalent to <code>timedelta(-t1.days, -t1.seconds, -t1.microseconds)</code>, and to $t1**-1$. (1)(4)</td>
</tr>
<tr>
<td><code>abs(t)</code></td>
<td>Equivalent to $+t$ when $t.days \geq 0$, and to $-t$ when $t.days &lt; 0$. (2)</td>
</tr>
</tbody>
</table>

Notes:

(1) This is exact, but may overflow.

(2) This is exact, and cannot overflow.

(3) Division by 0 raises `ZeroDivisionError`.

(4) `$timedelta\.max$` is not representable as a `timedelta` object.

In addition to the operations listed above `timedelta` objects support certain additions and subtractions with `date` and `datetime` objects (see below).

Comparisons of `timedelta` objects are supported with the `timedelta` object representing the smaller duration considered to be the smaller `timedelta`. In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a `timedelta` object is compared to an object of a different type, `TypeError` is raised unless the comparison is `==` or `!=`. The latter cases return `False` or `True`, respectively.

timedelta objects are hashable (usable as dictionary keys), support efficient pickling, and in Boolean contexts, a `timedelta` object is considered to be true if and only if it isn't equal to `timedelta(0)`.
5.1.3 date Objects

A date object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions. January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on. This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold’s book *Calendrical Calculations*, where it's the base calendar for all computations. See the book for algorithms for converting between proleptic Gregorian ordinals and many other calendar systems.

```python
class date(year, month, day)
```

All arguments are required. Arguments may be ints or longs, in the following ranges:

- MINYEAR <= year <= MAXYEAR
- 1 <= month <= 12
- 1 <= day <= number of days in the given month and year

If an argument outside those ranges is given, ValueError is raised.

Other constructors, all class methods:

```python
today()
```

Return the current local date. This is equivalent to
```
date.fromtimestamp(time.time()).
```

```python
fromtimestamp(timestamp)
```

Return the local date corresponding to the POSIX timestamp, such as is returned by `time.time()`. This may raise ValueError, if the timestamp is out of the range of values supported by the platform C `localtime()` function. It’s common for this to be restricted to years from 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by `fromtimestamp()`.
fromordinal\,(\textit{ordinal})

Return the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. \texttt{ValueError} is raised unless \(1 \leq \textit{ordinal} \leq \text{date.max.toordinal}()\). For any date \(d\), \(\text{date.fromordinal}(d\text{.toordinal}()) == d\).

Class attributes:

\texttt{min}

The earliest representable date, \(\text{date(MINYEAR, 1, 1)}\).

\texttt{max}

The latest representable date, \(\text{date(MAXYEAR, 12, 31)}\).

\texttt{resolution}

The smallest possible difference between non-equal date objects, \(\text{timedelta(days=1)}\).

Instance attributes (read-only):

\texttt{year}

Between MINYEAR and MAXYEAR inclusive.

\texttt{month}

Between 1 and 12 inclusive.

\texttt{day}

Between 1 and the number of days in the given month of the given year.

Supported operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(date2 = date1 + \text{timedelta})</td>
<td>(date2) is (\text{timedelta}). \texttt{days} \texttt{days} removed from (date1). (1)</td>
</tr>
<tr>
<td>(date2 = date1 - \text{timedelta})</td>
<td>Computes (date2) such that (date2 + \text{timedelta} == date1). (2)</td>
</tr>
<tr>
<td>(\text{timedelta} = date1 - date2)</td>
<td>(3)</td>
</tr>
<tr>
<td>(date1 &lt; date2)</td>
<td>(date1) is considered less than (date2) when (date1)</td>
</tr>
</tbody>
</table>
Notes:

(1) \(\text{date2}\) is moved forward in time if \(\text{timedelta}.\text{days} > 0\), or backward if \(\text{timedelta}.\text{days} < 0\). Afterward \(\text{date2} - \text{date1} == \text{timedelta}.\text{days}\). \(\text{timedelta}.\text{seconds}\) and \(\text{timedelta}.\text{microseconds}\) are ignored. \text{OverflowError} is raised if \(\text{date2}.\text{year}\) would be smaller than \(\text{MINYEAR}\) or larger than \(\text{MAXYEAR}\).

(2) This isn't quite equivalent to \(\text{date1} + (-\text{timedelta})\), because -\(\text{timedelta}\) in isolation can overflow in cases where \(\text{date1} - \text{timedelta}\) does not. \(\text{timedelta}.\text{seconds}\) and \(\text{timedelta}.\text{microseconds}\) are ignored.

(3) This is exact, and cannot overflow. \(\text{timedelta}.\text{seconds}\) and \(\text{timedelta}.\text{microseconds}\) are 0, and \(\text{date2} + \text{timedelta} == \text{date1}\) after.

(4) In other words, \(\text{date1} < \text{date2}\) if and only if \(\text{date1}.\text{toordinal}() < \text{date2}.\text{toordinal}()\). In order to stop comparison from falling back to the default scheme of comparing object addresses, date comparison normally raises \text{TypeError} if the other comparand isn't also a \text{date} object. However, \text{NotImplemented} is returned instead if the other comparand has a \text{timetuple} attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a \text{date} object is compared to an object of a different type, \text{TypeError} is raised unless the comparison is == or !=. The latter cases return \text{False} or \text{True}, respectively.

Dates can be used as dictionary keys. In Boolean contexts, all \text{date} objects are considered to be true.

Instance methods:

\text{replace}(\text{year, month, day})

Return a date with the same value, except for those members given new
values by whichever keyword arguments are specified. For example, if d
== date(2002, 12, 31), then d.replace(day=26) ==
date(2002, 12, 26).

timetuple()
Return a time.struct_time such as returned by
time.localtime(). The hours, minutes and seconds are 0, and the
DST flag is -1. d.timetuple() is equivalent to
time.struct_time((d.year, d.month, d.day, 0, 0, 0,
d.weekday(), d.toordinal() - date(d.year, 1,
1).toordinal() + 1, -1))

toordinal()
Return the proleptic Gregorian ordinal of the date, where January 1 of year
1 has ordinal 1. For any date object d,
date.fromordinal(d.toordinal()) == d.

weekday()
Return the day of the week as an integer, where Monday is 0 and Sunday is
6. For example, date(2002, 12, 4).weekday() == 2, a
Wednesday. See also isoweekday().

isoweekday()
Return the day of the week as an integer, where Monday is 1 and Sunday is
7. For example, date(2002, 12, 4).isoweekday() == 3, a
Wednesday. See also weekday(), isocalendar().

isocalendar()
Return a 3-tuple, (ISO year, ISO week number, ISO weekday).

The ISO calendar is a widely used variant of the Gregorian calendar. See
http://www.phys.uu.nl/~vgent/calendar/isocalendar.htm for a good
explanation.

The ISO year consists of 52 or 53 full weeks, and where a week starts on a
Monday and ends on a Sunday. The first week of an ISO year is the first
(Gregorian) calendar week of a year containing a Thursday. This is called
week number 1, and the ISO year of that Thursday is the same as its Gregorian year.


**isoformat()**

Return a string representing the date in ISO 8601 format, 'YYYY-MM-DD'. For example, date(2002, 12, 4).isoformat() == '2002-12-04'.

**__str__()**

For a date d, str(d) is equivalent to d.isoformat().

**ctime()**

Return a string representing the date, for example date(2002, 12, 4).ctime() == 'Wed Dec 4 00:00:00 2002'. d.ctime() is equivalent to time.ctime(time.mktime(d.timetuple())) on platforms where the native C ctime() function (which time.ctime() invokes, but which date.ctime() does not invoke) conforms to the C standard.

**strftime(format)**

Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values. See section 5.1.7 - strftime() behavior.
5.1.4 datetime Objects

A datetime object is a single object containing all the information from a date object and a time object. Like a date object, datetime assumes the current Gregorian calendar extended in both directions; like a time object, datetime assumes there are exactly 3600*24 seconds in every day.

Constructor:

class datetime(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]])

The year, month and day arguments are required. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments may be ints or longs, in the following ranges:

- MINYEAR <= year <= MAXYEAR
- 1 <= month <= 12
- 1 <= day <= number of days in the given month and year
- 0 <= hour < 24
- 0 <= minute < 60
- 0 <= second < 60
- 0 <= microsecond < 1000000

If an argument outside those ranges is given, ValueError is raised.

Other constructors, all class methods:

today()

Return the current local datetime, with tzinfo None. This is equivalent to datetime.fromtimestamp(time.time()). See also now(), fromtimestamp().

now([tz])

Return the current local date and time. If optional argument tz is None or not specified, this is like today(), but, if possible, supplies more
precision than can be gotten from going through a \texttt{time.time()} timestamp (for example, this may be possible on platforms supplying the C \texttt{gettimeofday()} function).

Else \texttt{tz} must be an instance of a class \texttt{tzinfo} subclass, and the current date and time are converted to \texttt{tz}'s time zone. In this case the result is equivalent to \texttt{tz.fromutc(datetime.utcnow().replace(tzinfo=tz))}. See also \texttt{today()}, \texttt{utcnow()}.

\texttt{utcnow()}

Return the current UTC date and time, with \texttt{tzinfo None}. This is like \texttt{now()}, but returns the current UTC date and time, as a naive \texttt{datetime} object. See also \texttt{now()}.

\texttt{fromtimestamp( timestamp[, tz])}

Return the local date and time corresponding to the POSIX timestamp, such as is returned by \texttt{time.time()}. If optional argument \texttt{tz} is \texttt{None} or not specified, the timestamp is converted to the platform's local date and time, and the returned \texttt{datetime} object is naive.

Else \texttt{tz} must be an instance of a class \texttt{tzinfo} subclass, and the timestamp is converted to \texttt{tz}'s time zone. In this case the result is equivalent to \texttt{tz.fromutc(datetime.utctfromtimestamp(timestamp).replace(tzinfo=tz))}. \texttt{fromtimestamp()} may raise \texttt{ValueError}, if the timestamp is out of the range of values supported by the platform C \texttt{localtime()} or \texttt{gmtime()} functions. It's common for this to be restricted to years in 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by \texttt{fromtimestamp()}, and then it's possible to have two timestamps differing by a second that yield identical \texttt{datetime} objects. See also \texttt{utcffromtimestamp()}.

\texttt{utcffromtimestamp( timestamp)}

Return the UTC \texttt{datetime} corresponding to the POSIX timestamp, with \texttt{tzinfo None}. This may raise \texttt{ValueError}, if the timestamp is out of
the range of values supported by the platform C `gmtime()` function. It's common for this to be restricted to years in 1970 through 2038. See also `fromtimestamp()`.

**fromordinal**(*ordinal*)

Return the `datetime` corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. `ValueError` is raised unless `1 <= ordinal <= datetime.max.toordinal()`. The hour, minute, second and microsecond of the result are all 0, and `tzinfo` is `None`.

**combine**(*date, time*)

Return a new `datetime` object whose date members are equal to the given `date` object's, and whose time and `tzinfo` members are equal to the given `time` object's. For any `datetime` object `d`, `d == datetime.combine(d.date(), d.timetz())`. If `date` is a `datetime` object, its time and `tzinfo` members are ignored.

**strptime**(*date_string, format*)

Return a `datetime` corresponding to `date_string`, parsed according to `format`. This is equivalent to `datetime(* (time.strptime(date_string, format)[0:6]))`. `ValueError` is raised if the `date_string` and `format` can't be parsed by `time.strptime()` or if it returns a value which isn't a time tuple.

New in version 2.5.

Class attributes:

**min**

The earliest representable `datetime`, `datetime(MINYEAR, 1, 1, tzinfo=None)`.

**max**

The latest representable `datetime`, `datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)`.

**resolution**

The smallest possible difference between non-equal `datetime` objects,
timedelta(microseconds=1).

Instance attributes (read-only):

year
   Between MINYEAR and MAXYEAR inclusive.

month
   Between 1 and 12 inclusive.

day
   Between 1 and the number of days in the given month of the given year.

hour
   In range(24).

minute
   In range(60).

second
   In range(60).

microsecond
   In range(1000000).

tzinfo
   The object passed as the tzinfo argument to the datetime constructor, or None if none was passed.

Supported operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datetime2 = datetime1 + timedelta</code></td>
<td>(1)</td>
</tr>
<tr>
<td><code>datetime2 = datetime1 - timedelta</code></td>
<td>(2)</td>
</tr>
<tr>
<td><code>timedelta = datetime1 - datetime2</code></td>
<td>(3)</td>
</tr>
<tr>
<td><code>datetime1 &lt; datetime2</code></td>
<td>Compares datetime to datetime.</td>
</tr>
</tbody>
</table>
datetime2 is a duration of timedelta removed from datetime1, moving forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. The result has the same `tzinfo` member as the input datetime, and `datetime2 - datetime1 == timedelta` after. `OverflowError` is raised if `datetime2.year` would be smaller than `MINYEAR` or larger than `MAXYEAR`. Note that no time zone adjustments are done even if the input is an aware object.

Computes the datetime2 such that `datetime2 + timedelta == datetime1`. As for addition, the result has the same `tzinfo` member as the input datetime, and no time zone adjustments are done even if the input is aware. This isn't quite equivalent to `datetime1 + (-timedelta)`, because `-timedelta` in isolation can overflow in cases where `datetime1 - timedelta` does not.

Subtraction of a `datetime` from a `datetime` is defined only if both operands are naive, or if both are aware. If one is aware and the other is naive, `TypeError` is raised.

If both are naive, or both are aware and have the same `tzinfo` member, the `tzinfo` members are ignored, and the result is a `timedelta` object `t` such that `datetime2 + t == datetime1`. No time zone adjustments are done in this case.

If both are aware and have different `tzinfo` members, `a - b` acts as if `a` and `b` were first converted to naive UTC datetimes first. The result is `(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` except that the implementation never overflows.

`datetime1` is considered less than `datetime2` when `datetime1` precedes `datetime2` in time.
If one comparand is naive and the other is aware, TypeError is raised. If both comparands are aware, and have the same tzinfo member, the common tzinfo member is ignored and the base datetimes are compared. If both comparands are aware and have different tzinfo members, the comparands are first adjusted by subtracting their UTC offsets (obtained from self.utcoffset()). Note: In order to stop comparison from falling back to the default scheme of comparing object addresses, datetime comparison normally raises TypeError if the other comparand isn't also a datetime object. However, NotImplementedError is returned instead if the other comparand has a timetuple attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a datetime object is compared to an object of a different type, TypeError is raised unless the comparison is == or !=. The latter cases return False or True, respectively.

datetime objects can be used as dictionary keys. In Boolean contexts, all datetime objects are considered to be true.

Instance methods:

**date()**
Return date object with same year, month and day.

**time()**
Return time object with same hour, minute, second and microsecond. tzinfo is None. See also method timetz().

**timetz()**
Return time object with same hour, minute, second, microsecond, and tzinfo members. See also method time().

**replace([year[, month[, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]]]]])**
Return a datetime with the same members, except for those members given new values by whichever keyword arguments are specified. Note that tzinfo=None can be specified to create a naive datetime from an aware datetime with no conversion of date and time members.
astimezone(tz)

Return a datetime object with new tzinfo member tz, adjusting the
date and time members so the result is the same UTC time as self, but in tz's
local time.

tz must be an instance of a tzinfo subclass, and its utcoffset() and
dst() methods must not return None. self must be aware (self.tzinfo
must not be None, and self.utcoffset() must not return None).

If self.tzinfo is tz, self.astimezone(tz) is equal to self: no
adjustment of date or time members is performed. Else the result is local
time in time zone tz, representing the same UTC time as self: after astz =
dt.astimezone(tz), astz - astz.utcoffset() will usually have the
same date and time members as dt - dt.utcoffset(). The discussion
of class tzinfo explains the cases at Daylight Saving Time transition
boundaries where this cannot be achieved (an issue only if tz models both
standard and daylight time).

If you merely want to attach a time zone object tz to a datetime dt without
adjustment of date and time members, use dt.replace(tzinfo=tz). If
you merely want to remove the time zone object from an aware datetime dt
without conversion of date and time members, use
dt.replace(tzinfo=None).

Note that the default tzinfo.fromutc() method can be overridden in a
tzinfo subclass to affect the result returned by astimezone().
Ignoring error cases, astimezone() acts like:

def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new time zone object
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)

utcoffset()

If tzinfo is None, returns None, else returns
self.tzinfo.utcoffset(self), and raises an exception if the latter
doesn't return None, or a timedelta object representing a whole number
of minutes with magnitude less than one day.

**dst()**

If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.dst(self)`, and raises an exception if the latter doesn't return `None`, or a `timedelta` object representing a whole number of minutes with magnitude less than one day.

**tzname()**

If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.tzname(self)`, raises an exception if the latter doesn't return `None` or a string object,

**timetuple()**

Return a `time.struct_time` such as returned by

```
    time.localtime(d).timetuple()
```

is equivalent to

```
    time.localtime((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), d.toordinal() - date(d.year, 1, 1).toordinal() + 1, dst))
```

The `tm_isdst` flag of the result is set according to the `dst()` method:

- `tzinfo` is `None` or `dst()` returns `None`, `tm_isdst` is set to `-1`;
- else if `dst()` returns a non-zero value, `tm_isdst` is set to `1`;
- else `tm_isdst` is set to `0`.

**utctimetuple()**

If `datetime` instance `d` is naive, this is the same as `d.timetuple()` except that `tm_isdst` is forced to `0` regardless of what `d.dst()` returns. DST is never in effect for a UTC time.

If `d` is aware, `d` is normalized to UTC time, by subtracting `d.utcoffset()`, and a `time.struct_time` for the normalized time is returned. `tm_isdst` is forced to `0`. Note that the result's `tm_year` member may be `MINYEAR-1` or `MAXYEAR+1`, if `d.year` was `MINYEAR` or `MAXYEAR` and UTC adjustment spills over a year boundary.

**toordinal()**

Return the proleptic Gregorian ordinal of the date. The same as
self.date().toordinal().

weekday()
Return the day of the week as an integer, where Monday is 0 and Sunday is 6. The same as self.date().weekday(). See also isoweekday().

isoweekday()
Return the day of the week as an integer, where Monday is 1 and Sunday is 7. The same as self.date().isoweekday(). See also weekday(), isocalendar().

isocalendar()
Return a 3-tuple, (ISO year, ISO week number, ISO weekday). The same as self.date().isocalendar().

isoformat([sep])
Return a string representing the date and time in ISO 8601 format, YYYY-MM-DDTHH:MM:SS.mmmmmm or, if microsecond is 0, YYYY-MM-DDTHH:MM:SS

If utcoffset() does not return None, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM or, if microsecond is 0 YYYY-MM-DDTHH:MM:SS+HH:MM

The optional argument sep (default 'T') is a one-character separator, placed between the date and time portions of the result. For example,

```python
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(minutes=-399)
...     ...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat('T')
'2002-12-25 00:00:00-06:39'
```

__str__()
For a datetime instance d, str(d) is equivalent to d.isoformat('T').
ctime()

Return a string representing the date and time, for example
datetime(2002, 12, 4, 20, 30, 40).ctime() == 'Wed
Dec 4 20:30:40 2002'. d.ctime() is equivalent to
time.ctime(time.mktime(d.timetuple())) on platforms
where the native C ctime() function (which time.ctime() invokes,
but which datetime.ctime() does not invoke) conforms to the C
standard.

strftime( format)

Return a string representing the date and time, controlled by an explicit
format string. See section 5.1.7 - strftime() behavior.
5.1.5 time Objects

A time object represents a (local) time of day, independent of any particular day, and subject to adjustment via a tzinfo object.

```python
class time(hour[, minute[, second[, microsecond[, tzinfo]]]]):
```

All arguments are optional. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments may be ints or longs, in the following ranges:

- \(0 \leq \text{hour} < 24\)
- \(0 \leq \text{minute} < 60\)
- \(0 \leq \text{second} < 60\)
- \(0 \leq \text{microsecond} < 1000000\).

If an argument outside those ranges is given, `ValueError` is raised. All default to 0 except `tzinfo`, which defaults to None.

Class attributes:

- `min`
  The earliest representable time, `time(0, 0, 0, 0)`.

- `max`
  The latest representable time, `time(23, 59, 59, 999999)`.

- `resolution`
  The smallest possible difference between non-equal time objects, `timedelta(microseconds=1)`, although note that arithmetic on time objects is not supported.

Instance attributes (read-only):

- `hour`
  In `range(24)`.

- `minute`
The object passed as the tzinfo argument to the time constructor, or None if none was passed.

Supported operations:

- comparison of time to time, where a is considered less than b when a precedes b in time. If one comparand is naive and the other is aware, TypeError is raised. If both comparands are aware, and have the same tzinfo member, the common tzinfo member is ignored and the base times are compared. If both comparands are aware and have different tzinfo members, the comparands are first adjusted by subtracting their UTC offsets (obtained from self.utcoffset()). In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a time object is compared to an object of a different type, TypeError is raised unless the comparison is == or !=. The latter cases return False or True, respectively.

- hash, use as dict key

- efficient pickling

- in Boolean contexts, a time object is considered to be true if and only if, after converting it to minutes and subtracting utcoffset() (or 0 if that's None), the result is non-zero.

Instance methods:

replace([hour[, minute[, second[, microsecond[, tzinfo]]]]]])

Return a time with the same value, except for those members given new values by whichever keyword arguments are specified. Note that
tzinfo=None can be specified to create a naive time from an aware time, without conversion of the time members.

**isoformat()**

Return a string representing the time in ISO 8601 format, HH:MM:SS.mmmmmm or, if self.microsecond is 0, HH:MM:SS If utcoffset() does not return None, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: HH:MM:SS.mmmmmm+HH:MM or, if self.microsecond is 0, HH:MM:SS+HH:MM

```
__str__()
```

For a time t, str(t) is equivalent to t.isoformat().

**strftime**(format)

Return a string representing the time, controlled by an explicit format string. See section 5.17 - strftime() behavior.

**utcoffset()**

If tzinfo is None, returns None, else returns self.tzinfo.utcoffset(None), and raises an exception if the latter doesn't return None or a timedelta object representing a whole number of minutes with magnitude less than one day.

**dst()**

If tzinfo is None, returns None, else returns self.tzinfo.dst(None), and raises an exception if the latter doesn't return None, or a timedelta object representing a whole number of minutes with magnitude less than one day.

**tzname()**

If tzinfo is None, returns None, else returns self.tzinfo.tzname(None), or raises an exception if the latter doesn't return None or a string object.
5.1.6 tzinfo Objects

tzinfo is an abstract base class, meaning that this class should not be instantiated directly. You need to derive a concrete subclass, and (at least) supply implementations of the standard tzinfo methods needed by the datetime methods you use. The datetime module does not supply any concrete subclasses of tzinfo.

An instance of (a concrete subclass of) tzinfo can be passed to the constructors for datetime and time objects. The latter objects view their members as being in local time, and the tzinfo object supports methods revealing offset of local time from UTC, the name of the time zone, and DST offset, all relative to a date or time object passed to them.

Special requirement for pickling: A tzinfo subclass must have an __init__ method that can be called with no arguments, else it can be pickled but possibly not unpickled again. This is a technical requirement that may be relaxed in the future.

A concrete subclass of tzinfo may need to implement the following methods. Exactly which methods are needed depends on the uses made of aware datetime objects. If in doubt, simply implement all of them.

**utcoffset**(self, dt)

Return offset of local time from UTC, in minutes east of UTC. If local time is west of UTC, this should be negative. Note that this is intended to be the total offset from UTC; for example, if a tzinfo object represents both time zone and DST adjustments, utcoffset() should return their sum. If the UTC offset isn't known, return None. Else the value returned must be a timedelta object specifying a whole number of minutes in the range -1439 to 1439 inclusive (1440 = 24*60; the magnitude of the offset must be less than one day). Most implementations of utcoffset() will probably look like one of these two:

```
return CONSTANT  # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```
If `utcoffset()` does not return `None`, `dst()` should not return `None` either.

The default implementation of `utcoffset()` raises `NotImplementedError`.

```python
def dst(self, dt):
    # a fixed-offset class: doesn't account for DST
    return timedelta(0)
```

Return the daylight saving time (DST) adjustment, in minutes east of UTC, or `None` if DST information isn't known. Return `timedelta(0)` if DST is not in effect. If DST is in effect, return the offset as a `timedelta` object (see `utcoffset()` for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by `utcoffset()`, so there's no need to consult `dst()` unless you're interested in obtaining DST info separately. For example, `datetime.timetuple()` calls its `tzinfo` member's `dst()` method to determine how the `tm_isdst` flag should be set, and `tzinfo.fromutc()` calls `dst()` to account for DST changes when crossing time zones.

An instance `tz` of a `tzinfo` subclass that models both standard and daylight times must be consistent in this sense:

```python
tz.utcoffset(dt) - tz.dst(dt)
```

must return the same result for every `datetime dt` with `dt.tzinfo == tz` For sane `tzinfo` subclasses, this expression yields the time zone's "standard offset", which should not depend on the date or the time, but only on geographic location. The implementation of `datetime.astimezone()` relies on this, but cannot detect violations; it's the programmer's responsibility to ensure it. If a `tzinfo` subclass cannot guarantee this, it may be able to override the default implementation of `tzinfo.fromutc()` to work correctly with `astimezone()` regardless.

Most implementations of `dst()` will probably look like one of these two:
or

```python
def dst(self):
    # Code to set dst on and dst off to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time. Then
    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

The default implementation of `dst()` raises `NotImplementedError`.

`tzname(self, dt)`

Return the time zone name corresponding to the `datetime` object `dt`, as a string. Nothing about string names is defined by the `datetime` module, and there's no requirement that it mean anything in particular. For example, "GMT", "UTC", "-500", "-5:00", "EDT", "US/Eastern", "America/New York" are all valid replies. Return `None` if a string name isn't known. Note that this is a method rather than a fixed string primarily because some `tzinfo` subclasses will wish to return different names depending on the specific value of `dt` passed, especially if the `tzinfo` class is accounting for daylight time.

The default implementation of `tzname()` raises `NotImplementedError`.

These methods are called by a `datetime` or `time` object, in response to their methods of the same names. A `datetime` object passes itself as the argument, and a `time` object passes `None` as the argument. A `tzinfo` subclass's methods should therefore be prepared to accept a `dt` argument of `None`, or of class `datetime`.

When `None` is passed, it's up to the class designer to decide the best response. For example, returning `None` is appropriate if the class wishes to say that time objects don't participate in the `tzinfo` protocols. It may be more useful for `utcoffset(None)` to return the standard UTC offset, as there is no other convention for discovering the standard offset.

When a `datetime` object is passed in response to a `datetime` method,
dt.tzinfo is the same object as self.tzinfo methods can rely on this, unless user code calls tzinfo methods directly. The intent is that the tzinfo methods interpret dt as being in local time, and not need worry about objects in other timezones.

There is one more tzinfo method that a subclass may wish to override:

```python
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta  # convert to standard local time
        dtdst = dt.dst()  # raise ValueError if dtdst is None
        if dtdst:
            return dt + dtdst
    else:
        return dt
```

This is called from the default datetime.astimezone() implementation. When called from that, dt.tzinfo is self, and dt's date and time members are to be viewed as expressing a UTC time. The purpose of fromutc() is to adjust the date and time members, returning an equivalent datetime in self's local time.

Most tzinfo subclasses should be able to inherit the default fromutc() implementation without problems. It's strong enough to handle fixed-offset time zones, and time zones accounting for both standard and daylight time, and the latter even if the DST transition times differ in different years. An example of a time zone the default fromutc() implementation may not handle correctly in all cases is one where the standard offset (from UTC) depends on the specific date and time passed, which can happen for political reasons. The default implementations of astimezone() and fromutc() may not produce the result you want if the result is one of the hours straddling the moment the standard offset changes.

Skipping code for error cases, the default fromutc() implementation acts like:
Example tzinfo classes:

```python
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)

class UTC(tzinfo):
    """UTC""
    def utcoffset(self, dt):
        return ZERO
    def tzname(self, dt):
        return "UTC"
    def dst(self, dt):
        return ZERO
utc = UTC()

class FixedOffset(tzinfo):
    """Fixed offset in minutes east from UTC.""
    def __init__(self, offset, name):
        self.__offset = timedelta(minutes=offset)
        self.__name = name
    def utcoffset(self, dt):
        return self.__offset
    def tzname(self, dt):
        return self.__name
    def dst(self, dt):
        return ZERO

class LocalTZ(tzinfo):
    """A class capturing the platform's idea of local time."

import time as _time
STDOFFSET = timedelta(seconds=_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds=_time.altzone)
else:
    DSTOFFSET = STDOFFSET
```
DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):
    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

def dst(self, dt):
    if self._isdst(dt):
        return DSTDIFF
    else:
        return ZERO

def tzname(self, dt):
    return _time.tzname[self._isdst(dt)]

def _isdst(self, dt):
    tt = (dt.year, dt.month, dt.day,
          dt.hour, dt.minute, dt.second,
          dt.weekday(), 0, -1)
    stamp = _time.mktime(tt)
    tt = _time.localtime(stamp)
    return tt.tm_isdst > 0

Local = LocalTimezone()

# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt

# In the US, DST starts at 2am (standard time) on the first Sunday in April.
# DSTSTART = datetime(1, 4, 1, 2)
# and ends at 2am (DST time; 1am standard time) on the last Sunday of October.
# which is the first Sunday on or after Oct 25.
# DSTEND = datetime(1, 10, 25, 1)

class USTimeZone(tzinfo):
    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname
def __repr__(self):
    return self.reprname

def tzname(self, dt):
    if self.dst(dt):
        return self.dstname
    else:
        return self.stdname

def utcoffset(self, dt):
    return self.stdoffset + self.dst(dt)

def dst(self, dt):
    if dt is None or dt.tzinfo is None:
        # An exception may be sensible here, in one or both case
        # It depends on how you want to treat them. The default
        # fromutc() implementation (called by the default astime
        # implementation) passes a datetime with dt.tzinfo is se
        return ZERO
    assert dt.tzinfo is self

    # Find first Sunday in April & the last in October.
    start = first_sunday_on_or_after(DSTSTART.replace(year=dt.year))
    end = first_sunday_on_or_after(DSTEND.replace(year=dt.year))

    # Can't compare naive to aware objects, so strip the timezone
    # dt first.
    if start <= dt.replace(tzinfo=None) < end:
        return HOUR
    else:
        return ZERO

Eastern = USTimeZone(-5, "Eastern", "EST", "EDT")
Central = USTimeZone(-6, "Central", "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific = USTimeZone(-8, "Pacific", "PST", "PDT")

Note that there are unavoidable subtleties twice per year in a tzinfo subclass accounting for both standard and daylight time, at the DST transition points. For concreteness, consider US Eastern (UTC -0500), where EDT begins the minute after 1:59 (EST) on the first Sunday in April, and ends the minute after 1:59 (EDT) on the last Sunday in October:

|-------|-----|------|------|------|------|------|------|

|-------|-------|------|------|------|------|------|------|
When DST starts (the "start" line), the local wall clock leaps from 1:59 to 3:00. A wall time of the form 2:MM doesn't really make sense on that day, so `astimezone(Eastern)` won't deliver a result with `hour == 2` on the day DST begins. In order for `astimezone()` to make this guarantee, the `tzinfo.dst()` method must consider times in the "missing hour" (2:MM for Eastern) to be in daylight time.

When DST ends (the "end" line), there's a potentially worse problem: there's an hour that can't be spelled unambiguously in local wall time: the last hour of daylight time. In Eastern, that's times of the form 5:MM UTC on the day daylight time ends. The local wall clock leaps from 1:59 (daylight time) back to 1:00 (standard time) again. Local times of the form 1:MM are ambiguous. `astimezone()` mimics the local clock's behavior by mapping two adjacent UTC hours into the same local hour then. In the Eastern example, UTC times of the form 5:MM and 6:MM both map to 1:MM when converted to Eastern. In order for `astimezone()` to make this guarantee, the `tzinfo.dst()` method must consider times in the "repeated hour" to be in standard time. This is easily arranged, as in the example, by expressing DST switch times in the time zone's standard local time.

Applications that can't bear such ambiguities should avoid using hybrid `tzinfo` subclasses; there are no ambiguities when using UTC, or any other fixed-offset `tzinfo` subclass (such as a class representing only EST (fixed offset -5 hours), or only EDT (fixed offset -4 hours)).
5.1.7 strftime() Behavior

date, datetime, and time objects all support a strftime(format) method, to create a string representing the time under the control of an explicit format string. Broadly speaking, d.strftime(fmt) acts like the time module's time.strftime(fmt, d.timetuple()) although not all objects support a timetuple() method.

For time objects, the format codes for year, month, and day should not be used, as time objects have no such values. If they're used anyway, 1900 is substituted for the year, and 0 for the month and day.

For date objects, the format codes for hours, minutes, and seconds should not be used, as date objects have no such values. If they're used anyway, 0 is substituted for them.

For a naive object, the %Z and %Z format codes are replaced by empty strings.

For an aware object:

%Z
utcoffset() is transformed into a 5-character string of the form +HHMM or -HHMM, where HH is a 2-digit string giving the number of UTC offset hours, and MM is a 2-digit string giving the number of UTC offset minutes. For example, if utcoffset() returns timedelta(hours=-3, minutes=-30), %Z is replaced with the string '-0330'.

%Z
If tzname() returns None, %Z is replaced by an empty string. Otherwise %Z is replaced by the returned value, which must be a string.

The full set of format codes supported varies across platforms, because Python calls the platform C library's strftime() function, and platform variations are common. The documentation for Python's time module lists the format codes that the C standard (1989 version) requires, and those work on all platforms with a standard C implementation. Note that the 1999 version of the C
standard added additional format codes.

The exact range of years for which strftime() works also varies across platforms. Regardless of platform, years before 1900 cannot be used.
Previous: 5.1.7 strftime() Behavior  Up: 5.1 datetime  Next: 5.1.8.1 Creating Datetime Objects
5.1.8 Examples

Subsections

- 5.1.8.1 Creating Datetime Objects from Formatted Strings

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.2 calendar -- General calendar-related functions

This module allows you to output calendars like the UNIX `cal` program, and provides additional useful functions related to the calendar. By default, these calendars have Monday as the first day of the week, and Sunday as the last (the European convention). Use `setfirstweekday()` to set the first day of the week to Sunday (6) or to any other weekday. Parameters that specify dates are given as integers.

Most of these functions and classes rely on the `datetime` module which uses an idealized calendar, the current Gregorian calendar indefinitely extended in both directions. This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold's book "Calendrical Calculations", where it's the base calendar for all computations.

```python
class Calendar([firstweekday])

Creates a Calendar object. `firstweekday` is an integer specifying the first day of the week. 0 is Monday (the default), 6 is Sunday.

A Calendar object provides several methods that can be used for preparing the calendar data for formatting. This class doesn't do any formatting itself. This is the job of subclasses. New in version 2.5.
```

Calendar instances have the following methods:

```python
iterweekdays(weekday)

Return an iterator for the week day numbers that will be used for one week. The first number from the iterator will be the same as the number returned by `firstweekday()`.

itermonthdates(year, month)

Return an iterator for the month `month` (1-12) in the year `year`. This iterator will return all days (as `datetime.date` objects) for the month and all days before the start of the month or after the end of the month that are
required to get a complete week.

\texttt{itermonthdays2(year, month)}

Return an iterator for the month \textit{month} in the year \textit{year} similar to \texttt{itermonthdates()}. Days returned will be tuples consisting of a day number and a week day number.

\texttt{itermonthdays(year, month)}

Return an iterator for the month \textit{month} in the year \textit{year} similar to \texttt{itermonthdates()}. Days returned will simply be day numbers.

\texttt{monthdatescalendar(year, month)}

Return a list of the weeks in the month \textit{month} of the \textit{year} as full weeks. Weeks are lists of seven \texttt{datetime.date} objects.

\texttt{monthdays2calendar(year, month)}

Return a list of the weeks in the month \textit{month} of the \textit{year} as full weeks. Weeks are lists of seven tuples of day numbers and weekday numbers.

\texttt{monthdayscalendar(year, month)}

Return a list of the weeks in the month \textit{month} of the \textit{year} as full weeks. Weeks are lists of seven day numbers.

\texttt{yeardatescalendar(year, month[, width])}

Return the data for the specified year ready for formatting. The return value is a list of month rows. Each month row contains up to \textit{width} months (defaulting to 3). Each month contains between 4 and 6 weeks and each week contains 1-7 days. Days are \texttt{datetime.date} objects.

\texttt{yeardays2calendar(year, month[, width])}

Return the data for the specified year ready for formatting (similar to \texttt{yeardatescalendar()}). Entries in the week lists are tuples of day numbers and weekday numbers. Day numbers outside this month are zero.

\texttt{yeardayscalendar(year, month[, width])}

Return the data for the specified year ready for formatting (similar to \texttt{yeardatescalendar()}). Entries in the week lists are day numbers.
Day numbers outside this month are zero.

**class TextCalendar**([`firstweekday`])
This class can be used to generate plain text calendars.

New in version 2.5.

*TextCalendar* instances have the following methods:

- **formatmonth**( `theyear, themonth[, w[, l]]` )
  Return a month's calendar in a multi-line string. If `w` is provided, it specifies the width of the date columns, which are centered. If `l` is given, it specifies the number of lines that each week will use. Depends on the first weekday as set by `setfirstweekday()`.

- **prmonth**( `theyear, themonth[, w[, l]]` )
  Print a month's calendar as returned by `formatmonth()`.

- **formatyear**( `theyear, themonth[, w[, l[, c[, m]]]]` )
  Return a `m`-column calendar for an entire year as a multi-line string. Optional parameters `w`, `l`, and `c` are for date column width, lines per week, and number of spaces between month columns, respectively. Depends on the first weekday as set by `setfirstweekday()`.

- **pryear**( `theyear[, w[, l[, c[, m]]]]` )
  Print the calendar for an entire year as returned by `formatyear()`.

**class HTMLCalendar**([`firstweekday`])
This class can be used to generate HTML calendars.

New in version 2.5.

*HTMLCalendar* instances have the following methods:

- **formatmonth**( `theyear, themonth[, withyear]` )
  Return a month's calendar as an HTML table. If `withyear` is true the year
will be included in the header, otherwise just the month name will be used.

**formatyear***( *theyear, themonth[, width]*)***
Return a year's calendar as an HTML table. *width* (defaulting to 3) specifies the number of months per row.

**formatyearpage***( *theyear, themonth[, width[, css[, encoding]]]*)***
Return a year's calendar as a complete HTML page. *width* (defaulting to 3) specifies the number of months per row. *css* is the name for the cascading style sheet to be used. None can be passed if no style sheet should be used. *encoding* specifies the encoding to be used for the output (defaulting to the system default encoding).

**class LocaleTextCalendar***( *[firstweekday[, locale]*)***
This subclass of TextCalendar can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode. New in version 2.5.

**class LocaleHTMLCalendar***( *[firstweekday[, locale]*)***
This subclass of HTMLCalendar can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode. New in version 2.5.

For simple text calendars this module provides the following functions.

**setfirstweekday**( *weekday*)
Sets the weekday (0 is Monday, 6 is Sunday) to start each week. The values MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, and SUNDAY are provided for convenience. For example, to set the first weekday to Sunday:

```python
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

New in version 2.0.
firstweekday()
Returns the current setting for the weekday to start each week. New in version 2.0.

isleap(year)
Returns True if year is a leap year, otherwise False.

leapdays(y1, y2)
Returns the number of leap years in the range [y1...y2], where y1 and y2 are years. Changed in version 2.0: This function didn't work for ranges spanning a century change in Python 1.5.2.

weekday(year, month, day)
Returns the day of the week (0 is Monday) for year (1970-...), month (1-12), day (1-31).

weekheader(n)
Return a header containing abbreviated weekday names. n specifies the width in characters for one weekday.

monthrange(year, month)
Returns weekday of first day of the month and number of days in month, for the specified year and month.

monthcalendar(year, month)
Returns a matrix representing a month's calendar. Each row represents a week; days outside of the month a represented by zeros. Each week begins with Monday unless set by setfirstweekday().

prmonth(theyear, themonth[, w[, l]])
Prints a month's calendar as returned by month().

month(theyear, themonth[, w[, l]])
Returns a month's calendar in a multi-line string using the formatmonth of the TextCalendar class. New in version 2.0.
prcal(year[, w[, l[c]]])
Prints the calendar for an entire year as returned by calendar().

calendar(year[, w[, l[c]]])
Returns a 3-column calendar for an entire year as a multi-line string using the formatyear of the TextCalendar class. New in version 2.0.

timegm(tuple)
An unrelated but handy function that takes a time tuple such as returned by the gmtime() function in the time module, and returns the corresponding UNIX timestamp value, assuming an epoch of 1970, and the POSIX encoding. In fact, time.gmtime() and timegm() are each others' inverse. New in version 2.0.

The calendar module exports the following data attributes:

day_name
An array that represents the days of the week in the current locale.

day_abbr
An array that represents the abbreviated days of the week in the current locale.

month_name
An array that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and month_name[0] is the empty string.

month_abbr
An array that represents the abbreviated months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and month_abbr[0] is the empty string.

See Also:
Module datetime:
Object-oriented interface to dates and times with similar functionality
to the `time` module.

Module `time`:
Low-level time related functions.

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.3 collections -- High-performance container datatypes

New in version 2.4.

This module implements high-performance container datatypes. Currently, there are two datatypes, deque and defaultdict. Future additions may include balanced trees and ordered dictionaries. Changed in version 2.5: Added defaultdict.

Subsections

- 5.3.1 deque objects
  - 5.3.1.1 Recipes
- 5.3.2 defaultdict objects
  - 5.3.2.1 defaultdict Examples
5.3.1 deque objects

**deque([iterable])**

Returns a new deque objected initialized left-to-right (using `append()`) with data from `iterable`. If `iterable` is not specified, the new deque is empty.

Deques are a generalization of stacks and queues (the name is pronounced "`deck" and is short for "`double-ended queue"). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same \(O(1)\) performance in either direction.

Though `list` objects support similar operations, they are optimized for fast fixed-length operations and incur \(O(n)\) memory movement costs for "`pop(0)`" and "`insert(0, v)`" operations which change both the size and position of the underlying data representation. New in version 2.4.

Deque objects support the following methods:

**append(x)**

Add `x` to the right side of the deque.

**appendleft(x)**

Add `x` to the left side of the deque.

**clear()**

Remove all elements from the deque leaving it with length 0.

**extend(iterable)**

Extend the right side of the deque by appending elements from the iterable argument.

**extendleft(iterable)**

Extend the left side of the deque by appending elements from `iterable`. Note, the series of left appends results in reversing the order of elements in the iterable argument.
pop()
Remove and return an element from the right side of the deque. If no elements are present, raises an IndexError.

popleft()
Remove and return an element from the left side of the deque. If no elements are present, raises an IndexError.

remove(value)
Removed the first occurrence of value. If not found, raises a ValueError. New in version 2.5.

rotate(n)
Rotate the deque n steps to the right. If n is negative, rotate to the left. Rotating one step to the right is equivalent to:
"d.appendleft(d.pop())".

In addition to the above, deques support iteration, pickling, "len(d)", "reversed(d)", "copy.copy(d)", "copy.deepcopy(d)", membership testing with the in operator, and subscript references such as "d[-1]".

Example:

```python
>>> from collections import deque
>>> d = deque('ghi')                      # make a new deque with three items
>>> for elem in d:                        # iterate over the deque's elements
...     print elem.upper()
G
H
I

>>> d.append('j')                         # add a new entry to the right
>>> d.appendleft('f')                     # add a new entry to the left side
>>> d                                  # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                               # return and remove the rightmost item
'j'
>>> d.popleft()                           # return and remove the leftmost item
'f'
>>> list(d)                               # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                                 # peek at leftmost item
'g'
```
>>> d[-1]  # peek at rightmost item
'i'

>>> list(reversed(d))  # list the contents of a deque
['i', 'h', 'g']

>>> 'h' in d  # search the deque
True

>>> d.extend('jkl')  # add multiple elements at once

deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> d.rotate(1)  # right rotation

deque(['l', 'g', 'h', 'i', 'j', 'k'])

>>> d.rotate(-1)  # left rotation

deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))  # make a new deque in reverse order

deque(['l', 'k', 'j', 'i', 'h', 'g'])

>>> d.clear()  # empty the deque

>>> d.pop()  # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')  # extendleft() reverses the input

deque(['c', 'b', 'a'])
Examples
**5.3.2 defaultdict objects**

`defaultdict([default_factory[, ...]])`

Returns a new dictionary-like object. `defaultdict` is a subclass of the builtin `dict` class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the `dict` class and is not documented here.

The first argument provides the initial value for the `default_factory` attribute; it defaults to `None`. All remaining arguments are treated the same as if they were passed to the `dict` constructor, including keyword arguments.

New in version 2.5.

`defaultdict` objects support the following method in addition to the standard `dict` operations:

`__missing__(key)`

If the `default_factory` attribute is `None`, this raises a `KeyError` exception with the `key` as argument.

If `default_factory` is not `None`, it is called without arguments to provide a default value for the given `key`, this value is inserted in the dictionary for the `key`, and returned.

If calling `default_factory` raises an exception this exception is propagated unchanged.

This method is called by the `__getitem__` method of the `dict` class when the requested key is not found; whatever it returns or raises is then returned or raised by `__getitem__`.

`defaultdict` objects support the following instance variable:

`default_factory`

This attribute is used by the `__missing__` method; it is initialized from
the first argument to the constructor, if present, or to \texttt{None}, if absent.
5.4 heapq -- Heap queue algorithm

New in version 2.3.

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

Heaps are arrays for which \( heap[k] \leq heap[2k+1] \) and \( heap[k] \leq heap[2k+2] \) for all \( k \), counting elements from zero. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that \( heap[0] \) is always its smallest element.

The API below differs from textbook heap algorithms in two aspects: (a) We use zero-based indexing. This makes the relationship between the index for a node and the indexes for its children slightly less obvious, but is more suitable since Python uses zero-based indexing. (b) Our pop method returns the smallest item, not the largest (called a "min heap" in textbooks; a "max heap" is more common in texts because of its suitability for in-place sorting).

These two make it possible to view the heap as a regular Python list without surprises: \( heap[0] \) is the smallest item, and \( heap.sort() \) maintains the heap invariant!

To create a heap, use a list initialized to [], or you can transform a populated list into a heap via function `heapify()`.

The following functions are provided:

```
heappush( heap, item )
```

Push the value `item` onto the `heap`, maintaining the heap invariant.

```
heappop( heap )
```

Pop and return the smallest item from the `heap`, maintaining the heap invariant. If the heap is empty, `IndexError` is raised.

```
heapify(x)
```
Transform list \( x \) into a heap, in-place, in linear time.

**heapreplace** \((heap, item)\)

Pop and return the smallest item from the \( heap \), and also push the new \( item \). The heap size doesn't change. If the heap is empty, \texttt{IndexError} is raised. This is more efficient than \texttt{heappop()} followed by \texttt{heappush()}, and can be more appropriate when using a fixed-size heap. Note that the value returned may be larger than \( item \)! That constrains reasonable uses of this routine unless written as part of a conditional replacement:

```python
    if item > heap[0]:
        item = heapreplace(heap, item)
```

Example of use:

```python
>>> from heapq import heappush, heappop
>>> heap = []
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> for item in data:
...    heappush(heap, item)
...
>>> sorted = []
>>> while heap:
...    sorted.append(heappop(heap))
...
>>> print sorted
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> data.sort()
>>> print data == sorted
True
>>>```

The module also offers two general purpose functions based on heaps.

**nlargest** \((n, iterable[, key])\)

Return a list with the \( n \) largest elements from the dataset defined by \( iterable \). \texttt{key}, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the iterable:

```
"key=str.lower"
```

Equivalent to:

```
"sorted(iterable, key=key, reverse=True)[:n]"
```

New in version 2.4. Changed in version 2.5: Added the optional \texttt{key} argument.

**nsmallest** \((n, iterable[, key])\)
Return a list with the $n$ smallest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the iterable:

"key=str.lower" Equivalent to: "sorted(iterable, key=key)[:n]" New in version 2.4. Changed in version 2.5: Added the optional *key* argument.

Both functions perform best for smaller values of $n$. For larger values, it is more efficient to use the `sorted()` function. Also, when $n==1$, it is more efficient to use the builtin `min()` and `max()` functions.

---

**Subsections**

- [5.4.1 Theory](#)

---

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
5.4.1 Theory

(This explanation is due to François Pinard. The Python code for this module was contributed by Kevin O'Connor.)

Heaps are arrays for which $a[k] \leq a[2*k+1]$ and $a[k] \leq a[2*k+2]$ for all $k$, counting elements from 0. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that $a[0]$ is always its smallest element.

The strange invariant above is meant to be an efficient memory representation for a tournament. The numbers below are $k$, not $a[k]$:

```
       0
      / \  \\
     1   2  \\
    /   / \  \\
   3   4   5  \\
  /  /  /  \  \\
 7  8  9  10  11  12  13  14
```

In the tree above, each cell $k$ is topping $2*k+1$ and $2*k+2$. In an usual binary tournament we see in sports, each cell is the winner over the two cells it tops, and we can trace the winner down the tree to see all opponents s/he had. However, in many computer applications of such tournaments, we do not need to trace the history of a winner. To be more memory efficient, when a winner is promoted, we try to replace it by something else at a lower level, and the rule becomes that a cell and the two cells it tops contain three different items, but the top cell "wins" over the two topped cells.

If this heap invariant is protected at all time, index 0 is clearly the overall winner. The simplest algorithmic way to remove it and find the "next" winner is to move some loser (let's say cell 30 in the diagram above) into the 0 position, and then percolate this new 0 down the tree, exchanging values, until the invariant is re-established. This is clearly logarithmic on the total number of items in the tree. By iterating over all items, you get an $O(n \log n)$ sort.

A nice feature of this sort is that you can efficiently insert new items while the
sort is going on, provided that the inserted items are not "better" than the last 0'th element you extracted. This is especially useful in simulation contexts, where the tree holds all incoming events, and the "win" condition means the smallest scheduled time. When an event schedule other events for execution, they are scheduled into the future, so they can easily go into the heap. So, a heap is a good structure for implementing schedulers (this is what I used for my MIDI sequencer :-).

Various structures for implementing schedulers have been extensively studied, and heaps are good for this, as they are reasonably speedy, the speed is almost constant, and the worst case is not much different than the average case. However, there are other representations which are more efficient overall, yet the worst cases might be terrible.

Heaps are also very useful in big disk sorts. You most probably all know that a big sort implies producing "runs" (which are pre-sorted sequences, which size is usually related to the amount of CPU memory), followed by a merging passes for these runs, which merging is often very cleverly organised\footnote{5}. It is very important that the initial sort produces the longest runs possible. Tournaments are a good way to that. If, using all the memory available to hold a tournament, you replace and percolate items that happen to fit the current run, you'll produce runs which are twice the size of the memory for random input, and much better for input fuzzily ordered.

Moreover, if you output the 0'th item on disk and get an input which may not fit in the current tournament (because the value "wins" over the last output value), it cannot fit in the heap, so the size of the heap decreases. The freed memory could be cleverly reused immediately for progressively building a second heap, which grows at exactly the same rate the first heap is melting. When the first heap completely vanishes, you switch heaps and start a new run. Clever and quite effective!

In a word, heaps are useful memory structures to know. I use them in a few applications, and I think it is good to keep a `heap' module around. :-)

---

**Footnotes**
... organised

The disk balancing algorithms which are current, nowadays, are more annoying than clever, and this is a consequence of the seeking capabilities of the disks. On devices which cannot seek, like big tape drives, the story was quite different, and one had to be very clever to ensure (far in advance) that each tape movement will be the most effective possible (that is, will best participate at "progressing" the merge). Some tapes were even able to read backwards, and this was also used to avoid the rewinding time. Believe me, real good tape sorts were quite spectacular to watch! From all times, sorting has always been a Great Art! :-)
5.5 **bisect -- Array bisection algorithm**

This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. For long lists of items with expensive comparison operations, this can be an improvement over the more common approach. The module is called `bisect` because it uses a basic bisection algorithm to do its work. The source code may be most useful as a working example of the algorithm (the boundary conditions are already right!).

The following functions are provided:

**bisect_left** *(list, item[, lo[, hi]])*

Locate the proper insertion point for `item` in `list` to maintain sorted order. The parameters `lo` and `hi` may be used to specify a subset of the list which should be considered; by default the entire list is used. If `item` is already present in `list`, the insertion point will be before (to the left of) any existing entries. The return value is suitable for use as the first parameter to `list.insert()`. This assumes that `list` is already sorted. New in version 2.1.

**bisect_right** *(list, item[, lo[, hi]])*

Similar to `bisect_left()`, but returns an insertion point which comes after (to the right of) any existing entries of `item` in `list`. New in version 2.1.

**bisect(...)**

Alias for `bisect_right()`.

**insort_left** *(list, item[, lo[, hi]])*

Insert `item` in `list` in sorted order. This is equivalent to `list.insert(bisect.bisect_left(list, item, lo, hi), item)`. This assumes that `list` is already sorted. New in version 2.1.

**insort_right** *(list, item[, lo[, hi]])*
Similar to `insort_left()`, but inserting `item` in `list` after any existing entries of `item`. New in version 2.1.

```python
insort(...)  
Alias for `insort_right()`.
```

Subsections

- [5.5.1 Examples](#)

---

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
5.5.1 Examples

The `bisect()` function is generally useful for categorizing numeric data. This example uses `bisect()` to look up a letter grade for an exam total (say) based on a set of ordered numeric breakpoints: 85 and up is an `A', 75..84 is a `B', etc.

```python
>>> grades = "FEDCBA"
>>> breakpoints = [30, 44, 66, 75, 85]
>>> from bisect import bisect
>>> def grade(total):
...     return grades[bisect(breakpoints, total)]
... 
>>> grade(66)
'C'
>>> map(grade, [33, 99, 77, 44, 12, 88])
['E', 'A', 'B', 'D', 'F', 'A']
```
5.6 array -- Efficient arrays of numeric values

This module defines an object type which can efficiently represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a \textit{type code}, which is a single character. The following type codes are defined:

<table>
<thead>
<tr>
<th>Type code</th>
<th>C Type</th>
<th>Python Type</th>
<th>Minimum size in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'c'</td>
<td>char</td>
<td>character</td>
<td>1</td>
</tr>
<tr>
<td>'b'</td>
<td>signed char</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>'B'</td>
<td>unsigned char</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>'u'</td>
<td>Py_UNICODE</td>
<td>Unicode character</td>
<td>2</td>
</tr>
<tr>
<td>'h'</td>
<td>signed short</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>'H'</td>
<td>unsigned short</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>'i'</td>
<td>signed int</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>'I'</td>
<td>unsigned int</td>
<td>long</td>
<td>2</td>
</tr>
<tr>
<td>'l'</td>
<td>signed long</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>'L'</td>
<td>unsigned long</td>
<td>long</td>
<td>4</td>
</tr>
<tr>
<td>'f'</td>
<td>float</td>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>'d'</td>
<td>double</td>
<td>float</td>
<td>8</td>
</tr>
</tbody>
</table>

The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the \textit{itemsize} attribute. The values stored for 'L' and 'I' items will be represented as Python long integers when retrieved, because Python's plain integer type cannot represent the full range of C's unsigned (long) integers.

The module defines the following type:

\begin{verbatim}
array( typecode[, initializer])
Return a new array whose items are restricted by typecode, and initialized
\end{verbatim}
from the optional *initializer* value, which must be a list, string, or iterable over elements of the appropriate type. Changed in version 2.4: Formerly, only lists or strings were accepted. If given a list or string, the initializer is passed to the new array's *fromlist()*,*fromstring()*, or *fromunicode()* method (see below) to add initial items to the array. Otherwise, the iterable initializer is passed to the *extend()* method.

**ArrayType**

Obsolete alias for *array*.

Array objects support the ordinary sequence operations of indexing, slicing, concatenation, and multiplication. When using slice assignment, the assigned value must be an array object with the same type code; in all other cases, *TypeError* is raised. Array objects also implement the buffer interface, and may be used wherever buffer objects are supported.

The following data items and methods are also supported:

**typecode**

The typecode character used to create the array.

**itemsize**

The length in bytes of one array item in the internal representation.

**append** *(x)*

Append a new item with value *x* to the end of the array.

**buffer_info()**

Return a tuple *(address, length)* giving the current memory address and the length in elements of the buffer used to hold array's contents. The size of the memory buffer in bytes can be computed as *array.buffer_info()[1] * *array.itemsize*. This is occasionally useful when working with low-level (and inherently unsafe) I/O interfaces that require memory addresses, such as certain *ioctl()* operations. The returned numbers are valid as long as the array exists and no length-changing operations are applied to it.

**Note:** When using array objects from code written in C or C++ (the only way to effectively make use of this information), it makes more sense to use
the buffer interface supported by array objects. This method is maintained for backward compatibility and should be avoided in new code. The buffer interface is documented in the Python/C API Reference Manual.

byteswap()
``Byteswap'' all items of the array. This is only supported for values which are 1, 2, 4, or 8 bytes in size; for other types of values, RuntimeError is raised. It is useful when reading data from a file written on a machine with a different byte order.

count(x)
Return the number of occurrences of x in the array.

extend(iterable)
Append items from iterable to the end of the array. If iterable is another array, it must have exactly the same type code; if not, TypeError will be raised. If iterable is not an array, it must be iterable and its elements must be the right type to be appended to the array. Changed in version 2.4: Formerly, the argument could only be another array.

fromfile(f, n)
Read n items (as machine values) from the file object f and append them to the end of the array. If less than n items are available, EOFError is raised, but the items that were available are still inserted into the array. f must be a real built-in file object; something else with a read() method won't do.

fromlist(list)
Append items from the list. This is equivalent to "for x in list: a.append(x)" except that if there is a type error, the array is unchanged.

fromstring(s)
Appends items from the string, interpreting the string as an array of machine values (as if it had been read from a file using the fromfile() method).

fromunicode(s)
Extends this array with data from the given unicode string. The array must
be a type 'u' array; otherwise a ValueError is raised. Use "array.fromstring(ustr.decode(enc))" to append Unicode data to an array of some other type.

**index(x)**

Return the smallest $i$ such that $i$ is the index of the first occurrence of $x$ in the array.

**insert(i, x)**

Insert a new item with value $x$ in the array before position $i$. Negative values are treated as being relative to the end of the array.

**pop([i])**

Removes the item with the index $i$ from the array and returns it. The optional argument defaults to -1, so that by default the last item is removed and returned.

**read(f, n)**

*Deprecated since release 1.5.1.* Use the fromfile() method.

Read $n$ items (as machine values) from the file object $f$ and append them to the end of the array. If less than $n$ items are available, EOFError is raised, but the items that were available are still inserted into the array. $f$ must be a real built-in file object; something else with a read() method won't do.

**remove(x)**

Remove the first occurrence of $x$ from the array.

**reverse()**

Reverse the order of the items in the array.

**tofile(f)**

Write all items (as machine values) to the file object $f$.

**tolist()**

Convert the array to an ordinary list with the same items.
tostring()
Convert the array to an array of machine values and return the string representation (the same sequence of bytes that would be written to a file by the tofile() method.)

toungcode()
Convert the array to a unicode string. The array must be a type 'u' array; otherwise a ValueError is raised. Use "array.tostring().decode(enc)" to obtain a unicode string from an array of some other type.

write(f)
Deprecated since release 1.5.1. Use the tofile() method.

Write all items (as machine values) to the file object f.

When an array object is printed or converted to a string, it is represented as array(typecode, initializer). The initializer is omitted if the array is empty, otherwise it is a string if the typecode is 'c', otherwise it is a list of numbers. The string is guaranteed to be able to be converted back to an array with the same type and value using reverse quotes (``), so long as the array() function has been imported using from array import array. Examples:

    array('l')
    array('c', 'hello world')
    array('u', u'hello \textbackslash u2641')
    array('l', [1, 2, 3, 4, 5])
    array('d', [1.0, 2.0, 3.14])

See Also:
Module struct:
    Packing and unpacking of heterogeneous binary data.

Module xdrlib:
    Packing and unpacking of External Data Representation (XDR) data as used in some remote procedure call systems.

The Numerical Python Manual
5.7 sets -- Unordered collections of unique elements

New in version 2.3.

The sets module provides classes for constructing and manipulating unordered collections of unique elements. Common uses include membership testing, removing duplicates from a sequence, and computing standard math operations on sets such as intersection, union, difference, and symmetric difference.

Like other collections, sets support `x in set`, `len(set)`, and `for x in set`. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

Most set applications use the Set class which provides every set method except for `__hash__()`. For advanced applications requiring a hash method, the ImmutableSet class adds a `__hash__()` method but omits methods which alter the contents of the set. Both Set and ImmutableSet derive from BaseSet, an abstract class useful for determining whether something is a set: `isinstance(obj, BaseSet)`. The set classes are implemented using dictionaries. Accordingly, the requirements for set elements are the same as those for dictionary keys; namely, that the element defines both `__eq__` and `__hash__`. As a result, sets cannot contain mutable elements such as lists or dictionaries. However, they can contain immutable collections such as tuples or instances of ImmutableSet. For convenience in implementing sets of sets, inner sets are automatically converted to immutable form, for example, `Set([Set(['dog'])])` is transformed to `Set([ImmutableSet(['dog'])])`.

```
class Set([iterable])
    Constructs a new empty Set object. If the optional iterable parameter is supplied, updates the set with elements obtained from iteration. All of the elements in iterable should be immutable or be transformable to an
```
immutable using the protocol described in section 5.7.3.

```python
class ImmutableSet([iterable])

Constructs a new empty ImmutableSet object. If the optional iterable parameter is supplied, updates the set with elements obtained from iteration. All of the elements in iterable should be immutable or be transformable to an immutable using the protocol described in section 5.7.3.

Because ImmutableSet objects provide a __hash__() method, they can be used as set elements or as dictionary keys. ImmutableSet objects do not have methods for adding or removing elements, so all of the elements must be known when the constructor is called.
```

---

**Subsections**

- [5.7.1 Set Objects](#)
- [5.7.2 Example](#)
- [5.7.3 Protocol for automatic conversion to immutable](#)
- [5.7.4 Comparison to the built-in set types](#)

---

**Python Library Reference**

Previous: [5.6 array](#)  Up: [5. Data Types](#)  Next: [5.7.1 Set Objects](#)

*Release 2.5, documentation updated on 19th September, 2006.*  
See [About this document...](#) for information on suggesting changes.
## 5.7.1 Set Objects

Instances of `Set` and `ImmutableSet` both provide the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Equivalent</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(s)</code></td>
<td></td>
<td>cardinality of set <code>s</code></td>
</tr>
<tr>
<td><code>x in s</code></td>
<td></td>
<td>test <code>x</code> for membership in <code>s</code></td>
</tr>
<tr>
<td><code>x not in s</code></td>
<td></td>
<td>test <code>x</code> for non-membership in <code>s</code></td>
</tr>
<tr>
<td><code>s.issubset(t)</code></td>
<td><code>s &lt;= t</code></td>
<td>test whether every element in <code>s</code> is in <code>t</code></td>
</tr>
<tr>
<td><code>s.issuperset(t)</code></td>
<td><code>s &gt;= t</code></td>
<td>test whether every element in <code>t</code> is in <code>s</code></td>
</tr>
<tr>
<td><code>s.union(t)</code></td>
<td>`s</td>
<td>t`</td>
</tr>
<tr>
<td><code>s.intersection(t)</code></td>
<td><code>s &amp; t</code></td>
<td>new set with elements common to <code>s</code> and <code>t</code></td>
</tr>
<tr>
<td><code>s.difference(t)</code></td>
<td><code>s - t</code></td>
<td>new set with elements in <code>s</code> but not in <code>t</code></td>
</tr>
<tr>
<td><code>s.symmetric_difference(t)</code></td>
<td><code>s ^ t</code></td>
<td>new set with elements in either <code>s</code> or <code>t</code> but not both</td>
</tr>
<tr>
<td><code>s.copy()</code></td>
<td></td>
<td>new set with a shallow copy of <code>s</code></td>
</tr>
</tbody>
</table>

Note, the non-operator versions of `union()`, `intersection()`, `difference()`, and `symmetric_difference()` will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like `Set('abc') & 'cbs'` in favor of the more readable `Set('abc').intersection('cbs')`. Changed in version 2.3.1: Formerly all arguments were required to be sets.

In addition, both `Set` and `ImmutableSet` support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is
greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

The subset and equality comparisons do not generalize to a complete ordering function. For example, any two disjoint sets are not equal and are not subsets of each other, so all of the following return False: $a<b$, $a==b$, or $a>b$. Accordingly, sets do not implement the __cmp__ method.

Since sets only define partial ordering (subset relationships), the output of the list.sort() method is undefined for lists of sets.

The following table lists operations available in ImmutableSet but not found in Set:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash(s)</td>
<td>returns a hash value for s</td>
</tr>
</tbody>
</table>

The following table lists operations available in Set but not found in ImmutableSet:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Equivalent</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.update(t)</td>
<td>$s</td>
<td>t</td>
</tr>
<tr>
<td>s.intersection_update(t)</td>
<td>$s &amp; t$</td>
<td>return set $s$ keeping only elements also found in $t$</td>
</tr>
<tr>
<td>s.difference_update(t)</td>
<td>$s - t$</td>
<td>return set $s$ after removing elements found in $t$</td>
</tr>
<tr>
<td>s.symmetric_difference_update(t)</td>
<td>$s ^ t$</td>
<td>return set $s$ with elements from $s$ or $t$ but not both</td>
</tr>
<tr>
<td>s.add(x)</td>
<td></td>
<td>add element $x$ to set $s$</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><code>s.remove(x)</code></td>
<td>remove <code>x</code> from set <code>s</code>; raises <code>KeyError</code> if not present</td>
<td></td>
</tr>
<tr>
<td><code>s.discard(x)</code></td>
<td>removes <code>x</code> from set <code>s</code> if present</td>
<td></td>
</tr>
<tr>
<td><code>s.pop()</code></td>
<td>remove and return an arbitrary element from <code>s</code>; raises <code>KeyError</code> if empty</td>
<td></td>
</tr>
<tr>
<td><code>s.clear()</code></td>
<td>remove all elements from set <code>s</code></td>
<td></td>
</tr>
</tbody>
</table>

Note, the non-operator versions of `update()`, `intersection_update()`, `difference_update()`, and `symmetric_difference_update()` will accept any iterable as an argument. Changed in version 2.3.1: Formerly all arguments were required to be sets.

Also note, the module also includes a `union_update()` method which is an alias for `update()`. The method is included for backwards compatibility. Programmers should prefer the `update()` method because it is supported by the builtin `set()` and `frozenset()` types.
Example

```python
>>> from sets import Set
>>> engineers = Set(['John', 'Jane', 'Jack', 'Janice'])
>>> programmers = Set(['Jack', 'Sam', 'Susan', 'Janice'])
>>> managers = Set(['Jane', 'Jack', 'Susan', 'Zack'])
>>> employees = engineers | programmers | managers  # union
>>> engineering_management = engineers & managers  # inter
>>> fulltime_management = managers - engineers - programmers  # diff
>>> engineers.add('Marvin')  # add e
>>> print engineers
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
>>> employees.issuperset(engineers)  # superset test
False
>>> employees.union_update(engineers)  # update from another
>>> employees.issuperset(engineers)
True
>>> for group in [engineers, programmers, managers, employees]:
...    group.discard('Susan')  # unconditionally remo
...    print group
...              
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
Set(['Janice', 'Jack', 'Sam'])
Set(['Jane', 'Zack', 'Jack'])
Set(['Jack', 'Sam', 'Jane', 'Marvin', 'Janice', 'John', 'Zack'])
```

Python Library Reference

Previous: 5.7.1 Set Objects  Up: 5.7 sets  Next: 5.7.3 Protocol for automatic

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.7.3 Protocol for automatic conversion to immutable

Sets can only contain immutable elements. For convenience, mutable Set objects are automatically copied to an ImmutableSet before being added as a set element.

The mechanism is to always add a hashable element, or if it is not hashable, the element is checked to see if it has an __asImmutable__() method which returns an immutable equivalent.

Since Set objects have a __asImmutable__() method returning an instance of ImmutableSet, it is possible to construct sets of sets.

A similar mechanism is needed by the __contains__() and remove() methods which need to hash an element to check for membership in a set. Those methods check an element for hashability and, if not, check for a __asTemporarilyImmutable__() method which returns the element wrapped by a class that provides temporary methods for __hash__(), __eq__(), and __ne__().

The alternate mechanism spares the need to build a separate copy of the original mutable object.

Set objects implement the __asTemporarilyImmutable__() method which returns the Set object wrapped by a new class _TemporarilyImmutableSet.

The two mechanisms for adding hashability are normally invisible to the user; however, a conflict can arise in a multi-threaded environment where one thread is updating a set while another has temporarily wrapped it in _TemporarilyImmutableSet. In other words, sets of mutable sets are not thread-safe.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.7.4 Comparison to the built-in set types

The built-in `set` and `frozenset` types were designed based on lessons learned from the `sets` module. The key differences are:

- Set and `ImmutableSet` were renamed to `set` and `frozenset`.
- There is no equivalent to `BaseSet`. Instead, use `isinstance(x, (set, frozenset))`.
- The hash algorithm for the built-ins performs significantly better (fewer collisions) for most datasets.
- The built-in versions have more space efficient pickles.
- The built-in versions do not have a `union_update()` method. Instead, use the `update()` method which is equivalent.
- The built-in versions do not have a `__repr__(sorted=True)` method. Instead, use the built-in `repr()` and `sorted()` functions: `repr(sorted(s))`.
- The built-in version does not have a protocol for automatic conversion to immutable. Many found this feature to be confusing and no one in the community reported having found real uses for it.
5.8 sched -- Event scheduler

The sched module defines a class which implements a general purpose event scheduler:

```python
class scheduler(timefunc, delayfunc)
```

The `scheduler` class defines a generic interface to scheduling events. It needs two functions to actually deal with the ``outside world'' -- `timefunc` should be callable without arguments, and return a number (the ``time'', in any units whatsoever). The `delayfunc` function should be callable with one argument, compatible with the output of `timefunc`, and should delay that many time units. `delayfunc` will also be called with the argument 0 after each event is run to allow other threads an opportunity to run in multi-threaded applications.

Example:

```python
>>> import sched, time
>>> s=sched.scheduler(time.time, time.sleep)
>>> def print_time(): print "From print_time", time.time()
... >>> def print_some_times():
... print time.time()
... s.enter(5, 1, print_time, ())
... s.enter(10, 1, print_time, ())
... s.run()
... print time.time()
... >>> print_some_times()
930343690.257
From print_time 930343695.274
From print_time 930343700.273
930343700.276
```

Subsections

- 5.8.1 Scheduler Objects
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.8.1 Scheduler Objects

scheduler instances have the following methods:

**enterabs** *(time, priority, action, argument)*
Schedule a new event. The *time* argument should be a numeric type compatible with the return value of the *timefunc* function passed to the constructor. Events scheduled for the same *time* will be executed in the order of their *priority*.

Executing the event means executing *action*( *argument*). *argument* must be a sequence holding the parameters for *action*.

Return value is an event which may be used for later cancellation of the event (see **cancel()**).

**enter** *(delay, priority, action, argument)*
Schedule an event for *delay* more time units. Other then the relative time, the other arguments, the effect and the return value are the same as those for **enterabs()**.

**cancel** *(event)*
Remove the event from the queue. If *event* is not an event currently in the queue, this method will raise a **RuntimeError**.

**empty** *
Return true if the event queue is empty.

**run** *
Run all scheduled events. This function will wait (using the *delayfunc* function passed to the constructor) for the next event, then execute it and so on until there are no more scheduled events.

Either *action* or *delayfunc* can raise an exception. In either case, the scheduler will maintain a consistent state and propagate the exception. If an exception is raised by *action*, the event will not be attempted in future calls.
to `run()`.

If a sequence of events takes longer to run than the time available before the next event, the scheduler will simply fall behind. No events will be dropped; the calling code is responsible for canceling events which are no longer pertinent.
5.9 mutex -- Mutual exclusion support

The mutex module defines a class that allows mutual-exclusion via acquiring and releasing locks. It does not require (or imply) threading or multi-tasking, though it could be useful for those purposes.

The mutex module defines the following class:

class mutex()

Create a new (unlocked) mutex.

A mutex has two pieces of state -- a ``locked'' bit and a queue. When the mutex is not locked, the queue is empty. Otherwise, the queue contains zero or more (function, argument) pairs representing functions (or methods) waiting to acquire the lock. When the mutex is unlocked while the queue is not empty, the first queue entry is removed and its function(argument) pair called, implying it now has the lock.

Of course, no multi-threading is implied - hence the funny interface for lock(), where a function is called once the lock is acquired.

Subsections

- 5.9.1 Mutex Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.9.1 Mutex Objects

mutex objects have following methods:

**test()**
Check whether the mutex is locked.

**testandset()**
``Atomic'' test-and-set, grab the lock if it is not set, and return True, otherwise, return False.

**lock**(function, argument)
Execute function( argument ), unless the mutex is locked. In the case it is locked, place the function and argument on the queue. See **unlock** for explanation of when function( argument ) is executed in that case.

**unlock()**
Unlock the mutex if queue is empty, otherwise execute the first element in the queue.
5.10 Queue -- A synchronized queue class

The Queue module implements a multi-producer, multi-consumer FIFO queue. It is especially useful in threads programming when information must be exchanged safely between multiple threads. The Queue class in this module implements all the required locking semantics. It depends on the availability of thread support in Python.

The Queue module defines the following class and exception:

```python
class Queue(maxsize)
    Constructor for the class. maxsize is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If maxsize is less than or equal to zero, the queue size is infinite.
```

exception Empty
    Exception raised when non-blocking get() (or get_nowait()) is called on a Queue object which is empty.

exception Full
    Exception raised when non-blocking put() (or put_nowait()) is called on a Queue object which is full.

---

Subsections

- 5.10.1 Queue Objects

---

Python Library Reference

Previous: 5.9.1 Mutex Objects  Up: 5. Data Types  Next: 5.10.1 Queue Objects

Release 2.5, documentation updated on 19th September, 2006. 
See About this document... for information on suggesting changes.
5.10.1 Queue Objects

Class Queue implements queue objects and has the methods described below. This class can be derived from in order to implement other queue organizations (e.g. stack) but the inheritable interface is not described here. See the source code for details. The public methods are:

**qsize()**
Return the approximate size of the queue. Because of multithreading semantics, this number is not reliable.

**empty()**
Return True if the queue is empty, False otherwise. Because of multithreading semantics, this is not reliable.

**full()**
Return True if the queue is full, False otherwise. Because of multithreading semantics, this is not reliable.

**put(item[, block[, timeout]])**
Put item into the queue. If optional args block is true and timeout is None (the default), block if necessary until a free slot is available. If timeout is a positive number, it blocks at most timeout seconds and raises the Full exception if no free slot was available within that time. Otherwise (block is false), put an item on the queue if a free slot is immediately available, else raise the Full exception (timeout is ignored in that case).

New in version 2.3: the timeout parameter.

**put_nowait(item)**
Equivalent to put(item, False).

**get([block[, timeout]])**
Remove and return an item from the queue. If optional args block is true and timeout is None (the default), block if necessary until an item is available. If timeout is a positive number, it blocks at most timeout seconds
and raises the \texttt{Empty} exception if no item was available within that time. Otherwise (\texttt{block} is false), return an item if one is immediately available, else raise the \texttt{Empty} exception (\texttt{timeout} is ignored in that case).

New in version 2.3: the \texttt{timeout} parameter.

\texttt{get\_nowait()} \\
Equivalent to \texttt{get(False)}.

Two methods are offered to support tracking whether enqueued tasks have been fully processed by daemon consumer threads.

\texttt{task\_done()} \\
Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each \texttt{get()} used to fetch a task, a subsequent call to \texttt{task\_done()} tells the queue that the processing on the task is complete.

If a \texttt{join()} is currently blocking, it will resume when all items have been processed (meaning that a \texttt{task\_done()} call was received for every item that had been \texttt{put()} into the queue).

Raises a \texttt{ValueError} if called more times than there were items placed in the queue. New in version 2.5.

\texttt{join()} \\
Blocks until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls \texttt{task\_done()} to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, \texttt{join()} unblocks. New in version 2.5.

Example of how to wait for enqueued tasks to be completed:

```python
def worker():
    while True:
        item = q.get()
        do_work(item)
```
q.task_done()

q = Queue()
for i in range(num_worker_threads):
    t = Thread(target=worker)
    t.setDaemon(True)
    t.start()

for item in source():
    q.put(item)

q.join()  # block until all tasks are done
5.11 `weakref` -- Weak references

New in version 2.1.

The `weakref` module allows the Python programmer to create weak references to objects.

In the following, the term *referent* means the object which is referred to by a weak reference.

A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, garbage collection is free to destroy the referent and reuse its memory for something else. A primary use for weak references is to implement caches or mappings holding large objects, where it's desired that a large object not be kept alive solely because it appears in a cache or mapping. For example, if you have a number of large binary image objects, you may wish to associate a name with each. If you used a Python dictionary to map names to images, or images to names, the image objects would remain alive just because they appeared as values or keys in the dictionaries. The `WeakKeyDictionary` and `WeakValueDictionary` classes supplied by the `weakref` module are an alternative, using weak references to construct mappings that don't keep objects alive solely because they appear in the mapping objects. If, for example, an image object is a value in a `WeakValueDictionary`, then when the last remaining references to that image object are the weak references held by weak mappings, garbage collection can reclaim the object, and its corresponding entries in weak mappings are simply deleted.

`WeakKeyDictionary` and `WeakValueDictionary` use weak references in their implementation, setting up callback functions on the weak references that notify the weak dictionaries when a key or value has been reclaimed by garbage collection. Most programs should find that using one of these weak dictionary types is all they need - it's not usually necessary to create your own weak references directly. The low-level machinery used by the weak dictionary implementations is exposed by the `weakref` module for the benefit of advanced uses.
Not all objects can be weakly referenced; those objects which can include class instances, functions written in Python (but not in C), methods (both bound and unbound), sets, frozensets, file objects, generators, type objects, DBcursor objects from the bsddb module, sockets, arrays, deques, and regular expression pattern objects. Changed in version 2.4: Added support for files, sockets, arrays, and patterns.

Several builtin types such as list and dict do not directly support weak references but can add support through subclassing:

```python
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)  # this object is weak referencable
```

Extension types can easily be made to support weak references; see `Weak Reference Support` in *Extending and Embedding the Python Interpreter*.

```python
class ref(object[, callback])
```

Return a weak reference to object. The original object can be retrieved by calling the reference object if the referent is still alive; if the referent is no longer alive, calling the reference object will cause None to be returned. If callback is provided and not None, and the returned weakref object is still alive, the callback will be called when the object is about to be finalized; the weak reference object will be passed as the only parameter to the callback; the referent will no longer be available.

It is allowable for many weak references to be constructed for the same object. Callbacks registered for each weak reference will be called from the most recently registered callback to the oldest registered callback.

Exceptions raised by the callback will be noted on the standard error output, but cannot be propagated; they are handled in exactly the same way as exceptions raised from an object's `__del__( )` method.

Weak references are hashable if the object is hashable. They will maintain their hash value even after the object was deleted. If `hash( )` is called the first time only after the object was deleted, the call will raise TypeError.

Weak references support tests for equality, but not ordering. If the referents
are still alive, two references have the same equality relationship as their referents (regardless of the callback). If either referent has been deleted, the references are equal only if the reference objects are the same object.

Changed in version 2.4: This is now a subclassable type rather than a factory function; it derives from object.

**proxy**(object[, callback])

Return a proxy to object which uses a weak reference. This supports use of the proxy in most contexts instead of requiring the explicit dereferencing used with weak reference objects. The returned object will have a type of either ProxyType or CallableProxyType, depending on whether object is callable. Proxy objects are not hashable regardless of the referent; this avoids a number of problems related to their fundamentally mutable nature, and prevent their use as dictionary keys. callback is the same as the parameter of the same name to the ref() function.

**getweakrefcount**(object)

Return the number of weak references and proxies which refer to object.

**getweakrefs**(object)

Return a list of all weak reference and proxy objects which refer to object.

**class WeakKeyDictionary**(dict)

Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key. This can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

**Note:** Caution: Because a WeakKeyDictionary is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a WeakKeyDictionary because actions performed by the program during iteration may cause items in the dictionary to vanish "by magic" (as a side effect of garbage collection).

WeakKeyDictionary objects have the following additional methods. These expose the internal references directly. The references are not guaranteed to be
``live'' at the time they are used, so the result of calling the references needs to be checked before being used. This can be used to avoid creating references that will cause the garbage collector to keep the keys around longer than needed.

**iterkeyrefs()**

Return an iterator that yields the weak references to the keys. New in version 2.5.

**keyrefs()**

Return a list of weak references to the keys. New in version 2.5.

**class WeakValueDictionary([dict])**

Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

**Note:** Caution: Because a WeakValueDictionary is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a WeakValueDictionary because actions performed by the program during iteration may cause items in the dictionary to vanish "by magic" (as a side effect of garbage collection).

WeakValueDictionary objects have the following additional methods. These method have the same issues as the **iterkeyrefs()** and **keyrefs()** methods of WeakKeyDictionary objects.

**itervaluerefs()**

Return an iterator that yields the weak references to the values. New in version 2.5.

**valuerefs()**

Return a list of weak references to the values. New in version 2.5.

**ReferenceType**

The type object for weak references objects.

**ProxyType**

The type object for proxies of objects which are not callable.
CallableProxyType
The type object for proxies of callable objects.

ProxyTypes
Sequence containing all the type objects for proxies. This can make it simpler to test if an object is a proxy without being dependent on naming both proxy types.

exception ReferenceError
Exception raised when a proxy object is used but the underlying object has been collected. This is the same as the standard ReferenceError exception.

See Also:

PEP 0205, Weak References
The proposal and rationale for this feature, including links to earlier implementations and information about similar features in other languages.

Subsections

- 5.11.1 Weak Reference Objects
- 5.11.2 Example
5.11.1 Weak Reference Objects

Weak reference objects have no attributes or methods, but do allow the referent to be obtained, if it still exists, by calling it:

```python
>>> import weakref
>>> class Object:
...    pass
...>>> o = Object()
...>>> r = weakref.ref(o)
...>>> o2 = r()
...>>> o is o2
True
```

If the referent no longer exists, calling the reference object returns `None`:

```python
>>> del o, o2
>>> print r()
None
```

Testing that a weak reference object is still live should be done using the expression `ref() is not None`. Normally, application code that needs to use a reference object should follow this pattern:

```python
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print "Object has been deallocated; can't frobnicate."
else:
    print "Object is still live!"
    o.do_something_useful()
```

Using a separate test for "liveness" creates race conditions in threaded applications; another thread can cause a weak reference to become invalidated before the weak reference is called; the idiom shown above is safe in threaded applications as well as single-threaded applications.

Specialized versions of `ref` objects can be created through subclassing. This is used in the implementation of the `WeakValueDictionary` to reduce the memory overhead for each entry in the mapping. This may be most useful to associate additional information with a reference, but could also be used to insert
additional processing on calls to retrieve the referent.

This example shows how a subclass of `ref` can be used to store additional information about an object and affect the value that's returned when the referent is accessed:

```python
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, **annotations):
        super(ExtendedRef, self).__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.iteritems():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of times the reference has been called."
        ob = super(ExtendedRef, self).__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

Python Library Reference

Previous: 5.11 weakref  Up: 5.11 weakref  Next: 5.11.2 Example

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.11.2 Example

This simple example shows how an application can use objects IDs to retrieve objects that it has seen before. The IDs of the objects can then be used in other data structures without forcing the objects to remain alive, but the objects can still be retrieved by ID if they do.

```python
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```
5.12 UserDict -- Class wrapper for dictionary objects

The module defines a mixin, DictMixin, defining all dictionary methods for classes that already have a minimum mapping interface. This greatly simplifies writing classes that need to be substitutable for dictionaries (such as the shelve module).

This also module defines a class, UserDict, that acts as a wrapper around dictionary objects. The need for this class has been largely supplanted by the ability to subclass directly from dict (a feature that became available starting with Python version 2.2). Prior to the introduction of dict, the UserDict class was used to create dictionary-like sub-classes that obtained new behaviors by overriding existing methods or adding new ones.

The UserDict module defines the UserDict class and DictMixin:

```python
class UserDict([initialdata])
   Class that simulates a dictionary. The instance's contents are kept in a
   regular dictionary, which is accessible via the data attribute of
   UserDict instances. If initialdata is provided, data is initialized with its
   contents; note that a reference to initialdata will not be kept, allowing it be
   used for other purposes. Note: For backward compatibility, instances of
   UserDict are not iterable.
```

```python
class IterableUserDict([initialdata])
   Subclass of UserDict that supports direct iteration (e.g. for key in
   myDict).
```

In addition to supporting the methods and operations of mappings (see section 3.8), UserDict and IterableUserDict instances provide the following attribute:

```python
data
   A real dictionary used to store the contents of the UserDict class.
```
class DictMixin()

    Mixin defining all dictionary methods for classes that already have a minimum dictionary interface including __getitem__(), __setitem__(), __delitem__(), and keys().

    This mixin should be used as a superclass. Adding each of the above methods adds progressively more functionality. For instance, defining all but __delitem__ will preclude only pop and popitem from the full interface.

    In addition to the four base methods, progressively more efficiency comes with defining __contains__(), __iter__(), and iteritems().

    Since the mixin has no knowledge of the subclass constructor, it does not define __init__() or copy().
5.13 UserList -- Class wrapper for list objects

**Note:** This module is available for backward compatibility only. If you are writing code that does not need to work with versions of Python earlier than Python 2.2, please consider subclassing directly from the built-in list type.

This module defines a class that acts as a wrapper around list objects. It is a useful base class for your own list-like classes, which can inherit from them and override existing methods or add new ones. In this way one can add new behaviors to lists.

The UserList module defines the UserList class:

```python
class UserList([list])
    Class that simulates a list. The instance's contents are kept in a regular list, which is accessible via the data attribute of UserList instances. The instance's contents are initially set to a copy of list, defaulting to the empty list []. list can be either a regular Python list, or an instance of UserList (or a subclass).
```

In addition to supporting the methods and operations of mutable sequences (see section 3.6), UserList instances provide the following attribute:

**data**

A real Python list object used to store the contents of the UserList class.

**Subclassing requirements:** Subclasses of UserList are expect to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided
in that case.

Changed in version 2.0: Python versions 1.5.2 and 1.6 also required that the constructor be callable with no parameters, and offer a mutable data attribute. Earlier versions of Python did not attempt to create instances of the derived class.
5.14 UserString -- Class wrapper for string objects

Note: This UserString class from this module is available for backward compatibility only. If you are writing code that does not need to work with versions of Python earlier than Python 2.2, please consider subclassing directly from the built-in str type instead of using UserString (there is no built-in equivalent to MutableString).

This module defines a class that acts as a wrapper around string objects. It is a useful base class for your own string-like classes, which can inherit from them and override existing methods or add new ones. In this way one can add new behaviors to strings.

It should be noted that these classes are highly inefficient compared to real string or Unicode objects; this is especially the case forMutableString.

The UserString module defines the following classes:

class UserString([sequence])

Class that simulates a string or a Unicode string object. The instance's content is kept in a regular string or Unicode string object, which is accessible via the data attribute of UserString instances. The instance's contents are initially set to a copy of sequence. sequence can be either a regular Python string or Unicode string, an instance of UserString (or a subclass) or an arbitrary sequence which can be converted into a string using the built-in str() function.

class MutableString([sequence])

This class is derived from the UserString above and redefines strings to be mutable. Mutable strings can't be used as dictionary keys, because dictionaries require immutable objects as keys. The main intention of this class is to serve as an educational example for inheritance and necessity to remove (override) the __hash__() method in order to trap attempts to use a mutable object as dictionary key, which would be otherwise very error
prone and hard to track down.

In addition to supporting the methods and operations of string and Unicode objects (see section 3.6.1, ``String Methods''), UserString instances provide the following attribute:

**data**

A real Python string or Unicode object used to store the content of the UserString class.
5.15 types -- Names for built-in types

This module defines names for some object types that are used by the standard Python interpreter, but not for the types defined by various extension modules. Also, it does not include some of the types that arise during processing such as the listiterator type. It is safe to use "from types import *" -- the module does not export any names besides the ones listed here. New names exported by future versions of this module will all end in "Type".

Typical use is for functions that do different things depending on their argument types, like the following:

```python
from types import *
def delete(mylist, item):
    if type(item) is IntType:
        del mylist[item]
    else:
        mylist.remove(item)
```

Starting in Python 2.2, built-in factory functions such as int() and str() are also names for the corresponding types. This is now the preferred way to access the type instead of using the types module. Accordingly, the example above should be written as follows:

```python
def delete(mylist, item):
    if isinstance(item, int):
        del mylist[item]
    else:
        mylist.remove(item)
```

The module defines the following names:

**NoneType**
The type of None.

**TypeType**
The type of type objects (such as returned by type( )).
**BooleanType**
The type of the `bool` values `True` and `False`; this is an alias of the built-in `bool()` function. New in version 2.3.

**IntType**
The type of integers (e.g. `1`).

**LongType**
The type of long integers (e.g. `1L`).

**FloatType**
The type of floating point numbers (e.g. `1.0`).

**ComplexType**
The type of complex numbers (e.g. `1.0j`). This is not defined if Python was built without complex number support.

**StringType**
The type of character strings (e.g. `'Spam'`).

**UnicodeType**
The type of Unicode character strings (e.g. `u'Spam'`). This is not defined if Python was built without Unicode support.

**TupleType**
The type of tuples (e.g. `(1, 2, 3, 'Spam')`).

**ListType**
The type of lists (e.g. `[0, 1, 2, 3]`).

**DictType**
The type of dictionaries (e.g. `{ 'Bacon': 1, 'Ham': 0 }`).

**DictionaryType**
An alternate name for `DictType`.

**FunctionType**
The type of user-defined functions and lambdas.

**LambdaType**
An alternate name for FunctionType.

**GeneratorType**
The type of generator-iterator objects, produced by calling a generator function. New in version 2.2.

**CodeType**
The type for code objects such as returned by `compile()`.

**ClassType**
The type of user-defined classes.

**InstanceType**
The type of instances of user-defined classes.

**MethodType**
The type of methods of user-defined class instances.

**UnboundMethodType**
An alternate name for MethodType.

**BuiltinFunctionType**
The type of built-in functions like `len()` or `sys.exit()`.

**BuiltinMethodType**
An alternate name for BuiltinFunction.

**ModuleType**
The type of modules.

**FileType**
The type of open file objects such as `sys.stdout`.

**XRangeType**
The type of range objects returned by `xrange()`.

**SliceType**
The type of objects returned by `slice()`.

**EllipsisType**
The type of **Ellipsis**.

**TracebackType**
The type of traceback objects such as found in `sys.exc_traceback`.

**FrameType**
The type of frame objects such as found in `tb.tb_frame` if `tb` is a traceback object.

**BufferType**
The type of buffer objects created by the `buffer()` function.

**DictProxyType**
The type of dict proxies, such as `TypeError.__dict__`.

**NotImplementedType**
The type of `NotImplemented`.

**GetSetDescriptorType**
The type of objects defined in extension modules with `PyGetSetDef`, such as `FrameType.f_locals` or `array.array.typecode`. This constant is not defined in implementations of Python that do not have such extension types, so for portable code use `hasattr(types, 'GetSetDescriptorType')`. New in version 2.5.

**MemberDescriptorType**
The type of objects defined in extension modules with `PyMemberDef`, such as `datetime.timedelta.days`. This constant is not defined in implementations of Python that do not have such extension types, so for portable code use `hasattr(types, 'MemberDescriptorType')`. New in version 2.5.

**StringTypes**
A sequence containing `StringType` and `UnicodeType` used to facilitate easier checking for any string object. Using this is more portable than using a sequence of the two string types constructed elsewhere since it only contains `UnicodeType` if it has been built in the running version of Python. For example: `isinstance(s, types.StringTypes)`. New in version 2.2.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.16 new -- Creation of runtime internal objects

The `new` module allows an interface to the interpreter object creation functions. This is for use primarily in marshal-type functions, when a new object needs to be created ``magically'' and not by using the regular creation functions. This module provides a low-level interface to the interpreter, so care must be exercised when using this module. It is possible to supply non-sensical arguments which crash the interpreter when the object is used.

The `new` module defines the following functions:

**instance**(`class[, dict]`)  
This function creates an instance of `class` with dictionary `dict` without calling the `__init__() constructor. If `dict` is omitted or `None`, a new, empty dictionary is created for the new instance. Note that there are no guarantees that the object will be in a consistent state.

**instancemethod**(`function, instance, class`)  
This function will return a method object, bound to `instance`, or unbound if `instance` is `None`. `function` must be callable.

**function**(`code, globals[, name[, argdefs[, closure]]]]`)  
Returns a (Python) function with the given code and globals. If `name` is given, it must be a string or `None`. If it is a string, the function will have the given name, otherwise the function name will be taken from `code.co_name`. If `argdefs` is given, it must be a tuple and will be used to determine the default values of parameters. If `closure` is given, it must be `None` or a tuple of cell objects containing objects to bind to the names in `code.co_freevars`.

**code**(`argcount, nlocals, stacksize, flags, codestring, constants, names, varnames, filename, name, firstlineno, lnotab`)  
This function is an interface to the PyCode_New( ) C function.
module( name[, doc])
   This function returns a new module object with name name. name must be a string. The optional doc argument can have any type.

classobj( name, baseclasses, dict)
   This function returns a new class object, with name name, derived from baseclasses (which should be a tuple of classes) and with namespace dict.
5.17 copy -- Shallow and deep copy operations

This module provides generic (shallow and deep) copying operations.

Interface summary:

```python
import copy

x = copy.copy(y)  # make a shallow copy of y
x = copy.deepcopy(y)  # make a deep copy of y
```

For module specific errors, `copy.error` is raised.

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A *shallow copy* constructs a new compound object and then (to the extent possible) inserts *references* into it to the objects found in the original.

- A *deep copy* constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original.

Two problems often exist with deep copy operations that don't exist with shallow copy operations:

- Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.

- Because deep copy copies *everything* it may copy too much, e.g., administrative data structures that should be shared even between copies.

The `deepcopy()` function avoids these problems by:

- keeping a ``memo`` dictionary of objects already copied during the current copying pass; and

- letting user-defined classes override the copying operation or the set of
components copied.

This module does not copy types like module, method, stack trace, stack frame, file, socket, window, array, or any similar types. It does ```copy``` functions and classes (shallow and deeply), by returning the original object unchanged; this is compatible with the way these are treated by the pickle module. Changed in version 2.5: Added copying functions.

Classes can use the same interfaces to control copying that they use to control pickling. See the description of module pickle for information on these methods. The copy module does not use the copy_reg registration module.

In order for a class to define its own copy implementation, it can define special methods __copy__() and __deepcopy__(). The former is called to implement the shallow copy operation; no additional arguments are passed. The latter is called to implement the deep copy operation; it is passed one argument, the memo dictionary. If the __deepcopy__() implementation needs to make a deep copy of a component, it should call the deepcopy() function with the component as first argument and the memo dictionary as second argument.

See Also:

Module pickle:
Discussion of the special methods used to support object state retrieval and restoration.
5.18 pprint -- Data pretty printer

The pprint module provides a capability to `''pretty-print''` arbitrary Python data structures in a form which can be used as input to the interpreter. If the formatted structures include objects which are not fundamental Python types, the representation may not be loadable. This may be the case if objects such as files, sockets, classes, or instances are included, as well as many other built-in objects which are not representable as Python constants.

The formatted representation keeps objects on a single line if it can, and breaks them onto multiple lines if they don't fit within the allowed width. Construct PrettyPrinter objects explicitly if you need to adjust the width constraint.

Changed in version 2.5: Dictionaries are sorted by key before the display is computed; before 2.5, a dictionary was sorted only if its display required more than one line, although that wasn't documented.

The pprint module defines one class:

```python
class PrettyPrinter(...)

Construct a PrettyPrinter instance. This constructor understands several keyword parameters. An output stream may be set using the stream keyword; the only method used on the stream object is the file protocol's write() method. If not specified, the PrettyPrinter adopts sys.stdout. Three additional parameters may be used to control the formatted representation. The keywords are indent, depth, and width. The amount of indentation added for each recursive level is specified by indent; the default is one. Other values can cause output to look a little odd, but can make nesting easier to spot. The number of levels which may be printed is controlled by depth; if the data structure being printed is too deep, the next contained level is replaced by "...". By default, there is no constraint on the depth of the objects being formatted. The desired output width is constrained using the width parameter; the default is eighty characters. If a structure cannot be formatted within the constrained width, a best effort will be made.
```

```python
>>> import pprint, sys
```
>>> stuff = sys.path[:]
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[  
   ['/usr/local/lib/python1.5',
    '/usr/local/lib/python1.5/test',
    '/usr/local/lib/python1.5/sunos5',
    '/usr/local/lib/python1.5/sharedmodules',
    '/usr/local/lib/python1.5/tkinter'],
   '',
   ['/usr/local/lib/python1.5',
    '/usr/local/lib/python1.5/test',
    '/usr/local/lib/python1.5/sunos5',
    '/usr/local/lib/python1.5/sharedmodules',
    '/usr/local/lib/python1.5/tkinter']
]

The `PrettyPrinter` class supports several derivative functions:

**pformat**( `object[, indent[, width[, depth]]]` )

Return the formatted representation of `object` as a string. `indent`, `width` and `depth` will be passed to the `PrettyPrinter` constructor as formatting parameters. Changed in version 2.4: The parameters `indent`, `width` and `depth` were added.

**pprint**( `object[, stream[, indent[, width[, depth]]]]` )

Prints the formatted representation of `object` on `stream`, followed by a newline. If `stream` is omitted, `sys.stdout` is used. This may be used in the interactive interpreter instead of a `print` statement for inspecting values. `indent`, `width` and `depth` will be passed to the `PrettyPrinter` constructor as formatting parameters.

```python
>>> stuff = sys.path[:]
>>> stuff.insert(0, stuff[:])
>>> pprint.pprint(stuff)
[<Recursion on list with id=869440>,
   '',
   '/usr/local/lib/python1.5',
   '/usr/local/lib/python1.5/test',
]```
Changed in version 2.4: The parameters \textit{indent}, \textit{width} and \textit{depth} were added.

\textbf{isreadable}(\textit{object})
Determine if the formatted representation of \textit{object} is `\texttt{readable}," or can be used to reconstruct the value using \texttt{eval}(). This always returns false for recursive objects.

\begin{verbatim}
>>> pprint.isreadable(stuff)
False
\end{verbatim}

\textbf{isrecursive}(\textit{object})
Determine if \textit{object} requires a recursive representation.

One more support function is also defined:

\textbf{saferepr}(\textit{object})
Return a string representation of \textit{object}, protected against recursive data structures. If the representation of \textit{object} exposes a recursive entry, the recursive reference will be represented as "\texttt{<Recursion on typename with id=number>}". The representation is not otherwise formatted.

\begin{verbatim}
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=682968>, '', '/usr/local/lib/python1.5/lib/python1.5/test', '/usr/local/lib/python1.5/sunos5', '/usr/local/lib/python1.5/sharedmodules', '/usr/local/lib/python1.5/tkinter']"
\end{verbatim}
5.18.1 PrettyPrinter Objects

PrettyPrinter instances have the following methods:

**pformat**(object)
  Return the formatted representation of *object*. This takes into account the options passed to the PrettyPrinter constructor.

**pprint**(object)
  Print the formatted representation of *object* on the configured stream, followed by a newline.

The following methods provide the implementations for the corresponding functions of the same names. Using these methods on an instance is slightly more efficient since new PrettyPrinter objects don't need to be created.

**isreadable**(object)
  Determine if the formatted representation of the object is "readable," or can be used to reconstruct the value using `eval()`. Note that this returns false for recursive objects. If the *depth* parameter of the PrettyPrinter is set and the object is deeper than allowed, this returns false.

**isrecursive**(object)
  Determine if the object requires a recursive representation.

This method is provided as a hook to allow subclasses to modify the way objects are converted to strings. The default implementation uses the internals of the `saferepr()` implementation.

**format**(object, context, maxlevels, level)
  Returns three values: the formatted version of *object* as a string, a flag indicating whether the result is readable, and a flag indicating whether recursion was detected. The first argument is the object to be presented. The second is a dictionary which contains the `id()` of objects that are part of the current presentation context (direct and indirect containers for *object* that are affecting the presentation) as the keys; if an object needs to be
presented which is already represented in context, the third return value should be true. Recursive calls to the format( ) method should add additional entries for containers to this dictionary. The third argument, maxlevels, gives the requested limit to recursion; this will be 0 if there is no requested limit. This argument should be passed unmodified to recursive calls. The fourth argument, level, gives the current level; recursive calls should be passed a value less than that of the current call. New in version 2.3.
5.19 repr -- Alternate repr() implementation

The `repr` module provides a means for producing object representations with limits on the size of the resulting strings. This is used in the Python debugger and may be useful in other contexts as well.

This module provides a class, an instance, and a function:

```python
class Repr()
    Class which provides formatting services useful in implementing functions similar to the built-in `repr()`; size limits for different object types are added to avoid the generation of representations which are excessively long.
```

```python
aRepr
    This is an instance of Repr which is used to provide the `repr()` function described below. Changing the attributes of this object will affect the size limits used by `repr()` and the Python debugger.
```

```python
repr(obj)
    This is the `repr()` method of aRepr. It returns a string similar to that returned by the built-in function of the same name, but with limits on most sizes.
```

Subsections

- 5.19.1 Repr Objects
- 5.19.2 Subclassing Repr Objects
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.19.1 Repr Objects

Repr instances provide several members which can be used to provide size limits for the representations of different object types, and methods which format specific object types.

**maxlevel**
Depth limit on the creation of recursive representations. The default is 6.

**maxdict**
**maxlist**
**maxtuple**
**maxset**
**maxfrozenset**
**maxdeque**
**maxarray**
Limits on the number of entries represented for the named object type. The default is 4 for `maxdict`, 5 for `maxarray`, and 6 for the others. New in version 2.4: `maxset`, `maxfrozenset`, and `set`.

**maxlong**
Maximum number of characters in the representation for a long integer. Digits are dropped from the middle. The default is 40.

**maxstring**
Limit on the number of characters in the representation of the string. Note that the ``normal`` representation of the string is used as the character source: if escape sequences are needed in the representation, these may be mangled when the representation is shortened. The default is 30.

**maxother**
This limit is used to control the size of object types for which no specific formatting method is available on the Repr object. It is applied in a similar manner as `maxstring`. The default is 20.

**repr(obj)**
The equivalent to the built-in `repr()` that uses the formatting imposed by
the instance.

**repr1**( *obj, level*)

Recursive implementation used by **repr**(). This uses the type of *obj* to determine which formatting method to call, passing it *obj* and *level*. The type-specific methods should call **repr1()** to perform recursive formatting, with *level* − 1 for the value of *level* in the recursive call.

**repr_type**( *obj, level*)

Formatting methods for specific types are implemented as methods with a name based on the type name. In the method name, *type* is replaced by

```
string.join(string.split(type(obj).__name__, '_'))
```

Dispatch to these methods is handled by **repr1()**. Type-specific methods which need to recursively format a value should call

```
"self.repr1(subobj, level - 1)"
```
5.19.2 Subclassing Repr Objects

The use of dynamic dispatching by `Repr.repr1()` allows subclasses of `Repr` to add support for additional built-in object types or to modify the handling of types already supported. This example shows how special support for file objects could be added:

```python
import repr
import sys

class MyRepr(repr.Repr):
    def repr_file(self, obj, level):
        if obj.name in ['<stdin>', '<stdout>', '<stderr>']:
            return obj.name
        else:
            return `obj`

aRepr = MyRepr()
print aRepr.repr(sys.stdin)  # prints '<stdin>'
```

---

**Python Library Reference**

Previous: [5.19.1 Repr Objects](#)  Up: [5.19 repr](#)  Next: [6. Numeric and Mathematical](#)

*Release 2.5, documentation updated on 19th September, 2006.*

See [About this document...](#) for information on suggesting changes.
6. Numeric and Mathematical Modules

The modules described in this chapter provide numeric and math-related functions and data types. The `math` and `cmath` contain various mathematical functions for floating-point and complex numbers. For users more interested in decimal accuracy than in speed, the `decimal` module supports exact representations of decimal numbers.

The following modules are documented in this chapter:

- **math**: Mathematical functions (\(\sin()\) etc.).
- **cmath**: Mathematical functions for complex numbers.
- **random**: Generate pseudo-random numbers with various common distributions.
- **itertools**: Functions creating iterators for efficient looping.
- **functools**: Higher-order functions and operations on callable objects.
- **operator**: All Python's standard operators as built-in functions.
6.1 math -- Mathematical functions

This module is always available. It provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the cmath module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

Number-theoretic and representation functions:

**ceil(x)**

Return the ceiling of \( x \) as a float, the smallest integer value greater than or equal to \( x \).

**fabs(x)**

Return the absolute value of \( x \).

**floor(x)**

Return the floor of \( x \) as a float, the largest integer value less than or equal to \( x \).

**fmod(x, y)**

Return \( \text{fmod}(x, y) \), as defined by the platform C library. Note that the Python expression \( x \% y \) may not return the same result. The intent of the C standard is that \( \text{fmod}(x, y) \) be exactly (mathematically; to infinite precision) equal to \( x - n \cdot y \) for some integer \( n \) such that the result has the same sign as \( x \) and magnitude less than \( \text{abs}(y) \). Python's \( x \% y \) returns a
result with the sign of $y$ instead, and may not be exactly computable for float arguments. For example, $\text{fmod}(-1\times10^{-100}, 1\times10^{100})$ is $-1\times10^{-100}$, but the result of Python's $\text{mod}(-1\times10^{-100}, 1\times10^{100})$ is $1\times10^{-100} - 1\times10^{-100}$, which cannot be represented exactly as a float, and rounds to the surprising $1\times10^{100}$. For this reason, function $\text{fmod}()$ is generally preferred when working with floats, while Python's $x \mod y$ is preferred when working with integers.

$f\text{rexp}(x)$

Return the mantissa and exponent of $x$ as the pair $(m, e)$. $m$ is a float and $e$ is an integer such that $x = m \times 2^e$ exactly. If $x$ is zero, returns $(0.0, 0)$, otherwise $0.5 \leq |m| < 1$. This is used to "pick apart" the internal representation of a float in a portable way.

$l\text{dexp}(x, i)$

Return $x \times (2^i)$. This is essentially the inverse of function $f\text{rexp}()$.

$\text{modf}(x)$

Return the fractional and integer parts of $x$. Both results carry the sign of $x$, and both are floats.

Note that $f\text{rexp}()$ and $\text{modf}()$ have a different call/return pattern than their C equivalents: they take a single argument and return a pair of values, rather than returning their second return value through an `output parameter' (there is no such thing in Python).

For the $\text{ceil}()$, $\text{floor}()$, and $\text{modf}()$ functions, note that all floating-point numbers of sufficiently large magnitude are exact integers. Python floats typically carry no more than 53 bits of precision (the same as the platform C double type), in which case any float $x$ with $\text{abs}(x) \geq 2^{52}$ necessarily has no fractional bits.

Power and logarithmic functions:

$\text{exp}(x)$

Return $e^{x}$.

$\text{log}(x[, \text{base}])$
Return the logarithm of $x$ to the given base. If the base is not specified, return the natural logarithm of $x$ (that is, the logarithm to base $e$). Changed in version 2.3: base argument added.

\texttt{log10}(x)

Return the base-10 logarithm of $x$.

\texttt{pow}(x, y)

Return $x^{**}y$.

\texttt{sqrt}(x)

Return the square root of $x$.

Trigonometric functions:

\texttt{acos}(x)

Return the arc cosine of $x$, in radians.

\texttt{asin}(x)

Return the arc sine of $x$, in radians.

\texttt{atan}(x)

Return the arc tangent of $x$, in radians.

\texttt{atan2}(y, x)

Return $\texttt{atan}(y / x)$, in radians. The result is between -$\pi$ and $\pi$. The vector in the plane from the origin to point $(x, y)$ makes this angle with the positive $X$ axis. The point of $\texttt{atan2}()$ is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, $\texttt{atan}(1)$ and $\texttt{atan2}(1, 1)$ are both $\pi/4$, but $\texttt{atan2}(-1, -1)$ is $-3\pi/4$.

\texttt{cos}(x)

Return the cosine of $x$ radians.

\texttt{hypot}(x, y)

Return the Euclidean norm, $\sqrt{x^x + y^y}$. This is the length of the
vector from the origin to point \((x, y)\).

\(\sin(x)\)
Return the sine of \(x\) radians.

\(\tan(x)\)
Return the tangent of \(x\) radians.

Angular conversion:

\(\text{degrees}(x)\)
Converts angle \(x\) from radians to degrees.

\(\text{radians}(x)\)
Converts angle \(x\) from degrees to radians.

Hyperbolic functions:

\(\cosh(x)\)
Return the hyperbolic cosine of \(x\).

\(\sinh(x)\)
Return the hyperbolic sine of \(x\).

\(\tanh(x)\)
Return the hyperbolic tangent of \(x\).

The module also defines two mathematical constants:

\(\pi\)
The mathematical constant \(\pi\).

\(e\)
The mathematical constant \(e\).

**Note:** The `math` module consists mostly of thin wrappers around the platform C math library functions. Behavior in
exceptional cases is loosely specified by the C standards, and Python inherits much of its math-function error-reporting behavior from the platform C implementation. As a result, the specific exceptions raised in error cases (and even whether some arguments are considered to be exceptional at all) are not defined in any useful cross-platform or cross-release way. For example, whether `math.log(0)` returns `-Inf` or raises `ValueError` or `OverflowError` isn't defined, and in cases where `math.log(0)` raises `OverflowError`, `math.log(0L)` may raise `ValueError` instead.

See Also:

Module **cmath**: Complex number versions of many of these functions.
6.2 cmath -- Mathematical functions for complex numbers

This module is always available. It provides access to mathematical functions for complex numbers. The functions are:

acos(x)

Return the arc cosine of x. There are two branch cuts: One extends right from 1 along the real axis to $\infty$, continuous from below. The other extends left from -1 along the real axis to -$\infty$, continuous from above.

acosh(x)

Return the hyperbolic arc cosine of x. There is one branch cut, extending left from 1 along the real axis to -$\infty$, continuous from above.

asin(x)

Return the arc sine of x. This has the same branch cuts as acos().

asinh(x)

Return the hyperbolic arc sine of x. There are two branch cuts, extending left from $\pm 1j$ to $\pm -\infty j$, both continuous from above. These branch cuts should be considered a bug to be corrected in a future release. The correct branch cuts should extend along the imaginary axis, one from $1j$ up to $\infty j$ and continuous from the right, and one from $-1j$ down to $-\infty j$ and continuous from the left.

atan(x)

Return the arc tangent of x. There are two branch cuts: One extends from $1j$ along the imaginary axis to $\infty j$, continuous from the left. The other extends from $-1j$ along the imaginary axis to -$\infty j$, continuous from the left. (This should probably be changed so the upper cut becomes continuous from the other side.)

atanh(x)
Return the hyperbolic arc tangent of \( x \). There are two branch cuts: One extends from 1 along the real axis to \( \infty \), continuous from above. The other extends from -1 along the real axis to \( -\infty \), continuous from above. (This should probably be changed so the right cut becomes continuous from the other side.)

\[
\text{cos}(x)
\]
Return the cosine of \( x \).

\[
\text{cosh}(x)
\]
Return the hyperbolic cosine of \( x \).

\[
\text{exp}(x)
\]
Return the exponential value \( e^{**}x \).

\[
\text{log}(x[, \text{base}])
\]
Returns the logarithm of \( x \) to the given \( \text{base} \). If the \( \text{base} \) is not specified, returns the natural logarithm of \( x \). There is one branch cut, from 0 along the negative real axis to \( -\infty \), continuous from above. Changed in version 2.4: \( \text{base} \) argument added.

\[
\text{log10}(x)
\]
Return the base-10 logarithm of \( x \). This has the same branch cut as \( \text{log()} \).

\[
\text{sin}(x)
\]
Return the sine of \( x \).

\[
\text{sinh}(x)
\]
Return the hyperbolic sine of \( x \).

\[
\text{sqrt}(x)
\]
Return the square root of \( x \). This has the same branch cut as \( \text{log()} \).

\[
\text{tan}(x)
\]
Return the tangent of \( x \).

\[
\text{tanh}(x)
\]
Return the hyperbolic tangent of \( x \).

The module also defines two mathematical constants:

\textbf{\textit{\texttt{pi}}}

The mathematical constant \( \pi \), as a real.

\textbf{\textit{\texttt{e}}}

The mathematical constant \( e \), as a real.

Note that the selection of functions is similar, but not identical, to that in module \texttt{math}. The reason for having two modules is that some users aren't interested in complex numbers, and perhaps don't even know what they are. They would rather have \texttt{math.sqrt(-1)} raise an exception than return a complex number. Also note that the functions defined in \texttt{cmath} always return a complex number, even if the answer can be expressed as a real number (in which case the complex number has an imaginary part of zero).

A note on branch cuts: They are curves along which the given function fails to be continuous. They are a necessary feature of many complex functions. It is assumed that if you need to compute with complex functions, you will understand about branch cuts. Consult almost any (not too elementary) book on complex variables for enlightenment. For information of the proper choice of branch cuts for numerical purposes, a good reference should be the following:

\begin{quote}
\textbf{See Also:}

\end{quote}
The `decimal` module provides support for decimal floating point arithmetic. It offers several advantages over the `float()` datatype:

- Decimal numbers can be represented exactly. In contrast, numbers like `1.1` do not have an exact representation in binary floating point. End users typically would not expect `1.1` to display as `1.1000000000000001` as it does with binary floating point.

- The exactness carries over into arithmetic. In decimal floating point, "0.1 + 0.1 + 0.1 - 0.3" is exactly equal to zero. In binary floating point, result is `5.5511151231257827e-017`. While near to zero, the differences prevent reliable equality testing and differences can accumulate. For this reason, decimal would be preferred in accounting applications which have strict equality invariants.

- The decimal module incorporates a notion of significant places so that "1.30 + 1.20" is `2.50`. The trailing zero is kept to indicate significance. This is the customary presentation for monetary applications. For multiplication, the ``schoolbook`` approach uses all the figures in the multiplicands. For instance, "1.3 * 1.2" gives `1.56` while "1.30 * 1.20" gives `1.5600`.

- Unlike hardware based binary floating point, the decimal module has a user settable precision (defaulting to 28 places) which can be as large as needed for a given problem:

```python
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal("0.142857")
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal("0.1428571428571428571428571429")
```
Both binary and decimal floating point are implemented in terms of published standards. While the built-in float type exposes only a modest portion of its capabilities, the decimal module exposes all required parts of the standard. When needed, the programmer has full control over rounding and signal handling.

The module design is centered around three concepts: the decimal number, the context for arithmetic, and signals.

A decimal number is immutable. It has a sign, coefficient digits, and an exponent. To preserve significance, the coefficient digits do not truncate trailing zeroes. Decimals also include special values such as \texttt{Infinity}, \texttt{-Infinity}, and \texttt{NaN}. The standard also differentiates \texttt{-0} from \texttt{+0}.

The context for arithmetic is an environment specifying precision, rounding rules, limits on exponents, flags indicating the results of operations, and trap enablers which determine whether signals are treated as exceptions. Rounding options include \texttt{ROUND_CEILING}, \texttt{ROUND_DOWN}, \texttt{ROUND_FLOOR}, \texttt{ROUND_HALF_DOWN}, \texttt{ROUND_HALF_EVEN}, \texttt{ROUND_HALF_UP}, and \texttt{ROUND_UP}.

Signals are groups of exceptional conditions arising during the course of computation. Depending on the needs of the application, signals may be ignored, considered as informational, or treated as exceptions. The signals in the decimal module are: \texttt{Clamped}, \texttt{InvalidOperation}, \texttt{DivisionByZero}, \texttt{Inexact}, \texttt{Rounded}, \texttt{Subnormal}, \texttt{Overflow}, and \texttt{Underflow}.

For each signal there is a flag and a trap enabler. When a signal is encountered, its flag is incremented from zero and, then, if the trap enabler is set to one, an exception is raised. Flags are sticky, so the user needs to reset them before monitoring a calculation.

\textbf{See Also:}


Subsections

- **6.3.1 Quick-start Tutorial**
- **6.3.2 Decimal objects**
- **6.3.3 Context objects**
- **6.3.4 Signals**
- **6.3.5 Floating Point Notes**
  - **6.3.5.1 Mitigating round-off error with increased precision**
  - **6.3.5.2 Special values**
- **6.3.6 Working with threads**
- **6.3.7 Recipes**
- **6.3.8 Decimal FAQ**
6.3.1 Quick-start Tutorial

The usual start to using decimals is importing the module, viewing the current context with `getcontext()` and, if necessary, setting new values for precision, rounding, or enabled traps:

```python
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999, capitals=1, flags=[], traps=[Overflow, InvalidOperation, DivisionByZero])
```

```python
>>> getcontext().prec = 7  # Set a new precision
```

Decimal instances can be constructed from integers, strings, or tuples. To create a Decimal from a float, first convert it to a string. This serves as an explicit reminder of the details of the conversion (including representation error).

Decimal numbers include special values such as NaN which stands for ```Not a number''`, positive and negative Infinity, and -0.

```python
>>> Decimal(10)
Decimal("10")
>>> Decimal("3.14")
Decimal("3.14")
>>> Decimal((0, (3, 1, 4), -2))
Decimal("3.14")
>>> Decimal(str(2.0 ** 0.5))
Decimal("1.41421356237")
>>> Decimal("NaN")
Decimal("NaN")
>>> Decimal("-Infinity")
Decimal("-Infinity")
```

The significance of a new Decimal is determined solely by the number of digits input. Context precision and rounding only come into play during arithmetic operations.

```python
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal("3.0")
>>> Decimal('3.1415926535')
Decimal("3.1415926535")
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal("5.85987")
>>> getcontext().rounding = ROUND_UP
```
Decimals interact well with much of the rest of Python. Here is a small decimal floating point flying circus:

```python
>>> data = map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split())
>>> max(data)
Decimal("9.25")
>>> min(data)
Decimal("0.03")
>>> sorted(data)
[Decimal("0.03"), Decimal("1.00"), Decimal("1.34"), Decimal("1.87"),
 Decimal("2.35"), Decimal("3.45"), Decimal("9.25")]
>>> sum(data)
Decimal("19.29")
>>> a, b, c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.3400000000000001
>>> round(a, 1)  # round() first converts to binary floating point
1.3
>>> int(a)
1
>>> a * 5
Decimal("6.70")
>>> a * b
Decimal("2.5058")
>>> c % a
Decimal("0.77")
```

The `quantize()` method rounds a number to a fixed exponent. This method is useful for monetary applications that often round results to a fixed number of places:

```python
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal("7.32")
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal("8")
```

As shown above, the `getcontext()` function accesses the current context and allows the settings to be changed. This approach meets the needs of most applications.

For more advanced work, it may be useful to create alternate contexts using the
Context() constructor. To make an alternate active, use the setcontext() function.

In accordance with the standard, the Decimal module provides two ready to use standard contexts, BasicContext and ExtendedContext. The former is especially useful for debugging because many of the traps are enabled:

```python
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal("0.142857142857142857142857142857142857142857142857142857142857")
>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999, capitals=1, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal("0.142857143")
>>> Decimal(42) / Decimal(0)
Decimal("Infinity")
>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

Contexts also have signal flags for monitoring exceptional conditions encountered during computations. The flags remain set until explicitly cleared, so it is best to clear the flags before each set of monitored computations by using the clear_flags() method.

```python
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal("3.14159292")
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999, capitals=1, flags=[Inexact, Rounded], traps=[])```

The flags entry shows that the rational approximation to Pi was rounded (digits beyond the context precision were thrown away) and that the result is inexact (some of the discarded digits were non-zero).

Individual traps are set using the dictionary in the traps field of a context:
>>> Decimal(1) / Decimal(0)
Decimal("Infinity")

>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0

Most programs adjust the current context only once, at the beginning of the program. And, in many applications, data is converted to Decimal with a single cast inside a loop. With context set and decimals created, the bulk of the program manipulates the data no differently than with other Python numeric types.
6.3.2 Decimal objects

**class Decimal([value [, context]])**

Constructs a new Decimal object based from `value`.

`value` can be an integer, string, tuple, or another Decimal object. If no `value` is given, returns Decimal("0"). If `value` is a string, it should conform to the decimal numeric string syntax:

```
sign ::= '+' | '-'
digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6'
indicator ::= 'e' | 'E'
digits ::= digit [digit]...
decimal-part ::= digits '.' [digits] | ['.' ]digits
exponent-part ::= indicator [sign] digits
infinity ::= 'Infinity' | 'Inf'
nan ::= 'NaN' [digits] | 'sNaN' [digits]
numeric-value ::= decimal-part [exponent-part] | infinity
numeric-string ::= [sign] numeric-value | [sign] nan
```

If `value` is a tuple, it should have three components, a sign (0 for positive or 1 for negative), a tuple of digits, and an integer exponent. For example, "Decimal((0, (1, 4, 1, 4), -3))" returns Decimal("1.414").

The `context` precision does not affect how many digits are stored. That is determined exclusively by the number of digits in `value`. For example, "Decimal("3.00000")" records all five zeroes even if the context precision is only three.

The purpose of the `context` argument is determining what to do if `value` is a malformed string. If the context traps `InvalidOperation`, an exception is raised; otherwise, the constructor returns a new Decimal with the value of NaN.

Once constructed, Decimal objects are immutable.

Decimal floating point objects share many properties with the other builtin numeric types such as float and int. All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed,
used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as float or long).

In addition to the standard numeric properties, decimal floating point objects also have a number of specialized methods:

adjusted()  
Return the adjusted exponent after shifting out the coefficient's rightmost digits until only the lead digit remains:  
Decimal("321e+5").adjusted() returns seven. Used for determining the position of the most significant digit with respect to the decimal point.

as_tuple()  
Returns a tuple representation of the number: "(sign, digittuple, exponent)".

compare(other[, context])  
Compares like __cmp__() but returns a decimal instance:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a or b is a NaN</td>
<td>Decimal(&quot;NaN&quot;)</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>Decimal(&quot;-1&quot;)</td>
</tr>
<tr>
<td>a == b</td>
<td>Decimal(&quot;0&quot;)</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>Decimal(&quot;1&quot;)</td>
</tr>
</tbody>
</table>

max(other[, context])  
Like "max(self, other)" except that the context rounding rule is applied before returning and that NaN values are either signalled or ignored (depending on the context and whether they are signaling or quiet).

min(other[, context])  
Like "min(self, other)" except that the context rounding rule is applied before returning and that NaN values are either signalled or ignored (depending on the context and whether they are signaling or quiet).

normalize([context])  
Normalize the number by stripping the rightmost trailing zeroes and converting any result equal to Decimal("0") to Decimal("0e0").
Used for producing canonical values for members of an equivalence class. For example, Decimal("32.100") and Decimal("0.321000e+2") both normalize to the equivalent value Decimal("32.1").

**quantize**(*exp[, rounding[, context[, watchexp]]]*)
Quantize makes the exponent the same as exp. Searches for a rounding method in rounding, then in context, and then in the current context.

If watchexp is set (default), then an error is returned whenever the resulting exponent is greater than Emax or less than Etiny.

**remainder_near**(*other[, context]*)
Computes the modulo as either a positive or negative value depending on which is closest to zero. For instance, "Decimal(10).remainder_near(6)" returns Decimal("-2") which is closer to zero than Decimal("4").

If both are equally close, the one chosen will have the same sign as self.

**same_quantum**(*other[, context]*)
Test whether self and other have the same exponent or whether both are NaN.

**sqrt**([context])
Return the square root to full precision.

**to_eng_string**([context])
Convert to an engineering-type string.

Engineering notation has an exponent which is a multiple of 3, so there are up to 3 digits left of the decimal place. For example, converts Decimal('123E+1') to Decimal("1.23E+3")

**to_integral**([rounding[, context]])
Rounds to the nearest integer without signaling Inexact or Rounded. If given, applies rounding; otherwise, uses the rounding method in either the
supplied context or the current context.
6.3.3 Context objects

Contexts are environments for arithmetic operations. They govern precision, set rules for rounding, determine which signals are treated as exceptions, and limit the range for exponents.

Each thread has its own current context which is accessed or changed using the `getcontext()` and `setcontext()` functions:

**getcontext()**

Return the current context for the active thread.

**setcontext(c)**

Set the current context for the active thread to `c`.

Beginning with Python 2.5, you can also use the `with` statement and the `localcontext()` function to temporarily change the active context.

**localcontext([c])**

Return a context manager that will set the current context for the active thread to a copy of `c` on entry to the with-statement and restore the previous context when exiting the with-statement. If no context is specified, a copy of the current context is used. New in version 2.5.

For example, the following code sets the current decimal precision to 42 places, performs a calculation, and then automatically restores the previous context:

```python
from __future__ import with_statement
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42  # Perform a high precision calculation
    s = calculate_something()
    s = +s  # Round the final result back to the default precision
```

New contexts can also be created using the `Context` constructor described below. In addition, the module provides three pre-made contexts:
class BasicContext
This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to ROUND_HALF_UP. All flags are cleared. All traps are enabled (treated as exceptions) except Inexact, Rounded, and Subnormal.

Because many of the traps are enabled, this context is useful for debugging.

class ExtendedContext
This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to ROUND_HALF_EVEN. All flags are cleared. No traps are enabled (so that exceptions are not raised during computations).

Because the trapped are disabled, this context is useful for applications that prefer to have result value of NaN or Infinity instead of raising exceptions. This allows an application to complete a run in the presence of conditions that would otherwise halt the program.

class DefaultContext
This context is used by the Context constructor as a prototype for new contexts. Changing a field (such a precision) has the effect of changing the default for new contexts creating by the Context constructor.

This context is most useful in multi-threaded environments. Changing one of the fields before threads are started has the effect of setting system-wide defaults. Changing the fields after threads have started is not recommended as it would require thread synchronization to prevent race conditions.

In single threaded environments, it is preferable to not use this context at all. Instead, simply create contexts explicitly as described below.

The default values are precision=28, rounding=ROUND_HALF_EVEN, and enabled traps for Overflow, InvalidOperation, and DivisionByZero.

In addition to the three supplied contexts, new contexts can be created with the Context constructor.

prec=None, rounding=None, traps=None, flags=None,
class Context(Emin=None, Emax=None, capitals=1)

Creates a new context. If a field is not specified or is None, the default values are copied from the DefaultContext. If the flags field is not specified or is None, all flags are cleared.

The prec field is a positive integer that sets the precision for arithmetic operations in the context.

The rounding option is one of:

- ROUND_CEILING (towards Infinity),
- ROUND_DOWN (towards zero),
- ROUND_FLOOR (towards -Infinity),
- ROUND_HALF_DOWN (to nearest with ties going towards zero),
- ROUND_HALF_EVEN (to nearest with ties going to nearest even integer),
- ROUND_HALF_UP (to nearest with ties going away from zero), or
- ROUND_UP (away from zero).

The traps and flags fields list any signals to be set. Generally, new contexts should only set traps and leave the flags clear.

The Emin and Emax fields are integers specifying the outer limits allowable for exponents.

The capitals field is either 0 or 1 (the default). If set to 1, exponents are printed with a capital E; otherwise, a lowercase e is used: Decimal('6.02e+23').

The Context class defines several general purpose methods as well as a large number of methods for doing arithmetic directly in a given context.

**clear_flags()**

Resets all of the flags to 0.

**copy()**

Return a duplicate of the context.
**create_decimal**( *num*)  
Creates a new Decimal instance from *num* but using *self* as context. Unlike the Decimal constructor, the context precision, rounding method, flags, and traps are applied to the conversion.

This is useful because constants are often given to a greater precision than is needed by the application. Another benefit is that rounding immediately eliminates unintended effects from digits beyond the current precision. In the following example, using unrounded inputs means that adding zero to a sum can change the result:

```python  
>>> getcontext().prec = 3  
>>> Decimal("3.4445") + Decimal("1.0023")  
Decimal("4.45")  
>>> Decimal("3.4445") + Decimal(0) + Decimal("1.0023")  
Decimal("4.44")
```

**Etiny()**  
Returns a value equal to "Emin - prec + 1" which is the minimum exponent value for subnormal results. When underflow occurs, the exponent is set to Etiny.

**Etop()**  
Returns a value equal to "Emax - prec + 1".

The usual approach to working with decimals is to create Decimal instances and then apply arithmetic operations which take place within the current context for the active thread. An alternate approach is to use context methods for calculating within a specific context. The methods are similar to those for the Decimal class and are only briefly recounted here.

**abs**( *x*)  
Returns the absolute value of *x*.

**add**( *x*, *y*)  
Return the sum of *x* and *y*.

**compare**( *x*, *y*)  
Compares values numerically.
Like \_\_cmp\_\_() but returns a decimal instance:

\begin{align*}
  &a \text{ or } b \text{ is a NaN} \implies \text{Decimal("NaN")} \\
  &a < b \implies \text{Decimal("-1")} \\
  &a == b \implies \text{Decimal("0")} \\
  &a > b \implies \text{Decimal("1")}
\end{align*}

divide(x, y)
Return x divided by y.

divmod(x, y)
Divides two numbers and returns the integer part of the result.

max(x, y)
Compare two values numerically and return the maximum.
If they are numerically equal then the left-hand operand is chosen as the result.

min(x, y)
Compare two values numerically and return the minimum.
If they are numerically equal then the left-hand operand is chosen as the result.

minus(x)
Minus corresponds to the unary prefix minus operator in Python.

multiply(x, y)
Return the product of x and y.

normalize(x)
Normalize reduces an operand to its simplest form.
Essentially a plus operation with all trailing zeros removed from the result.

plus(x)
Plus corresponds to the unary prefix plus operator in Python. This operation
applies the context precision and rounding, so it is not an identity operation.

**power**(*x*, *y*[,*modulo]*)

Return "x ** y" to the *modulo* if given.

The right-hand operand must be a whole number whose integer part (after any exponent has been applied) has no more than 9 digits and whose fractional part (if any) is all zeros before any rounding. The operand may be positive, negative, or zero; if negative, the absolute value of the power is used, and the left-hand operand is inverted (divided into 1) before use.

If the increased precision needed for the intermediate calculations exceeds the capabilities of the implementation then an *InvalidOperation* condition is signaled.

If, when raising to a negative power, an underflow occurs during the division into 1, the operation is not halted at that point but continues.

**quantize**(*x*, *y*)

Returns a value equal to *x* after rounding and having the exponent of *y*.

Unlike other operations, if the length of the coefficient after the quantize operation would be greater than precision, then an *InvalidOperation* is signaled. This guarantees that, unless there is an error condition, the quantized exponent is always equal to that of the right-hand operand.

Also unlike other operations, quantize never signals Underflow, even if the result is subnormal and inexact.

**remainder**(*x*, *y*)

Returns the remainder from integer division.

The sign of the result, if non-zero, is the same as that of the original dividend.

**remainder_near**(*x*, *y*)

Computed the modulo as either a positive or negative value depending on which is closest to zero. For instance,
"Decimal(10).remainder_near(6)" returns Decimal("-2")
which is closer to zero than Decimal("4").

If both are equally close, the one chosen will have the same sign as self.

same_quantum(x, y)
Test whether x and y have the same exponent or whether both are NaN.

sqrt(x)
Return the square root of x to full precision.

subtract(x, y)
Return the difference between x and y.

to_eng_string()
Convert to engineering-type string.

Engineering notation has an exponent which is a multiple of 3, so there are
up to 3 digits left of the decimal place. For example, converts
Decimal('123E+1') to Decimal("1.23E+3")

to_integral(x)
Rounds to the nearest integer without signaling Inexact or Rounded.

to_sci_string(x)
Converts a number to a string using scientific notation.
6.3.4 Signals

Signals represent conditions that arise during computation. Each corresponds to one context flag and one context trap enabler.

The context flag is incremented whenever the condition is encountered. After the computation, flags may be checked for informational purposes (for instance, to determine whether a computation was exact). After checking the flags, be sure to clear all flags before starting the next computation.

If the context's trap enabler is set for the signal, then the condition causes a Python exception to be raised. For example, if the DivisionByZero trap is set, then a DivisionByZero exception is raised upon encountering the condition.

class Clamped
   Altered an exponent to fit representation constraints.

   Typically, clamping occurs when an exponent falls outside the context's Emin and Emax limits. If possible, the exponent is reduced to fit by adding zeroes to the coefficient.

class DecimalException
   Base class for other signals and a subclass of ArithmeticError.

class DivisionByZero
   Signals the division of a non-infinite number by zero.

   Can occur with division, modulo division, or when raising a number to a negative power. If this signal is not trapped, returns Infinity or -Infinity with the sign determined by the inputs to the calculation.

class Inexact
   Indicates that rounding occurred and the result is not exact.

   Signals when non-zero digits were discarded during rounding. The rounded result is returned. The signal flag or trap is used to detect when results are inexact.
class InvalidOperation
An invalid operation was performed.

Indicates that an operation was requested that does not make sense. If not trapped, returns NaN. Possible causes include:

\[
\begin{align*}
\text{Infinity} - \text{Infinity} \\
0 * \text{Infinity} \\
\text{Infinity} / \text{Infinity} \\
x \% 0 \\
\text{Infinity} \% x \\
x._\text{rescale}(\text{non-integer}) \\
\text{sqrt}(-x) \text{ and } x > 0 \\
0 ** 0 \\
x ** (\text{non-integer}) \\
x ** \text{Infinity}
\end{align*}
\]

class Overflow
Numerical overflow.

Indicates the exponent is larger than Emax after rounding has occurred. If not trapped, the result depends on the rounding mode, either pulling inward to the largest representable finite number or rounding outward to Infinity. In either case, Inexact and Rounded are also signaled.

class Rounded
Rounding occurred though possibly no information was lost.

Signaled whenever rounding discards digits; even if those digits are zero (such as rounding 5.00 to 5.0). If not trapped, returns the result unchanged. This signal is used to detect loss of significant digits.

class Subnormal
Exponent was lower than Emin prior to rounding.

Occurs when an operation result is subnormal (the exponent is too small). If not trapped, returns the result unchanged.

class Underflow
Numerical underflow with result rounded to zero.

Occurs when a subnormal result is pushed to zero by rounding. Inexact
and **Subnormal** are also signaled.

The following table summarizes the hierarchy of signals:

<table>
<thead>
<tr>
<th>exceptions.ArithmeticError(exceptions.StandardError)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DecimalException</td>
</tr>
<tr>
<td>Clamped</td>
</tr>
<tr>
<td>DivisionByZero(DecimalException, exceptions.ZeroDivision)</td>
</tr>
<tr>
<td>Inexact</td>
</tr>
<tr>
<td>Overflow(Inexact, Rounded)</td>
</tr>
<tr>
<td>Underflow(Inexact, Rounded, Subnormal)</td>
</tr>
<tr>
<td>InvalidOperation</td>
</tr>
<tr>
<td>Rounded</td>
</tr>
<tr>
<td>Subnormal</td>
</tr>
</tbody>
</table>
6.3.5 Floating Point Notes

Subsections

- 6.3.5.1 Mitigating round-off error with increased precision
- 6.3.5.2 Special values

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
6.3.6 Working with threads

The `getcontext()` function accesses a different `Context` object for each thread. Having separate thread contexts means that threads may make changes (such as `getcontext.prec=10`) without interfering with other threads.

Likewise, the `setcontext()` function automatically assigns its target to the current thread.

If `setcontext()` has not been called before `getcontext()`, then `getcontext()` will automatically create a new context for use in the current thread.

The new context is copied from a prototype context called `DefaultContext`. To control the defaults so that each thread will use the same values throughout the application, directly modify the `DefaultContext` object. This should be done before any threads are started so that there won't be a race condition between threads calling `getcontext()`. For example:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
...
```
6.3.7 Recipes

Here are a few recipes that serve as utility functions and that demonstrate ways to work with the `Decimal` class:

```python
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
pos='', neg='-', trailneg=''):  
    """Convert Decimal to a money formatted string.

    places: required number of places after the decimal point
    curr: optional currency symbol before the sign (may be blank)
    sep: optional grouping separator (comma, period, space, or blank)
    dp: decimal point indicator (comma or period)
        only specify as blank when places is zero
    pos: optional sign for positive numbers: '+', space or blank
    neg: optional sign for negative numbers: '-', '(', space or blank
    trailneg: optional trailing minus indicator: '-', ')'
    
    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')  
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<.02>'
    """

    >>> q = Decimal((0, (1,), -places))  # 2 places --> '0.01'
    >>> sign, digits, exp = value.quantize(q).as_tuple()
    >>> assert exp == -places
    >>> result = []
    >>> digits = map(str, digits)
    >>> build, next = result.append, digits.pop
    >>> if sign:
    ...     build(trailneg)
    ... for i in range(places):
    ...     if digits:
    ...         build(next())
    ...     else:
    ...         build('0')
    >>> build(dp)
    >>> i = 0
    >>> while digits:
    ...     build(next())
    ...     i += 1
```

```
if i == 3 and digits:
    i = 0
    build(sep)
build(curr)
if sign:
    build(neg)
else:
    build(pos)
result.reverse()
return ''.join(result)

def pi():
    """Compute Pi to the current precision."""
    >>> print pi()
    3.141592653589793238462643383
    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2  # unary plus applies the new precision
    return +s

def exp(x):
    """Return e raised to the power of x. Result type matches input"

    >>> print exp(Decimal(1))
    2.718281828459045235360287471
    >>> print exp(Decimal(2))
    7.389056098930650227230427461
    >>> print exp(2.0)
    7.38905609893
    >>> print exp(2+0j)
    (7.38905609893+0j)
    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    >>> print cos(Decimal('0.5'))
    0.8775825618903727161162815826
    >>> print cos(0.5)
    0.87758256189
    >>> print cos(0.5+0j)
    (0.87758256189+0j)
    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    >>> print sin(Decimal('0.5'))
    0.4794255386042030002732879352
    >>> print sin(0.5)
    0.479425538604
    >>> print sin(0.5+0j)
    (0.479425538604+0j)
    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
FAQ

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
6.3.8 Decimal FAQ

Q. It is cumbersome to type `decimal.Decimal('1234.5')`. Is there a way to minimize typing when using the interactive interpreter?

A. Some users abbreviate the constructor to just a single letter:

```python
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal("4.68")
```

Q. In a fixed-point application with two decimal places, some inputs have many places and need to be rounded. Others are not supposed to have excess digits and need to be validated. What methods should be used?

A. The `quantize()` method rounds to a fixed number of decimal places. If the `Inexact` trap is set, it is also useful for validation:

```python
>>> TWOPLACES = Decimal(10) ** -2    # same as Decimal('0.01')

>>> # Round to two places
>>> Decimal("3.214").quantize(TWOPLACES)
Decimal("3.21")

>>> # Validate that a number does not exceed two places
>>> Decimal("3.21").quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal("3.21")

>>> Decimal("3.214").quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
...
Inexact: Changed in rounding
```

Q. Once I have valid two place inputs, how do I maintain that invariant throughout an application?

A. Some operations like addition and subtraction automatically preserve fixed point. Others, like multiplication and division, change the number of decimal places and need to be followed-up with a `quantize()` step.

Q. There are many ways to express the same value. The numbers 200, 200.000, 2E2, and .02E+4 all have the same value at various precisions. Is
there a way to transform them to a single recognizable canonical value?

A. The `normalize()` method maps all equivalent values to a single representative:

```python
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal("2E+2"), Decimal("2E+2"), Decimal("2E+2"), Decimal("2E+2")]
```

Q. Some decimal values always print with exponential notation. Is there a way to get a non-exponential representation?

A. For some values, exponential notation is the only way to express the number of significant places in the coefficient. For example, expressing `5.0E+3` as `5000` keeps the value constant but cannot show the original's two-place significance.

Q. Is there a way to convert a regular float to a `Decimal`?

A. Yes, all binary floating point numbers can be exactly expressed as a Decimal. An exact conversion may take more precision than intuition would suggest, so trapping `Inexact` will signal a need for more precision:

```python
def floatToDecimal(f):
    """Convert a floating point number to a Decimal with no loss of in
    # Transform (exactly) a float to a mantissa (0.5 <= abs(m) < 1.0
    # exponent. Double the mantissa until it is an integer. Use th
    # mantissa and exponent to compute an equivalent Decimal. If th
    # be done exactly, then retry with more precision.
    #"
    mantissa, exponent = math.frexp(f)
    while mantissa != int(mantissa):
        mantissa *= 2.0
        exponent -= 1
    mantissa = int(mantissa)

    oldcontext = getcontext()
    setcontext(Context(traps=[Inexact]))
    try:
        try:
            return mantissa * Decimal(2) ** exponent
        except Inexact:
            getcontext().prec += 1
    finally:
        setcontext(oldcontext)
```
Q. Why isn't the `floatToDecimal()` routine included in the module?

A. There is some question about whether it is advisable to mix binary and decimal floating point. Also, its use requires some care to avoid the representation issues associated with binary floating point:

```python
>>> floatToDecimal(1.1)
Decimal("1.100000000000000088817841970012523233890533447265625")
```

Q. Within a complex calculation, how can I make sure that I haven't gotten a spurious result because of insufficient precision or rounding anomalies.

A. The decimal module makes it easy to test results. A best practice is to re-run calculations using greater precision and with various rounding modes. Widely differing results indicate insufficient precision, rounding mode issues, ill-conditioned inputs, or a numerically unstable algorithm.

Q. I noticed that context precision is applied to the results of operations but not to the inputs. Is there anything to watch out for when mixing values of different precisions?

A. Yes. The principle is that all values are considered to be exact and so is the arithmetic on those values. Only the results are rounded. The advantage for inputs is that `what you type is what you get`. A disadvantage is that the results can look odd if you forget that the inputs haven't been rounded:

```python
>>> getcontext().prec = 3
>>> Decimal('3.104') + D('2.104')
Decimal("5.21")
>>> Decimal('3.104') + D('0.000') + D('2.104')
Decimal("5.20")
```

The solution is either to increase precision or to force rounding of inputs using the unary plus operation:

```python
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')  # unary plus triggers rounding
Decimal("1.23")
```

Alternatively, inputs can be rounded upon creation using the `Context.create_decimal()` method:

```python
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
```
Decimal("1.2345")
6.4 random -- Generate pseudo-random numbers

This module implements pseudo-random number generators for various distributions.

For integers, uniform selection from a range. For sequences, uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

Almost all module functions depend on the basic function random(), which generates a random float uniformly in the semi-open range [0.0, 1.0). Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of $2^{19937}$-1. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The functions supplied by this module are actually bound methods of a hidden instance of the random.Random class. You can instantiate your own instances of Random to get generators that don't share state. This is especially useful for multi-threaded programs, creating a different instance of Random for each thread, and using the jumpahead() method to make it likely that the generated sequences seen by each thread don't overlap.

Class Random can also be subclassed if you want to use a different basic generator of your own devising: in that case, override the random(), seed(), getstate(), setstate() and jumpahead() methods. Optionally, a new generator can supply a getrandombits() method -- this allows randrange() to produce selections over an arbitrarily large range. New in version 2.4: the getrandombits() method.
As an example of subclassing, the random module provides the WichmannHill class that implements an alternative generator in pure Python. The class provides a backward compatible way to reproduce results from earlier versions of Python, which used the Wichmann-Hill algorithm as the core generator. Note that this Wichmann-Hill generator can no longer be recommended: its period is too short by contemporary standards, and the sequence generated is known to fail some stringent randomness tests. See the references below for a recent variant that repairs these flaws. Changed in version 2.3: Substituted MersenneTwister for Wichmann-Hill.

Bookkeeping functions:

**seed([x])**

Initialize the basic random number generator. Optional argument x can be any hashable object. If x is omitted or None, current system time is used; current system time is also used to initialize the generator when the module is first imported. If randomness sources are provided by the operating system, they are used instead of the system time (see the os.urandom() function for details on availability). Changed in version 2.4: formerly, operating system resources were not used. If x is not None or an int or long, hash(x) is used instead. If x is an int or long, x is used directly.

**getstate()**

Return an object capturing the current internal state of the generator. This object can be passed to setstate() to restore the state. New in version 2.1.

**setstate(state)**

state should have been obtained from a previous call to getstate(), and setstate() restores the internal state of the generator to what it was at the time setstate() was called. New in version 2.1.

**jumpahead(n)**

Change the internal state to one different from and likely far away from the current state. n is a non-negative integer which is used to scramble the current state vector. This is most useful in multi-threaded programs, in conjunction with multiple instances of the Random class: setstate() or
seed() can be used to force all instances into the same internal state, and
then jumpahead() can be used to force the instances' states far apart.
New in version 2.1. Changed in version 2.3: Instead of jumping to a
specific state, \( n \) steps ahead, jumpahead\( (n) \) jumps to another state likely
to be separated by many steps.

getrandbits(\( k \))
Returns a python long int with \( k \) random bits. This method is supplied
with the MersenneTwister generator and some other generators may also
provide it as an optional part of the API. When available,
getrandbits() enables randrange() to handle arbitrarily large
ranges. New in version 2.4.

functions for integers:

randrange( [\( start, \) \( stop[, step] \)])
Return a randomly selected element from range\( (\text{\texttt{start, stop, step}}) \).
This is equivalent to choice(range\( (\text{\texttt{start, stop, step}}) \)), but doesn't
actually build a range object. New in version 1.5.2.

randint( \( a, b \))
Return a random integer \( N \) such that \( a \leq N \leq b \).

functions for sequences:

choice(seq)
Return a random element from the non-empty sequence \( \text{seq} \). If \( \text{seq} \) is empty,
raises IndexError.

shuffle(x[, random])
Shuffle the sequence \( x \) in place. The optional argument \( \text{random} \) is a 0-
argument function returning a random float in \([0.0, 1.0)\); by default, this is
the function random().

Note that for even rather small \( \text{len}(x) \), the total number of permutations
of \( x \) is larger than the period of most random number generators; this
implies that most permutations of a long sequence can never be generated.
**sample(population, k)**

Return a \( k \) length list of unique elements chosen from the population sequence. Used for random sampling without replacement. New in version 2.3.

Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).

Members of the population need not be hashable or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.

To choose a sample from a range of integers, use an `xrange()` object as an argument. This is especially fast and space efficient for sampling from a large population: `sample(xrange(10000000), 60)`.

The following functions generate specific real-valued distributions. Function parameters are named after the corresponding variables in the distribution's equation, as used in common mathematical practice; most of these equations can be found in any statistics text.

**random()**

Return the next random floating point number in the range \([0.0, 1.0)\).

**uniform(a, b)**

Return a random real number \( N \) such that \( a \leq N < b \).

**betavariate(alpha, beta)**

Beta distribution. Conditions on the parameters are \( alpha > -1 \) and \( beta > -1 \). Returned values range between 0 and 1.

**expovariate(lambd)**

Exponential distribution. \( lambd \) is 1.0 divided by the desired mean. (The parameter would be called \`\`lambda\`, but that is a reserved word in Python.) Returned values range from 0 to positive infinity.
**gammavariate** *(alpha, beta)*

Gamma distribution. *(Not the gamma function!)* Conditions on the parameters are *alpha > 0* and *beta > 0*.

**gauss** *(mu, sigma)*

Gaussian distribution. *mu* is the mean, and *sigma* is the standard deviation. This is slightly faster than the *normalvariate()* function defined below.

**lognormvariate** *(mu, sigma)*

Log normal distribution. If you take the natural logarithm of this distribution, you'll get a normal distribution with mean *mu* and standard deviation *sigma*. *mu* can have any value, and *sigma* must be greater than zero.

**normalvariate** *(mu, sigma)*

Normal distribution. *mu* is the mean, and *sigma* is the standard deviation.

**vonmisesvariate** *(mu, kappa)*

*mu* is the mean angle, expressed in radians between 0 and 2*π*, and *kappa* is the concentration parameter, which must be greater than or equal to zero. If *kappa* is equal to zero, this distribution reduces to a uniform random angle over the range 0 to 2*π*.

**paretovariate** *(alpha)*

Pareto distribution. *alpha* is the shape parameter.

**weibullvariate** *(alpha, beta)*

Weibull distribution. *alpha* is the scale parameter and *beta* is the shape parameter.

Alternative Generators:

**class WichmannHill** *[seed]*

Class that implements the Wichmann-Hill algorithm as the core generator. Has all of the same methods as Random plus the *whseed()* method described below. Because this class is implemented in pure Python, it is not
threadsafe and may require locks between calls. The period of the generator is 6,953,607,871,644 which is small enough to require care that two independent random sequences do not overlap.

**whseed([x])**

This is obsolete, supplied for bit-level compatibility with versions of Python prior to 2.1. See seed() for details. whseed() does not guarantee that distinct integer arguments yield distinct internal states, and can yield no more than about 2**24 distinct internal states in all.

**class SystemRandom([seed])**

Class that uses the os.urandom() function for generating random numbers from sources provided by the operating system. Not available on all systems. Does not rely on software state and sequences are not reproducible. Accordingly, the seed() and jumpahead() methods have no effect and are ignored. The getstate() and setstate() methods raise `NotImplementedError` if called. New in version 2.4.

Examples of basic usage:

```python
>>> random.random()  # Random float x, 0.0 <= x < 1.0
0.37444887175646646
>>> random.uniform(1, 10)  # Random float x, 1.0 <= x < 10.0
1.1800146073117523
>>> random.randint(1, 10)  # Integer from 1 to 10, endpoints included
7
>>> random.randrange(0, 101, 2)  # Even integer from 0 to 100
26
>>> random.choice('abcdefghij')  # Choose a random element 'c'
'
>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> random.shuffle(items)
>>> items
[7, 3, 2, 5, 6, 4, 1]

>>> random.sample([1, 2, 3, 4, 5], 3)  # Choose 3 elements
[4, 1, 5]
```

**See Also:**

M. Matsumoto and T. Nishimura, ``Mersenne Twister: A 623-
dimensionally equidistributed uniform pseudorandom number generator",
*ACM Transactions on Modeling and Computer Simulation* Vol. 8, No. 1,

Wichmann, B. A. & Hill, I. D., ``Algorithm AS 183: An efficient and
188-190.

[http://www.npl.co.uk/ssfm/download/abstracts.html#196](http://www.npl.co.uk/ssfm/download/abstracts.html#196)

A modern variation of the Wichmann-Hill generator that greatly
increases the period, and passes now-standard statistical tests that the
original generator failed.
6.5 itertools -- Functions creating iterators for efficient looping

New in version 2.3.

This module implements a number of iterator building blocks inspired by constructs from the Haskell and SML programming languages. Each has been recast in a form suitable for Python.

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Standardization helps avoid the readability and reliability problems which arise when many different individuals create their own slightly varying implementations, each with their own quirks and naming conventions.

The tools are designed to combine readily with one another. This makes it easy to construct more specialized tools succinctly and efficiently in pure Python.

For instance, SML provides a tabulation tool: \texttt{tabulate(f)} which produces a sequence \( f(0), f(1), \ldots \). This toolbox provides \texttt{imap()} and \texttt{count()} which can be combined to form \texttt{imap(f, count())} and produce an equivalent result.

Likewise, the functional tools are designed to work well with the high-speed functions provided by the \texttt{operator} module.

The module author welcomes suggestions for other basic building blocks to be added to future versions of the module.

Whether cast in pure python form or compiled code, tools that use iterators are more memory efficient (and faster) than their list based counterparts. Adopting the principles of just-in-time manufacturing, they create data when and where needed instead of consuming memory with the computer equivalent of "inventory".

The performance advantage of iterators becomes more acute as the number of

elements increases - at some point, lists grow large enough to severely impact memory cache performance and start running slowly.

See Also:


Subsections

- 6.5.1 Itertool functions
- 6.5.2 Examples
- 6.5.3 Recipes

Release 2.5, documentation updated on 19th September, 2006.
See *About this document...* for information on suggesting changes.
6.5.1 Itertool functions

The following module functions all construct and return iterators. Some provide streams of infinite length, so they should only be accessed by functions or loops that truncate the stream.

chain(*iterables)
Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a single sequence. Equivalent to:

```python
def chain(*iterables):
    for it in iterables:
        for element in it:
            yield element
```

count([n])
Make an iterator that returns consecutive integers starting with n. If not specified n defaults to zero. Does not currently support python long integers. Often used as an argument to `imap()` to generate consecutive data points. Also, used with `izip()` to add sequence numbers. Equivalent to:

```python
def count(n=0):
    while True:
        yield n
        n += 1
```

Note, `count()` does not check for overflow and will return negative numbers after exceeding `sys.maxint`. This behavior may change in the future.

cycle(iterable)
Make an iterator returning elements from the iterable and saving a copy of each. When the iterable is exhausted, return elements from the saved copy. Repeats indefinitely. Equivalent to:

```python
def cycle(iterable):
```
```
saved = []
for element in iterable:
    yield element
    saved.append(element)
while saved:
    for element in saved:
        yield element
```

Note, this member of the toolkit may require significant auxiliary storage (depending on the length of the iterable).

**dropwhile**( *predicate, iterable*)

Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element. Note, the iterator does not produce *any* output until the predicate is true, so it may have a lengthy start-up time. Equivalent to:

```
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
        break
    for x in iterable:
        yield x
```

**groupby**( *iterable[, key]*)

Make an iterator that returns consecutive keys and groups from the iterable. The *key* is a function computing a key value for each element. If not specified or is *None*, *key* defaults to an identity function and returns the element unchanged. Generally, the iterable needs to already be sorted on the same key function.

The returned group is itself an iterator that shares the underlying iterable with groupby(). Because the source is shared, when the groupby object is advanced, the previous group is no longer visible. So, if that data is needed later, it should be stored as a list:

```
groups = []
uniquekeys = []
for k, g in groupby(data, keyfunc):
    groups.append(list(g))  # Store group iterator as a
    uniquekeys.append(k)
```
groupby() is equivalent to:

```python
class groupby(object):
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue = xrange(0)
    def __iter__(self):
        return self
    def next(self):
        while self.currkey == self.tgtkey:
            self.currvalue = self.it.next()  # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
            self.tgtkey = self.currkey
        return (self.currkey, self._grouper(self.tgtkey))
    def _grouper(self, tgtkey):
        while self.currkey == tgtkey:
            yield self.currvalue
            self.currvalue = self.it.next()  # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)

New in version 2.4.
```

**ifilter**(predicate, iterable)

Make an iterator that filters elements from iterable returning only those for which the predicate is True. If predicate is None, return the items that are true. Equivalent to:

```python
def ifilter(predicate, iterable):
    if predicate is None:
        predicate = bool
    for x in iterable:
        if predicate(x):
            yield x
```

**ifilterfalse**(predicate, iterable)

Make an iterator that filters elements from iterable returning only those for which the predicate is False. If predicate is None, return the items that are false. Equivalent to:

```python
def ifilterfalse(predicate, iterable):
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate:
```
def imap(function, *iterables):
    iterables = map(iter, iterables)
    while True:
        args = [i.next() for i in iterables]
        if function is None:
            yield tuple(args)
        else:
            yield function(*args)

def islice(iterable, *args):
    s = slice(*args)
    it = iter(xrange(s.start or 0, s.stop or sys.maxint, s.step or 1))
    nexti = it.next()
    for i, element in enumerate(iterable):
        if i == nexti:
            yield element
        nexti = it.next()
**start** and **step**.

**izip( *iterables)**

Make an iterator that aggregates elements from each of the iterables. Like `zip()` except that it returns an iterator instead of a list. Used for lock-step iteration over several iterables at a time. Equivalent to:

```python
def izip(*iterables):
    iterables = map(iter, iterables)
    while iterables:
        result = [it.next() for it in iterables]
        yield tuple(result)
```

Changed in version 2.4: When no iterables are specified, returns a zero length iterator instead of raising a `TypeError` exception.

Note, the left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using "izip(*[iter(s)]*n)". For data that doesn't fit n-length groups exactly, the last tuple can be pre-padded with fill values using "izip(*[chain(s, [None]*(n-1))]*n)".

Note, when `izip()` is used with unequal length inputs, subsequent iteration over the longer iterables cannot reliably be continued after `izip()` terminates. Potentially, up to one entry will be missing from each of the left-over iterables. This occurs because a value is fetched from each iterator in-turn, but the process ends when one of the iterators terminates. This leaves the last fetched values in limbo (they cannot be returned in a final, incomplete tuple and they are cannot be pushed back into the iterator for retrieval with `it.next()`). In general, `izip()` should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables.

**repeat( object[, times])**

Make an iterator that returns **object** over and over again. Runs indefinitely unless the **times** argument is specified. Used as argument to `imap()` for invariant parameters to the called function. Also used with `izip()` to create an invariant part of a tuple record. Equivalent to:

```python
def repeat(object, times=None):
```
if times is None:
  while True:
    yield object
else:
  for i in xrange(times):
    yield object

**starmap**( function, iterable )

Make an iterator that computes the function using arguments tuples obtained from the iterable. Used instead of `imap()` when argument parameters are already grouped in tuples from a single iterable (the data has been ``pre-zipped``). The difference between `imap()` and `starmap()` parallels the distinction between `function(a, b)` and `function(*c)`. Equivalent to:

```python
def starmap(function, iterable):
  iterable = iter(iterable)
  while True:
    yield function(*iterable.next())
```

**takewhile**( predicate, iterable )

Make an iterator that returns elements from the iterable as long as the predicate is true. Equivalent to:

```python
def takewhile(predicate, iterable):
  for x in iterable:
    if predicate(x):
      yield x
    else:
      break
```

**tee**( iterable[, n=2])

Return n independent iterators from a single iterable. The case where n==2 is equivalent to:

```python
def tee(iterable):
  def gen(next, data={}, cnt=[0]):
    for i in count():
      if i == cnt[0]:
        item = data[i] = next()
        cnt[0] += 1
      else:
        item = data.pop(i)
        yield item
  it = iter(iterable)
  ```
return (gen(it.next), gen(it.next))

Note, once tee() has made a split, the original iterable should not be used anywhere else; otherwise, the iterable could get advanced without the tee objects being informed.

Note, this member of the toolkit may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator is going to use most or all of the data before the other iterator, it is faster to use list() instead of tee(). New in version 2.4.
6.5.2 Examples

The following examples show common uses for each tool and demonstrate ways they can be combined.

>>> amounts = [120.15, 764.05, 823.14]
>>> for checknum, amount in izip(count(1200), amounts):
...     print 'Check %d is for $%.2f' % (checknum, amount)
...
Check 1200 is for $120.15
Check 1201 is for $764.05
Check 1202 is for $823.14

>>> import operator
>>> for cube in imap(operator.pow, xrange(1,5), repeat(3)):
...     print cube
...
1
8
27
64

>>> reportlines = ['EuroPython', 'Roster', '', 'alex', '', 'laura', '', 'martin', '', 'walter', '', 'mark']
>>> for name in islice(reportlines, 3, None, 2):
...     print name.title()
...
Alex
Laura
Martin
Walter
Mark

# Show a dictionary sorted and grouped by value
>>> from operator import itemgetter
>>> d = dict(a=1, b=2, c=1, d=2, e=1, f=2, g=3)
>>> di = sorted(d.iteritems(), key=itemgetter(1))
>>> for k, g in groupby(di, key=itemgetter(1)):
...     print k, map(itemgetter(0), g)
...
1 ['a', 'c', 'e']
2 ['b', 'd', 'f']
3 ['g']

# Find runs of consecutive numbers using groupby. The key to the so
# is differencing with a range so that consecutive numbers all appea
# same group.
>>> data = [ 1, 4, 5, 6, 10, 15,16,17,18, 22, 25,26,27,28]
>>> for k, g in groupby(enumerate(data), lambda (i,x):i-x):
... print map(operator.itemgetter(1), g)
...
[1]
[4, 5, 6]
[10]
[15, 16, 17, 18]
[22]
[25, 26, 27, 28]

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
6.5.3 Recipes

This section shows recipes for creating an extended toolset using the existing itertools as building blocks.

The extended tools offer the same high performance as the underlying toolset. The superior memory performance is kept by processing elements one at a time rather than bringing the whole iterable into memory all at once. Code volume is kept small by linking the tools together in a functional style which helps eliminate temporary variables. High speed is retained by preferring ``vectorized building blocks over the use of for-loops and generators which incur interpreter overhead.

```python
def take(n, seq):
    return list(islice(seq, n))

def enumerate(iterable):
    return izip(count(), iterable)

def tabulate(function):
    """Return function(0), function(1), ...""
    return imap(function, count())

def iteritems(mapping):
    return izip(mapping.iterkeys(), mapping.itervalues())

def nth(iterable, n):
    """Returns the nth item""
    return list(islice(iterable, n, n+1))

def all(seq, pred=None):
    """Returns True if pred(x) is true for every element in the iterab""
    for elem in ifilterfalse(pred, seq):
        return False
    return True

def any(seq, pred=None):
    """Returns True if pred(x) is true for at least one element in the""
    for elem in ifilter(pred, seq):
        return True
    return False

def no(seq, pred=None):
    """Returns True if pred(x) is false for every element in the itera""
    for elem in ifilter(pred, seq):
        return False
```
return True

def quantify(seq, pred=None):
    "Count how many times the predicate is true in the sequence"
    return sum(imap(pred, seq))

def padnone(seq):
    """Returns the sequence elements and then returns None indefinitely.
    Useful for emulating the behavior of the built-in map() function
    """
    return chain(seq, repeat(None))

def ncycles(seq, n):
    "Returns the sequence elements n times"
    return chain(*repeat(seq, n))

def dotproduct(vec1, vec2):
    return sum(imap(operator.mul, vec1, vec2))

def flatten(listOfLists):
    return list(chain(*listOfLists))

def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.
    Example: repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
    else:
        return starmap(func, repeat(args, times))

def pairwise(iterable):
    """s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    try:
        b.next()
    except StopIteration:
        pass
    return izip(a, b)

def grouper(n, iterable, padvalue=None):
    """grouper(3, 'abcedfg', 'x') --> ('a','b','c'), ('d','e','f'), ('
    return izip(*[chain(iterable, repeat(padvalue, n-1))]\*n)"""
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
6.6 functools -- Higher order functions and operations on callable objects.

New in version 2.5.

The functools module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module.

The functools module defines the following function:

**partial** (*func*, *args*, **keywords*)

Return a new partial object which when called will behave like *func* called with the positional arguments *args* and keyword arguments *keywords*. If more arguments are supplied to the call, they are appended to *args*. If additional keyword arguments are supplied, they extend and override *keywords*. Roughly equivalent to:

```python
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*(args + fargs), **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

The partial is used for partial function application which "freezes" some portion of a function's arguments and/or keywords resulting in a new object with a simplified signature. For example, partial can be used to create a callable that behaves like the int function where the base argument defaults to two:

```python
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
```
**update_wrapper***(wrapper, wrapped[, assigned][, updated])

Update a wrapper function to look like the wrapped function. The optional arguments are tuples to specify which attributes of the original function are assigned directly to the matching attributes on the wrapper function and which attributes of the wrapper function are updated with the corresponding attributes from the original function. The default values for these arguments are the module level constants **WRAPPER_ASSIGNMENTS** (which assigns to the wrapper function's name, module and documentation string) and **WRAPPER_UPDATES** (which updates the wrapper function's instance dictionary).

The main intended use for this function is in decorator functions which wrap the decorated function and return the wrapper. If the wrapper function is not updated, the metadata of the returned function will reflect the wrapper definition rather than the original function definition, which is typically less than helpful.

**wraps***(wrapped[, assigned][, updated])

This is a convenience function for invoking **partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)** as a function decorator when defining a wrapper function. For example:

```python
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print 'Calling decorated function'
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     print 'Called example function'
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
```

Without the use of this decorator factory, the name of the example function
would have been 'wrapper'.

Subsections

- 6.6.1 partial Objects
6.6.1 partial Objects

partial objects are callable objects created by `partial()`. They have three read-only attributes:

**func**
A callable object or function. Calls to the `partial` object will be forwarded to `func` with new arguments and keywords.

**args**
The leftmost positional arguments that will be prepended to the positional arguments provided to a `partial` object call.

**keywords**
The keyword arguments that will be supplied when the `partial` object is called.

partial objects are like `function` objects in that they are callable, weak referencable, and can have attributes. There are some important differences. For instance, the `__name__` and `__doc__` attributes are not created automatically. Also, `partial` objects defined in classes behave like static methods and do not transform into bound methods during instance attribute look-up.
6.7 operator -- Standard operators as functions.

The `operator` module exports a set of functions implemented in C corresponding to the intrinsic operators of Python. For example, `operator.add(x, y)` is equivalent to the expression `x+y`. The function names are those used for special class methods; variants without leading and trailing "__" are also provided for convenience.

The functions fall into categories that perform object comparisons, logical operations, mathematical operations, sequence operations, and abstract type tests.

The object comparison functions are useful for all objects, and are named after the rich comparison operators they support:

```
lt(a, b)
le(a, b)
eq(a, b)
ge(a, b)
gt(a, b)
__lt__(a, b)
__le__(a, b)
__eq__(a, b)
__ne__(a, b)
__ge__(a, b)
__gt__(a, b)
```

Perform "rich comparisons" between `a` and `b`. Specifically, `lt(a, b)` is equivalent to `a < b`, `le(a, b)` is equivalent to `a <= b`, `eq(a, b)` is equivalent to `a == b`, `ne(a, b)` is equivalent to `a != b`, `gt(a, b)` is equivalent to `a > b` and `ge(a, b)` is equivalent to `a >= b`. Note that
unlike the built-in `cmp()`, these functions can return any value, which may or may not be interpretable as a Boolean value. See the Python Reference Manual for more information about rich comparisons. New in version 2.2.

The logical operations are also generally applicable to all objects, and support truth tests, identity tests, and boolean operations:

```python
not_(o)
__not__(o)
    Return the outcome of `not o`. (Note that there is no `__not__(o)` method for object instances; only the interpreter core defines this operation. The result is affected by the `__nonzero__(o)` and `__len__(o)` methods.)

truth(o)
    Return `True` if `o` is true, and `False` otherwise. This is equivalent to using the `bool` constructor.

is_(a, b)
    Return `a is b`. Tests object identity. New in version 2.3.

is_not(a, b)
    Return `a is not b`. Tests object identity. New in version 2.3.
```

The mathematical and bitwise operations are the most numerous:

```python
abs(o)
__abs__(o)
    Return the absolute value of `o`.

add(a, b)
__add__(a, b)
    Return `a + b`, for `a` and `b` numbers.

and_(a, b)
__and__(a, b)
    Return the bitwise and of `a` and `b`.
```
**div**(\(a, b\))

\[\text{\_\_div\_\_}(a, b)\]

Return \(a / b\) when `__future__.division` is not in effect. This is also known as ``classic'' division.

**floordiv**(\(a, b\))

\[\text{\_\_floordiv\_\_}(a, b)\]

Return \(a // b\). New in version 2.2.

**inv**(\(o\))

\[\text{\_\_inv\_\_}(o)\]

\[\text{\_\_invert\_\_}(o)\]

Return the bitwise inverse of the number \(o\). This is equivalent to \(\sim o\). The names `invert()` and `\_\_invert\_\_()` were added in Python 2.0.

**lshift**(\(a, b\))

\[\text{\_\_lshift\_\_}(a, b)\]

Return \(a\) shifted left by \(b\).

**mod**(\(a, b\))

\[\text{\_\_mod\_\_}(a, b)\]

Return \(a \% b\).

**mul**(\(a, b\))

\[\text{\_\_mul\_\_}(a, b)\]

Return \(a * b\), for \(a\) and \(b\) numbers.

**neg**(\(o\))

\[\text{\_\_neg\_\_}(o)\]

Return \(o\) negated.

**or**(\(a, b\))

\[\text{\_\_or\_\_}(a, b)\]
Return the bitwise or of \( a \) and \( b \).

\[
\text{pos}(o)
\]

__pos__(o)

Return \( o \) positive.

\[
pow(a, b)
\]

__pow__(a, b)

Return \( a \ ^{**} \ b \), for \( a \) and \( b \) numbers. New in version 2.3.

\[
rshift(a, b)
\]

__rshift__(a, b)

Return \( a \) shifted right by \( b \).

\[
sub(a, b)
\]

__sub__(a, b)

Return \( a - b \).

\[
truediv(a, b)
\]

__truediv__(a, b)

Return \( a \ / \ b \) when \_
\_future\_.division is in effect. This is also known as ```true``` division. New in version 2.2.

\[
xor(a, b)
\]

__xor__(a, b)

Return the bitwise exclusive or of \( a \) and \( b \).

\[
index(a)
\]

__index__(a)

Return \( a \) converted to an integer. Equivalent to \( a . \_\_index\_() \). New in version 2.5.

Operations which work with sequences include:

\[
concat(a, b)
\]
__concat__(a, b)
Return a + b for a and b sequences.

contains(a, b)
__contains__(a, b)
Return the outcome of the test b in a. Note the reversed operands. The name __contains__() was added in Python 2.0.

countOf(a, b)
Return the number of occurrences of b in a.

delitem(a, b)
__delitem__(a, b)
Remove the value of a at index b.

delslice(a, b, c)
__delslice__(a, b, c)
Delete the slice of a from index b to index c-1.

getitem(a, b)
__getitem__(a, b)
Return the value of a at index b.

getslice(a, b, c)
__getslice__(a, b, c)
Return the slice of a from index b to index c-1.

indexOf(a, b)
Return the index of the first of occurrence of b in a.

repeat(a, b)
__repeat__(a, b)
Return a * b where a is a sequence and b is an integer.

sequenceIncludes(...)
Deprecated since release 2.0. Use contains() instead.

Alias for contains().

```
setitem(a, b, c)
__setitem__(a, b, c)
  Set the value of a at index b to c.
```

```
setslice(a, b, c, v)
__setslice__(a, b, c, v)
  Set the slice of a from index b to index c-1 to the sequence v.
```

Many operations have an "in-place" version. The following functions provide a more primitive access to in-place operators than the usual syntax does; for example, the statement `x += y` is equivalent to `x = operator.iadd(x, y)`. Another way to put it is to say that `z = operator.iadd(x, y)` is equivalent to the compound statement `z = x; z += y`.

```
iadd(a, b)
__iadd__(a, b)
  a = iadd(a, b) is equivalent to a += b. New in version 2.5.
```

```
iand(a, b)
__iand__(a, b)
  a = iand(a, b) is equivalent to a &= b. New in version 2.5.
```

```
iconcat(a, b)
__iconcat__(a, b)
  a = iconcat(a, b) is equivalent to a += b for a and b sequences. New in version 2.5.
```

```
idiv(a, b)
__idiv__(a, b)
  a = idiv(a, b) is equivalent to a /= b when __future__.division is not in effect. New in version 2.5.
```
ifloordiv\( (a, b) \)

\__ifloordiv__\( (a, b) \)

\( a = \text{ifloordiv}(a, b) \) is equivalent to \( a \ // = b \). New in version 2.5.

ilshift\( (a, b) \)

\__ilshift__\( (a, b) \)

\( a = \text{ilshift}(a, b) \) is equivalent to \( a <<= b \). New in version 2.5.

imod\( (a, b) \)

\__imod__\( (a, b) \)

\( a = \text{imod}(a, b) \) is equivalent to \( a \ %= b \). New in version 2.5.

imul\( (a, b) \)

\__imul__\( (a, b) \)

\( a = \text{imul}(a, b) \) is equivalent to \( a \ *= b \). New in version 2.5.

ior\( (a, b) \)

\__ior__\( (a, b) \)

\( a = \text{ior}(a, b) \) is equivalent to \( a \ |= b \). New in version 2.5.

ipow\( (a, b) \)

\__ipow__\( (a, b) \)

\( a = \text{ipow}(a, b) \) is equivalent to \( a \ **= b \). New in version 2.5.

irepeat\( (a, b) \)

\__irepeat__\( (a, b) \)

\( a = \text{irepeat}(a, b) \) is equivalent to \( a \ *= b \) where \( a \) is a sequence and \( b \) is an integer. New in version 2.5.

irshift\( (a, b) \)

\__irshift__\( (a, b) \)

\( a = \text{irshift}(a, b) \) is equivalent to \( a \ >>= b \). New in version 2.5.
**isub**\( (a, b) \)

\`
__isub__(a, b)
\`

\( a = \text{isub}(a, b) \) is equivalent to \( a -= b \). New in version 2.5.

**itruediv**\( (a, b) \)

\`
__itruediv__(a, b)
\`

\( a = \text{itruediv}(a, b) \) is equivalent to \( a /= b \) when **__future__.division** is in effect. New in version 2.5.

**ixor**\( (a, b) \)

\`
__ixor__(a, b)
\`

\( a = \text{ixor}(a, b) \) is equivalent to \( a ^= b \). New in version 2.5.

The operator module also defines a few predicates to test the type of objects.

**Note:** Be careful not to misinterpret the results of these functions; only **isCallable()** has any measure of reliability with instance objects. For example:

```python
>>> class C:
...     pass
... >>> import operator
>>> o = C()
>>> operator.isMappingType(o)
True
```

**isCallable**\( (o) \)

**Deprecated since release 2.0.** Use the **callable()** built-in function instead.

Returns true if the object \( o \) can be called like a function, otherwise it returns false. True is returned for functions, bound and unbound methods, class objects, and instance objects which support the **__call__()** method.

**isMappingType**\( (o) \)

Returns true if the object \( o \) supports the mapping interface. This is true for dictionaries and all instance objects defining **__getitem__**. **Warning:** There is no reliable way to test if an instance supports the complete mapping protocol since the interface itself is ill-defined. This makes this
test less useful than it otherwise might be.

**isNumberType(o)**
Returns true if the object o represents a number. This is true for all numeric types implemented in C. **Warning:** There is no reliable way to test if an instance supports the complete numeric interface since the interface itself is ill-defined. This makes this test less useful than it otherwise might be.

**isSequenceType(o)**
Returns true if the object o supports the sequence protocol. This returns true for all objects which define sequence methods in C, and for all instance objects defining `__getitem__`. **Warning:** There is no reliable way to test if an instance supports the complete sequence interface since the interface itself is ill-defined. This makes this test less useful than it otherwise might be.

Example: Build a dictionary that maps the ordinals from 0 to 255 to their character equivalents.

```python
>>> import operator
>>> d = {}
>>> keys = range(256)
>>> vals = map(chr, keys)
>>> map(operator.setitem, [d]*len(keys), keys, vals)
```

The `operator` module also defines tools for generalized attribute and item lookups. These are useful for making fast field extractors as arguments for `map()`, `sorted()`, `itertools.groupby()`, or other functions that expect a function argument.

**attrgetter(attr[, args...])**
Return a callable object that fetches `attr` from its operand. If more than one attribute is requested, returns a tuple of attributes. After, "f=attrgetter('name')", the call "f(b)" returns "b.name". After, "f=attrgetter('name', 'date')", the call "f(b)" returns "(b.name, b.date)". New in version 2.4. Changed in version 2.5: Added support for multiple attributes.

**itemgetter(item[, args...])**
Return a callable object that fetches *item* from its operand. If more than one item is requested, returns a tuple of items. After, "f=itemgetter(2)" , the call "f(b)" returns "b[2]". After, "f=itemgetter(2,5,3)" , the call "f(b)" returns "(b[2], b[5], b[3])". New in version 2.4. Changed in version 2.5: Added support for multiple item extraction.

Examples:

```python
>>> from operator import itemgetter
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> map(getcount, inventory)
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```
6.7.1 Mapping Operators to Functions

This table shows how abstract operations correspond to operator symbols in the Python syntax and the functions in the `operator` module.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>$a + b$</td>
<td><code>add(a, b)</code></td>
</tr>
<tr>
<td>Concatenation</td>
<td><code>seq1 + seq2</code></td>
<td><code>concat(seq1, seq2)</code></td>
</tr>
<tr>
<td>Containment Test</td>
<td>$o \text{ in } seq$</td>
<td><code>contains(seq, o)</code></td>
</tr>
<tr>
<td>Division</td>
<td>$a / b$</td>
<td><code>div(a, b) # without __future__.division</code></td>
</tr>
<tr>
<td>Division</td>
<td><code>a / b</code></td>
<td><code>truediv(a, b) # with __future__.division</code></td>
</tr>
<tr>
<td>Division</td>
<td><code>a // b</code></td>
<td><code>floordiv(a, b)</code></td>
</tr>
<tr>
<td>Bitwise And</td>
<td>$a &amp; b$</td>
<td><code>and_(a, b)</code></td>
</tr>
<tr>
<td>Bitwise Or</td>
<td>$a</td>
<td>b$</td>
</tr>
<tr>
<td>Exponentiation</td>
<td>$a ** b$</td>
<td><code>pow(a, b)</code></td>
</tr>
<tr>
<td>Identity</td>
<td>$a \text{ is } b$</td>
<td><code>is_(a, b)</code></td>
</tr>
<tr>
<td>Identity</td>
<td>$a \text{ is not } b$</td>
<td><code>is_not(a, b)</code></td>
</tr>
<tr>
<td>Indexed Assignment</td>
<td>$o[k] = v$</td>
<td><code>setitem(o, k, v)</code></td>
</tr>
<tr>
<td>Indexed Deletion</td>
<td><code>del o[k]</code></td>
<td><code>delitem(o, k)</code></td>
</tr>
<tr>
<td>Indexing</td>
<td>$o[k]$</td>
<td><code>getitem(o, k)</code></td>
</tr>
<tr>
<td>Left Shift</td>
<td>$a &lt;&lt; b$</td>
<td><code>lshift(a, b)</code></td>
</tr>
<tr>
<td>Modulo</td>
<td>$a % b$</td>
<td><code>mod(a, b)</code></td>
</tr>
<tr>
<td>Multiplication</td>
<td>$a * b$</td>
<td><code>mul(a, b)</code></td>
</tr>
<tr>
<td>Negation</td>
<td>$- a$</td>
<td><code>neg(a)</code></td>
</tr>
<tr>
<td>Arithmetic (Arithmetic)</td>
<td>Logical Negation</td>
<td>Logical Not</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Right Shift</td>
<td>a &gt;&gt; b</td>
<td>rshift(a, b)</td>
</tr>
<tr>
<td>Sequence Repitition</td>
<td>seq * i</td>
<td>repeat(seq, i)</td>
</tr>
<tr>
<td>Slice Assignment</td>
<td>seq[i:j] =</td>
<td>setslice(seq, i, j, values)</td>
</tr>
<tr>
<td>Slice Deletion</td>
<td>del</td>
<td>delslice(seq, i, j)</td>
</tr>
<tr>
<td>Slicing</td>
<td>seq[i:j]</td>
<td>getslice(seq, i, j)</td>
</tr>
<tr>
<td>String Formatting</td>
<td>s % o</td>
<td>mod(s, o)</td>
</tr>
<tr>
<td>Subtraction</td>
<td>a - b</td>
<td>sub(a, b)</td>
</tr>
<tr>
<td>Truth Test</td>
<td>o</td>
<td>truth(o)</td>
</tr>
<tr>
<td>Ordering</td>
<td>a &lt; b</td>
<td>lt(a, b)</td>
</tr>
<tr>
<td>Ordering</td>
<td>a &lt;= b</td>
<td>le(a, b)</td>
</tr>
<tr>
<td>Equality</td>
<td>a == b</td>
<td>eq(a, b)</td>
</tr>
<tr>
<td>Difference</td>
<td>a != b</td>
<td>ne(a, b)</td>
</tr>
<tr>
<td>Ordering</td>
<td>a &gt;= b</td>
<td>ge(a, b)</td>
</tr>
<tr>
<td>Ordering</td>
<td>a &gt; b</td>
<td>gt(a, b)</td>
</tr>
</tbody>
</table>
7. Internet Data Handling

This chapter describes modules which support handling data formats commonly used on the Internet.

- **email**: Package supporting the parsing, manipulating, and generating email messages, including MIME documents.
- **email.message**: The base class representing email messages. Parse flat text email messages to produce a message object structure.
- **email.parser**: Generate flat text email messages from a message structure.
- **email.mime**: Representing non-ASCII headers
- **email.mime.base**: Character Sets
- **email.mime.nonmultipart**: Encoders for email message payloads.
- **email.mime.multipart**: The exception classes used by the email package.
- **email.mime.audio**: Miscellaneous email package utilities.
- **email.mime.image**: Iterate over a message object tree.
- **email.mime.message**: Mailcap file handling.
- **email.mime.text**: Manipulate mailboxes in various formats.
- **email.header**: Manipulate MH mailboxes from Python.
- **email.charset**: Tools for parsing MIME-style message bodies.
- **email.encoders**: Mimetypes
- **email.errors**: Tools for parsing MIME-style message bodies.
- **email.utils**: mailcap
- **email.iterators**: mailbox
- **mhlib**: mimetools
mimetypes
Mapping of filename extensions to MIME types.

MimeWriter
Generic MIME file writer.

mimify
Mimification and unmimification of mail messages.

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

rfc822
Parse RFC 2822 style mail messages.

base64
RFC 3548: Base16, Base32, Base64 Data Encodings

binhex
Encode and decode files in binhex4 format.

binascii
Tools for converting between binary and various ASCII-encoded binary representations.

quopri
Encode and decode files using the MIME quoted-printable encoding.

uu
Encode and decode files in uuencode format.

---

Python Library Reference

Previous: 6.7.1 Mapping Operators to  Up: Python Library Reference  Next: 7.1 email

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
Representing an email
7.1 email -- An email and MIME handling package

New in version 2.2.

The email package is a library for managing email messages, including MIME and other RFC2822-based message documents. It subsumes most of the functionality in several older standard modules such as rfc822, mimetypes, multifile, and other non-standard packages such as mimecntl. It is specifically not designed to do any sending of email messages to SMTP (RFC 2821), NNTP, or other servers; those are functions of modules such as smtplib and nntplib. The email package attempts to be as RFC-compliant as possible, supporting in addition to RFC 2822, such MIME-related RFCs as RFC 2045, RFC 2046, RFC 2047, and RFC 2231.

The primary distinguishing feature of the email package is that it splits the parsing and generating of email messages from the internal object model representation of email. Applications using the email package deal primarily with objects; you can add sub-objects to messages, remove sub-objects from messages, completely re-arrange the contents, etc. There is a separate parser and a separate generator which handles the transformation from flat text to the object model, and then back to flat text again. There are also handy subclasses for some common MIME object types, and a few miscellaneous utilities that help with such common tasks as extracting and parsing message field values, creating RFC-compliant dates, etc.

The following sections describe the functionality of the email package. The ordering follows a progression that should be common in applications: an email message is read as flat text from a file or other source, the text is parsed to produce the object structure of the email message, this structure is manipulated, and finally, the object tree is rendered back into flat text.

It is perfectly feasible to create the object structure out of whole cloth -- i.e. completely from scratch. From there, a similar progression can be taken as above.
Also included are detailed specifications of all the classes and modules that the email package provides, the exception classes you might encounter while using the email package, some auxiliary utilities, and a few examples. For users of the older mimelib package, or previous versions of the email package, a section on differences and porting is provided.

See Also:

Module **smtplib**:
   SMTP protocol client.

Module **nntplib**:
   NNTP protocol client.

---

Subsections

- 7.1.1 Representing an email message
- 7.1.2 Parsing email messages
   - 7.1.2.1 FeedParser API
   - 7.1.2.2 Parser class API
   - 7.1.2.3 Additional notes
- 7.1.3 Generating MIME documents
- 7.1.4 Creating email and MIME objects from scratch
- 7.1.5 Internationalized headers
- 7.1.6 Representing character sets
- 7.1.7 Encoders
- 7.1.8 Exception and Defect classes
- 7.1.9 Miscellaneous utilities
- 7.1.10 Iterators
- 7.1.11 Package History
- 7.1.12 Differences from mimelib
- 7.1.13 Examples
Representing an email

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.1.1 Representing an email message

The central class in the email package is the Message class, imported from the email.message module. It is the base class for the email object model. Message provides the core functionality for setting and querying header fields, and for accessing message bodies.

Conceptually, a Message object consists of headers and payloads. Headers are RFC 2822 style field names and values where the field name and value are separated by a colon. The colon is not part of either the field name or the field value.

Headers are stored and returned in case-preserving form but are matched case-insensitively. There may also be a single envelope header, also known as the Unix-From header or the From_ header. The payload is either a string in the case of simple message objects or a list of Message objects for MIME container documents (e.g. multipart/* and message/rfc822).

Message objects provide a mapping style interface for accessing the message headers, and an explicit interface for accessing both the headers and the payload. It provides convenience methods for generating a flat text representation of the message object tree, for accessing commonly used header parameters, and for recursively walking over the object tree.

Here are the methods of the Message class:

```python
class Message()
    The constructor takes no arguments.

as_string([unixfrom])
    Return the entire message flatten as a string. When optional unixfrom is True, the envelope header is included in the returned string. unixfrom defaults to False.

    Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it mangles lines that begin with From. For more flexibility, instantiate a
```
Generator instance and use its `flatten()` method directly. For example:

```python
from cStringIO import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=False, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

```python
__str__()
Equivalent to as_string(unixfrom=True).
```

```python
is_multipart()
Return True if the message's payload is a list of sub-Message objects, otherwise return False. When is_multipart() returns False, the payload should be a string object.
```

```python
set_unixfrom(unixfrom)
Set the message's envelope header to `unixfrom`, which should be a string.
```

```python
get_unixfrom()
Return the message's envelope header. Defaults to None if the envelope header was never set.
```

```python
attach(payload)
Add the given `payload` to the current payload, which must be None or a list of Message objects before the call. After the call, the payload will always be a list of Message objects. If you want to set the payload to a scalar object (e.g. a string), use `set_payload()` instead.
```

```python
get_payload([i, decode])
Return a reference the current payload, which will be a list of Message objects when is_multipart() is True, or a string when is_multipart() is False. If the payload is a list and you mutate the list object, you modify the message's payload in place.
```

With optional argument `i`, `get_payload()` will return the `i`-th element of the payload, counting from zero, if is_multipart() is True. An
IndexError will be raised if i is less than 0 or greater than or equal to the number of items in the payload. If the payload is a string (i.e. is_multipart() is False) and i is given, a TypeError is raised.

Optional decode is a flag indicating whether the payload should be decoded or not, according to the Content-Transfer-Encoding: header. When True and the message is not a multipart, the payload will be decoded if this header's value is "quoted-printable" or "base64". If some other encoding is used, or Content-Transfer-Encoding: header is missing, or if the payload has bogus base64 data, the payload is returned as-is (undecoded). If the message is a multipart and the decode flag is True, then None is returned. The default for decode is False.

set_payload(payload[, charset])

Set the entire message object's payload to payload. It is the client's responsibility to ensure the payload invariants. Optional charset sets the message's default character set; see set_charset() for details.

Changed in version 2.2.2: charset argument added.

set_charset(charset)

Set the character set of the payload to charset, which can either be a Charset instance (see email.charset), a string naming a character set, or None. If it is a string, it will be converted to a Charset instance. If charset is None, the charset parameter will be removed from the Content-Type: header. Anything else will generate a TypeError.

The message will be assumed to be of type text/* encoded with charset.input_charset. It will be converted to charset.output_charset and encoded properly, if needed, when generating the plain text representation of the message. MIME headers (MIME-Version:, Content-Type:, Content-Transfer-Encoding:) will be added as needed.

New in version 2.2.2.

get_charset()

Return the Charset instance associated with the message's payload. New in version 2.2.2.
The following methods implement a mapping-like interface for accessing the message's RFC 2822 headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by keys(), but in a Message object, headers are always returned in the order they appeared in the original message, or were added to the message later. Any header deleted and then re-added are always appended to the end of the header list.

These semantic differences are intentional and are biased toward maximal convenience.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

__len__()  
Return the total number of headers, including duplicates.

__contains__ (name)  
Return true if the message object has a field named name. Matching is done case-insensitively and name should not include the trailing colon. Used for the in operator, e.g.:

```python
if 'message-id' in myMessage:
   print 'Message-ID:', myMessage['message-id']
```

__getitem__ (name)  
Return the value of the named header field. name should not include the colon field separator. If the header is missing, None is returned; a KeyError is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the get_all() method to get the values of all the extant named headers.

__setitem__ (name, val)  
Add a header to the message with field name name and value val. The field
is appended to the end of the message's existing fields.

Note that this does not overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name name, delete the field first, e.g.:

```python
def del msg['subject']
msg['subject'] = 'Python roolz'
```

__delitem__(name)
Delete all occurrences of the field with name name from the message's headers. No exception is raised if the named field isn't present in the headers.

has_key(name)
Return true if the message contains a header field named name, otherwise return false.

keys()
Return a list of all the message's header field names.

values()
Return a list of all the message's field values.

items()
Return a list of 2-tuples containing all the message's field headers and values.

get(name[, failobj])
Return the value of the named header field. This is identical to __getitem__() except that optional failobj is returned if the named header is missing (defaults to None).

Here are some additional useful methods:

get_all(name[, failobj])
Return a list of all the values for the field named name. If there are no such named headers in the message, failobj is returned (defaults to None).
add_header(_name, _value, **_params)

Extended header setting. This method is similar to __setitem__() except that additional header parameters can be provided as keyword arguments. _name is the header field to add and _value is the primary value for the header.

For each item in the keyword argument dictionary _params, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as key="value" unless the value is None, in which case only the key will be added.

Here's an example:

    msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')

This will add a header that looks like

    Content-Disposition: attachment; filename="bud.gif"

replace_header(_name, _value)

Replace a header. Replace the first header found in the message that matches _name, retaining header order and field name case. If no matching header was found, a KeyError is raised.

New in version 2.2.2.

get_content_type()

Return the message's content type. The returned string is coerced to lower case of the form maintype/subtype. If there was no Content-Type: header in the message the default type as given by get_default_type() will be returned. Since according to RFC 2045, messages always have a default type, get_content_type() will always return a value.

RFC 2045 defines a message's default type to be text/plain unless it appears inside a multipart/digest container, in which case it would be message/rfc822. If the Content-Type: header has an invalid type specification, RFC 2045 mandates that the default type be text/plain.
New in version 2.2.2.

**get_content_maintype()**
Return the message's main content type. This is the maintype part of the string returned by `get_content_type()`.

New in version 2.2.2.

**get_content_subtype()**
Return the message's sub-content type. This is the subtype part of the string returned by `get_content_type()`.

New in version 2.2.2.

**get_default_type()**
Return the default content type. Most messages have a default content type of text/plain, except for messages that are subparts of multipart/digest containers. Such subparts have a default content type of message/rfc822.

New in version 2.2.2.

**set_default_type(ctype)**
Set the default content type. *ctype* should either be text/plain or message/rfc822, although this is not enforced. The default content type is not stored in the Content-Type: header.

New in version 2.2.2.

**get_params([[failobj[, header[, unquote]]]])**
Return the message's Content-Type: parameters, as a list. The elements of the returned list are 2-tuples of key/value pairs, as split on the "=" sign. The left hand side of the "=" is the key, while the right hand side is the value. If there is no "=" sign in the parameter the value is the empty string, otherwise the value is as described in `get_param()` and is unquoted if optional *unquote* is True (the default).

Optional *failobj* is the object to return if there is no Content-Type: header. Optional *header* is the header to search instead of Content-Type:
Changed in version 2.2.2: `unquote` argument added.

**get_param**( `param[, failobj[, header[, unquote]]]` )  
Return the value of the Content-Type: header's parameter `param` as a string. If the message has no Content-Type: header or if there is no such parameter, then `failobj` is returned (defaults to None).

Optional `header` if given, specifies the message header to use instead of Content-Type:.

Parameter keys are always compared case insensitively. The return value can either be a string, or a 3-tuple if the parameter was RFC 2231 encoded. When it's a 3-tuple, the elements of the value are of the form (CHARSET, LANGUAGE, VALUE). Note that both CHARSET and LANGUAGE can be None, in which case you should consider VALUE to be encoded in the us-ascii charset. You can usually ignore LANGUAGE.

If your application doesn't care whether the parameter was encoded as in RFC 2231, you can collapse the parameter value by calling `email.Utils.collapse_rfc2231_value()`, passing in the return value from `get_param()`. This will return a suitably decoded Unicode string when the value is a tuple, or the original string unquoted if it isn't. For example:

```
rawparam = msg.get_param('foo')
param = email.Utils.collapse_rfc2231_value(rawparam)
```

In any case, the parameter value (either the returned string, or the VALUE item in the 3-tuple) is always unquoted, unless `unquote` is set to False.

Changed in version 2.2.2: `unquote` argument added, and 3-tuple return value possible.

**set_param**( `param, value[, header[, reqquote[, charset[, language]]]]` )

Set a parameter in the Content-Type: header. If the parameter already exists in the header, its value will be replaced with `value`. If the Content-Type: header as not yet been defined for this message, it will be set to text/plain and the new parameter value will be appended as per RFC 2045.
Optional header specifies an alternative header to Content-Type:, and all parameters will be quoted as necessary unless optional requote is False (the default is True).

If optional charset is specified, the parameter will be encoded according to RFC 2231. Optional language specifies the RFC 2231 language, defaulting to the empty string. Both charset and language should be strings.

New in version 2.2.2.

```
del_param( param[, header[, requote]])
```

Remove the given parameter completely from the Content-Type: header. The header will be re-written in place without the parameter or its value. All values will be quoted as necessary unless requote is False (the default is True). Optional header specifies an alternative to Content-Type:.

New in version 2.2.2.

```
set_type( type[, header][, requote])
```

Set the main type and subtype for the Content-Type: header. type must be a string in the form maintype/subtype, otherwise a ValueError is raised.

This method replaces the Content-Type: header, keeping all the parameters in place. If requote is False, this leaves the existing header's quoting as is, otherwise the parameters will be quoted (the default).

An alternative header can be specified in the header argument. When the Content-Type: header is set a MIME-Version: header is also added.

New in version 2.2.2.

```
get_filename([failobj])
```

Return the value of the filename parameter of the Content-Disposition: header of the message. If the header does not have a filename parameter, this method falls back to looking for the name parameter. If neither is found, or the header is missing, then failobj is returned. The returned string will always be unquoted as per Utils.unquote().
get_boundary([failobj])
Return the value of the boundary parameter of the Content-Type: header of the message, or failobj if either the header is missing, or has no boundary parameter. The returned string will always be unquoted as per Utils.unquote().

set_boundary(boundary)
Set the boundary parameter of the Content-Type: header to boundary. set_boundary() will always quote boundary if necessary. A HeaderParseError is raised if the message object has no Content-Type: header.

Note that using this method is subtly different than deleting the old Content-Type: header and adding a new one with the new boundary via add_header(), because set_boundary() preserves the order of the Content-Type: header in the list of headers. However, it does not preserve any continuation lines which may have been present in the original Content-Type: header.

get_content_charset([failobj])
Return the charset parameter of the Content-Type: header, coerced to lower case. If there is no Content-Type: header, or if that header has no charset parameter, failobj is returned.

Note that this method differs from get_charset() which returns the Charset instance for the default encoding of the message body.

New in version 2.2.2.

getCharsets([failobj])
Return a list containing the character set names in the message. If the message is a multipart, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the charset parameter in the Content-Type: header for the represented subpart. However, if the subpart has no Content-Type: header, no charset
parameter, or is not of the text main MIME type, then that item in the returned list will be failobj.

**walk()**

The `walk()` method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use `walk()` as the iterator in a `for` loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure:

```python
>>> for part in msg.walk():
...     print part.get_content_type()
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
```

Changed in version 2.5: The previously deprecated methods `get_type()`, `get_main_type()`, and `get_subtype()` were removed.

**Message** objects can also optionally contain two instance attributes, which can be used when generating the plain text of a MIME message.

**preamble**

The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The `preamble` attribute contains this leading extra-armor text for MIME documents. When the `Parser` discovers some text after the headers but before the first boundary string, it assigns this text to the message's `preamble` attribute. When the `Generator` is writing out the plain text representation of a MIME message, and it finds the message has a `preamble` attribute, it will write this text in the area between the headers and the first
boundary. See `email.parser` and `email.generator` for details.

Note that if the message object has no preamble, the `preamble` attribute will be `None`.

**epilogue**

The `epilogue` attribute acts the same way as the `preamble` attribute, except that it contains text that appears between the last boundary and the end of the message.

Changed in version 2.5: You do not need to set the epilogue to the empty string in order for the `Generator` to print a newline at the end of the file.

**defects**

The `defects` attribute contains a list of all the problems found when parsing this message. See `email.errors` for a detailed description of the possible parsing defects.

New in version 2.4.
7.1.2 Parsing email messages

Message object structures can be created in one of two ways: they can be created from whole cloth by instantiating Message objects and stringing them together via attach() and set_payload() calls, or they can be created by parsing a flat text representation of the email message.

The email package provides a standard parser that understands most email document structures, including MIME documents. You can pass the parser a string or a file object, and the parser will return to you the root Message instance of the object structure. For simple, non-MIME messages the payload of this root object will likely be a string containing the text of the message. For MIME messages, the root object will return True from its is_multipart() method, and the subparts can be accessed via the get_payload() and walk() methods.

There are actually two parser interfaces available for use, the classic Parser API and the incremental FeedParser API. The classic Parser API is fine if you have the entire text of the message in memory as a string, or if the entire message lives in a file on the file system. FeedParser is more appropriate for when you're reading the message from a stream which might block waiting for more input (e.g. reading an email message from a socket). The FeedParser can consume and parse the message incrementally, and only returns the root object when you close the parser.

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. There is no magical connection between the email package's bundled parser and the Message class, so your custom parser can create message object trees any way it finds necessary.

Footnotes

... parser

As of email package version 3.0, introduced in Python 2.4, the classic
Parser was re-implemented in terms of the FeedParser, so the semantics and results are identical between the two parsers.

Subsections

- 7.1.2.1 FeedParser API
- 7.1.2.2 Parser class API
- 7.1.2.3 Additional notes

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.1.3 Generating MIME documents

One of the most common tasks is to generate the flat text of the email message represented by a message object structure. You will need to do this if you want to send your message via the smtplib module or the nntplib module, or print the message on the console. Taking a message object structure and producing a flat text document is the job of the Generator class.

Again, as with the email.parser module, you aren't limited to the functionality of the bundled generator; you could write one from scratch yourself. However the bundled generator knows how to generate most email in a standards-compliant way, should handle MIME and non-MIME email messages just fine, and is designed so that the transformation from flat text, to a message structure via the Parser class, and back to flat text, is idempotent (the input is identical to the output).

Here are the public methods of the Generator class, imported from the email.generator module:

```
class Generator(outfp[, mangle_from_, [maxheaderlen]])
```

The constructor for the Generator class takes a file-like object called `outfp` for an argument. `outfp` must support the `write()` method and be usable as the output file in a Python extended print statement.

Optional `mangle_from_` is a flag that, when `True`, puts a "->" character in front of any line in the body that starts exactly as "From ", i.e. From followed by a space at the beginning of the line. This is the only guaranteed portable way to avoid having such lines be mistaken for a UNIX mailbox format envelope header separator (see Why The Content-Length Format Is Bad for details). `mangle_from_` defaults to `True`, but you might want to set this to `False` if you are not writing UNIX mailbox format files.

Optional `maxheaderlen` specifies the longest length for a non-continued header. When a header line is longer than `maxheaderlen` (in characters, with tabs expanded to 8 spaces), the header will be split as defined in the email.header.Header class. Set to zero to disable header wrapping.
The default is 78, as recommended (but not required) by RFC 2822.

The other public Generator methods are:

**flatten**(msg[, unixfrom])

Print the textual representation of the message object structure rooted at `msg` to the output file specified when the Generator instance was created. Subparts are visited depth-first and the resulting text will be properly MIME encoded.

Optional `unixfrom` is a flag that forces the printing of the envelope header delimiter before the first RFC 2822 header of the root message object. If the root object has no envelope header, a standard one is crafted. By default, this is set to `False` to inhibit the printing of the envelope delimiter.

Note that for subparts, no envelope header is ever printed.

New in version 2.2.2.

**clone**(fp)

Return an independent clone of this Generator instance with the exact same options.

New in version 2.2.2.

**write**(s)

Write the string `s` to the underlying file object, i.e. `outfp` passed to Generator's constructor. This provides just enough file-like API for Generator instances to be used in extended print statements.

As a convenience, see the methods `Message.as_string()` and `str(aMessage)`, a.k.a. `Message.__str__()`, which simplify the generation of a formatted string representation of a message object. For more detail, see `email.message`.

The email.generator module also provides a derived class, called DecodedGenerator which is like the Generator base class, except that non-text parts are substituted with a format string representing the part.
class DecodedGenerator(outfp[, mangle_from_, maxheaderlen[, fmt]])

This class, derived from Generator walks through all the subparts of a message. If the subpart is of main type text, then it prints the decoded payload of the subpart. Optional _mangle_from_ and maxheaderlen are as with the Generator base class.

If the subpart is not of main type text, optional fmt is a format string that is used instead of the message payload. fmt is expanded with the following keywords, "%(keyword)s" format:

- **type** - Full MIME type of the non-text part
- **maintype** - Main MIME type of the non-text part
- **subtype** - Sub-MIME type of the non-text part
- **filename** - Filename of the non-text part
- **description** - Description associated with the non-text part
- **encoding** - Content transfer encoding of the non-text part

The default value for fmt is None, meaning

[Non-text (%(type)s) part of message omitted, filename %(filename)s]

New in version 2.2.2.

Changed in version 2.5: The previously deprecated method __call__() was removed.
7.1.3 Generating MIME documents  Up: 7.1 email  Next: 7.1.5 Internationalized headers
7.1.4 Creating email and MIME objects from scratch

Ordinarily, you get a message object structure by passing a file or some text to a parser, which parses the text and returns the root message object. However you can also build a complete message structure from scratch, or even individual Message objects by hand. In fact, you can also take an existing structure and add new Message objects, move them around, etc. This makes a very convenient interface for slicing-and-dicing MIME messages.

You can create a new object structure by creating Message instances, adding attachments and all the appropriate headers manually. For MIME messages though, the email package provides some convenient subclasses to make things easier.

Here are the classes:

**class MIMEBase(_maintype, _subtype, **_params)**

    Module: email.mime.base

    This is the base class for all the MIME-specific subclasses of Message. Ordinarily you won't create instances specifically of MIMEBase, although you could. MIMEBase is provided primarily as a convenient base class for more specific MIME-aware subclasses.

    _maintype is the Content-Type: major type (e.g. text or image), and _subtype is the Content-Type: minor type (e.g. plain or gif). _params is a parameter key/value dictionary and is passed directly to Message.add_header().

    The MIMEBase class always adds a Content-Type: header (based on _maintype, _subtype, and _params), and a MIME-Version: header (always set to 1.0).

**class MIMENonMultipart()**

    Module: email.mime.nonmultipart
A subclass of MIMEBase, this is an intermediate base class for MIME messages that are not multipart. The primary purpose of this class is to prevent the use of the attach() method, which only makes sense for multipart messages. If attach() is called, a MultipartConversionError exception is raised.

New in version 2.2.2.

```
class MIMEMultipart([subtype[, boundary[, _subparts[, _params]]]])
```

Module: email.mime.multipart

A subclass of MIMEBase, this is an intermediate base class for MIME messages that are multipart. Optional _subtype defaults to mixed, but can be used to specify the subtype of the message. A Content-Type: header of multipart/_subtype will be added to the message object. A MIME-Version: header will also be added.

Optional boundary is the multipart boundary string. When None (the default), the boundary is calculated when needed.

_subparts is a sequence of initial subparts for the payload. It must be possible to convert this sequence to a list. You can always attach new subparts to the message by using the Message.attach() method.

Additional parameters for the Content-Type: header are taken from the keyword arguments, or passed into the _params argument, which is a keyword dictionary.

New in version 2.2.2.

```
class MIMEApplication(_data[, _subtype[, _encoder[, **_params]]])
```

Module: email.mime.application

A subclass of MIMENonMultipart, the MIMEApplication class is used to represent MIME message objects of major type application. _data is a string containing the raw byte data. Optional _subtype specifies the MIME subtype and defaults to octet-stream.

Optional _encoder is a callable (i.e. function) which will perform the actual
encoding of the data for transport. This callable takes one argument, which is the MIMEApplication instance. It should use get_payload() and set_payload() to change the payload to encoded form. It should also add any Content-Transfer-Encoding: or other headers to the message object as necessary. The default encoding is base64. See the email.encoders module for a list of the built-in encoders.

_params are passed straight through to the base class constructor. New in version 2.5.

class MIMEAudio(_audiodata[, _subtype[, _encoder[, **_params]]])
Module: email.mime.audio

A subclass of MIMENonMultipart, the MIMEAudio class is used to create MIME message objects of major type audio. _audiodata is a string containing the raw audio data. If this data can be decoded by the standard Python module sndhdr, then the subtype will be automatically included in the Content-Type: header. Otherwise you can explicitly specify the audio subtype via the _subtype parameter. If the minor type could not be guessed and _subtype was not given, then TypeError is raised.

Optional _encoder is a callable (i.e. function) which will perform the actual encoding of the audio data for transport. This callable takes one argument, which is the MIMEAudio instance. It should use get_payload() and set_payload() to change the payload to encoded form. It should also add any Content-Transfer-Encoding: or other headers to the message object as necessary. The default encoding is base64. See the email.encoders module for a list of the built-in encoders.

_params are passed straight through to the base class constructor.

class MIMEImage(_imagedata[, _subtype[, _encoder[, **_params]]])
Module: email.mime.image

A subclass of MIMENonMultipart, the MIMEImage class is used to create MIME message objects of major type image. _imagedata is a string containing the raw image data. If this data can be decoded by the standard Python module imghdr, then the subtype will be automatically included in
the Content-Type: header. Otherwise you can explicitly specify the image subtype via the _subtype parameter. If the minor type could not be guessed and _subtype was not given, then TypeError is raised.

Optional _encoder is a callable (i.e. function) which will perform the actual encoding of the image data for transport. This callable takes one argument, which is the MIMEImage instance. It should use get_payload() and set_payload() to change the payload to encoded form. It should also add any Content-Transfer-Encoding: or other headers to the message object as necessary. The default encoding is base64. See the email.encoders module for a list of the built-in encoders.

__params are passed straight through to the MIMEBase constructor.

class MIMEMessage(_msg[, _subtype])
Module: email.mime.message

A subclass of MIMENonMultipart, the MIMEMessage class is used to create MIME objects of main type message. _msg is used as the payload, and must be an instance of class Message (or a subclass thereof), otherwise a TypeError is raised.

Optional _subtype sets the subtype of the message; it defaults to rfc822.

class MIMEText(_text[, _subtype[, _charset]])
Module: email.mime.text

A subclass of MIMENonMultipart, the MIMEText class is used to create MIME objects of major type text. _text is the string for the payload. _subtype is the minor type and defaults to plain. _charset is the character set of the text and is passed as a parameter to the MIMENonMultipart constructor; it defaults to us-ascii. No guessing or encoding is performed on the text data.

Changed in version 2.4: The previously deprecated _encoding argument has been removed. Encoding happens implicitly based on the _charset argument.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.1.5 Internationalized headers
RFC 2822 is the base standard that describes the format of email messages. It
derives from the older RFC 822 standard which came into widespread use at a
time when most email was composed of ASCII characters only. RFC 2822 is a
specification written assuming email contains only 7-bit ASCII characters.
Of course, as email has been deployed worldwide, it has become
internationalized, such that language specific character sets can now be used in
email messages. The base standard still requires email messages to be transferred
using only 7-bit ASCII characters, so a slew of RFCs have been written
describing how to encode email containing non-ASCII characters into RFC
2822-compliant format. These RFCs include RFC 2045, RFC 2046, RFC 2047,
and RFC 2231. The email package supports these standards in its
email.header and email.charset modules.
If you want to include non-ASCII characters in your email headers, say in the
Subject: or To: fields, you should use the Header class and assign the field in
the Message object to an instance of Header instead of using a string for the
header value. Import the Header class from the email.header module. For
example:
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> print msg.as_string()
Subject: =?iso-8859-1?q?p=F6stal?=

Notice here how we wanted the Subject: field to contain a non-ASCII
character? We did this by creating a Header instance and passing in the
character set that the byte string was encoded in. When the subsequent
Message instance was flattened, the Subject: field was properly RFC 2047
encoded. MIME-aware mail readers would show this header using the embedded
ISO-8859-1 character.
New in version 2.2.2.


Here is the `Header` class description:

```python
class Header([s[, charset[, maxlinelen[, header_name[, continuation_ws[, errors]]]]]])
```

Create a MIME-compliant header that can contain strings in different character sets.

Optional `s` is the initial header value. If `None` (the default), the initial header value is not set. You can later append to the header with `append()` method calls. `s` may be a byte string or a Unicode string, but see the `append()` documentation for semantics.

Optional `charset` serves two purposes: it has the same meaning as the `charset` argument to the `append()` method. It also sets the default character set for all subsequent `append()` calls that omit the `charset` argument. If `charset` is not provided in the constructor (the default), the `us-ascii` character set is used both as `s`'s initial charset and as the default for subsequent `append()` calls.

The maximum line length can be specified explicit via `maxlinelen`. For splitting the first line to a shorter value (to account for the field header which isn't included in `s`, e.g. `Subject:`) pass in the name of the field in `header_name`. The default `maxlinelen` is 76, and the default value for `header_name` is `None`, meaning it is not taken into account for the first line of a long, split header.

Optional `continuation_ws` must be RFC 2822-compliant folding whitespace, and is usually either a space or a hard tab character. This character will be prepended to continuation lines.

Optional `errors` is passed straight through to the `append()` method.

```python
append(s[, charset[, errors]])
```

Append the string `s` to the MIME header.

Optional `charset`, if given, should be a `Charset` instance (see `email.charset`) or the name of a character set, which will be converted to a `Charset` instance. A value of `None` (the default) means that the
charset given in the constructor is used.

s may be a byte string or a Unicode string. If it is a byte string (i.e. \texttt{isinstance(s, str)} is true), then \texttt{charset} is the encoding of that byte string, and a \texttt{UnicodeError} will be raised if the string cannot be decoded with that character set.

If \texttt{s} is a Unicode string, then \texttt{charset} is a hint specifying the character set of the characters in the string. In this case, when producing an \texttt{RFC 2822}\textsuperscript{compliant header using \texttt{RFC 2047}} rules, the Unicode string will be encoded using the following charsets in order: \texttt{us-ascii}, the \texttt{charset} hint, \texttt{utf-8}. The first character set to not provoke a \texttt{UnicodeError} is used.

Optional \texttt{errors} is passed through to any \texttt{unicode()} or \texttt{ustr.encode()} call, and defaults to ``strict''.

\texttt{encode([splitchars])}

Encode a message header into an RFC-compliant format, possibly wrapping long lines and encapsulating non-ASCII parts in base64 or quoted-printable encodings. Optional \texttt{splitchars} is a string containing characters to split long ASCII lines on, in rough support of \texttt{RFC 2822}'s highest level syntactic breaks. This doesn't affect \texttt{RFC 2047} encoded lines.

The \texttt{Header} class also provides a number of methods to support standard operators and built-in functions.

\texttt{__str__()} 
A synonym for \texttt{Header.encode()}. Useful for \texttt{str(aHeader)}.

\texttt{__unicode__()} 
A helper for the built-in \texttt{unicode()} function. Returns the header as a Unicode string.

\texttt{__eq__(other)} 
This method allows you to compare two \texttt{Header} instances for equality.

\texttt{__ne__(other)} 
This method allows you to compare two \texttt{Header} instances for inequality.
The email.header module also provides the following convenient functions.
decode_header( header)
Decode a message header value without converting the character set. The
header value is in header.
This function returns a list of (decoded_string, charset) pairs
containing each of the decoded parts of the header. charset is None for
non-encoded parts of the header, otherwise a lower case string containing
the name of the character set specified in the encoded string.
Here's an example:
>>> from email.header import decode_header
[('p\xf6stal', 'iso-8859-1')]

decoded_seq[, maxlinelen[, header_name[,
continuation_ws]]])
Create a Header instance from a sequence of pairs as returned by
decode_header().

make_header(

decode_header() takes a header value string and returns a sequence of
pairs of the format (decoded_string, charset) where charset is
the name of the character set.
This function takes one of those sequence of pairs and returns a Header
instance. Optional maxlinelen, header_name, and continuation_ws are as in
the Header constructor.

Python Library Reference
7.1.4 Creating email and Up: 7.1 email Next: 7.1.6 Representing
character sets
Previous:

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.


7.1.6 Representing character sets

This module provides a class `Charset` for representing character sets and character set conversions in email messages, as well as a character set registry and several convenience methods for manipulating this registry. Instances of `Charset` are used in several other modules within the `email` package.

Import this class from the `email.charset` module.

New in version 2.2.2.

```python
class Charset([input_charset])
    Map character sets to their email properties.

    This class provides information about the requirements imposed on email for a specific character set. It also provides convenience routines for converting between character sets, given the availability of the applicable codecs. Given a character set, it will do its best to provide information on how to use that character set in an email message in an RFC-compliant way.

    Certain character sets must be encoded with quoted-printable or base64 when used in email headers or bodies. Certain character sets must be converted outright, and are not allowed in email.

    Optional `input_charset` is as described below; it is always coerced to lower case. After being alias normalized it is also used as a lookup into the registry of character sets to find out the header encoding, body encoding, and output conversion codec to be used for the character set. For example, if `input_charset` is `iso-8859-1`, then headers and bodies will be encoded using quoted-printable and no output conversion codec is necessary. If `input_charset` is `euc-jp`, then headers will be encoded with base64, bodies will not be encoded, but output text will be converted from the `euc-jp` character set to the `iso-2022-jp` character set.

    `Charset` instances have the following data attributes:

    `input_charset`
```
The initial character set specified. Common aliases are converted to their official email names (e.g. latin_1 is converted to iso-8859-1). Defaults to 7-bit us-ascii.

**header_encoding**
If the character set must be encoded before it can be used in an email header, this attribute will be set to Charset.QP (for quoted-printable), Charset.BASE64 (for base64 encoding), or Charset.SHORTEST for the shortest of QP or BASE64 encoding. Otherwise, it will be None.

**body_encoding**
Same as header_encoding, but describes the encoding for the mail message's body, which indeed may be different than the header encoding. Charset.SHORTEST is not allowed for body_encoding.

**output_charset**
Some character sets must be converted before they can be used in email headers or bodies. If the input_charset is one of them, this attribute will contain the name of the character set output will be converted to. Otherwise, it will be None.

**input_codec**
The name of the Python codec used to convert the input_charset to Unicode. If no conversion codec is necessary, this attribute will be None.

**output_codec**
The name of the Python codec used to convert Unicode to the output_charset. If no conversion codec is necessary, this attribute will have the same value as the input_codec.

Charset instances also have the following methods:

**get_body_encoding()**
Return the content transfer encoding used for body encoding.

This is either the string "quoted-printable" or "base64" depending on the encoding used, or it is a function, in which case you should call the function with a single argument, the Message object being encoded. The function should then set the Content-Transfer-Encoding: header itself to
whatever is appropriate.

Returns the string "quoted-printable" if body_encoding is QP, returns the string "base64" if body_encoding is BASE64, and returns the string "7bit" otherwise.

**convert(s)**

Convert the string s from the input_codec to the output_codec.

**to_splittable(s)**

Convert a possibly multibyte string to a safely splittable format. s is the string to split.

Uses the input_codec to try and convert the string to Unicode, so it can be safely split on character boundaries (even for multibyte characters).

Returns the string as-is if it isn't known how to convert s to Unicode with the input_charset.

Characters that could not be converted to Unicode will be replaced with the Unicode replacement character "U+FFFD".

**from_splittable(ustr[, to_output])**

Convert a splittable string back into an encoded string. ustr is a Unicode string to ``unsplit``.

This method uses the proper codec to try and convert the string from Unicode back into an encoded format. Return the string as-is if it is not Unicode, or if it could not be converted from Unicode.

Characters that could not be converted from Unicode will be replaced with an appropriate character (usually "?").

If to_output is True (the default), uses output_codec to convert to an encoded format. If to_output is False, it uses input_codec.

**get_output_charset()**

Return the output character set.
This is the `output_charset` attribute if that is not `None`, otherwise it is `input_charset`.

**encoded_header_len()**

Return the length of the encoded header string, properly calculating for quoted-printable or base64 encoding.

**header_encode(s[, convert])**

Header-encode the string `s`.

If `convert` is `True`, the string will be converted from the input charset to the output charset automatically. This is not useful for multibyte character sets, which have line length issues (multibyte characters must be split on a character, not a byte boundary); use the higher-level `Header` class to deal with these issues (see `email.header`). `convert` defaults to `False`.

The type of encoding (base64 or quoted-printable) will be based on the `header_encoding` attribute.

**body_encode(s[, convert])**

Body-encode the string `s`.

If `convert` is `True` (the default), the string will be converted from the input charset to output charset automatically. Unlike `header_encode()`, there are no issues with byte boundaries and multibyte charsets in email bodies, so this is usually pretty safe.

The type of encoding (base64 or quoted-printable) will be based on the `body_encoding` attribute.

The `Charset` class also provides a number of methods to support standard operations and built-in functions.

**__str__()**

Returns `input_charset` as a string coerced to lower case. **__repr__()** is an alias for **__str__()**.

**__eq__(other)**
This method allows you to compare two `Charset` instances for equality.

```
__ne__(other)
```

This method allows you to compare two `Charset` instances for inequality.

The `email.charset` module also provides the following functions for adding new entries to the global character set, alias, and codec registries:

```
add_charset(charset[, header_enc[, body_enc[, output_charset]]])
```

Add character properties to the global registry.

`charset` is the input character set, and must be the canonical name of a character set.

Optional `header_enc` and `body_enc` is either `Charset.QP` for quoted-printable, `Charset.BASE64` for base64 encoding, `Charset.SHORTEST` for the shortest of quoted-printable or base64 encoding, or `None` for no encoding. `SHORTEST` is only valid for `header_enc`. The default is `None` for no encoding.

Optional `output_charset` is the character set that the output should be in. Conversions will proceed from input charset, to Unicode, to the output charset when the method `Charset.convert()` is called. The default is to output in the same character set as the input.

Both `input_charset` and `output_charset` must have Unicode codec entries in the module's character set-to-codec mapping; use `add_codec()` to add codecs the module does not know about. See the `codecs` module's documentation for more information.

The global character set registry is kept in the module global dictionary `CHARSETS`.

```
add_alias(alias, canonical)
```

Add a character set alias. `alias` is the alias name, e.g. `latin-1`. `canonical` is the character set's canonical name, e.g. `iso-8859-1`.

The global charset alias registry is kept in the module global dictionary
ALIASES.

add_codec( charset, codecname)
Add a codec that map characters in the given character set to and from Unicode.

charset is the canonical name of a character set. codecname is the name of a Python codec, as appropriate for the second argument to the unicode() built-in, or to the encode() method of a Unicode string.
7.1.7 Encoders

When creating `Message` objects from scratch, you often need to encode the payloads for transport through compliant mail servers. This is especially true for `image/*` and `text/*` type messages containing binary data.

The `email` package provides some convenient encodings in its `encoders` module. These encoders are actually used by the `MIMEAudio` and `MIMEImage` class constructors to provide default encodings. All encoder functions take exactly one argument, the message object to encode. They usually extract the payload, encode it, and reset the payload to this newly encoded value. They should also set the `Content-Transfer-Encoding:` header as appropriate.

Here are the encoding functions provided:

`encode_quopri(msg)`

Encodes the payload into quoted-printable form and sets the `Content-Transfer-Encoding:` header to `quoted-printable`. This is a good encoding to use when most of your payload is normal printable data, but contains a few unprintable characters.

`encode_base64(msg)`

Encodes the payload into base64 form and sets the `Content-Transfer-Encoding:` header to `base64`. This is a good encoding to use when most of your payload is unprintable data since it is a more compact form than quoted-printable. The drawback of base64 encoding is that it renders the text non-human readable.

`encode_7or8bit(msg)`

This doesn't actually modify the message's payload, but it does set the `Content-Transfer-Encoding:` header to either `7bit` or `8bit` as appropriate, based on the payload data.

`encode_noop(msg)`

This does nothing; it doesn't even set the `Content-Transfer-Encoding:` header.
Footnotes

...quoted-printable\footnote{7.2}

Note that encoding with \texttt{encode_quopri()} also encodes all tabs and space characters in the data.

---

Release 2.5, documentation updated on 19th September, 2006.
See \textit{About this document...} for information on suggesting changes.
7.1.8 Exception and Defect classes

The following exception classes are defined in the email.errors module:

**exception MessageError()**
This is the base class for all exceptions that the email package can raise. It is derived from the standard Exception class and defines no additional methods.

**exception MessageParseError()**
This is the base class for exceptions thrown by the Parser class. It is derived from MessageError.

**exception HeaderParseError()**
Raised under some error conditions when parsing the RFC 2822 headers of a message, this class is derived from MessageParseError. It can be raised from the Parser.parse() or Parser.parsestr() methods.

Situations where it can be raised include finding an envelope header after the first RFC 2822 header of the message, finding a continuation line before the first RFC 2822 header is found, or finding a line in the headers which is neither a header or a continuation line.

**exception BoundaryError()**
Raised under some error conditions when parsing the RFC 2822 headers of a message, this class is derived from MessageParseError. It can be raised from the Parser.parse() or Parser.parsestr() methods.

Situations where it can be raised include not being able to find the starting or terminating boundary in a multipart/* message when strict parsing is used.

**exception MultipartConversionError()**
Raised when a payload is added to a Message object using add_payload(), but the payload is already a scalar and the message's Content-Type: main type is not either multipart or missing.
MultipartConversionError multiply inherits from MessageError and the built-in TypeError.

Since Message.add_payload() is deprecated, this exception is rarely raised in practice. However the exception may also be raised if the attach() method is called on an instance of a class derived from MIMENonMultipart (e.g. MIMEImage).

Here's the list of the defects that the FeedParser can find while parsing messages. Note that the defects are added to the message where the problem was found, so for example, if a message nested inside a multipart/alternative had a malformed header, that nested message object would have a defect, but the containing messages would not.

All defect classes are subclassed from email.errors.MessageDefect, but this class is not an exception!

New in version 2.4: All the defect classes were added.

- **NoBoundaryInMultipartDefect** - A message claimed to be a multipart, but had no boundary parameter.

- **StartBoundaryNotFoundDefect** - The start boundary claimed in the Content-Type: header was never found.

- **FirstHeaderLineIsContinuationDefect** - The message had a continuation line as its first header line.

- **MisplacedEnvelopeHeaderDefect** - A `Unix From` header was found in the middle of a header block.

- **MalformedHeaderDefect** - A header was found that was missing a colon, or was otherwise malformed.

- **MultipartInvariantViolationDefect** - A message claimed to be a multipart, but no subparts were found. Note that when a message has this defect, its is_multipart() method may return false even though its content type claims to be multipart.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.1.9 Miscellaneous utilities

There are several useful utilities provided in the `email.utils` module:

**quote**( `str` )

Return a new string with backslashes in `str` replaced by two backslashes, and double quotes replaced by backslash-double quote.

**unquote**( `str` )

Return a new string which is an unquoted version of `str`. If `str` ends and begins with double quotes, they are stripped off. Likewise if `str` ends and begins with angle brackets, they are stripped off.

**parseaddr**( `address` )

Parse address - which should be the value of some address-containing field such as `To:` or `Cc:` - into its constituent `realname` and `email address` parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple of (',', '') is returned.

**formataddr**( `pair` )

The inverse of `parseaddr()`, this takes a 2-tuple of the form (`realname`, `email_address`) and returns the string value suitable for a `To:` or `Cc:` header. If the first element of `pair` is false, then the second element is returned unmodified.

**getaddresses**( `fieldvalues` )

This method returns a list of 2-tuples of the form returned by `parseaddr()`. `fieldvalues` is a sequence of header field values as might be returned by `Message.get_all()`. Here's a simple example that gets all the recipients of a message:

```python
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```
**parsedate** *(date)*
Attempts to parse a date according to the rules in [RFC 2822](https://www.rfc-editor.org/rfc/rfc2822). However, some mailers don't follow that format as specified, so parsedate() tries to guess correctly in such cases. *date* is a string containing an [RFC 2822](https://www.rfc-editor.org/rfc/rfc2822) date, such as "Mon, 20 Nov 1995 19:12:08 -0500". If it succeeds in parsing the date, parsedate() returns a 9-tuple that can be passed directly to `time.mktime()`; otherwise None will be returned. Note that fields 6, 7, and 8 of the result tuple are not usable.

**parsedate_tz** *(date)*
Performs the same function as parsedate(), but returns either None or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to `time.mktime()`, and the tenth is the offset of the date's timezone from UTC (which is the official term for Greenwich Mean Time). If the input string has no timezone, the last element of the tuple returned is None. Note that fields 6, 7, and 8 of the result tuple are not usable.

**mktime_tz** *(tuple)*
Turn a 10-tuple as returned by parsedate_tz() into a UTC timestamp. If the timezone item in the tuple is None, assume local time. Minor deficiency: mktime_tz() interprets the first 8 elements of *tuple* as a local time and then compensates for the timezone difference. This may yield a slight error around changes in daylight savings time, though not worth worrying about for common use.

**formatdate** *(timeval[, localtime][, usegmt])*  
Returns a date string as per [RFC 2822](https://www.rfc-editor.org/rfc/rfc2822), e.g.:

```
Fri, 09 Nov 2001 01:08:47 -0000
```

Optional *timeval* if given is a floating point time value as accepted by `time.gmtime()` and `time.localtime()`, otherwise the current time is used.

Optional *localtime* is a flag that when True, interprets *timeval*, and returns a date relative to the local timezone instead of UTC, properly taking daylight savings time into account. The default is False meaning UTC is used.
Optional usegmt is a flag that when True, outputs a date string with the timezone as an ascii string GMT, rather than a numeric -0000. This is needed for some protocols (such as HTTP). This only applies when localtime is False. New in version 2.4.

make_msgid([idstring])

Returns a string suitable for an RFC 2822-compliant Message-ID: header. Optional idstring if given, is a string used to strengthen the uniqueness of the message id.

decode_rfc2231(s)

Decode the string s according to RFC 2231.

encode_rfc2231(s[, charset[, language]])

Encode the string s according to RFC 2231. Optional charset and language, if given is the character set name and language name to use. If neither is given, s is returned as-is. If charset is given but language is not, the string is encoded using the empty string for language.

collapse_rfc2231_value(value[, errors[, fallback_charset]])

When a header parameter is encoded in RFC 2231 format, Message.get_param() may return a 3-tuple containing the character set, language, and value. collapse_rfc2231_value() turns this into a unicode string. Optional errors is passed to the errors argument of the built-in unicode() function; it defaults to replace. Optional fallback_charset specifies the character set to use if the one in the RFC 2231 header is not known by Python; it defaults to us-ascii.

For convenience, if the value passed to collapse_rfc2231_value() is not a tuple, it should be a string and it is returned unquoted.

decode_params(params)

Decode parameters list according to RFC 2231. params is a sequence of 2-tuples containing elements of the form (content-type, string-value).

Changed in version 2.4: The dump_address_pair() function has been
removed; use `formataddr()` instead.

Changed in version 2.4: The `decode()` function has been removed; use the `Header.decode_header()` method instead.

Changed in version 2.4: The `encode()` function has been removed; use the `Header.encode()` method instead.

---

**Footnotes**

... Time)\(^7.3\)

Note that the sign of the timezone offset is the opposite of the sign of the `time.timezone` variable for the same timezone; the latter variable follows the POSIX standard while this module follows [RFC 2822](https://www.rfc-editor.org/rfc/rfc2822).

---

**Python Library Reference**

Previous: [7.1.8 Exception and Defect](#) Up: [7.1 email](#) Next: [7.1.10 Iterators](#)

---

*Release 2.5, documentation updated on 19th September, 2006.*

See [About this document...](#) for information on suggesting changes.
7.1.10 Iterators

Iterating over a message object tree is fairly easy with the `Message.walk()` method. The `email.iterators` module provides some useful higher level iterations over message object trees.

**body_line_iterator** (`msg[, decode]`)  
This iterates over all the payloads in all the subparts of `msg`, returning the string payloads line-by-line. It skips over all the subpart headers, and it skips over any subpart with a payload that isn't a Python string. This is somewhat equivalent to reading the flat text representation of the message from a file using `readline()`, skipping over all the intervening headers.

Optional `decode` is passed through to `Message.get_payload()`.

**typed_subpart_iterator** (`msg[, maintype[, subtype]]`)  
This iterates over all the subparts of `msg`, returning only those subparts that match the MIME type specified by `maintype` and `subtype`.

Note that `subtype` is optional; if omitted, then subpart MIME type matching is done only with the main type. `maintype` is optional too; it defaults to `text`.

Thus, by default `typed_subpart_iterator()` returns each subpart that has a MIME type of `text/*`.

The following function has been added as a useful debugging tool. It should *not* be considered part of the supported public interface for the package.

**_structure** (`msg[, fp[, level]]`)  
Prints an indented representation of the content types of the message object structure. For example:

```python
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
  text/plain
  text/plain
  multipart/digest
```
Optional \( fp \) is a file-like object to print the output to. It must be suitable for Python's extended print statement. \( level \) is used internally.
7.1.11 Package History

This table describes the release history of the email package, corresponding to the version of Python that the package was released with. For purposes of this document, when you see a note about change or added versions, these refer to the Python version the change was made in, not the email package version. This table also describes the Python compatibility of each version of the package.

<table>
<thead>
<tr>
<th>email version</th>
<th>distributed with</th>
<th>compatible with</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.x</td>
<td>Python 2.2.0 to Python 2.2.1</td>
<td>no longer supported</td>
</tr>
<tr>
<td>2.5</td>
<td>Python 2.2.2+ and Python 2.3</td>
<td>Python 2.1 to 2.5</td>
</tr>
<tr>
<td>3.0</td>
<td>Python 2.4</td>
<td>Python 2.3 to 2.5</td>
</tr>
<tr>
<td>4.0</td>
<td>Python 2.5</td>
<td>Python 2.3 to 2.5</td>
</tr>
</tbody>
</table>

Here are the major differences between email version 4 and version 3:

- All modules have been renamed according to [PEP 8](https://www.python.org/dev/peps/pep-0008/) standards. For example, the version 3 module `email.Message` was renamed to `email.message` in version 4.

- A new subpackage `email.mime` was added and all the version 3 `email.MIME*` modules were renamed and situated into the `email.mime` subpackage. For example, the version 3 module `email.MIMEText` was renamed to `email.mime.text`.

  *Note that the version 3 names will continue to work until Python 2.6.*

- The `email.mime.application` module was added, which contains the MIMEApplication class.

- Methods that were deprecated in version 3 have been removed. These include `Generator.__call__()`, `Message.get_type()`, `Message.get_main_type()`, `Message.get_subtype()`.

- Fixes have been added for [RFC 2231](https://tools.ietf.org/html/rfc2231) support which can change some of the return types for `Message.get_param()` and friends. Under some circumstances, values which used to return a 3-tuple now return simple
strings (specifically, if all extended parameter segments were unencoded, there is no language and charset designation expected, so the return type is now a simple string). Also, %-decoding used to be done for both encoded and unencoded segments; this decoding is now done only for encoded segments.

Here are the major differences between email version 3 and version 2:

- The FeedParser class was introduced, and the Parser class was implemented in terms of the FeedParser. All parsing therefore is non-strict, and parsing will make a best effort never to raise an exception. Problems found while parsing messages are stored in the message's defect attribute.

- All aspects of the API which raised DeprecationWarnings in version 2 have been removed. These include the _encoder argument to the MIMEText constructor, the Message.add_payload() method, the Utils.dump_address_pair() function, and the functions Utils.decode() and Utils.encode().

- New DeprecationWarnings have been added to: Generator.__call__(), Message.get_type(), Message.get_main_type(), Message.get_subtype(), and the strict argument to the Parser class. These are expected to be removed in future versions.

- Support for Pythons earlier than 2.3 has been removed.

Here are the differences between email version 2 and version 1:

- The email.Header and email.Charset modules have been added.

- The pickle format for Message instances has changed. Since this was never (and still isn't) formally defined, this isn't considered a backward incompatibility. However if your application pickles and unpickles Message instances, be aware that in email version 2, Message instances now have private variables _charset and _default_type.

- Several methods in the Message class have been deprecated, or their
signatures changed. Also, many new methods have been added. See the documentation for the Message class for details. The changes should be completely backward compatible.

- The object structure has changed in the face of message/rfc822 content types. In email version 1, such a type would be represented by a scalar payload, i.e. the container message's is_multipart() returned false, get_payload() was not a list object, but a single Message instance.

This structure was inconsistent with the rest of the package, so the object representation for message/rfc822 content types was changed. In email version 2, the container does return True from is_multipart(), and get_payload() returns a list containing a single Message item.

Note that this is one place that backward compatibility could not be completely maintained. However, if you're already testing the return type of get_payload(), you should be fine. You just need to make sure your code doesn't do a set_payload() with a Message instance on a container with a content type of message/rfc822.

- The Parser constructor's strict argument was added, and its parse() and parsestr() methods grew a headersonly argument. The strict flag was also added to functions email.message_from_file() and email.message_from_string().

- Generator.__call__() is deprecated; use Generator.flatten() instead. The Generator class has also grown the clone() method.

- The DecodedGenerator class in the email.Generator module was added.

- The intermediate base classes MIMENonMultipart and MIME_Multipart have been added, and interposed in the class hierarchy for most of the other MIME-related derived classes.

- The _encoder argument to the MIMEText constructor has been deprecated. Encoding now happens implicitly based on the _charset argument.
The following functions in the `email.Utils` module have been deprecated: `dump_address_pairs()`, `decode()`, and `encode()`. The following functions have been added to the module: `make_msgid()`, `decode_rfc2231()`, `encode_rfc2231()`, and `decode_params()`.

The non-public function `email.Iterators._structure()` was added.
7.1.12 Differences from mimelib

The email package was originally prototyped as a separate library called mimelib. Changes have been made so that method names are more consistent, and some methods or modules have either been added or removed. The semantics of some of the methods have also changed. For the most part, any functionality available in mimelib is still available in the email package, albeit often in a different way. Backward compatibility between the mimelib package and the email package was not a priority.

Here is a brief description of the differences between the mimelib and the email packages, along with hints on how to port your applications.

Of course, the most visible difference between the two packages is that the package name has been changed to email. In addition, the top-level package has the following differences:

- messageFromString() has been renamed to message_from_string().
- messageFromFile() has been renamed to message_from_file().

The Message class has the following differences:

- The method asString() was renamed to as_string().
- The method ismultipart() was renamed to is_multipart().
- The get_payload() method has grown a decode optional argument.
- The method getall() was renamed to get_all().
- The method addheader() was renamed to add_header().
- The method gettype() was renamed to get_type().
- The method getmaintype() was renamed to get_main_type().
• The method `getsubtype()` was renamed to `get_subtype()`.

• The method `getparams()` was renamed to `get_params()`. Also, whereas `getparams()` returned a list of strings, `get_params()` returns a list of 2-tuples, effectively the key/value pairs of the parameters, split on the "=" sign.

• The method `getparam()` was renamed to `get_param()`.

• The method `getcharsets()` was renamed to `get_charsets()`.

• The method `getfilename()` was renamed to `get_filename()`.

• The method `getboundary()` was renamed to `get_boundary()`.

• The method `setboundary()` was renamed to `set_boundary()`.

• The method `getdecodedpayload()` was removed. To get similar functionality, pass the value 1 to the `decode` flag of the `get_payload()` method.

• The method `getpayloadastext()` was removed. Similar functionality is supported by the `DecodedGenerator` class in the `email.generator` module.

• The method `getbodyastext()` was removed. You can get similar functionality by creating an iterator with `typed_subpart_iterator()` in the `email.iterators` module.

The `Parser` class has no differences in its public interface. It does have some additional smarts to recognize `message/delivery-status` type messages, which it represents as a `Message` instance containing separate `Message` subparts for each header block in the delivery status notification.\(^4\)

The `Generator` class has no differences in its public interface. There is a new class in the `email.generator` module though, called `DecodedGenerator` which provides most of the functionality previously available in the `Message.getpayloadastext()` method.
The following modules and classes have been changed:

- The MIMEBase class constructor arguments _major and _minor have changed to _maintype and _subtype respectively.

- The Image class/module has been renamed to MIMEImage. The _minor argument has been renamed to _subtype.

- The Text class/module has been renamed to MIMEText. The _minor argument has been renamed to _subtype.

- The MessageRFC822 class/module has been renamed to MIMEMessage. Note that an earlier version of mimelib called this class/module RFC822, but that clashed with the Python standard library module rfc822 on some case-insensitive file systems.

  Also, the MIMEMessage class now represents any kind of MIME message with main type message. It takes an optional argument _subtype which is used to set the MIME subtype. _subtype defaults to rfc822.

mimelib provided some utility functions in its address and date modules. All of these functions have been moved to the email.utils module.

The MsgReader class/module has been removed. Its functionality is most closely supported in the body_line_iterator() function in the email.iterators module.

---

Footnotes

... notification\(^7.4\)  
Delivery Status Notifications (DSN) are defined in RFC 1894.
See About this document... for information on suggesting changes.
7.1.13 Examples

Here are a few examples of how to use the email package to read, write, and send simple email messages, as well as more complex MIME messages.

First, let's see how to create and send a simple text message:

```python
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.mime.text import MIMEText
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart

# Open a plain text file for reading. For this example, assume that
# the text file contains only ASCII characters.
fp = open(textfile, 'rb')
# Create a text/plain message
msg = MIMEText(fp.read())
fp.close()

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = 'The contents of %s' % textfile
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server, but don't include the
# envelope header.
s = smtplib.SMTP()
s.connect()
s.sendmail(me, [you], msg.as_string())
s.close()
```

Download as text (original file name: email-simple.py).

Here's an example of how to send a MIME message containing a bunch of family pictures that may be residing in a directory:

```python
# Import smtplib for the actual sending function
import smtplib

# Here are the email package modules we'll need
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart

COMMASPACE = ', '
```
# Create the container (outer) email message.
msg = MIME_Multipart()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = COMMASPACE.join(family)
msg.preamble = 'Our family reunion'

# Assume we know that the image files are all in PNG format
for file in pngfiles:
    # Open the files in binary mode. Let the MIMEImage class automa
    # guess the specific image type.
    fp = open(file, 'rb')
    img = MIMEImage(fp.read())
    fp.close()
    msg.attach(img)

# Send the email via our own SMTP server.
s = smtplib.SMTP()
s.connect()
s.sendmail(me, family, msg.as_string())
s.close()

Download as text (original file name: email-mime.py).

Here's an example of how to send the entire contents of a directory as an email message.

#!/usr/bin/env python

"""Send the contents of a directory as a MIME message.""

import os
import sys
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from optparse import OptionParser

from email import encoders
from email.message import Message
from email.mime.audio import MIMEAudio
from email.mime.base import MIMEBase
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

COMMASPACE = ' ', ','
def main():
    parser = OptionParser(usage="""Send the contents of a directory as a MIME message.
Usage: %prog [options]"

Unless the -o option is given, the email is sent by forwarding to your SMTP server, which then does the normal delivery process. Your local machine must be running an SMTP server.
""")
    parser.add_option('-d', '--directory',
                      type='string', action='store',
                      help="Mail the contents of the specified dir otherwise use the current directory. Only the files in the directory are sent, and we don't recurse to subdirectories.""")
    parser.add_option('-o', '--output',
                      type='string', action='store', metavar='FILE',
                      help="Print the composed message to FILE instead of sending the message to the SMTP server.""")
    parser.add_option('-s', '--sender',
                      type='string', action='store', metavar='SENDER',
                      help='The value of the From: header (required)')
    parser.add_option('-r', '--recipient',
                      type='string', action='append', metavar='RECIPIENT',
                      default=[], dest='recipients',
                      help='A To: header value (at least one required)')

    opts, args = parser.parse_args()
    if not opts.sender or not opts.recipients:
        parser.print_help()
        sys.exit(1)

directory = opts.directory
    if not directory:
        directory = '.

    # Create the enclosing (outer) message
    outer = MIME(Multipart())
    outer['Subject'] = 'Contents of directory %s' % os.path.abspath(directory)
    outer['To'] = COMMASPACE.join(opts.recipients)
    outer['From'] = opts.sender
    outer.preamble = 'You will not see this in a MIME-aware mail reader

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension. Enc will be ignored, although we should check for simple thing # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compre
            # use a generic bag-of-bits type.
And finally, here's an example of how to unpack a MIME message like the one above, into a directory of files:

```
#!/usr/bin/env python

"""Unpack a MIME message into a directory of files."""

import os
import sys
```
import email
import errno
import mimetypes

from optparse import OptionParser

def main():
    parser = OptionParser(usage="Unpack a MIME message into a directory of files.
Usage: \%prog [options] msgfile"
)
    parser.add_option('-d', '--directory',
                      type='string', action='store',
                      help="Unpack the MIME message into the named directory, which will be created if it doesn't exist."
)
    opts, args = parser.parse_args()
    if not opts.directory:
        parser.print_help()
        sys.exit(1)

    try:
        msgfile = args[0]
    except IndexError:
        parser.print_help()
        sys.exit(1)

    try:
        os.mkdir(opts.directory)
    except OSError, e:
        # Ignore directory exists error
        if e.errno <> errno.EEXIST:
            raise

    fp = open(msgfile)
    msg = email.message_from_file(fp)
    fp.close()

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename so
        # email message can't be used to overwrite important files
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_type())
            if not ext:
                # Use a generic bag-of-bits extension
                ext = '.bin'
filename = 'part-%03d%s' % (counter, ext)
counter += 1
fp = open(os.path.join(opts.directory, filename), 'wb')
fp.write(part.get_payload(decode=True))
fp.close()

if __name__ == '__main__':
    main()

Download as text (original file name: email-unpack.py).

Footnotes

... message:7.5
    Thanks to Matthew Dixon Cowles for the original inspiration and examples.

Python Library Reference

Previous: 7.1.12 Differences from mimelib  Up: 7.1 email  Next: 7.2 mailcap

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.2 mailcap -- Mailcap file handling.

Mailcap files are used to configure how MIME-aware applications such as mail readers and Web browsers react to files with different MIME types. (The name `mailcap" is derived from the phrase "mail capability".) For example, a mailcap file might contain a line like "video/mpeg; xmpeg %s". Then, if the user encounters an email message or Web document with the MIME type video/mpeg, "%s" will be replaced by a filename (usually one belonging to a temporary file) and the xmpeg program can be automatically started to view the file.

The mailcap format is documented in RFC 1524, "A User Agent Configuration Mechanism For Multimedia Mail Format Information," but is not an Internet standard. However, mailcap files are supported on most UNIX systems.

\[\text{findmatch}(caps, \text{MIMEtype}[, key[, filename[, plist]])}\]

Return a 2-tuple; the first element is a string containing the command line to be executed (which can be passed to `os.system()`), and the second element is the mailcap entry for a given MIME type. If no matching MIME type can be found, (None, None) is returned.

\(key\) is the name of the field desired, which represents the type of activity to be performed; the default value is 'view', since in the most common case you simply want to view the body of the MIME-typed data. Other possible values might be 'compose' and 'edit', if you wanted to create a new body of the given MIME type or alter the existing body data. See RFC 1524 for a complete list of these fields.

\(filename\) is the filename to be substituted for "%s" in the command line; the default value is '/dev/null' which is almost certainly not what you want, so usually you'll override it by specifying a filename.

\(plist\) can be a list containing named parameters; the default value is simply an empty list. Each entry in the list must be a string containing the parameter name, an equals sign ('='), and the parameter's value. Mailcap entries can contain named parameters like \%{foo}, which will be replaced
by the value of the parameter named 'foo'. For example, if the command line "showpartial %id %number %total" was in a mailcap file, and $plist was set to ['id=1', 'number=2', 'total=3'], the resulting command line would be 'showpartial 1 2 3'.

In a mailcap file, the ``test'' field can optionally be specified to test some external condition (such as the machine architecture, or the window system in use) to determine whether or not the mailcap line applies. $findmatch() will automatically check such conditions and skip the entry if the check fails.

$getcaps()

Returns a dictionary mapping MIME types to a list of mailcap file entries. This dictionary must be passed to the $findmatch() function. An entry is stored as a list of dictionaries, but it shouldn't be necessary to know the details of this representation.

The information is derived from all of the mailcap files found on the system. Settings in the user's mailcap file $HOME/.mailcap will override settings in the system mailcap files /etc/mailcap, /usr/etc/mailcap, and /usr/local/etc/mailcap.

An example usage:

```python
>>> import mailcap
>>> d=mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='/tmp/tmp1223')
('xmpeg /tmp/tmp1223', {'view': 'xmpeg %s'})
```
7.3 mailbox -- Manipulate mailboxes in various formats

This module defines two classes, Mailbox and Message, for accessing and manipulating on-disk mailboxes and the messages they contain. Mailbox offers a dictionary-like mapping from keys to messages. Message extends the email.Message module's Message class with format-specific state and behavior. Supported mailbox formats are Maildir, mbox, MH, Babyl, and MMDF.

See Also:
Module email:
   Represent and manipulate messages.

Subsections

- 7.3.1 Mailbox objects
  - 7.3.1.1 Maildir
  - 7.3.1.2 mbox
  - 7.3.1.3 MH
  - 7.3.1.4 Babyl
  - 7.3.1.5 MMDF
- 7.3.2 Message objects
  - 7.3.2.1 MaildirMessage
  - 7.3.2.2 mboxMessage
  - 7.3.2.3 MHMessage
  - 7.3.2.4 BabylMessage
  - 7.3.2.5 MMDFMessage
- 7.3.3 Exceptions
- 7.3.4 Deprecated classes and methods
- 7.3.5 Examples
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.3.1 Mailbox objects

class Mailbox
    A mailbox, which may be inspected and modified.

The Mailbox interface is dictionary-like, with small keys corresponding to messages. Keys are issued by the Mailbox instance with which they will be used and are only meaningful to that Mailbox instance. A key continues to identify a message even if the corresponding message is modified, such as by replacing it with another message. Messages may be added to a Mailbox instance using the set-like method add() and removed using a del statement or the set-like methods remove() and discard().

Mailbox interface semantics differ from dictionary semantics in some noteworthy ways. Each time a message is requested, a new representation (typically a Message instance) is generated, based upon the current state of the mailbox. Similarly, when a message is added to a Mailbox instance, the provided message representation's contents are copied. In neither case is a reference to the message representation kept by the Mailbox instance.

The default Mailbox iterator iterates over message representations, not keys as the default dictionary iterator does. Moreover, modification of a mailbox during iteration is safe and well-defined. Messages added to the mailbox after an iterator is created will not be seen by the iterator. Messages removed from the mailbox before the iterator yields them will be silently skipped, though using a key from an iterator may result in a KeyError exception if the corresponding message is subsequently removed.

Mailbox itself is intended to define an interface and to be inherited from by format-specific subclasses but is not intended to be instantiated. Instead, you should instantiate a subclass.

Mailbox instances have the following methods:

add( message)
    Add message to the mailbox and return the key that has been assigned to it.
Parameter `message` may be a `Message` instance, an `email.Message.Message` instance, a string, or a file-like object (which should be open in text mode). If `message` is an instance of the appropriate format-specific `Message` subclass (e.g., if it's an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, reasonable defaults for format-specific information are used.

`remove(key)`
`__delitem__(key)`
`discard(key)`
Delete the message corresponding to `key` from the mailbox.

If no such message exists, a `KeyError` exception is raised if the method was called as `remove()` or `__delitem__()`, but no exception is raised if the method was called as `discard()`. The behavior of `discard()` may be preferred if the underlying mailbox format supports concurrent modification by other processes.

`__setitem__(key, message)`
Replace the message corresponding to `key` with `message`. Raise a `KeyError` exception if no message already corresponds to `key`.

As with `add()`, parameter `message` may be a `Message` instance, an `email.Message.Message` instance, a string, or a file-like object (which should be open in text mode). If `message` is an instance of the appropriate format-specific `Message` subclass (e.g., if it's an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, the format-specific information of the message that currently corresponds to `key` is left unchanged.

`iterkeys()`
`keys()`
Return an iterator over all keys if called as `iterkeys()` or return a list of keys if called as `keys()`.

`itervalues()`
__iter__() values()
Return an iterator over representations of all messages if called as itervalues() or __iter__() or return a list of such representations if called as values(). The messages are represented as instances of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized. Note: The behavior of __iter__() is unlike that of dictionaries, which iterate over keys.

iteritems() items()
Return an iterator over (key, message) pairs, where key is a key and message is a message representation, if called as iteritems() or return a list of such pairs if called as items(). The messages are represented as instances of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

get( key[, default=None])
__getitem__(key)
Return a representation of the message corresponding to key. If no such message exists, default is returned if the method was called as get() and a KeyError exception is raised if the method was called as __getitem__(). The message is represented as an instance of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

get_message( key)
Return a representation of the message corresponding to key as an instance of the appropriate format-specific Message subclass, or raise a KeyError exception if no such message exists.

get_string( key)
Return a string representation of the message corresponding to key, or raise a KeyError exception if no such message exists.
get_file( key)
    Return a file-like representation of the message corresponding to key, or raise a KeyError exception if no such message exists. The file-like object behaves as if open in binary mode. This file should be closed once it is no longer needed.

    Note: Unlike other representations of messages, file-like representations are not necessarily independent of the Mailbox instance that created them or of the underlying mailbox. More specific documentation is provided by each subclass.

has_key( key)
__contains__( key)
    Return True if key corresponds to a message, False otherwise.

__len__()
    Return a count of messages in the mailbox.

clear()
    Delete all messages from the mailbox.

pop( key[, default])
    Return a representation of the message corresponding to key and delete the message. If no such message exists, return default if it was supplied or else raise a KeyError exception. The message is represented as an instance of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

popitem()
    Return an arbitrary (key, message) pair, where key is a key and message is a message representation, and delete the corresponding message. If the mailbox is empty, raise a KeyError exception. The message is represented as an instance of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

update( arg)
Parameter `arg` should be a `key-to-message` mapping or an iterable of `(key, message)` pairs. Updates the mailbox so that, for each given `key` and `message`, the message corresponding to `key` is set to `message` as if by using `__setitem__()`. As with `__setitem__()`, each `key` must already correspond to a message in the mailbox or else a `KeyError` exception will be raised, so in general it is incorrect for `arg` to be a `Mailbox` instance. **Note:** Unlike with dictionaries, keyword arguments are not supported.

**flush()**
Write any pending changes to the filesystem. For some `Mailbox` subclasses, changes are always written immediately and this method does nothing.

**lock()**
Acquire an exclusive advisory lock on the mailbox so that other processes know not to modify it. An `ExternalClashError` is raised if the lock is not available. The particular locking mechanisms used depend upon the mailbox format.

**unlock()**
Release the lock on the mailbox, if any.

**close()**
Flush the mailbox, unlock it if necessary, and close any open files. For some `Mailbox` subclasses, this method does nothing.

---

**Subsections**

- [7.3.1.1 Maildir](#)
- [7.3.1.2 mbox](#)
- [7.3.1.3 MH](#)
- [7.3.1.4 Babyl](#)
- [7.3.1.5 MMDF](#)
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.3.2 Message objects

class Message([message])

If message is omitted, the new instance is created in a default, empty state. If message is an email.Message.Message instance, its contents are copied; furthermore, any format-specific information is converted insofar as possible if message is a Message instance. If message is a string or a file, it should contain an RFC 2822-compliant message, which is read and parsed.

The format-specific state and behaviors offered by subclasses vary, but in general it is only the properties that are not specific to a particular mailbox that are supported (although presumably the properties are specific to a particular mailbox format). For example, file offsets for single-file mailbox formats and file names for directory-based mailbox formats are not retained, because they are only applicable to the original mailbox. But state such as whether a message has been read by the user or marked as important is retained, because it applies to the message itself.

There is no requirement that Message instances be used to represent messages retrieved using Mailbox instances. In some situations, the time and memory required to generate Message representations might not not acceptable. For such situations, Mailbox instances also offer string and file-like representations, and a custom message factory may be specified when a Mailbox instance is initialized.

---

Subsections

- 7.3.2.1 MaildirMessage
- 7.3.2.2 mboxMessage
- 7.3.2.3 MHMessage
7.3.2.4 BabylMessage
7.3.2.5 MMDFMessage

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.3.3 Exceptions

The following exception classes are defined in the mailbox module:

**class Error()**
The based class for all other module-specific exceptions.

**class NoSuchMailboxError()**
Raised when a mailbox is expected but is not found, such as when instantiating a Mailbox subclass with a path that does not exist (and with the create parameter set to False), or when opening a folder that does not exist.

**class NotEmptyError()**
Raised when a mailbox is not empty but is expected to be, such as when deleting a folder that contains messages.

**class ExternalClashError()**
Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely-generated file name already exists.

**class FormatError()**
Raised when the data in a file cannot be parsed, such as when an MH instance attempts to read a corrupted .mh_sequences file.
7.3.4 Deprecated classes and methods

Older versions of the mailbox module do not support modification of mailboxes, such as adding or removing message, and do not provide classes to represent format-specific message properties. For backward compatibility, the older mailbox classes are still available, but the newer classes should be used in preference to them.

Older mailbox objects support only iteration and provide a single public method:

next()
Return the next message in the mailbox, created with the optional factory argument passed into the mailbox object's constructor. By default this is an rfc822.Message object (see the rfc822 module). Depending on the mailbox implementation the fp attribute of this object may be a true file object or a class instance simulating a file object, taking care of things like message boundaries if multiple mail messages are contained in a single file, etc. If no more messages are available, this method returns None.

Most of the older mailbox classes have names that differ from the current mailbox class names, except for Maildir. For this reason, the new Maildir class defines a next() method and its constructor differs slightly from those of the other new mailbox classes.

The older mailbox classes whose names are not the same as their newer counterparts are as follows:

class UnixMailbox( fp[, factory])
Access to a classic UNIX-style mailbox, where all messages are contained in a single file and separated by "From "(a.k.a. "From_") lines. The file object fp points to the mailbox file. The optional factory parameter is a callable that should create new message objects. factory is called with one argument, fp by the next() method of the mailbox object. The default is the rfc822.Message class (see the rfc822 module - and the note below).
Note: For reasons of this module's internal implementation, you will probably want to open the fp object in binary mode. This is especially important on Windows.

For maximum portability, messages in a UNIX-style mailbox are separated by any line that begins exactly with the string 'From ' (note the trailing space) if preceded by exactly two newlines. Because of the wide-range of variations in practice, nothing else on the From_line should be considered. However, the current implementation doesn't check for the leading two newlines. This is usually fine for most applications.

The UnixMailbox class implements a more strict version of From_line checking, using a regular expression that usually correctly matched From_delimiters. It considers delimiter line to be separated by "From name time" lines. For maximum portability, use the PortableUnixMailbox class instead. This class is identical to UnixMailbox except that individual messages are separated by only "From " lines.

For more information, see Configuring Netscape Mail on UNIX: Why the Content-Length Format is Bad.

class PortableUnixMailbox(fp[, factory])
A less-strict version of UnixMailbox, which considers only the "From " at the beginning of the line separating messages. The ``name time" portion of the From line is ignored, to protect against some variations that are observed in practice. This works since lines in the message which begin with 'From ' are quoted by mail handling software at delivery-time.

class MmdfMailbox(fp[, factory])
Access an MMDF-style mailbox, where all messages are contained in a single file and separated by lines consisting of 4 control-A characters. The file object fp points to the mailbox file. Optional factory is as with the UnixMailbox class.

class MHMailbox(dirname[, factory])
Access an MH mailbox, a directory with each message in a separate file with a numeric name. The name of the mailbox directory is passed in \textit{dirname}. \textit{factory} is as with the \texttt{UnixMailbox} class.

\begin{verbatim}
class BabylMailbox((fp, factory))
  Access a Babyl mailbox, which is similar to an MMDF mailbox. In Babyl format, each message has two sets of headers, the \textit{original} headers and the \textit{visible} headers. The original headers appear before a line containing only '*** EOOH ***' (End-Of-Original-Headers) and the visible headers appear after the EOOH line. Babyl-compliant mail readers will show you only the visible headers, and \texttt{BabylMailbox} objects will return messages containing only the visible headers. You'll have to do your own parsing of the mailbox file to get at the original headers. Mail messages start with the EOOH line and end with a line containing only '\037\014'. \textit{factory} is as with the \texttt{UnixMailbox} class.
\end{verbatim}

If you wish to use the older mailbox classes with the \texttt{email} module rather than the deprecated \texttt{rfc822} module, you can do so as follows:

\begin{verbatim}
import email
import email.Errors
import mailbox

def msgfactory(fp):
  try:
    return email.message_from_file(fp)
  except email.Errors.MessageParseError:
    # Don't return None since that will
    # stop the mailbox iterator
    return ''

mbox = mailbox.UnixMailbox(fp, msgfactory)
\end{verbatim}

Alternatively, if you know your mailbox contains only well-formed MIME messages, you can simplify this to:

\begin{verbatim}
import email
import mailbox

mbox = mailbox.UnixMailbox(fp, email.message_from_file)
\end{verbatim}
7.3.5 Examples

A simple example of printing the subjects of all messages in a mailbox that seem interesting:

```python
import mailbox
for message in mailbox.mbox('~/mbox '):
    subject = message['subject']  # Could possibly be None.
    if subject and 'python' in subject.lower():
        print subject
```

To copy all mail from a Babyl mailbox to an MH mailbox, converting all of the format-specific information that can be converted:

```python
import mailbox
destination = mailbox.MH('~/Mail')
for message in mailbox.Babyl('~/RMAIL '):
    destination.add(MHMessage(message))
```

An example of sorting mail from numerous mailing lists, being careful to avoid mail corruption due to concurrent modification by other programs, mail loss due to interruption of the program, or premature termination due to malformed messages in the mailbox:

```python
import mailbox
import email.Errors
list_names = ('python-list', 'python-dev', 'python-bugs')
boxes = dict((name, mailbox.mbox('~/email/%s' % name)) for name in list_names)
inbox = mailbox.Maildir('~/Maildir', None)
for key in inbox.iterkeys():
    try:
        message = inbox[key]
        except email.Errors.MessageParseError:
            continue  # The message is malformed. Just leave it.
    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            box = boxes[name]
            box.lock()
            box.add(message)
            box.flush()  # Write copy to disk before removing
            box.unlock()
            inbox.discard(key)
            break  # Found destination, so stop looking
    for box in boxes.itervalues():
        box.close()
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.4 mhlib -- Access to MH mailboxes

The mhlib module provides a Python interface to MH folders and their contents.

The module contains three basic classes, MH, which represents a particular collection of folders, Folder, which represents a single folder, and Message, which represents a single message.

```python
class MH([path[, profile]])
    MH represents a collection of MH folders.

class Folder(mh, name)
    The Folder class represents a single folder and its messages.

class Message(folder, number[, name])
    Message objects represent individual messages in a folder. The Message class is derived from mimetools.Message.
```

Subsections

- 7.4.1 MH Objects
- 7.4.2 Folder Objects
- 7.4.3 Message Objects

Python Library Reference

Previous: 7.3.5 Examples Up: 7. Internet Data Handling Next: 7.4.1 MH Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.4.1 MH Objects

MH instances have the following methods:

**error**(*format[, ...]*)
Print an error message - can be overridden.

**getprofile**(*key*)
Return a profile entry (None if not set).

**getpath**()
Return the mailbox pathname.

**getcontext**()
Return the current folder name.

**setcontext**(*name*)
Set the current folder name.

**listfolders**()
Return a list of top-level folders.

**listallfolders**()
Return a list of all folders.

**listsubfolders**(*name*)
Return a list of direct subfolders of the given folder.

**listallsubfolders**(*name*)
Return a list of all subfolders of the given folder.

**makefolder**(*name*)
Create a new folder.

**deletefolder**(*name*)
Delete a folder - must have no subfolders.

**openfolder**(*name*)

Return a new open folder object.
7.4.2 Folder Objects

Folder instances represent open folders and have the following methods:

**error**(*format*, ...)  
Print an error message - can be overridden.

**getfullname**()  
Return the folder's full pathname.

**getsequencesfilename**()  
Return the full pathname of the folder's sequences file.

**getmessagefilename**(*n*)  
Return the full pathname of message *n* of the folder.

**listmessages**()  
Return a list of messages in the folder (as numbers).

**getcurrent**()  
Return the current message number.

**setcurrent**(*n*)  
Set the current message number to *n*.

**parsesequence**(*seq*)  
Parse msgs syntax into list of messages.

**getlast**()  
Get last message, or 0 if no messages are in the folder.

**setlast**(*n*)  
Set last message (internal use only).

**getsequences**()
Return dictionary of sequences in folder. The sequence names are used as keys, and the values are the lists of message numbers in the sequences.

**putsequences**( *dict*)
Return dictionary of sequences in folder name: list.

**removemessages**( *list*)
Remove messages in list from folder.

**refilemessages**( *list, tofolder*)
Move messages in list to other folder.

**movemessage**( *n, tofolder, ton*)
Move one message to a given destination in another folder.

**copymessage**( *n, tofolder, ton*)
Copy one message to a given destination in another folder.
7.4.3 Message Objects

The Message class adds one method to those of mimetools.Message:

`openmessage(n)`

Return a new open message object (costs a file descriptor).
7.5 mime tools -- Tools for parsing MIME messages

Deprecated since release 2.3. The email package should be used in preference to the mimetools module. This module is present only to maintain backward compatibility.

This module defines a subclass of the rfc822 module's Message class and a number of utility functions that are useful for the manipulation for MIME multipart or encoded message.

It defines the following items:

**class Message** (fp[, seekable])
Return a new instance of the Message class. This is a subclass of the rfc822.Message class, with some additional methods (see below). The seekable argument has the same meaning as for rfc822.Message.

**choose_boundary()**
Return a unique string that has a high likelihood of being usable as a part boundary. The string has the form
'hostipaddr.uid.pid.timestamp.random'.

**decode**( input, output, encoding)
Read data encoded using the allowed MIME encoding from open file object input and write the decoded data to open file object output. Valid values for encoding include 'base64','quoted-printable','uuencode','x-uuencode','uue','x-uue','7bit', and '8bit'. Decoding messages encoded in '7bit' or '8bit' has no effect. The input is simply copied to the output.

**encode**( input, output, encoding)
Read data from open file object input and write it encoded using the allowed MIME encoding to open file object output. Valid values for
encoding are the same as for decode().

**copyliteral** *(input, output)*
Read lines from open file *input* until EOF and write them to open file *output*.

**copybinary** *(input, output)*
Read blocks until EOF from open file *input* and write them to open file *output*. The block size is currently fixed at 8192.

**See Also:**

Module **email**: Comprehensive email handling package; supersedes the mimetools module.

Module **rfc822**: Provides the base class for mimetools.Message.

Module **multifile**: Support for reading files which contain distinct parts, such as MIME data.

The MIME Frequently Asked Questions document. For an overview of MIME, see the answer to question 1.1 in Part 1 of this document.

---

Subsections

- **7.5.1 Additional Methods of Message Objects**

---

Python Library Reference

Previous: **7.4.3 Message Objects**  Up: **7. Internet Data Handling**  Next: **7.5.1 Additional Methods of**
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
7.5.1 Additional Methods of Message Objects

The Message class defines the following methods in addition to the rfc822.Message methods:

getplist()
Return the parameter list of the Content-Type: header. This is a list of strings. For parameters of the form "key=value", key is converted to lower case but value is not. For example, if the message contains the header "Content-type: text/html; spam=1; Spam=2; Spam" then getplist() will return the Python list ['spam=1', 'spam=2', 'Spam'].

getparam(name)
Return the value of the first parameter (as returned by getplist()) of the form "name=value" for the given name. If value is surrounded by quotes of the form `<...>' or "...", these are removed.

getencoding()
Return the encoding specified in the Content-Transfer-Encoding: message header. If no such header exists, return '7bit'. The encoding is converted to lower case.

gettype()
Return the message type (of the form "type/subtype") as specified in the Content-Type: header. If no such header exists, return 'text/plain'. The type is converted to lower case.

getmaintype()
Return the main type as specified in the Content-Type: header. If no such header exists, return 'text'. The main type is converted to lower case.

getsubtype()
Return the subtype as specified in the Content-Type: header. If no such header exists, return 'plain'. The subtype is converted to lower case.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.6 mimetypes -- Map filenames to MIME types

The mimetypes module converts between a filename or URL and the MIME type associated with the filename extension. Conversions are provided from filename to MIME type and from MIME type to filename extension; encodings are not supported for the latter conversion.

The module provides one class and a number of convenience functions. The functions are the normal interface to this module, but some applications may be interested in the class as well.

The functions described below provide the primary interface for this module. If the module has not been initialized, they will call init() if they rely on the information init() sets up.

**guess_type** *(filename[, strict]*)

Guess the type of a file based on its filename or URL, given by *filename*. The return value is a tuple *(type, encoding)* where *type* is None if the type can't be guessed (missing or unknown suffix) or a string of the form 'type/subtype', usable for a MIME content-type: header.

*encoding* is None for no encoding or the name of the program used to encode (e.g. **compress** or **gzip**). The encoding is suitable for use as a Content-Encoding: header, not as a Content-Transfer-Encoding: header. The mappings are table driven. Encoding suffixes are case sensitive; type suffixes are first tried case sensitively, then case insensitively.

Optional *strict* is a flag specifying whether the list of known MIME types is limited to only the official types [registered with IANA](https://www.iana.org/assignments/media-types) are recognized. When *strict* is true (the default), only the IANA types are supported; when *strict* is false, some additional non-standard but commonly used MIME types are also recognized.

**guess_all_extensions** *(type[, strict])*


Guess the extensions for a file based on its MIME type, given by *type*. The return value is a list of strings giving all possible filename extensions, including the leading dot ("."). The extensions are not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by `guess_type()`.

Optional `strict` has the same meaning as with the `guess_type()` function.

```python
guess_extension(*type*, strict)
```

Guess the extension for a file based on its MIME type, given by *type*. The return value is a string giving a filename extension, including the leading dot ("."). The extension is not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by `guess_type()`. If no extension can be guessed for *type*, `None` is returned.

Optional `strict` has the same meaning as with the `guess_type()` function.

Some additional functions and data items are available for controlling the behavior of the module.

```python
init(*files*)
```

Initialize the internal data structures. If given, *files* must be a sequence of file names which should be used to augment the default type map. If omitted, the file names to use are taken from `knownfiles`. Each file named in *files* or `knownfiles` takes precedence over those named before it. Calling `init()` repeatedly is allowed.

```python
read_mime_types(filename)
```

Load the type map given in the file *filename*, if it exists. The type map is returned as a dictionary mapping filename extensions, including the leading dot ("."), to strings of the form 'type/subtype'. If the file *filename* does not exist or cannot be read, `None` is returned.

```python
add_type(*type*, ext*, strict*)
```
Add a mapping from the mimetype `type` to the extension `ext`. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When `strict` is the mapping will added to the official MIME types, otherwise to the non-standard ones.

**inited**
Flag indicating whether or not the global data structures have been initialized. This is set to true by `init()`.

**knownfiles**
List of type map file names commonly installed. These files are typically named `mime.types` and are installed in different locations by different packages.

**suffix_map**
Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately.

**encodings_map**
Dictionary mapping filename extensions to encoding types.

**types_map**
Dictionary mapping filename extensions to MIME types.

**common_types**
Dictionary mapping filename extensions to non-standard, but commonly found MIME types.

The `MimeTypes` class may be useful for applications which may want more than one MIME-type database:

```python
class MimeTypes([filenames])
```
This class represents a MIME-types database. By default, it provides access to the same database as the rest of this module. The initial database is a
copy of that provided by the module, and may be extended by loading additional `mime.types`-style files into the database using the `read()` or `readfp()` methods. The mapping dictionaries may also be cleared before loading additional data if the default data is not desired.

The optional `filenames` parameter can be used to cause additional files to be loaded ``on top'' of the default database.

New in version 2.2.

An example usage of the module:

```python
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ...]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

---

**Subsections**

- 7.6.1 MimeTypes Objects

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.6.1 MimeTypes Objects

MimeTypes instances provide an interface which is very like that of the mimetypes module.

**suffix_map**
Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the .tgz extension is mapped to .tar.gz to allow the encoding and type to be recognized separately. This is initially a copy of the global suffix_map defined in the module.

**encodings_map**
Dictionary mapping filename extensions to encoding types. This is initially a copy of the global encodings_map defined in the module.

**types_map**
Dictionary mapping filename extensions to MIME types. This is initially a copy of the global types_map defined in the module.

**common_types**
Dictionary mapping filename extensions to non-standard, but commonly found MIME types. This is initially a copy of the global common_types defined in the module.

**guess_extension**(type[, strict])
Similar to the guess_extension() function, using the tables stored as part of the object.

**guess_type**(url[, strict])
Similar to the guess_type() function, using the tables stored as part of the object.

**read**(path)
Load MIME information from a file named path. This uses readfp() to parse the file.
readfp(file)

Load MIME type information from an open file. The file must have the format of the standard mime.types files.
7.7 MimeWriter -- Generic MIME file writer

**Deprecated since release 2.3.** The `email` package should be used in preference to the `MimeWriter` module. This module is present only to maintain backward compatibility.

This module defines the class `MimeWriter`. The `MimeWriter` class implements a basic formatter for creating MIME multi-part files. It doesn't seek around the output file nor does it use large amounts of buffer space. You must write the parts out in the order that they should occur in the final file. `MimeWriter` does buffer the headers you add, allowing you to rearrange their order.

```python
class MimeWriter(fp)
```

Return a new instance of the `MimeWriter` class. The only argument passed, `fp`, is a file object to be used for writing. Note that a `StringIO` object could also be used.

---

**Subsections**

- [7.7.1 MimeWriter Objects](#)

---

Release 2.5, documentation updated on 19th September, 2006. See [About this document...](#) for information on suggesting changes.
7.7.1 MimeWriter Objects

MimeWriter instances have the following methods:

addheader(`key`, `value`, `[prefix]`)

Add a header line to the MIME message. The `key` is the name of the header, where the `value` obviously provides the value of the header. The optional argument `prefix` determines where the header is inserted; "0" means append at the end, "1" is insert at the start. The default is to append.

flushheaders()

Causes all headers accumulated so far to be written out (and forgotten). This is useful if you don't need a body part at all, e.g. for a subpart of type `message/rfc822` that's (mis)used to store some header-like information.

startbody(`ctype`, `[plist`, `[prefix]`])

Returns a file-like object which can be used to write to the body of the message. The content-type is set to the provided `ctype`, and the optional parameter `plist` provides additional parameters for the content-type declaration. `prefix` functions as in `addheader()` except that the default is to insert at the start.

startmultipartbody(`subtype`, `[boundary`, `[plist`, `[prefix]`])

Returns a file-like object which can be used to write to the body of the message. Additionally, this method initializes the multi-part code, where `subtype` provides the multipart subtype, `boundary` may provide a user-defined boundary specification, and `plist` provides optional parameters for the subtype. `prefix` functions as in `startbody()`. Subparts should be created using `nextpart()`.

nextpart()

Returns a new instance of MimeWriter which represents an individual part in a multipart message. This may be used to write the part as well as used for creating recursively complex multipart messages. The message must first be initialized with `startmultipartbody()` before using
nextpart().

lastpart()

This is used to designate the last part of a multipart message, and should always be used when writing multipart messages.
7.8 mimify -- MIME processing of mail messages

Deprecated since release 2.3. The `email` package should be used in preference to the `mimify` module. This module is present only to maintain backward compatibility.

The `mimify` module defines two functions to convert mail messages to and from MIME format. The mail message can be either a simple message or a so-called multipart message. Each part is treated separately. Mimifying (a part of) a message entails encoding the message as quoted-printable if it contains any characters that cannot be represented using 7-bit ASCII. Unmimifying (a part of) a message entails undoing the quoted-printable encoding. Mimify and unmimify are especially useful when a message has to be edited before being sent. Typical use would be:

```
unmimify message
edit message
mimify message
send message
```

The modules defines the following user-callable functions and user-settable variables:

**mimify**( `infile`, `outfile` )

Copy the message in `infile` to `outfile`, converting parts to quoted-printable and adding MIME mail headers when necessary. `infile` and `outfile` can be file objects (actually, any object that has a `readline()` method (for `infile`) or a `write()` method (for `outfile`)) or strings naming the files. If `infile` and `outfile` are both strings, they may have the same value.

**unmimify**( `infile`, `outfile`, `decode_base64` )

Copy the message in `infile` to `outfile`, decoding all quoted-printable parts. `infile` and `outfile` can be file objects (actually, any object that has a `readline()` method (for `infile`) or a `write()` method (for `outfile`)) or strings naming the files. If `infile` and `outfile` are both strings, they may have
the same value. If the \texttt{decode_base64} argument is provided and tests true, any parts that are coded in the base64 encoding are decoded as well.

\textbf{mime\_decode\_header(} \textit{line})

Return a decoded version of the encoded header line in \textit{line}. This only supports the ISO 8859-1 charset (Latin-1).

\textbf{mime\_encode\_header(} \textit{line})

Return a MIME-encoded version of the header line in \textit{line}.

\textbf{MAXLEN}

By default, a part will be encoded as quoted-printable when it contains any non-ASCII characters (characters with the 8th bit set), or if there are any lines longer than \textbf{MAXLEN} characters (default value 200).

\textbf{CHARSET}

When not specified in the mail headers, a character set must be filled in. The string used is stored in \textbf{CHARSET}, and the default value is ISO-8859-1 (also known as Latin1 (latin-one)).

This module can also be used from the command line. Usage is as follows:

\begin{verbatim}
mimify.py -e [-l length] [infile [outfile]]
mimify.py -d [-b] [infile [outfile]]
\end{verbatim}

to encode (mimify) and decode (unmimify) respectively. \textit{inile} defaults to standard input, \textit{oufile} defaults to standard output. The same file can be specified for input and output.

If the \textbf{-l} option is given when encoding, if there are any lines longer than the specified \textit{length}, the containing part will be encoded.

If the \textbf{-b} option is given when decoding, any base64 parts will be decoded as well.

\textbf{See Also:}

Module \textbf{quopri}:

Encode and decode MIME quoted-printable files.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.9 multifile -- Support for files containing distinct parts

**Deprecated since release 2.5.** The `email` package should be used in preference to the `multifile` module. This module is present only to maintain backward compatibility.

The `MultiFile` object enables you to treat sections of a text file as file-like input objects, with '' being returned by `readline()` when a given delimiter pattern is encountered. The defaults of this class are designed to make it useful for parsing MIME multipart messages, but by subclassing it and overriding methods it can be easily adapted for more general use.

```python
class MultiFile(fp[, seekable])
    Create a multi-file. You must instantiate this class with an input object
    argument for the `MultiFile` instance to get lines from, such as a file
    object returned by `open()`.

    `MultiFile` only ever looks at the input object's `readline()`, `seek()`
    and `tell()` methods, and the latter two are only needed if you want
    random access to the individual MIME parts. To use `MultiFile` on a
    non-seekable stream object, set the optional `seekable` argument to false; this
    will prevent using the input object's `seek()` and `tell()` methods.
```

It will be useful to know that in `MultiFile`'s view of the world, text is composed of three kinds of lines: data, section-dividers, and end-markers. `MultiFile` is designed to support parsing of messages that may have multiple nested message parts, each with its own pattern for section-divider and end-marker lines.

**See Also:**

Module `email`:

Comprehensive email handling package; supersedes the `multifile`
module.

Subsections

- 7.9.1 MultiFile Objects
- 7.9.2 MultiFile Example

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.9.1 MultiFile Objects

A MultiFile instance has the following methods:

**readline**( *str*)
Read a line. If the line is data (not a section-divider or end-marker or real EOF) return it. If the line matches the most-recently-stacked boundary, return '' and set `self.last` to 1 or 0 according as the match is or is not an end-marker. If the line matches any other stacked boundary, raise an error. On encountering end-of-file on the underlying stream object, the method raises Error unless all boundaries have been popped.

**readlines**( *str*)
Return all lines remaining in this part as a list of strings.

**read**( )
Read all lines, up to the next section. Return them as a single (multiline) string. Note that this doesn't take a size argument!

**seek**( *pos*, *whence*)
Seek. Seek indices are relative to the start of the current section. The *pos* and *whence* arguments are interpreted as for a file seek.

**tell**( )
Return the file position relative to the start of the current section.

**next**( )
Skip lines to the next section (that is, read lines until a section-divider or end-marker has been consumed). Return true if there is such a section, false if an end-marker is seen. Re-enable the most-recently-pushed boundary.

**is_data**( *str*)
Return true if *str* is data and false if it might be a section boundary. As written, it tests for a prefix other than '-' at start of line (which all MIME boundaries have) but it is declared so it can be overridden in derived classes.
Note that this test is used intended as a fast guard for the real boundary tests; if it always returns false it will merely slow processing, not cause it to fail.

**push**( str)
Push a boundary string. When a decorated version of this boundary is found as an input line, it will be interpreted as a section-divider or end-marker (depending on the decoration, see [RFC 2045](https://tools.ietf.org/html/rfc2045)). All subsequent reads will return the empty string to indicate end-of-file, until a call to **pop()** removes the boundary a or **next()** call reenables it.

It is possible to push more than one boundary. Encountering the most-recently-pushed boundary will return EOF; encountering any other boundary will raise an error.

**pop()**
Pop a section boundary. This boundary will no longer be interpreted as EOF.

**section_divider**( str)
Turn a boundary into a section-divider line. By default, this method prepends ' - - ' (which MIME section boundaries have) but it is declared so it can be overridden in derived classes. This method need not append LF or CR-LF, as comparison with the result ignores trailing whitespace.

**end_marker**( str)
Turn a boundary string into an end-marker line. By default, this method prepends ' - - ' and appends ' - - ' (like a MIME-multipart end-of-message marker) but it is declared so it can be overridden in derived classes. This method need not append LF or CR-LF, as comparison with the result ignores trailing whitespace.

Finally, **MultiFile** instances have two public instance variables:

**level**
Nesting depth of the current part.

**last**
True if the last end-of-file was for an end-of-message marker.
7.9.2 MultiFile Example

```python
import mimetools
import multifile
import StringIO

def extract_mime_part_matching(stream, mimetype):
    """Return the first element in a multipart MIME message on stream
    matching mimetype."""

    msg = mimetools.Message(stream)
    msgtype = msg.gettype()
    params = msg.getplist()

    data = StringIO.StringIO()
    if msgtype[:10] == "multipart/":
        file = multifile.MultiFile(stream)
        file.push(msg.getparam("boundary"))
        while file.next():
            submsg = mimetools.Message(file)
            try:
                data = StringIO.StringIO()
                mimetools.decode(file, data, submsg.getencoding())
            except ValueError:
                continue
            if submsg.gettype() == mimetype:
                break
        file.pop()
    return data.getvalue()```

Python Library Reference

Previous: 7.9.1 MultiFile Objects  Up: 7.9 multifile  Next: 7.10 rfc822

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.10 rfc822 -- Parse RFC 2822 mail headers

 Deprecated since release 2.3. The `email` package should be used in preference to the rfc822 module. This module is present only to maintain backward compatibility.

 This module defines a class, `Message`, which represents an ``email message" as defined by the Internet standard [RFC 2822]. Such messages consist of a collection of message headers, and a message body. This module also defines a helper class `AddressList` for parsing [RFC 2822] addresses. Please refer to the RFC for information on the specific syntax of [RFC 2822] messages.

 The `mailbox` module provides classes to read mailboxes produced by various end-user mail programs.

 class `Message`(`file`, `seekable`)  

 A `Message` instance is instantiated with an input object as parameter. `Message` relies only on the input object having a `readline()` method; in particular, ordinary file objects qualify. Instantiation reads headers from the input object up to a delimiter line (normally a blank line) and stores them in the instance. The message body, following the headers, is not consumed.

 This class can work with any input object that supports a `readline()` method. If the input object has seek and tell capability, the `rewindbody()` method will work; also, illegal lines will be pushed back onto the input stream. If the input object lacks seek but has an `unread()` method that can push back a line of input, `Message` will use that to push back illegal lines. Thus this class can be used to parse messages coming from a buffered stream.

 The optional `seekable` argument is provided as a workaround for certain stdio libraries in which `tell()` discards buffered data before discovering that the `lseek()` system call doesn't work. For maximum portability, you should set the seekable argument to zero to prevent that initial `tell()`
when passing in an unseekable object such as a file object created from a socket object.

Input lines as read from the file may either be terminated by CR-LF or by a single linefeed; a terminating CR-LF is replaced by a single linefeed before the line is stored.

All header matching is done independent of upper or lower case; e.g. \['From'], \['from'] and \['FROM'] all yield the same result.

**class AddressList(field)**

You may instantiate the AddressList helper class using a single string parameter, a comma-separated list of RFC 2822 addresses to be parsed. (The parameter None yields an empty list.)

**quote(str)**

Return a new string with backslashes in str replaced by two backslashes and double quotes replaced by backslash-double quote.

**unquote(str)**

Return a new string which is an unquoted version of str. If str ends and begins with double quotes, they are stripped off. Likewise if str ends and begins with angle brackets, they are stripped off.

**parseaddr(address)**

Parse address, which should be the value of some address-containing field such as To: or Cc:, into its constituent "realname" and "email address" parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple (None, None) is returned.

**dump_address_pair(pair)**

The inverse of parseaddr(), this takes a 2-tuple of the form (realname, email_address) and returns the string value suitable for a To: or Cc: header. If the first element of pair is false, then the second element is returned unmodified.

**parsedate(date)**

Attempts to parse a date according to the rules in RFC 2822. However, some
mailers don't follow that format as specified, so `parsedate()` tries to
guess correctly in such cases. `date` is a string containing an [RFC 2822] date,
such as 'Mon, 20 Nov 1995 19:12:08 -0500'. If it succeeds in
parsing the date, `parsedate()` returns a 9-tuple that can be passed
directly to `time.mktime()`; otherwise `None` will be returned. Note that
fields 6, 7, and 8 of the result tuple are not usable.

`parsedate_tz(date)`
Performs the same function as `parsedate()`, but returns either `None` or
a 10-tuple; the first 9 elements make up a tuple that can be passed directly
to `time.mktime()`, and the tenth is the offset of the date's timezone
from UTC (which is the official term for Greenwich Mean Time). (Note
that the sign of the timezone offset is the opposite of the sign of the
time.timezone variable for the same timezone; the latter variable
follows the POSIX standard while this module follows [RFC 2822].) If the
input string has no timezone, the last element of the tuple returned is `None`. Note that fields 6, 7, and 8 of the result tuple are not usable.

`mktime_tz(tuple)`
Turn a 10-tuple as returned by `parsedate_tz()` into a UTC timestamp.
If the timezone item in the tuple is `None`, assume local time. Minor
deficiency: this first interprets the first 8 elements as a local time and then
compensates for the timezone difference; this may yield a slight error
around daylight savings time switch dates. Not enough to worry about for
common use.

See Also:

Module `email`:
  Comprehensive email handling package; supersedes the `rfc822`
  module.

Module `mailbox`:
  Classes to read various mailbox formats produced by end-user mail
  programs.

Module `mimetools`:
Subclass of \texttt{rfc822.Message} that handles MIME encoded messages.

Footnotes

...2822. 7.6
This module originally conformed to RFC 822, hence the name. Since then, RFC 2822 has been released as an update to RFC 822. This module should be considered RFC 2822-conformant, especially in cases where the syntax or semantics have changed since RFC 822.

Subsections

- 7.10.1 Message Objects
- 7.10.2 AddressList Objects

Python Library Reference

Previous: 7.9.2 MultiFile Example  Up: 7. Internet Data Handling  Next: 7.10.1 Message Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.10.1 Message Objects

A Message instance has the following methods:

**rewindbody()**

Seek to the start of the message body. This only works if the file object is seekable.

**isheader( line)**

Returns a line's canonicalized fieldname (the dictionary key that will be used to index it) if the line is a legal RFC 2822 header; otherwise returns None (implying that parsing should stop here and the line be pushed back on the input stream). It is sometimes useful to override this method in a subclass.

**islast( line)**

Return true if the given line is a delimiter on which Message should stop. The delimiter line is consumed, and the file object's read location positioned immediately after it. By default this method just checks that the line is blank, but you can override it in a subclass.

**iscomment( line)**

Return True if the given line should be ignored entirely, just skipped. By default this is a stub that always returns False, but you can override it in a subclass.

**getallmatchingheaders( name)**

Return a list of lines consisting of all headers matching name, if any. Each physical line, whether it is a continuation line or not, is a separate list item. Return the empty list if no header matches name.

**getfirstmatchingheader( name)**

Return a list of lines comprising the first header matching name, and its continuation line(s), if any. Return None if there is no header matching name.
getrawheader(name)
Return a single string consisting of the text after the colon in the first header matching name. This includes leading whitespace, the trailing linefeed, and internal linefeeds and whitespace if there any continuation line(s) were present. Return None if there is no header matching name.

getheader(name[, default])
Like getrawheader(name), but strip leading and trailing whitespace. Internal whitespace is not stripped. The optional default argument can be used to specify a different default to be returned when there is no header matching name.

get(name[, default])
An alias for getheader(), to make the interface more compatible with regular dictionaries.

getaddr(name)
Return a pair (full name, email address) parsed from the string returned by getheader(name). If no header matching name exists, return (None, None); otherwise both the full name and the address are (possibly empty) strings.

Example: If m's first From: header contains the string 'jack@cwi.nl (Jack Jansen)', then m.getaddr('From') will yield the pair ('Jack Jansen', 'jack@cwi.nl'). If the header contained 'Jack Jansen <jack@cwi.nl>' instead, it would yield the exact same result.

getaddrlist(name)
This is similar to getaddr(list), but parses a header containing a list of email addresses (e.g. a To: header) and returns a list of (full name, email address) pairs (even if there was only one address in the header). If there is no header matching name, return an empty list.

If multiple headers exist that match the named header (e.g. if there are several Cc: headers), all are parsed for addresses. Any continuation lines the named headers contain are also parsed.
getdate(name)
Retrieve a header using getheader() and parse it into a 9-tuple compatible with time.mktime(); note that fields 6, 7, and 8 are not usable. If there is no header matching name, or it is unparsable, return None.

Date parsing appears to be a black art, and not all mailers adhere to the standard. While it has been tested and found correct on a large collection of email from many sources, it is still possible that this function may occasionally yield an incorrect result.

getdate_tz(name)
Retrieve a header using getheader() and parse it into a 10-tuple; the first 9 elements will make a tuple compatible with time.mktime(), and the 10th is a number giving the offset of the date's timezone from UTC. Note that fields 6, 7, and 8 are not usable. Similarly to getdate(), if there is no header matching name, or it is unparsable, return None.

Message instances also support a limited mapping interface. In particular: m[name] is like m.getheader(name) but raises KeyError if there is no matching header; and len(m), m.get(name[, default]), m.has_key(name), m.keys(), m.values() m.items(), and m.setdefault(name[, default]) act as expected, with the one difference that setdefault() uses an empty string as the default value. Message instances also support the mapping writable interface m[name] = value and del m[name]. Message objects do not support the clear(), copy(), popitem(), or update() methods of the mapping interface. (Support for get() and setdefault() was only added in Python 2.2.)

Finally, Message instances have some public instance variables:

headers
A list containing the entire set of header lines, in the order in which they were read (except that setitem calls may disturb this order). Each line contains a trailing newline. The blank line terminating the headers is not contained in the list.

fp
The file or file-like object passed at instantiation time. This can be used to read the message content.

**unixfrom**

The UNIX "From " line, if the message had one, or an empty string. This is needed to regenerate the message in some contexts, such as an mbox-style mailbox file.
### 7.10.2 AddressList Objects

An `AddressList` instance has the following methods:

- **`__len__()`**
  Return the number of addresses in the address list.

- **`__str__()`**
  Return a canonicalized string representation of the address list. Addresses are rendered in "name" `<host@domain>` form, comma-separated.

- **`__add__(alist)`**
  Return a new `AddressList` instance that contains all addresses in both `AddressList` operands, with duplicates removed (set union).

- **`__iadd__(alist)`**
  In-place version of `__add__()`; turns this `AddressList` instance into the union of itself and the right-hand instance, `alist`.

- **`__sub__(alist)`**
  Return a new `AddressList` instance that contains every address in the left-hand `AddressList` operand that is not present in the right-hand address operand (set difference).

- **`__isub__(alist)`**
  In-place version of `__sub__()`, removing addresses in this list which are also in `alist`.

Finally, `AddressList` instances have one public instance variable:

- **`addresslist`**
  A list of tuple string pairs, one per address. In each member, the first is the canonicalized name part, the second is the actual route-address ("@"-separated username-host.domain pair).
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.11 base64 -- RFC 3548: Base16, Base32, Base64 Data Encodings

This module provides data encoding and decoding as specified in RFC 3548. This standard defines the Base16, Base32, and Base64 algorithms for encoding and decoding arbitrary binary strings into text strings that can be safely sent by email, used as parts of URLs, or included as part of an HTTP POST request. The encoding algorithm is not the same as the uuencode program.

There are two interfaces provided by this module. The modern interface supports encoding and decoding string objects using all three alphabets. The legacy interface provides for encoding and decoding to and from file-like objects as well as strings, but only using the Base64 standard alphabet.

The modern interface provides:

`b64encode(s[, altchars])`
Encode a string use Base64.

`s` is the string to encode. Optional `altchars` must be a string of at least length 2 (additional characters are ignored) which specifies an alternative alphabet for the + and / characters. This allows an application to e.g. generate URL or filesystem safe Base64 strings. The default is `None`, for which the standard Base64 alphabet is used.

The encoded string is returned.

`b64decode(s[, altchars])`
Decode a Base64 encoded string.

`s` is the string to decode. Optional `altchars` must be a string of at least length 2 (additional characters are ignored) which specifies the alternative alphabet used instead of the + and / characters.

The decoded string is returned. A `TypeError` is raised if `s` were incorrectly padded or if there are non-alphabet characters present in the
string.

**standard_b64encode(s)**
Encode string s using the standard Base64 alphabet.

**standard_b64decode(s)**
Decode string s using the standard Base64 alphabet.

**urlsafe_b64encode(s)**
Encode string s using a URL-safe alphabet, which substitutes - instead of + and _ instead of / in the standard Base64 alphabet.

**urlsafe_b64decode(s)**
Decode string s using a URL-safe alphabet, which substitutes - instead of + and _ instead of / in the standard Base64 alphabet.

**b32encode(s)**
Encode a string using Base32. s is the string to encode. The encoded string is returned.

**b32decode(s[, casefold[, map01]])**
Decode a Base32 encoded string.

s is the string to decode. Optional **casefold** is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is **False**.

**RFC 3548** allows for optional mapping of the digit 0 (zero) to the letter O (oh), and for optional mapping of the digit 1 (one) to either the letter I (eye) or letter L (el). The optional argument **map01** when not **None**, specifies which letter the digit 1 should be mapped to (when map01 is not **None**, the digit 0 is always mapped to the letter O). For security purposes the default is **None**, so that 0 and 1 are not allowed in the input.

The decoded string is returned. A **TypeError** is raised if s were incorrectly padded or if there are non-alphabet characters present in the string.
**b16encode**(s)

Encode a string using Base16.

s is the string to encode. The encoded string is returned.

**b16decode**(s[, *casefold*])

Decode a Base16 encoded string.

s is the string to decode. Optional *casefold* is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is *False*.

The decoded string is returned. A *TypeError* is raised if s were incorrectly padded or if there are non-alphabet characters present in the string.

The legacy interface:

**decode**(input, output)

Decode the contents of the *input* file and write the resulting binary data to the *output* file. *input* and *output* must either be file objects or objects that mimic the file object interface. *input* will be read until *input.read()* returns an empty string.

**decodestring**(s)

Decode the string s, which must contain one or more lines of base64 encoded data, and return a string containing the resulting binary data.

**encode**(input, output)

Encode the contents of the *input* file and write the resulting base64 encoded data to the *output* file. *input* and *output* must either be file objects or objects that mimic the file object interface. *input* will be read until *input.read()* returns an empty string. *encode()* returns the encoded data plus a trailing newline character ('\n').

**encodestring**(s)

Encode the string s, which can contain arbitrary binary data, and return a string containing one or more lines of base64-encoded data.
encodestring() returns a string containing one or more lines of base64-encoded data always including an extra trailing newline ("\n").

An example usage of the module:

```python
>>> import base64
>>> encoded = base64.b64encode('data to be encoded')
>>> encoded
'ZGF0YSB0byBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
'data to be encoded'
```

See Also:

Module **binascii**: Support module containing ASCII-to-binary and binary-to-ASCII conversions.

**RFC 1521, MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies**

Section 5.2, ``Base64 Content-Transfer-Encoding," provides the definition of the base64 encoding.

---

*Python Library Reference*

Previous: 7.10.2 AddressList Objects  Up: 7. Internet Data Handling  Next: 7.12 binhex

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.12 binhex -- Encode and decode binhex4 files

This module encodes and decodes files in binhex4 format, a format allowing representation of Macintosh files in ASCII. On the Macintosh, both forks of a file and the finder information are encoded (or decoded), on other platforms only the data fork is handled.

The binhex module defines the following functions:

**binhex**( input, output)

Convert a binary file with filename input to binhex file output. The output parameter can either be a filename or a file-like object (any object supporting a write() and close() method).

**hexbin**( input[, output])

Decode a binhex file input. input may be a filename or a file-like object supporting read() and close() methods. The resulting file is written to a file named output, unless the argument is omitted in which case the output filename is read from the binhex file.

The following exception is also defined:

**exception Error**

Exception raised when something can't be encoded using the binhex format (for example, a filename is too long to fit in the filename field), or when input is not properly encoded binhex data.

**See Also:**

Module [binascii]:

Support module containing ASCII-to-binary and binary-to-ASCII conversions.
Release 2.5, documentation updated on 19th September, 2006.

See About this document... for information on suggesting changes.
7.12.1 Notes

There is an alternative, more powerful interface to the coder and decoder, see the source for details.

If you code or decode textfiles on non-Macintosh platforms they will still use the Macintosh newline convention (carriage-return as end of line).

As of this writing, `hexbin()` appears to not work in all cases.
7.13 binascii -- Convert between binary and ASCII

The binascii module contains a number of methods to convert between binary and various ASCII-encoded binary representations. Normally, you will not use these functions directly but use wrapper modules like uu, base64, or binhex instead. The binascii module contains low-level functions written in C for greater speed that are used by the higher-level modules.

The binascii module defines the following functions:

**a2b_uu**( *string*)
Convert a single line of uuencoded data back to binary and return the binary data. Lines normally contain 45 (binary) bytes, except for the last line. Line data may be followed by whitespace.

**b2a_uu**( *data*)
Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of *data* should be at most 45.

**a2b_base64**( *string*)
Convert a block of base64 data back to binary and return the binary data. More than one line may be passed at a time.

**b2a_base64**( *data*)
Convert binary data to a line of ASCII characters in base64 coding. The return value is the converted line, including a newline char. The length of *data* should be at most 57 to adhere to the base64 standard.

**a2b_qp**( *string[, header]*)
Convert a block of quoted-printable data back to binary and return the binary data. More than one line may be passed at a time. If the optional argument *header* is present and true, underscores will be decoded as spaces.
**b2a_qp**(*data[, quotetabs, istext, header]*)

Convert binary data to a line(s) of ASCII characters in quoted-printable encoding. The return value is the converted line(s). If the optional argument *quotetabs* is present and true, all tabs and spaces will be encoded. If the optional argument *istext* is present and true, newlines are not encoded but trailing whitespace will be encoded. If the optional argument *header* is present and true, spaces will be encoded as underscores per RFC1522. If the optional argument *header* is present and false, newline characters will be encoded as well; otherwise linefeed conversion might corrupt the binary data stream.

**a2b_hqx**(*string*)

Convert binhex4 formatted ASCII data to binary, without doing RLE-decompression. The string should contain a complete number of binary bytes, or (in case of the last portion of the binhex4 data) have the remaining bits zero.

**rledecode_hqx**(*data*)

Perform RLE-decompression on the data, as per the binhex4 standard. The algorithm uses 0x90 after a byte as a repeat indicator, followed by a count. A count of 0 specifies a byte value of 0x90. The routine returns the decompressed data, unless data input data ends in an orphaned repeat indicator, in which case the Incomplete exception is raised.

**rlecode_hqx**(*data*)

Perform binhex4 style RLE-compression on *data* and return the result.

**b2a_hqx**(*data*)

Perform hexbin4 binary-to-ASCII translation and return the resulting string. The argument should already be RLE-coded, and have a length divisible by 3 (except possibly the last fragment).

**crc_hqx**(*data, crc*)

Compute the binhex4 crc value of *data*, starting with an initial *crc* and returning the result.

**crc32**(*data[, crc]*)
Compute CRC-32, the 32-bit checksum of data, starting with an initial crc. This is consistent with the ZIP file checksum. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm. Use as follows:

```
print binascii.crc32("hello world")  
# Or, in two pieces:  
crc = binascii.crc32("hello")  
crc = binascii.crc32(" world", crc)  
print crc
```

**b2a_hex** *(data)*

**hexlify** *(data)*

Return the hexadecimal representation of the binary *data*. Every byte of *data* is converted into the corresponding 2-digit hex representation. The resulting string is therefore twice as long as the length of *data*.

**a2b_hex** *(hexstr)*

**unhexlify** *(hexstr)*

Return the binary data represented by the hexadecimal string *hexstr*. This function is the inverse of **b2a_hex**(). *hexstr* must contain an even number of hexadecimal digits (which can be upper or lower case), otherwise a **TypeError** is raised.

**exception Error**

Exception raised on errors. These are usually programming errors.

**exception Incomplete**

Exception raised on incomplete data. These are usually not programming errors, but may be handled by reading a little more data and trying again.

**See Also:**

Module **base64**:

Support for base64 encoding used in MIME email messages.

Module **binhex**:

Support for the binhex format used on the Macintosh.
Module **uu**: Support for UU encoding used on UNIX.

Module **quopri**: Support for quoted-printable encoding used in MIME email messages.
7.14 quopri -- Encode and decode MIME quoted-printable data

This module performs quoted-printable transport encoding and decoding, as defined in RFC 1521: ``MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies''. The quoted-printable encoding is designed for data where there are relatively few nonprintable characters; the base64 encoding scheme available via the base64 module is more compact if there are many such characters, as when sending a graphics file.

**decode**( input, output[,header])

Decode the contents of the input file and write the resulting decoded binary data to the output file. input and output must either be file objects or objects that mimic the file object interface. input will be read until input.readline() returns an empty string. If the optional argument header is present and true, underscore will be decoded as space. This is used to decode ``Q''-encoded headers as described in RFC 1522: ``MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text''.

**encode**( input, output, quotetabs)

Encode the contents of the input file and write the resulting quoted-printable data to the output file. input and output must either be file objects or objects that mimic the file object interface. input will be read until input.readline() returns an empty string. quotetabs is a flag which controls whether to encode embedded spaces and tabs; when true it encodes such embedded whitespace, and when false it leaves them unencoded. Note that spaces and tabs appearing at the end of lines are always encoded, as per RFC 1521.

**decodestring**( s[,header])

Like decode(), except that it accepts a source string and returns the corresponding decoded string.
\texttt{encodestring(s[, quotetabs])}

Like \texttt{encode()}, except that it accepts a source string and returns the corresponding encoded string. \texttt{quotetabs} is optional (defaulting to 0), and is passed straight through to \texttt{encode()}.

\textbf{See Also:}

Module \texttt{mimify}:
General utilities for processing of MIME messages.

Module \texttt{base64}:
Encode and decode MIME base64 data.
7.15 uu -- Encode and decode uuencode files

This module encodes and decodes files in uuencode format, allowing arbitrary binary data to be transferred over ASCII-only connections. Wherever a file argument is expected, the methods accept a file-like object. For backwards compatibility, a string containing a pathname is also accepted, and the corresponding file will be opened for reading and writing; the pathname ' - ' is understood to mean the standard input or output. However, this interface is deprecated; it's better for the caller to open the file itself, and be sure that, when required, the mode is ' rb ' or ' wb ' on Windows.

This code was contributed by Lance Ellinghouse, and modified by Jack Jansen.

The uu module defines the following functions:

**encode( in_file, out_file[, name[, mode]])**

Uuencode file in_file into file out_file. The uuencoded file will have the header specifying name and mode as the defaults for the results of decoding the file. The default defaults are taken from in_file, or ' - ' and 0666 respectively.

**decode( in_file[, out_file[, mode[, quiet]]])**

This call decodes uuencoded file in_file placing the result on file out_file. If out_file is a pathname, mode is used to set the permission bits if the file must be created. Defaults for out_file and mode are taken from the uuencode header. However, if the file specified in the header already exists, a uu.Error is raised.

decode() may print a warning to standard error if the input was produced by an incorrect uuencoder and Python could recover from that error. Setting quiet to a true value silences this warning.

**exception Error()**

Subclass of Exception, this can be raised by uu.decode() under
various situations, such as described above, but also including a badly formatted header, or truncated input file.

See Also:

Module **binascii**: Support module containing ASCII-to-binary and binary-to-ASCII conversions.
8. Structured Markup Processing Tools

Python supports a variety of modules to work with various forms of structured data markup. This includes modules to work with the Standard Generalized Markup Language (SGML) and the Hypertext Markup Language (HTML), and several interfaces for working with the Extensible Markup Language (XML).

It is important to note that modules in the xml package require that there be at least one SAX-compliant XML parser available. Starting with Python 2.3, the Expat parser is included with Python, so the xml.parsers.expat module will always be available. You may still want to be aware of the PyXML add-on package; that package provides an extended set of XML libraries for Python.

The documentation for the xml.dom and xml.sax packages are the definition of the Python bindings for the DOM and SAX interfaces.

**HTMLParser**
A simple parser that can handle HTML and XHTML.

**sgmlib**
Only as much of an SGML parser as needed to parse HTML.

**htmlib**
A parser for HTML documents.

**htmlentitydefs**
Definitions of HTML general entities.

**xml.parsers.expat**
An interface to the Expat non-validating XML parser.

**xml.dom**
Document Object Model API for Python.

**xml.dom.minidom**
Lightweight Document Object Model (DOM) implementation.

**xml.dom.pulldom**
Support for building partial DOM trees from SAX events.

**xml.sax**
Package containing SAX2 base classes and convenience functions.

**xml.sax.handler**
Base classes for SAX event handlers.
Convenience functions and classes for use with
xml.sax.saxutils  
SAX.

xml.sax.xmlreader  
Interface which SAX-compliant XML parsers must implement.

xml.etree.ElementTree  
Implementation of the ElementTree API.

See Also:

Python/XML Libraries  
Home page for the PyXML package, containing an extension of xml package bundled with Python.
This module defines a class **HTMLParser** which serves as the basis for parsing text files formatted in HTML (HyperText Mark-up Language) and XHTML. Unlike the parser in **htmllib**, this parser is not based on the SGML parser in **sgmlib**.

**class HTMLParser()**

The **HTMLParser** class is instantiated without arguments.

An **HTMLParser** instance is fed HTML data and calls handler functions when tags begin and end. The **HTMLParser** class is meant to be overridden by the user to provide a desired behavior.

Unlike the parser in **htmllib**, this parser does not check that end tags match start tags or call the end-tag handler for elements which are closed implicitly by closing an outer element.

An exception is defined as well:

**exception HTMLParserError**

Exception raised by the **HTMLParser** class when it encounters an error while parsing. This exception provides three attributes: **msg** is a brief message explaining the error, **lineno** is the number of the line on which the broken construct was detected, and **offset** is the number of characters into the line at which the construct starts.

**HTMLParser** instances have the following methods:

**reset()**

Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.
feed( data)
    Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or close() is called.

close()
    Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call the HTMLParser base class method close().

getpos()
    Return current line number and offset.

get_starttag_text()
    Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML "as deployed" or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

handle_starttag( tag, attrs)
    This method is called to handle the start of a tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

    The tag argument is the name of the tag converted to lower case. The attrs argument is a list of (name, value) pairs containing the attributes found inside the tag's <> brackets. The name will be translated to lower case and double quotes and backslashes in the value have been interpreted. For instance, for the tag <A HREF="http://www.cwi.nl/">, this method would be called as "handle_starttag('a', [('href', 'http://www.cwi.nl/')])."

handle_startendtag( tag, attrs)
    Similar to handle_starttag(), but called when the parser encounters an XHTML-style empty tag (<a .../>). This method may be overridden by subclasses which require this particular lexical information; the default implementation simple calls handle_starttag() and
handle_endtag().

**handle_endtag(tag)**
This method is called to handle the end tag of an element. It is intended to be overridden by a derived class; the base class implementation does nothing. The tag argument is the name of the tag converted to lower case.

**handle_data(data)**
This method is called to process arbitrary data. It is intended to be overridden by a derived class; the base class implementation does nothing.

**handle_charref(name)**
This method is called to process a character reference of the form "&#ref;". It is intended to be overridden by a derived class; the base class implementation does nothing.

**handle_entityref(name)**
This method is called to process a general entity reference of the form "&name;" where name is an general entity reference. It is intended to be overridden by a derived class; the base class implementation does nothing.

**handle_comment(data)**
This method is called when a comment is encountered. The comment argument is a string containing the text between the "- -" and "- -" delimiters, but not the delimiters themselves. For example, the comment "<!--text-->" will cause this method to be called with the argument 'text'. It is intended to be overridden by a derived class; the base class implementation does nothing.

**handle_decl(decl)**
Method called when an SGML declaration is read by the parser. The decl parameter will be the entire contents of the declaration inside the <!...> markup. It is intended to be overridden by a derived class; the base class implementation does nothing.

**handle_pi(data)**
Method called when a processing instruction is encountered. The data
parameter will contain the entire processing instruction. For example, for the processing instruction `<?proc color='red'>`, this method would be called as `handle_pi("proc color='red'")`. It is intended to be overridden by a derived class; the base class implementation does nothing.

**Note:** The `HTMLParser` class uses the SGML syntactic rules for processing instructions. An XHTML processing instruction using the trailing "?" will cause the "?" to be included in `data`.

### Subsections

- [8.1.1 Example HTML Parser Application](#)
8.1.1 Example HTML Parser Application

As a basic example, below is a very basic HTML parser that uses the HTMLParser class to print out tags as they are encountered:

```python
from HTMLParser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print "Encountered the beginning of a %s tag" % tag

    def handle_endtag(self, tag):
        print "Encountered the end of a %s tag" % tag
```

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
8.2 sgmllib -- Simple SGML parser

This module defines a class *SGMLParser* which serves as the basis for parsing text files formatted in SGML (Standard Generalized Mark-up Language). In fact, it does not provide a full SGML parser -- it only parses SGML insofar as it is used by HTML, and the module only exists as a base for the *htmllib* module. Another HTML parser which supports XHTML and offers a somewhat different interface is available in the *HTMLParser* module.

**class SGMLParser()**

The *SGMLParser* class is instantiated without arguments. The parser is hardcoded to recognize the following constructs:

- Opening and closing tags of the form "<tag attr="value" ...>" and "</tag>", respectively.
- Numeric character references of the form "&#name;".
- Entity references of the form "&name;".
- SGML comments of the form "<! - - text - - >". Note that spaces, tabs, and newlines are allowed between the trailing ">" and the immediately preceding " - - ".

A single exception is defined as well:

**exception SGMLParseError**

Exception raised by the *SGMLParser* class when it encounters an error while parsing. New in version 2.1.

*SGMLParser* instances have the following methods:

**reset()**

Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.
setnomoretags()
Stop processing tags. Treat all following input as literal input (CDATA).
(This is only provided so the HTML tag `<PLAINTEXT>` can be implemented.)

setliteral()
Enter literal mode (CDATA mode).

feed( data)
Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or close() is called.

close()
Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call close().

get_starttag_text()
Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML "as deployed" or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

handle_starttag( tag, method, attributes)
This method is called to handle start tags for which either a start_tag() or do_tag() method has been defined. The tag argument is the name of the tag converted to lower case, and the method argument is the bound method which should be used to support semantic interpretation of the start tag. The attributes argument is a list of (name, value) pairs containing the attributes found inside the tag's <> brackets.

The name has been translated to lower case. Double quotes and backslashes in the value have been interpreted, as well as known character references and known entity references terminated by a semicolon (normally, entity references can be terminated by any non-alphanumerical character, but this
would break the very common case of `<A HREF="url?spam=1&eggs=2">` when `eggs` is a valid entity name).

For instance, for the tag `<A HREF="http://www.cwi.nl/">`, this method would be called as "unknown_starttag('a', [('href', 'http://www.cwi.nl/')])". The base implementation simply calls `method` with `attributes` as the only argument. New in version 2.5: Handling of entity and character references within attribute values.

**handle_endtag** *(tag, method)*

This method is called to handle endtags for which an `end_tag()` method has been defined. The `tag` argument is the name of the tag converted to lower case, and the `method` argument is the bound method which should be used to support semantic interpretation of the end tag. If no `end_tag()` method is defined for the closing element, this handler is not called. The base implementation simply calls `method`.

**handle_data** *(data)*

This method is called to process arbitrary data. It is intended to be overridden by a derived class; the base class implementation does nothing.

**handle_charref** *(ref)*

This method is called to process a character reference of the form "&#ref;". The base implementation uses `convert_charref()` to convert the reference to a string. If that method returns a string, it is passed to `handle_data()`, otherwise `unknown_charref(ref)` is called to handle the error. Changed in version 2.5: Use `convert_charref()` instead of hard-coding the conversion.

**convert_charref** *(ref)*

Convert a character reference to a string, or `None`. `ref` is the reference passed in as a string. In the base implementation, `ref` must be a decimal number in the range 0-255. It converts the code point found using the `convert_codepoint()` method. If `ref` is invalid or out of range, this method returns `None`. This method is called by the default `handle_charref()` implementation and by the attribute value parser. New in version 2.5.
**convert_codepoint** *(codepoint)*

Convert a codepoint to a str value. Encodings can be handled here if appropriate, though the rest of sgmllib is oblivious on this matter. New in version 2.5.

**handle_entityref** *(ref)*

This method is called to process a general entity reference of the form "&ref;" where *ref* is an general entity reference. It converts *ref* by passing it to **convert_entityref**(). If a translation is returned, it calls the method **handle_data**() with the translation; otherwise, it calls the method **unknown_entityref**(*ref*). The default **entitydefs** defines translations for &amp;, &apos, &gt;, &lt;, and &quot;. Changed in version 2.5: Use **convert_entityref**() instead of hard-coding the conversion.

**convert_entityref** *(ref)*

Convert a named entity reference to a str value, or None. The resulting value will not be parsed. *ref* will be only the name of the entity. The default implementation looks for *ref* in the instance (or class) variable **entitydefs** which should be a mapping from entity names to corresponding translations. If no translation is available for *ref*, this method returns None. This method is called by the default **handle_entityref**() implementation and by the attribute value parser. New in version 2.5.

**handle_comment** *(comment)*

This method is called when a comment is encountered. The *comment* argument is a string containing the text between the "<!--" and "-->" delimiters, but not the delimiters themselves. For example, the comment "<!--text-->" will cause this method to be called with the argument 'text'. The default method does nothing.

**handle_decl** *(data)*

Method called when an SGML declaration is read by the parser. In practice, the **DOCTYPE** declaration is the only thing observed in HTML, but the parser does not discriminate among different (or broken) declarations. Internal subsets in a **DOCTYPE** declaration are not supported. The *data*
The parameter will be the entire contents of the declaration inside the `<!...>` markup. The default implementation does nothing.

**report_unbalanced**(*tag*)
This method is called when an end tag is found which does not correspond to any open element.

**unknown_starttag**(*tag*, *attributes*)
This method is called to process an unknown start tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

**unknown_endtag**(*tag*)
This method is called to process an unknown end tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

**unknown_charref**(*ref*)
This method is called to process unresolvable numeric character references. Refer to **handle_charref()** to determine what is handled by default. It is intended to be overridden by a derived class; the base class implementation does nothing.

**unknown_entityref**(*ref*)
This method is called to process an unknown entity reference. It is intended to be overridden by a derived class; the base class implementation does nothing.

Apart from overriding or extending the methods listed above, derived classes may also define methods of the following form to define processing of specific tags. Tag names in the input stream are case independent; the *tag* occurring in method names must be in lower case:

**start_tag**(*attributes*)
This method is called to process an opening tag *tag*. It has preference over **do_tag()**. The *attributes* argument has the same meaning as described for **handle_starttag()** above.

**do_tag**(*attributes*)
This method is called to process an opening tag tag for which no
\texttt{start\_tag} method is defined. The attributes argument has the same
meaning as described for \texttt{handle\_start\_tag()} above.

\texttt{end\_tag()}  
This method is called to process a closing tag tag.

Note that the parser maintains a stack of open elements for which no end tag has
been found yet. Only tags processed by \texttt{start\_tag()} are pushed on this stack.
Definition of an \texttt{end\_tag()} method is optional for these tags. For tags
processed by \texttt{do\_tag()} or by \texttt{unknown\_tag()}, no \texttt{end\_tag()} method must
be defined; if defined, it will not be used. If both \texttt{start\_tag()} and \texttt{do\_tag()} methods exist for a tag, the \texttt{start\_tag()} method takes precedence.
8.2 sgmllib Up: 8. Structured Markup Processing Next: 8.3.1 HTMLParser Objects
8.3 htmllib -- A parser for HTML documents

This module defines a class which can serve as a base for parsing text files formatted in the HyperText Mark-up Language (HTML). The class is not directly concerned with I/O -- it must be provided with input in string form via a method, and makes calls to methods of a "formatter" object in order to produce output. The HTMLParser class is designed to be used as a base class for other classes in order to add functionality, and allows most of its methods to be extended or overridden. In turn, this class is derived from and extends the SGMLParser class defined in module sgmllib. The HTMLParser implementation supports the HTML 2.0 language as described in RFC 1866. Two implementations of formatter objects are provided in the formatter module; refer to the documentation for that module for information on the formatter interface.

The following is a summary of the interface defined by sgmllib.SGMLParser:

- The interface to feed data to an instance is through the feed() method, which takes a string argument. This can be called with as little or as much text at a time as desired; "p.feed(a); p.feed(b)" has the same effect as "p.feed(a+b)". When the data contains complete HTML markup constructs, these are processed immediately; incomplete constructs are saved in a buffer. To force processing of all unprocessed data, call the close() method.

For example, to parse the entire contents of a file, use:

```python
    parser.feed(open('myfile.html').read())
    parser.close()
```

- The interface to define semantics for HTML tags is very simple: derive a class and define methods called start_tag(), end_tag(), or do_tag(). The parser will call these at appropriate moments: start_tag or do_tag() is called when an opening tag of the form <tag ...> is
encountered; `end_tag()` is called when a closing tag of the form `<tag>` is encountered. If an opening tag requires a corresponding closing tag, like `<H1> ... </H1>`, the class should define the `start_tag()` method; if a tag requires no closing tag, like `<P>`, the class should define the `do_tag()` method.

The module defines a parser class and an exception:

**class HTMLParser**( `formatter`)  
This is the basic HTML parser class. It supports all entity names required by the XHTML 1.0 Recommendation ([http://www.w3.org/TR/xhtml1](http://www.w3.org/TR/xhtml1)). It also defines handlers for all HTML 2.0 and many HTML 3.0 and 3.2 elements.

**exception HTMLParseError**  
Exception raised by the HTMLParser class when it encounters an error while parsing. New in version 2.4.

See Also:

Module **formatter**:  
Interface definition for transforming an abstract flow of formatting events into specific output events on writer objects.

Module **HTMLParser**:  
Alternate HTML parser that offers a slightly lower-level view of the input, but is designed to work with XHTML, and does not implement some of the SGML syntax not used in ``HTML as deployed" and which isn't legal for XHTML.

Module **htmlentitydefs**:  
Definition of replacement text for XHTML 1.0 entities.

Module **sgmllib**:  
Base class for HTMLParser.
Subsections

- 8.3.1 HTMLParser Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
8.3.1 HTMLParser Objects

In addition to tag methods, the HTMLParser class provides some additional methods and instance variables for use within tag methods.

formatter
This is the formatter instance associated with the parser.

nofill
Boolean flag which should be true when whitespace should not be collapsed, or false when it should be. In general, this should only be true when character data is to be treated as "preformatted" text, as within a <PRE> element. The default value is false. This affects the operation of handle_data() and save_end().

anchor_bgn( href, name, type)
This method is called at the start of an anchor region. The arguments correspond to the attributes of the <A> tag with the same names. The default implementation maintains a list of hyperlinks (defined by the HREF attribute for <A> tags) within the document. The list of hyperlinks is available as the data attribute anchorlist.

anchor_end()
This method is called at the end of an anchor region. The default implementation adds a textual footnote marker using an index into the list of hyperlinks created by anchor_bgn().

handle_image( source, alt[, ismap[, align[, width[, height]]]])
This method is called to handle images. The default implementation simply passes the alt value to the handle_data() method.

save_bgn()
Begin saving character data in a buffer instead of sending it to the formatter object. Retrieve the stored data via save_end(). Use of the save_bgn() / save_end() pair may not be nested.
save_end()

Ends buffering character data and returns all data saved since the preceding call to save_bgn(). If the nofill flag is false, whitespace is collapsed to single spaces. A call to this method without a preceding call to save_bgn() will raise a TypeError exception.
8.4 htmlentitydefs -- Definitions of HTML general entities

This module defines three dictionaries, name2codepoint, codepoint2name, and entitydefs. entitydefs is used by the htmllib module to provide the entitydefs member of the HTMLParser class. The definition provided here contains all the entities defined by XHTML 1.0 that can be handled using simple textual substitution in the Latin-1 character set (ISO-8859-1).

entitydefs
   A dictionary mapping XHTML 1.0 entity definitions to their replacement text in ISO Latin-1.

name2codepoint
   A dictionary that maps HTML entity names to the Unicode codepoints. New in version 2.3.

codepoint2name
   A dictionary that maps Unicode codepoints to HTML entity names. New in version 2.3.
8.5 xml.parsers.expat -- Fast XML parsing using Expat

New in version 2.0.

The xml.parsers.expat module is a Python interface to the Expat non-validating XML parser. The module provides a single extension type, xmlparser, that represents the current state of an XML parser. After an xmlparser object has been created, various attributes of the object can be set to handler functions. When an XML document is then fed to the parser, the handler functions are called for the character data and markup in the XML document.

This module uses the pyexpat module to provide access to the Expat parser. Direct use of the pyexpat module is deprecated.

This module provides one exception and one type object:

**exception ExpatError**

The exception raised when Expat reports an error. See section 8.5.2, ```ExpatError Exceptions,`` for more information on interpreting Expat errors.

**exception error**

Alias for ExpatError.

**XMLParserType**

The type of the return values from the ParserCreate() function.

The xml.parsers.expat module contains two functions:

**ErrorString( errno)**

Returns an explanatory string for a given error number errno.

**ParserCreate( [encoding[, namespace_separator]])**
Creates and returns a new `xmlparser` object. *encoding*, if specified, must be a string naming the encoding used by the XML data. Expat doesn't support as many encodings as Python does, and its repertoire of encodings can't be extended; it supports UTF-8, UTF-16, ISO-8859-1 (Latin1), and ASCII. If *encoding* is given it will override the implicit or explicit encoding of the document.

Expat can optionally do XML namespace processing for you, enabled by providing a value for *namespace_separator*. The value must be a one-character string; a `ValueError` will be raised if the string has an illegal length (*None* is considered the same as omission). When namespace processing is enabled, element type names and attribute names that belong to a namespace will be expanded. The element name passed to the element handlers `StartElementHandler` and `EndElementHandler` will be the concatenation of the namespace URI, the namespace separator character, and the local part of the name. If the namespace separator is a zero byte (`chr(0)`)) then the namespace URI and the local part will be concatenated without any separator.

For example, if *namespace_separator* is set to a space character (" ") and the following document is parsed:

```
<?xml version="1.0"?>
<root xmlns = "http://default-namespace.org/
         xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

`StartElementHandler` will receive the following strings for each element:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

**See Also:**

*The Expat XML Parser*

Home page of the Expat project.
ExpatError Exceptions
8.5.1 XMLParser Objects

xmlparser objects have the following methods:

**Parse**(*data*, *isfinal*)

Parses the contents of the string *data*, calling the appropriate handler functions to process the parsed data. *isfinal* must be true on the final call to this method. *data* can be the empty string at any time.

**ParseFile**(*file*)

Parse XML data reading from the object *file*. *file* only needs to provide the read(*nbytes*) method, returning the empty string when there's no more data.

**SetBase**(*base*)

Sets the base to be used for resolving relative URIs in system identifiers in declarations. Resolving relative identifiers is left to the application: this value will be passed through as the *base* argument to the ExternalEntityRefHandler, NotationDeclHandler, and UnparsedEntityDeclHandler functions.

**GetBase**()

Returns a string containing the base set by a previous call to SetBase(), or None if SetBase() hasn't been called.

**GetInputContext**()

Returns the input data that generated the current event as a string. The data is in the encoding of the entity which contains the text. When called while an event handler is not active, the return value is None. New in version 2.1.

**ExternalEntityParserCreate**(*context*, *encoding*)

Create a "child" parser which can be used to parse an external parsed entity referred to by content parsed by the parent parser. The *context* parameter should be the string passed to the ExternalEntityRefHandler() handler function, described below. The child parser is created with the
ordered_attributes, returns_unicode and specified_attributes set to the values of this parser.

**UseForeignDTD**([flag])

Calling this with a true value for flag (the default) will cause Expat to call the `ExternalEntityRefHandler` with None for all arguments to allow an alternate DTD to be loaded. If the document does not contain a document type declaration, the `ExternalEntityRefHandler` will still be called, but the `StartDoctypeDeclHandler` and `EndDoctypeDeclHandler` will not be called.

Passing a false value for flag will cancel a previous call that passed a true value, but otherwise has no effect.

This method can only be called before the `Parse()` or `ParseFile()` methods are called; calling it after either of those have been called causes `ExpatError` to be raised with the `code` attribute set to `errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING`.

New in version 2.3.

**xmlparser** objects have the following attributes:

**buffer_size**

The size of the buffer used when buffer_text is true. This value cannot be changed at this time. New in version 2.3.

**buffer_text**

Setting this to true causes the *xmlparser* object to buffer textual content returned by Expat to avoid multiple calls to the `CharacterDataHandler()` callback whenever possible. This can improve performance substantially since Expat normally breaks character data into chunks at every line ending. This attribute is false by default, and may be changed at any time. New in version 2.3.

**buffer_used**

If buffer_text is enabled, the number of bytes stored in the buffer. These bytes represent UTF-8 encoded text. This attribute has no meaningful interpretation when buffer_text is false. New in version 2.3.
ordered_attributes
Setting this attribute to a non-zero integer causes the attributes to be reported as a list rather than a dictionary. The attributes are presented in the order found in the document text. For each attribute, two list entries are presented: the attribute name and the attribute value. (Older versions of this module also used this format.) By default, this attribute is false; it may be changed at any time. New in version 2.1.

returns_unicode
If this attribute is set to a non-zero integer, the handler functions will be passed Unicode strings. If returns_unicode is False, 8-bit strings containing UTF-8 encoded data will be passed to the handlers. This is True by default when Python is built with Unicode support. Changed in version 1.6: Can be changed at any time to affect the result type.

specified_attributes
If set to a non-zero integer, the parser will report only those attributes which were specified in the document instance and not those which were derived from attribute declarations. Applications which set this need to be especially careful to use what additional information is available from the declarations as needed to comply with the standards for the behavior of XML processors. By default, this attribute is false; it may be changed at any time. New in version 2.1.

The following attributes contain values relating to the most recent error encountered by an xmlparser object, and will only have correct values once a call to Parse() or ParseFile() has raised a xml.parsers.expat.ExpatError exception.

ErrorByteIndex
 Byte index at which an error occurred.

ErrorCode
 Numeric code specifying the problem. This value can be passed to the ErrorString() function, or compared to one of the constants defined in the errors object.

ErrorColumnNumber
 Column number at which an error occurred.
**ErrorLineNumber**
Line number at which an error occurred.

The following attributes contain values relating to the current parse location in an **xmlparser** object. During a callback reporting a parse event they indicate the location of the first of the sequence of characters that generated the event. When called outside of a callback, the position indicated will be just past the last parse event (regardless of whether there was an associated callback). New in version 2.4.

**CurrentByteIndex**
Current byte index in the parser input.

**CurrentColumnNumber**
Current column number in the parser input.

**CurrentLineNumber**
Current line number in the parser input.

Here is the list of handlers that can be set. To set a handler on an **xmlparser** object *o*, use *o.handlername = func*. *handlername* must be taken from the following list, and *func* must be a callable object accepting the correct number of arguments. The arguments are all strings, unless otherwise stated.

**XmlDeclHandler**(version, encoding, standalone)
Called when the XML declaration is parsed. The XML declaration is the (optional) declaration of the applicable version of the XML recommendation, the encoding of the document text, and an optional ``standalone`` declaration. *version* and *encoding* will be strings of the type dictated by the returns_unicode attribute, and *standalone* will be 1 if the document is declared standalone, 0 if it is declared not to be standalone, or -1 if the standalone clause was omitted. This is only available with Expat version 1.95.0 or newer. New in version 2.1.

**StartDoctypeDeclHandler**(doctypeName, systemId, publicId, has_internal_subset)
Called when Expat begins parsing the document type declaration (<!DOCTYPE ...>). The *doctypeName* is provided exactly as presented. The *systemId* and *publicId* parameters give the system and public identifiers
if specified, or None if omitted. has_internal_subset will be true if the document contains and internal document declaration subset. This requires Expat version 1.2 or newer.

**EndDoctypeDeclHandler()**
Called when Expat is done parsing the document type declaration. This requires Expat version 1.2 or newer.

**ElementDeclHandler( name, model)**
Called once for each element type declaration. name is the name of the element type, and model is a representation of the content model.

**AttlistDeclHandler( elname, attname, type, default, required)**
Called for each declared attribute for an element type. If an attribute list declaration declares three attributes, this handler is called three times, once for each attribute. elname is the name of the element to which the declaration applies and attname is the name of the attribute declared. The attribute type is a string passed as type; the possible values are 'CDATA', 'ID', 'IDREF', ... default gives the default value for the attribute used when the attribute is not specified by the document instance, or None if there is no default value (#IMPLIED values). If the attribute is required to be given in the document instance, required will be true. This requires Expat version 1.95.0 or newer.

**StartElementHandler( name, attributes)**
Called for the start of every element. name is a string containing the element name, and attributes is a dictionary mapping attribute names to their values.

**EndElementHandler( name)**
Called for the end of every element.

**ProcessingInstructionHandler( target, data)**
Called for every processing instruction.

**CharacterDataHandler( data)**
Called for character data. This will be called for normal character data,
CDATA marked content, and ignorable whitespace. Applications which must distinguish these cases can use the StartCdataSectionHandler, EndCdataSectionHandler, and ElementDeclHandler callbacks to collect the required information.

**UnparsedEntityDeclHandler**

```
entityName, base, systemId, publicId, notationName
```

Called for unparsed (NDATA) entity declarations. This is only present for version 1.2 of the Expat library; for more recent versions, use EntityDeclHandler instead. (The underlying function in the Expat library has been declared obsolete.)

**EntityDeclHandler**

```
entityName, is_parameter_entity, value, base, systemId, publicId, notationName
```

Called for all entity declarations. For parameter and internal entities, value will be a string giving the declared contents of the entity; this will be None for external entities. The notationName parameter will be None for parsed entities, and the name of the notation for unparsed entities. is_parameter_entity will be true if the entity is a parameter entity or false for general entities (most applications only need to be concerned with general entities). This is only available starting with version 1.95.0 of the Expat library. New in version 2.1.

**NotationDeclHandler**

```
notationName, base, systemId, publicId
```

Called for notation declarations. notationName, base, and systemId, and publicId are strings if given. If the public identifier is omitted, publicId will be None.

**StartNamespaceDeclHandler**

```
prefix, uri
```

Called when an element contains a namespace declaration. Namespace declarations are processed before the StartElementHandler is called for the element on which declarations are placed.

**EndNamespaceDeclHandler**

```
prefix
```

Called when the closing tag is reached for an element that contained a namespace declaration. This is called once for each namespace declaration on the element in the reverse of the order for which the StartNamespaceDeclHandler was called to indicate the start of each
namespace declaration's scope. Calls to this handler are made after the corresponding EndElementHandler for the end of the element.

**CommentHandler**( data)
Called for comments. data is the text of the comment, excluding the leading `<!-` and trailing `-->`.

**StartCdataSectionHandler()**
Called at the start of a CDATA section. This and EndCdataSectionHandler are needed to be able to identify the syntactical start and end for CDATA sections.

**EndCdataSectionHandler()**
Called at the end of a CDATA section.

**DefaultHandler**( data)
Called for any characters in the XML document for which no applicable handler has been specified. This means characters that are part of a construct which could be reported, but for which no handler has been supplied.

**DefaultHandlerExpand**( data)
This is the same as the DefaultHandler, but doesn't inhibit expansion of internal entities. The entity reference will not be passed to the default handler.

**NotStandaloneHandler()**
Called if the XML document hasn't been declared as being a standalone document. This happens when there is an external subset or a reference to a parameter entity, but the XML declaration does not set standalone to **yes** in an XML declaration. If this handler returns 0, then the parser will throw an **XML_ERROR_NOT_STANDALONE** error. If this handler is not set, no exception is raised by the parser for this condition.

**ExternalEntityRefHandler**( context, base, systemId, publicId)
Called for references to external entities. base is the current base, as set by a previous call to **SetBase()**. The public and system identifiers, **systemId**
and *publicId*, are strings if given; if the public identifier is not given, *publicId* will be *None*. The *context* value is opaque and should only be used as described below.

For external entities to be parsed, this handler must be implemented. It is responsible for creating the sub-parser using *ExternalEntityParserCreate*(*context*), initializing it with the appropriate callbacks, and parsing the entity. This handler should return an integer; if it returns 0, the parser will throw an *XML_ERROR_EXTERNAL_ENTITY_HANDLING* error, otherwise parsing will continue.

If this handler is not provided, external entities are reported by the *DefaultHandler* callback, if provided.
Example
8.5.2 ExpatError Exceptions

ExpatError exceptions have a number of interesting attributes:

**code**
Expat's internal error number for the specific error. This will match one of the constants defined in the *errors* object from this module. New in version 2.1.

**lineno**
Line number on which the error was detected. The first line is numbered 1. New in version 2.1.

**offset**
Character offset into the line where the error occurred. The first column is numbered 0. New in version 2.1.
8.5.3 Example

The following program defines three handlers that just print out their arguments.

```python
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print 'Start element:', name, attrs
def end_element(name):
    print 'End element:', name
def char_data(data):
    print 'Character data:', repr(data)

p = xml.parsers.expat.ParserCreate()
p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

The output from this program is:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

---

**Python Library Reference**

Previous: 8.5.2 ExpatError Exceptions  Up: 8.5 xml.parsers.expat  Next: 8.5.4
Content Model Descriptions

*Release 2.5, documentation updated on 19th September, 2006.*

See *About this document...* for information on suggesting changes.
8.5.4 Content Model Descriptions

Content modules are described using nested tuples. Each tuple contains four values: the type, the quantifier, the name, and a tuple of children. Children are simply additional content module descriptions.

The values of the first two fields are constants defined in the `model` object of the `xml.parsers.expat` module. These constants can be collected in two groups: the model type group and the quantifier group.

The constants in the model type group are:

**XML_CTYPE_ANY**
- The element named by the model name was declared to have a content model of ANY.

**XML_CTYPE_CHOICE**
- The named element allows a choice from a number of options; this is used for content models such as `(A | B | C)`.

**XML_CTYPE_EMPTY**
- Elements which are declared to be EMPTY have this model type.

**XML_CTYPE_MIXED**

**XML_CTYPE_NAME**

**XML_CTYPE_SEQ**
- Models which represent a series of models which follow one after the other are indicated with this model type. This is used for models such as `(A, B, C)`.

The constants in the quantifier group are:

**XML_CQUANT_NONE**
- No modifier is given, so it can appear exactly once, as for A.

**XML_CQUANT_OPT**
The model is optional: it can appear once or not at all, as for A?.

**XML_CQUANT_PLUS**
The model must occur one or more times (like A+).

**XML_CQUANT_REP**
The model must occur zero or more times, as for A*.
8.5.5 Expat error constants

The following constants are provided in the `errors` object of the `xml.parsers.expat` module. These constants are useful in interpreting some of the attributes of the `ExpatError` exception objects raised when an error has occurred.

The `errors` object has the following attributes:

- **XML_ERROR_ASYNC_ENTITY**
- **XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF**
  - An entity reference in an attribute value referred to an external entity instead of an internal entity.
- **XML_ERROR_BAD_CHAR_REF**
  - A character reference referred to a character which is illegal in XML (for example, character 0, or `&#0;`).
- **XML_ERROR_BINARY_ENTITY_REF**
  - An entity reference referred to an entity which was declared with a notation, so cannot be parsed.
- **XML_ERROR_DUPLICATE_ATTRIBUTE**
  - An attribute was used more than once in a start tag.
- **XML_ERROR_INCORRECT_ENCODING**
- **XML_ERROR_INVALID_TOKEN**
  - Raised when an input byte could not properly be assigned to a character; for example, a NUL byte (value 0) in a UTF-8 input stream.
- **XML_ERROR_JUNK_AFTER_DOC_ELEMENT**
  - Something other than whitespace occurred after the document element.
- **XML_ERROR_MISPLACED_XML_PI**
  - An XML declaration was found somewhere other than the start of the input data.
XML_ERROR_NO_ELEMENTS
   The document contains no elements (XML requires all documents to contain exactly one top-level element).

XML_ERROR_NO_MEMORY
   Expat was not able to allocate memory internally.

XML_ERROR_PARAM_ENTITY_REF
   A parameter entity reference was found where it was not allowed.

XML_ERROR_PARTIAL_CHAR
   An incomplete character was found in the input.

XML_ERROR_RECURSIVE_ENTITY_REF
   An entity reference contained another reference to the same entity; possibly via a different name, and possibly indirectly.

XML_ERROR_SYNTAX
   Some unspecified syntax error was encountered.

XML_ERROR_TAG_MISMATCH
   An end tag did not match the innermost open start tag.

XML_ERROR_UNCLOSED_TOKEN
   Some token (such as a start tag) was not closed before the end of the stream or the next token was encountered.

XML_ERROR_UNDEFINED_ENTITY
   A reference was made to a entity which was not defined.

XML_ERROR_UNKNOWN_ENCODING
   The document encoding is not supported by Expat.

XML_ERROR_UNCLOSED_CDATA_SECTION
   A CDATA marked section was not closed.

XML_ERROR_EXTERNAL_ENTITY_HANDLING

XML_ERROR_NOT_STANDALONE
   The parser determined that the document was not "standalone" though it
declared itself to be in the XML declaration, and the NotStandaloneHandler was set and returned 0.

**XML_ERROR_UNEXPECTED_STATE**

**XML_ERROR_ENTITY_DECLARED_IN_PE**

**XML_ERROR_FEATURE_REQUIRES_XML_DTD**
An operation was requested that requires DTD support to be compiled in, but Expat was configured without DTD support. This should never be reported by a standard build of the xml.parsers.expat module.

**XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING**
A behavioral change was requested after parsing started that can only be changed before parsing has started. This is (currently) only raised by UseForeignDTD().

**XML_ERROR_UNBOUND_PREFIX**
An undeclared prefix was found when namespace processing was enabled.

**XML_ERROR_UNDECLARING_PREFIX**
The document attempted to remove the namespace declaration associated with a prefix.

**XML_ERROR_INCOMPLETE_PE**
A parameter entity contained incomplete markup.

**XML_ERROR_XML_DECL**
The document contained no document element at all.

**XML_ERROR_TEXT_DECL**
There was an error parsing a text declaration in an external entity.

**XML_ERROR_PUBLICID**
Characters were found in the public id that are not allowed.

**XML_ERROR_SUSPENDED**
The requested operation was made on a suspended parser, but isn't allowed. This includes attempts to provide additional input or to stop the parser.
XML_ERROR_NOT_SUSPENDED
An attempt to resume the parser was made when the parser had not been suspended.

XML_ERROR_ABORTED
This should not be reported to Python applications.

XML_ERROR_FINISHED
The requested operation was made on a parser which was finished parsing input, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

XML_ERROR_SUSPEND_PE
8.6 xml.dom -- The Document Object Model API

New in version 2.0.

The Document Object Model, or ``DOM,'' is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents. A DOM implementation presents an XML document as a tree structure, or allows client code to build such a structure from scratch. It then gives access to the structure through a set of objects which provided well-known interfaces.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another. If you are looking at a text node, you have no access to a containing element. When you write a SAX application, you need to keep track of your program's position in the document somewhere in your own code. SAX does not do it for you. Also, if you need to look ahead in the XML document, you are just out of luck.

Some applications are simply impossible in an event driven model with no access to a tree. Of course you could build some sort of tree yourself in SAX events, but the DOM allows you to avoid writing that code. The DOM is a standard tree representation for XML data.

The Document Object Model is being defined by the W3C in stages, or ``levels'' in their terminology. The Python mapping of the API is substantially based on the DOM Level 2 recommendation. The mapping of the Level 3 specification, currently only available in draft form, is being developed by the Python XML Special Interest Group as part of the PyXML package. Refer to the documentation bundled with that package for information on the current state of DOM Level 3 support.

DOM applications typically start by parsing some XML into a DOM. How this is accomplished is not covered at all by DOM Level 1, and Level 2 provides only limited improvements: There is a DOMImplementation object class
which provides access to Document creation methods, but no way to access an XML reader/parser/Document builder in an implementation-independent way. There is also no well-defined way to access these methods without an existing Document object. In Python, each DOM implementation will provide a function `getDOMImplementation()`. DOM Level 3 adds a Load/Store specification, which defines an interface to the reader, but this is not yet available in the Python standard library.

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification; this portion of the reference manual describes the interpretation of the specification in Python.

The specification provided by the W3C defines the DOM API for Java, ECMAScript, and OMG IDL. The Python mapping defined here is based in large part on the IDL version of the specification, but strict compliance is not required (though implementations are free to support the strict mapping from IDL). See section 8.6.3, "Conformance," for a detailed discussion of mapping requirements.

**See Also:**

*Document Object Model (DOM) Level 2 Specification*
  The W3C recommendation upon which the Python DOM API is based.

*Document Object Model (DOM) Level 1 Specification*
  The W3C recommendation for the DOM supported by `xml.dom.minidom`.

*PyXML*
  Users that require a full-featured implementation of DOM should use the PyXML package.

*Python Language Mapping Specification*
  This specifies the mapping from OMG IDL to Python.
Subsections

- **8.6.1 Module Contents**
- **8.6.2 Objects in the DOM**
  - 8.6.2.1 DOMImplementation Objects
  - 8.6.2.2 Node Objects
  - 8.6.2.3 NodeList Objects
  - 8.6.2.4 DocumentType Objects
  - 8.6.2.5 Document Objects
  - 8.6.2.6 Element Objects
  - 8.6.2.7 Attr Objects
  - 8.6.2.8 NamedNodeMap Objects
  - 8.6.2.9 Comment Objects
  - 8.6.2.10 Text and CDATASection Objects
  - 8.6.2.11 ProcessingInstruction Objects
  - 8.6.2.12 Exceptions
- **8.6.3 Conformance**
  - 8.6.3.1 Type Mapping
  - 8.6.3.2 Accessor Methods

---

**Python Library Reference**

Previous: **8.5.5 Expat error constants**
Up: **8. Structured Markup Processing**
Next: **8.6.1 Module Contents**

*Release 2.5, documentation updated on 19th September, 2006.*
*See [About this document...](#) for information on suggesting changes.*
8.6.1 Module Contents

The xml.dom contains the following functions:

**registerDOMImplementation**( *name*, *factory*)
Register the *factory* function with the name *name*. The factory function should return an object which implements the DOMImplementation interface. The factory function can return the same object every time, or a new one for each call, as appropriate for the specific implementation (e.g. if that implementation supports some customization).

**getDOMImplementation**( *name*, *features*)
Return a suitable DOM implementation. The *name* is either well-known, the module name of a DOM implementation, or *None*. If it is not *None*, imports the corresponding module and returns a DOMImplementation object if the import succeeds. If no name is given, and if the environment variable PYTHON_DOM is set, this variable is used to find the implementation.

If name is not given, this examines the available implementations to find one with the required feature set. If no implementation can be found, raise an *ImportError*. The features list must be a sequence of (*feature*, *version*) pairs which are passed to the hasFeature() method on available DOMImplementation objects.

Some convenience constants are also provided:

**EMPTY_NAMESPACE**
The value used to indicate that no namespace is associated with a node in the DOM. This is typically found as the namespaceURI of a node, or used as the namespaceURI parameter to a namespaces-specific method. New in version 2.2.

**XML_NAMESPACE**
The namespace URI associated with the reserved prefix xml, as defined by *Namespaces in XML* (section 4). New in version 2.2.
XMLNS_NAMESPACE

The namespace URI for namespace declarations, as defined by Document Object Model (DOM) Level 2 Core Specification (section 1.1.8). New in version 2.2.

XHTML_NAMESPACE

The URI of the XHTML namespace as defined by XHTML 1.0: The Extensible HyperText Markup Language (section 3.1.1). New in version 2.2.

In addition, xml.dom contains a base Node class and the DOM exception classes. The Node class provided by this module does not implement any of the methods or attributes defined by the DOM specification; concrete DOM implementations must provide those. The Node class provided as part of this module does provide the constants used for the nodeType attribute on concrete Node objects; they are located within the class rather than at the module level to conform with the DOM specifications.
8.6.2 Objects in the DOM

The definitive documentation for the DOM is the DOM specification from the W3C.

Note that DOM attributes may also be manipulated as nodes instead of as simple strings. It is fairly rare that you must do this, however, so this usage is not yet documented.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Section</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMImplementation</td>
<td>8.6.2</td>
<td>Interface to the underlying implementation.</td>
</tr>
<tr>
<td>Node</td>
<td>8.6.2</td>
<td>Base interface for most objects in a document.</td>
</tr>
<tr>
<td>NodeList</td>
<td>8.6.2</td>
<td>Interface for a sequence of nodes.</td>
</tr>
<tr>
<td>DocumentType</td>
<td>8.6.2</td>
<td>Information about the declarations needed to process a document.</td>
</tr>
<tr>
<td>Document</td>
<td>8.6.2</td>
<td>Object which represents an entire document.</td>
</tr>
<tr>
<td>Element</td>
<td>8.6.2</td>
<td>Element nodes in the document hierarchy.</td>
</tr>
<tr>
<td>Attr</td>
<td>8.6.2</td>
<td>Attribute value nodes on element nodes.</td>
</tr>
<tr>
<td>Comment</td>
<td>8.6.2</td>
<td>Representation of comments in the source document.</td>
</tr>
<tr>
<td>Text</td>
<td>8.6.2</td>
<td>Nodes containing textual content from the document.</td>
</tr>
<tr>
<td>ProcessingInstruction</td>
<td>8.6.2</td>
<td>Processing instruction representation.</td>
</tr>
</tbody>
</table>

An additional section describes the exceptions defined for working with the DOM in Python.
Subsections

- 8.6.2.1 DOMImplementation Objects
- 8.6.2.2 Node Objects
- 8.6.2.3 NodeList Objects
- 8.6.2.4 DocumentType Objects
- 8.6.2.5 Document Objects
- 8.6.2.6 Element Objects
- 8.6.2.7 Attr Objects
- 8.6.2.8 NamedNodeMap Objects
- 8.6.2.9 Comment Objects
- 8.6.2.10 Text and CDATASection Objects
- 8.6.2.11 ProcessingInstruction Objects
- 8.6.2.12 Exceptions

Release 2.5, documentation updated on 19th September, 2006.  
See About this document... for information on suggesting changes.
8.6.3 Conformance

This section describes the conformance requirements and relationships between the Python DOM API, the W3C DOM recommendations, and the OMG IDL mapping for Python.

Subsections

- 8.6.3.1 Type Mapping
- 8.6.3.2 Accessor Methods
8.7 xml.dom.minidom -- Lightweight DOM implementation

New in version 2.0.

xml.dom.minidom is a light-weight implementation of the Document Object Model interface. It is intended to be simpler than the full DOM and also significantly smaller.

DOM applications typically start by parsing some XML into a DOM. With xml.dom.minidom, this is done through the parse functions:

```python
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\temp\mydata.xml')  # parse an XML file by name

datasource = open('c:\temp\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>
```

The `parse()` function can take either a filename or an open file object.

**parse** *(filename_or_file, parser)*

Return a Document from the given input. `filename_or_file` may be either a file name, or a file-like object. `parser`, if given, must be a SAX2 parser object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the `parseString()` function instead:

**parseString**(string[, parser])

Return a Document that represents the string. This method creates a StringIO object for the string and passes that on to parse.

Both functions return a Document object representing the content of the
document.

What the parse() and parseString() functions do is connect an XML parser with a ``DOM builder'' that can accept parse events from any SAX parser and convert them into a DOM tree. The name of the functions are perhaps misleading, but are easy to grasp when learning the interfaces. The parsing of the document will be completed before these functions return; it's simply that these functions do not provide a parser implementation themselves.

You can also create a Document by calling a method on a ``DOM Implementation'' object. You can get this object either by calling the getDOMImplementation() function in the xml.dom package or the xml.dom.minidom module. Using the implementation from the xml.dom.minidom module will always return a Document instance from the minidom implementation, while the version from xml.dom may provide an alternate implementation (this is likely if you have the PyXML package installed). Once you have a Document, you can add child nodes to it to populate the DOM:

```python
from xml.dom.minidom import getDOMImplementation
impl = getDOMImplementation()
newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content. ')
top_element.appendChild(text)
```

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification. The main property of the document object is the documentElement property. It gives you the main element in the XML document: the one that holds all others. Here is an example program:

```python
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

When you are finished with a DOM, you should clean it up. This is necessary because some versions of Python do not support garbage collection of objects that refer to each other in a cycle. Until this restriction is removed from all versions of Python, it is safest to write your code as if cycles would not be cleaned up.
The way to clean up a DOM is to call its `unlink()` method:

```python
dom1.unlink()
dom2.unlink()
dom3.unlink()
```

`unlink()` is a `xml.dom.minidom`-specific extension to the DOM API. After calling `unlink()` on a node, the node and its descendants are essentially useless.

**See Also:**

*Document Object Model (DOM) Level 1 Specification*

The W3C recommendation for the DOM supported by `xml.dom.minidom`.

---

**Subsections**

- 8.7.1 DOM Objects
- 8.7.2 DOM Example
- 8.7.3 minidom and the DOM standard

---

**Python Library Reference**

Previous: 8.6.3.2 Accessor Methods  Up: 8. Structured Markup Processing  Next: 8.7.1 DOM Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
8.7.1 DOM Objects

The definition of the DOM API for Python is given as part of the `xml.dom` module documentation. This section lists the differences between the API and `xml.dom.minidom`.

unlink()

Break internal references within the DOM so that it will be garbage collected on versions of Python without cyclic GC. Even when cyclic GC is available, using this can make large amounts of memory available sooner, so calling this on DOM objects as soon as they are no longer needed is good practice. This only needs to be called on the Document object, but may be called on child nodes to discard children of that node.

writexml( writer[, indent="", [addindent="", [newl=""]]])

Write XML to the writer object. The writer should have a write() method which matches that of the file object interface. The indent parameter is the indentation of the current node. The addindent parameter is the incremental indentation to use for subnodes of the current one. The newl parameter specifies the string to use to terminate newlines.

Changed in version 2.1: The optional keyword parameters indent, addindent, and newl were added to support pretty output.

Changed in version 2.3: For the Document node, an additional keyword argument encoding can be used to specify the encoding field of the XML header.

toxml([encoding])

Return the XML that the DOM represents as a string.

With no argument, the XML header does not specify an encoding, and the result is Unicode string if the default encoding cannot represent all characters in the document. Encoding this string in an encoding other than UTF-8 is likely incorrect, since UTF-8 is the default encoding of XML.

With an explicit encoding argument, the result is a byte string in the
specified encoding. It is recommended that this argument is always specified. To avoid UnicodeError exceptions in case of unrepresentable text data, the encoding argument should be specified as "utf-8".

Changed in version 2.3: the encoding argument was introduced.

toprettyxml([indent[, newl]])
    Return a pretty-printed version of the document. indent specifies the indentation string and defaults to a tabulator; newl specifies the string emitted at the end of each line and defaults to \n.

    New in version 2.1. Changed in version 2.3: the encoding argument; see toxml().

The following standard DOM methods have special considerations with xml.dom.minidom:

cloneNode(deep)
    Although this method was present in the version of xml.dom.minidom packaged with Python 2.0, it was seriously broken. This has been corrected for subsequent releases.

---

Python Library Reference

Previous: 8.7 xml.dom.minidom Up: 8.7 xml.dom.minidom Next: 8.7.2 DOM Example

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
and the
8.7.2 DOM Example

This example program is a fairly realistic example of a simple program. In this particular case, we do not take much advantage of the flexibility of the DOM.

```python
import xml.dom.minidom

document = """
<slideshow>
<title>Demo slideshow</title>
<slide>
<title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point></slide>

<slide>
<title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>"

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = ""
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc = rc + node.data
    return rc

def handleSlideshow(slideshow):
    print "<html>"
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print "</html>"

def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
"""
print "<title>%s</title>" % getText(title.childNodes)

def handleSlideTitle(title):
    print "<h2>%s</h2>" % getText(title.childNodes)

def handlePoints(points):
    print "<ul>
    for point in points:
        handlePoint(point)
    print "</ul>"

def handlePoint(point):
    print "<li>%s</li>" % getText(point.childNodes)

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print "<p>%s</p>" % getText(title.childNodes)

handleSlideshow(dom)

Download as text (original file name: minidom-example.py).
8.7.3 minidom and the DOM standard

The `xml.dom.minidom` module is essentially a DOM 1.0-compatible DOM with some DOM 2 features (primarily namespace features).

Usage of the DOM interface in Python is straight-forward. The following mapping rules apply:

- Interfaces are accessed through instance objects. Applications should not instantiate the classes themselves; they should use the creator functions available on the `Document` object. Derived interfaces support all operations (and attributes) from the base interfaces, plus any new operations.

- Operations are used as methods. Since the DOM uses only `in` parameters, the arguments are passed in normal order (from left to right). There are no optional arguments. `void` operations return `None`.

- IDL attributes map to instance attributes. For compatibility with the OMG IDL language mapping for Python, an attribute `foo` can also be accessed through accessor methods `_get_foo()` and `_set_foo()`. Readonly attributes must not be changed; this is not enforced at runtime.

- The types `short` int, `unsigned int`, `unsigned long long`, and `boolean` all map to Python integer objects.

- The type `DOMString` maps to Python strings. `xml.dom.minidom` supports either byte or Unicode strings, but will normally produce Unicode strings. Values of type `DOMString` may also be `None` where allowed to have the IDL `null` value by the DOM specification from the W3C.

- `const` declarations map to variables in their respective scope (e.g. `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`); they must not be changed.

- `DOMException` is currently not supported in `xml.dom.minidom`. Instead, `xml.dom.minidom` uses standard Python exceptions such as
TypeError and AttributeError.

- NodeList objects are implemented using Python's built-in list type. Starting with Python 2.2, these objects provide the interface defined in the DOM specification, but with earlier versions of Python they do not support the official API. They are, however, much more "Pythonic" than the interface defined in the W3C recommendations.

The following interfaces have no implementation in xml.dom.minidom:

- DOMTimeStamp
- DocumentType (added in Python 2.1)
- DOMImplementation (added in Python 2.1)
- CharacterData
- CDATASection
- Notation
- Entity
- EntityReference
- DocumentFragment

Most of these reflect information in the XML document that is not of general utility to most DOM users.
8.8 `xml.dom.pulldom` -- Support for building partial DOM trees

New in version 2.0.

`xml.dom.pulldom` allows building only selected portions of a Document Object Model representation of a document from SAX events.

```python
class PullDOM([documentFactory])
    xml.sax.handler.ContentHandler implementation that ...

class DOMEventStream(stream, parser, bufsize)
    ...

class SAX2DOM([documentFactory])
    xml.sax.handler.ContentHandler implementation that ...

parse(stream_or_string[, parser[, bufsize]])
    ...

parseString(string[, parser])
    ...

default_bufsize
    Default value for the bufsize parameter to parse(). Changed in version 2.1: The value of this variable can be changed before calling parse() and the new value will take effect.
```

Subsections

- 8.8.1 DOMEventStream Objects
8.8.1 DOMEventStream Objects

getEvent()
...

expandNode( node)
...

reset()
...
8.9 xml.sax -- Support for SAX2 parsers

New in version 2.0.

The xml.sax package provides a number of modules which implement the Simple API for XML (SAX) interface for Python. The package itself provides the SAX exceptions and the convenience functions which will be most used by users of the SAX API.

The convenience functions are:

**make_parser([parser_list])**
Create and return a SAX XMLReader object. The first parser found will be used. If parser_list is provided, it must be a sequence of strings which name modules that have a function named *create_parser()*(). Modules listed in parser_list will be used before modules in the default list of parsers.

**parse(filename_or_stream, handler[, error_handler])**
Create a SAX parser and use it to parse a document. The document, passed in as filename_or_stream, can be a filename or a file object. The handler parameter needs to be a SAX ContentHandler instance. If error_handler is given, it must be a SAX ErrorHandler instance; if omitted, SAXParseException will be raised on all errors. There is no return value; all work must be done by the handler passed in.

**parseString(string, handler[, error_handler])**
Similar to parse(), but parses from a buffer string received as a parameter.

A typical SAX application uses three kinds of objects: readers, handlers and input sources. ``Reader`` in this context is another term for parser, i.e. some piece of code that reads the bytes or characters from the input source, and produces a sequence of events. The events then get distributed to the handler objects, i.e. the reader invokes a method on the handler. A SAX application must therefore
obtain a reader object, create or open the input sources, create the handlers, and connect these objects all together. As the final step of preparation, the reader is called to parse the input. During parsing, methods on the handler objects are called based on structural and syntactic events from the input data.

For these objects, only the interfaces are relevant; they are normally not instantiated by the application itself. Since Python does not have an explicit notion of interface, they are formally introduced as classes, but applications may use implementations which do not inherit from the provided classes. The InputSource, Locator, Attributes, AttributesNS, and XMLReader interfaces are defined in the module `xml.sax.xmlreader`. The handler interfaces are defined in `xml.sax.handler`. For convenience, `InputSource` (which is often instantiated directly) and the handler classes are also available from `xml.sax`. These interfaces are described below.

In addition to these classes, `xml.sax` provides the following exception classes.

**exception SAXException** *(msg[, exception])*

Encapsulate an XML error or warning. This class can contain basic error or warning information from either the XML parser or the application: it can be subclassed to provide additional functionality or to add localization. Note that although the handlers defined in the ErrorHandler interface receive instances of this exception, it is not required to actually raise the exception -- it is also useful as a container for information.

When instantiated, *msg* should be a human-readable description of the error. The optional *exception* parameter, if given, should be `None` or an exception that was caught by the parsing code and is being passed along as information.

This is the base class for the other SAX exception classes.

**exception SAXParseException** *(msg, exception, locator)*

Subclass of `SAXException` raised on parse errors. Instances of this class are passed to the methods of the SAX ErrorHandler interface to provide information about the parse error. This class supports the SAX Locator interface as well as the SAXException interface.
exception SAXNotRecognizedException($msg[, exception])

Subclass of SAXException raised when a SAX XMLReader is confronted with an unrecognized feature or property. SAX applications and extensions may use this class for similar purposes.

exception SAXNotSupportedException($msg[, exception])

Subclass of SAXException raised when a SAX XMLReader is asked to enable a feature that is not supported, or to set a property to a value that the implementation does not support. SAX applications and extensions may use this class for similar purposes.

See Also:

[SAX: The Simple API for XML](#)

This site is the focal point for the definition of the SAX API. It provides a Java implementation and online documentation. Links to implementations and historical information are also available.

Module xml.sax.handler:

Definitions of the interfaces for application-provided objects.

Module xml.sax.saxutils:

Convenience functions for use in SAX applications.

Module xml.sax.xmlreader:

Definitions of the interfaces for parser-provided objects.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
8.9.1 SAXException Objects

The SAXException exception class supports the following methods:

**getMessage()**
Return a human-readable message describing the error condition.

**getException()**
Return an encapsulated exception object, or None.
8.10 xml.sax.handler -- Base classes for SAX handlers

New in version 2.0.

The SAX API defines four kinds of handlers: content handlers, DTD handlers, error handlers, and entity resolvers. Applications normally only need to implement those interfaces whose events they are interested in; they can implement the interfaces in a single object or in multiple objects. Handler implementations should inherit from the base classes provided in the module xml.sax.handler, so that all methods get default implementations.

**class ContentHandler**

This is the main callback interface in SAX, and the one most important to applications. The order of events in this interface mirrors the order of the information in the document.

**class DTDHandler**

Handle DTD events.

This interface specifies only those DTD events required for basic parsing (unparsed entities and attributes).

**class EntityResolver**

Basic interface for resolving entities. If you create an object implementing this interface, then register the object with your Parser, the parser will call the method in your object to resolve all external entities.

**class ErrorHandler**

Interface used by the parser to present error and warning messages to the application. The methods of this object control whether errors are immediately converted to exceptions or are handled in some other way.

In addition to these classes, xml.sax.handler provides symbolic constants for the feature and property names.
**feature_namespaces**
Value: "http://xml.org/sax/features/namespaces"
true: Perform Namespace processing.
false: Optionally do not perform Namespace processing (implies namespace-prefixes; default).
access: (parsing) read-only; (not parsing) read/write

**feature_namespace_prefixes**
Value: "http://xml.org/sax/features/namespace-prefixes"
true: Report the original prefixed names and attributes used for Namespace declarations.
false: Do not report attributes used for Namespace declarations, and optionally do not report original prefixed names (default).
access: (parsing) read-only; (not parsing) read/write

**feature_string_interning**
Value: "http://xml.org/sax/features/string-interning"
true: All element names, prefixes, attribute names, Namespace URIs, and local names are interned using the built-in intern function.
false: Names are not necessarily interned, although they may be (default).
access: (parsing) read-only; (not parsing) read/write

**feature_validation**
Value: "http://xml.org/sax/features/validation"
true: Report all validation errors (implies external-general-entities and external-parameter-entities).
false: Do not report validation errors.
access: (parsing) read-only; (not parsing) read/write

**feature_external_ges**
Value: "http://xml.org/sax/features/external-general-entities"
true: Include all external general (text) entities.
false: Do not include external general entities.
access: (parsing) read-only; (not parsing) read/write

**feature_external_pes**
Value: "http://xml.org/sax/features/external-
parameter-entities"
true: Include all external parameter entities, including the external DTD subset.
false: Do not include any external parameter entities, even the external DTD subset.
access: (parsing) read-only; (not parsing) read/write

all_features
List of all features.

property_lexical_handler
Value: "http://xml.org/sax/properties/lexical-handler"
data type: xml.sax.sax2lib.LexicalHandler (not supported in Python 2)
description: An optional extension handler for lexical events like comments.
access: read/write

property_declaration_handler
Value: "http://xml.org/sax/properties/declaration-handler"
data type: xml.sax.sax2lib.DeclHandler (not supported in Python 2)
description: An optional extension handler for DTD-related events other than notations and unparsed entities.
access: read/write

property_dom_node
Value: "http://xml.org/sax/properties/dom-node"
data type: org.w3c.dom.Node (not supported in Python 2)
description: When parsing, the current DOM node being visited if this is a DOM iterator; when not parsing, the root DOM node for iteration.
access: (parsing) read-only; (not parsing) read/write

property_xml_string
Value: "http://xml.org/sax/properties/xml-string"
data type: String
description: The literal string of characters that was the source for the current event.
access: read-only
all_properties
List of all known property names.

Subsections

- 8.10.1 ContentHandler Objects
- 8.10.2 DTDHandler Objects
- 8.10.3 EntityResolver Objects
- 8.10.4 ErrorHandler Objects

Python Library Reference
Previous: 8.9.1 SAXException Objects Up: 8. Structured Markup Processing Next: 8.10.1 ContentHandler Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
DTDHandler Objects
8.10.1 ContentHandler Objects

Users are expected to subclass ContentHandler to support their application. The following methods are called by the parser on the appropriate events in the input document:

**setDocumentLocator( locator)**

Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

**startDocument()**

Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for setDocumentLocator()).

**endDocument()**

Receive notification of the end of a document.

The SAX parser will invoke this method only once, and it will be the last
method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input.

**startPrefixMapping**( prefix, uri)

Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the feature_namespaces feature is enabled (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the **startPrefixMapping()** and **endPrefixMapping()** events supply the information to the application to expand prefixes in those contexts itself, if necessary.

Note that **startPrefixMapping()** and **endPrefixMapping()** events are not guaranteed to be properly nested relative to each-other: all **startPrefixMapping()** events will occur before the corresponding **startElement()** event, and all **endPrefixMapping()** events will occur after the corresponding **endElement()** event, but their order is not guaranteed.

**endPrefixMapping**( prefix)

End the scope of a prefix-URI mapping.

See **startPrefixMapping()** for details. This event will always occur after the corresponding **endElement()** event, but the order of **endPrefixMapping()** events is not otherwise guaranteed.

**startElement**( name, attrs)

Signals the start of an element in non-namespace mode.

The **name** parameter contains the raw XML 1.0 name of the element type as a string and the **attrs** parameter holds an object of the **Attributes** interface containing the attributes of the element. The object passed as **attrs**
may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the `copy()` method of the `attrs` object.

**endElement**(*name*)
Signals the end of an element in non-namespace mode.

The *name* parameter contains the name of the element type, just as with the `startElement()` event.

**startElementNS**(*name, qname, attrs*)
Signals the start of an element in namespace mode.

The *name* parameter contains the name of the element type as a `(uri, localname)` tuple, the *qname* parameter contains the raw XML 1.0 name used in the source document, and the *attrs* parameter holds an instance of the `AttributesNS` interface containing the attributes of the element. If no namespace is associated with the element, the *uri* component of *name* will be `None`. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the `copy()` method of the `attrs` object.

Parsers may set the *qname* parameter to `None`, unless the `feature_namespace_prefixes` feature is activated.

**endElementNS**(*name, qname*)
Signals the end of an element in namespace mode.

The *name* parameter contains the name of the element type, just as with the `startElementNS()` method, likewise the *qname* parameter.

**characters**(*content*)
Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any
single event must come from the same external entity so that the Locator provides useful information.

$content$ may be a Unicode string or a byte string; the $expat$ reader module produces always Unicode strings.

**Note:** The earlier SAX 1 interface provided by the Python XML Special Interest Group used a more Java-like interface for this method. Since most parsers used from Python did not take advantage of the older interface, the simpler signature was chosen to replace it. To convert old code to the new interface, use $content$ instead of slicing content with the old $offset$ and $length$ parameters.

**ignorableWhitespace**( $whitespace$ )
Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

**processingInstruction**( $target$, $data$ )
Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

**skippedEntity**( $name$ )
Receive notification of a skipped entity.
The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the \texttt{feature\_external\_ges} and the \texttt{feature\_external\_pes} properties.
8.10.2 DTDHandler Objects

DTDHandler instances provide the following methods:

notationDecl( name, publicId, systemId)

Handle a notation declaration event.

unparsedEntityDecl( name, publicId, systemId, ndata)

Handle an unparsed entity declaration event.
8.10.3 EntityResolver Objects

resolveEntity( publicId, systemId)
Resolve the system identifier of an entity and return either the system identifier to read from as a string, or an InputSource to read from. The default implementation returns systemId.
8.10.4 ErrorHandler Objects

Objects with this interface are used to receive error and warning information from the XMLReader. If you create an object that implements this interface, then register the object with your XMLReader, the parser will call the methods in your object to report all warnings and errors. There are three levels of errors available: warnings, (possibly) recoverable errors, and unrecoverable errors. All methods take a SAXParseException as the only parameter. Errors and warnings may be converted to an exception by raising the passed-in exception object.

**error**(*exception*)

Called when the parser encounters a recoverable error. If this method does not raise an exception, parsing may continue, but further document information should not be expected by the application. Allowing the parser to continue may allow additional errors to be discovered in the input document.

**fatalError**(*exception*)

Called when the parser encounters an error it cannot recover from; parsing is expected to terminate when this method returns.

**warning**(*exception*)

Called when the parser presents minor warning information to the application. Parsing is expected to continue when this method returns, and document information will continue to be passed to the application. Raising an exception in this method will cause parsing to end.
8.11 xml.sax.saxutils -- SAX Utilities

New in version 2.0.

The module xml.sax.saxutils contains a number of classes and functions that are commonly useful when creating SAX applications, either in direct use, or as base classes.

**escape**(data[, entities])

Escape "&", "<", and ">" in a string of data.

You can escape other strings of data by passing a dictionary as the optional entities parameter. The keys and values must all be strings; each key will be replaced with its corresponding value.

**unescape**(data[, entities])

Unescape "&amp;", "&lt;", and "&gt;" in a string of data.

You can unescape other strings of data by passing a dictionary as the optional entities parameter. The keys and values must all be strings; each key will be replaced with its corresponding value.

New in version 2.3.

**quoteattr**(data[, entities])

Similar to escape(), but also prepares data to be used as an attribute value. The return value is a quoted version of data with any additional required replacements. quoteattr() will select a quote character based on the content of data, attempting to avoid encoding any quote characters in the string. If both single- and double-quote characters are already in data, the double-quote characters will be encoded and data will be wrapped in double-quotes. The resulting string can be used directly as an attribute value:
>>> print "<element attr=%s>" % quoteattr("ab ' cd " ef")
<element attr="ab ' cd &quot; ef">

This function is useful when generating attribute values for HTML or any SGML using the reference concrete syntax. New in version 2.2.

class XMLGenerator([out[, encoding]])

This class implements the ContentHandler interface by writing SAX events back into an XML document. In other words, using an XMLGenerator as the content handler will reproduce the original document being parsed. out should be a file-like object which will default to sys.stdout. encoding is the encoding of the output stream which defaults to 'iso-8859-1'.

class XMLFilterBase(base)

This class is designed to sit between an XMLReader and the client application's event handlers. By default, it does nothing but pass requests up to the reader and events on to the handlers unmodified, but subclasses can override specific methods to modify the event stream or the configuration requests as they pass through.

prepare_input_source(source[, base])

This function takes an input source and an optional base URL and returns a fully resolved InputSource object ready for reading. The input source can be given as a string, a file-like object, or an InputSource object; parsers will use this function to implement the polymorphic source argument to their parse() method.
8.12 `xml.sax.xmlreader` -- Interface for XML parsers

New in version 2.0.

SAX parsers implement the `XMLReader` interface. They are implemented in a Python module, which must provide a function `create_parser()`. This function is invoked by `xml.sax.make_parser()` with no arguments to create a new parser object.

```python
class XMLReader()
    Base class which can be inherited by SAX parsers.
```

```python
class IncrementalParser()
    In some cases, it is desirable not to parse an input source at once, but to feed chunks of the document as they get available. Note that the reader will normally not read the entire file, but read it in chunks as well; still `parse()` won't return until the entire document is processed. So these interfaces should be used if the blocking behaviour of `parse()` is not desirable.

    When the parser is instantiated it is ready to begin accepting data from the feed method immediately. After parsing has been finished with a call to close the reset method must be called to make the parser ready to accept new data, either from feed or using the parse method.

    Note that these methods must `not` be called during parsing, that is, after parse has been called and before it returns.

    By default, the class also implements the parse method of the XMLReader interface using the feed, close and reset methods of the IncrementalParser interface as a convenience to SAX 2.0 driver writers.
```

```python
class Locator()
    Interface for associating a SAX event with a document location. A locator
object will return valid results only during calls to DocumentHandler methods; at any other time, the results are unpredictable. If information is not available, methods may return None.

**class InputSource([systemId])**

Encapsulation of the information needed by the XMLReader to read entities.

This class may include information about the public identifier, system identifier, byte stream (possibly with character encoding information) and/or the character stream of an entity.

Applications will create objects of this class for use in the XMLReader.parse() method and for returning from EntityResolver.resolveEntity.

An InputSource belongs to the application, the XMLReader is not allowed to modify InputSource objects passed to it from the application, although it may make copies and modify those.

**class AttributesImpl(attrs)**

This is an implementation of the Attributes interface (see section 8.12.5). This is a dictionary-like object which represents the element attributes in a startElement() call. In addition to the most useful dictionary operations, it supports a number of other methods as described by the interface. Objects of this class should be instantiated by readers; attrs must be a dictionary-like object containing a mapping from attribute names to attribute values.

**class AttributesNSImpl(attrs, qnames)**

Namespace-aware variant of AttributesImpl, which will be passed to startElementNS(). It is derived from AttributesImpl, but understands attribute names as two-tuples of namespaceURI and localname. In addition, it provides a number of methods expecting qualified names as they appear in the original document. This class implements the AttributesNS interface (see section 8.12.6).
Subsections

- 8.12.1 XMLReader Objects
- 8.12.2 IncrementalParser Objects
- 8.12.3 Locator Objects
- 8.12.4 InputSource Objects
- 8.12.5 The Attributes Interface
- 8.12.6 The AttributesNS Interface

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
8.12.1 XMLReader Objects

The XMLReader interface supports the following methods:

`parse(source)`
Process an input source, producing SAX events. The `source` object can be a system identifier (a string identifying the input source - typically a file name or an URL), a file-like object, or an `InputSource` object. When `parse()` returns, the input is completely processed, and the parser object can be discarded or reset. As a limitation, the current implementation only accepts byte streams; processing of character streams is for further study.

`getContentHandler()`
Return the current `ContentHandler`.

`setContentHandler(handler)`
Set the current `ContentHandler`. If no `ContentHandler` is set, content events will be discarded.

`getDTDHandler()`
Return the current `DTDHandler`.

`setDTDHandler(handler)`
Set the current `DTDHandler`. If no `DTDHandler` is set, DTD events will be discarded.

`getEntityResolver()`
Return the current `EntityResolver`.

`setEntityResolver(handler)`
Set the current `EntityResolver`. If no `EntityResolver` is set, attempts to resolve an external entity will result in opening the system identifier for the entity, and fail if it is not available.

`getErrorHandler()`
Return the current Error_Handler.

**setErrorHandler** *(handler)*
Set the current error handler. If no Error_Handler is set, errors will be raised as exceptions, and warnings will be printed.

**setLocale**(locale)
Allow an application to set the locale for errors and warnings.

SAX parsers are not required to provide localization for errors and warnings; if they cannot support the requested locale, however, they must throw a SAX exception. Applications may request a locale change in the middle of a parse.

**getFeature**(featurename)
Return the current setting for feature *featurename*. If the feature is not recognized, SAXNotRecognizedException is raised. The well-known featurenames are listed in the module `xml.sax.handler`.

**setFeature**(featurename, value)
Set the *featurename* to *value*. If the feature is not recognized, SAXNotRecognizedException is raised. If the feature or its setting is not supported by the parser, SAXNotSupportedException is raised.

**getProperty**(propertyname)
Return the current setting for property *propertyname*. If the property is not recognized, a SAXNotRecognizedException is raised. The well-known propertynames are listed in the module `xml.sax.handler`.

**setProperty**(propertyname, value)
Set the *propertyname* to *value*. If the property is not recognized, SAXNotRecognizedException is raised. If the property or its setting is not supported by the parser, SAXNotSupportedException is raised.
IncrementalParser Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
8.12.2 IncrementalParser Objects

Instances of `IncrementalParser` offer the following additional methods:

`feed(data)`
Process a chunk of `data`.

`close()`
Assume the end of the document. That will check well-formedness conditions that can be checked only at the end, invoke handlers, and may clean up resources allocated during parsing.

`reset()`
This method is called after `close` has been called to reset the parser so that it is ready to parse new documents. The results of calling `parse` or `feed` after `close` without calling `reset` are undefined.
8.12.3 Locator Objects

Instances of Locator provide these methods:

**getColumnNumber()**
Return the column number where the current event ends.

**getLineNumber()**
Return the line number where the current event ends.

**getPublicId()**
Return the public identifier for the current event.

**getSystemId()**
Return the system identifier for the current event.
8.12.4 InputSource Objects

**setPublicId**( *id*)
Sets the public identifier of this InputSource.

**getPublicId**()
Returns the public identifier of this InputSource.

**setSystemId**( *id*)
Sets the system identifier of this InputSource.

**getSystemId**()
Returns the system identifier of this InputSource.

**setEncoding**( *encoding*)
Sets the character encoding of this InputSource.

The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML recommendation).

The encoding attribute of the InputSource is ignored if the InputSource also contains a character stream.

**getEncoding**()
Get the character encoding of this InputSource.

**setByteStream**( *bytefile*)
Set the byte stream (a Python file-like object which does not perform byte-to-character conversion) for this input source.

The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.

If the application knows the character encoding of the byte stream, it should set it with the setEncoding method.
getByteStream()
Get the byte stream for this input source.

The getEncoding method will return the character encoding for this byte stream, or None if unknown.

setCharacterStream( charfile)
Set the character stream for this input source. (The stream must be a Python 1.6 Unicode-wrapped file-like that performs conversion to Unicode strings.)

If there is a character stream specified, the SAX parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier.

getCharacterStream()
Get the character stream for this input source.
8.12.5 The Attributes Interface

Attributes objects implement a portion of the mapping protocol, including the methods `copy()`, `get()`, `has_key()`, `items()`, `keys()`, and `values()`. The following methods are also provided:

`getLength()`
   Return the number of attributes.

`getNames()`
   Return the names of the attributes.

`getType(name)`
   Returns the type of the attribute `name`, which is normally 'CDATA'.

`getValue(name)`
   Return the value of attribute `name`. 
8.12.6 The AttributesNS Interface

This interface is a subtype of the Attributes interface (see section 8.12.5). All methods supported by that interface are also available on AttributesNS objects.

The following methods are also available:

**getValueByQName**(name)
Return the value for a qualified name.

**getNameByQName**(name)
Return the (namespace, localname) pair for a qualified name.

**getQNameByName**(name)
Return the qualified name for a (namespace, localname) pair.

**getQNames**( )
Return the qualified names of all attributes.
8.13 `xml.etree.ElementTree` -- The ElementTree XML API

New in version 2.5.

The Element type is a flexible container object, designed to store hierarchical data structures in memory. The type can be described as a cross between a list and a dictionary.

Each element has a number of properties associated with it:

- a tag which is a string identifying what kind of data this element represents (the element type, in other words).
- a number of attributes, stored in a Python dictionary.
- a text string.
- an optional tail string.
- a number of child elements, stored in a Python sequence.

To create an element instance, use the `Element` or `SubElement` factory functions.

The `ElementTree` class can be used to wrap an element structure, and convert it from and to XML.

A C implementation of this API is available as `xml.etree.cElementTree`.

**Subsections**

- [8.13.1 Functions](#)
- [8.13.2 ElementTree Objects](#)
- [8.13.3 QName Objects](#)
- [8.13.4 TreeBuilder Objects](#)
- [8.13.5 XMLTreeBuilder Objects](#)
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.


8.13.1 Functions

Comment([text])
Comment element factory. This factory function creates a special element that will be serialized as an XML comment. The comment string can be either an 8-bit ASCII string or a Unicode string. text is a string containing the comment string.

Returns:
An element instance, representing a comment.

dump(elem)
Writes an element tree or element structure to sys.stdout. This function should be used for debugging only.

The exact output format is implementation dependent. In this version, it's written as an ordinary XML file.

elem is an element tree or an individual element.

Element(tag[, attrib][, **extra])
Element factory. This function returns an object implementing the standard Element interface. The exact class or type of that object is implementation dependent, but it will always be compatible with the _ElementInterface class in this module.

The element name, attribute names, and attribute values can be either 8-bit ASCII strings or Unicode strings. tag is the element name. attrib is an optional dictionary, containing element attributes. extra contains additional attributes, given as keyword arguments.

Returns:
An element instance.

fromstring(text)
 Parses an XML section from a string constant. Same as XML. text is a string containing XML data.
**Returns:**
An Element instance.

**iselement** *(element)*
Checks if an object appears to be a valid element object. *element* is an element instance.

**Returns:**
A true value if this is an element object.

**iterparse** *(source[, events])*
Parses an XML section into an element tree incrementally, and reports what's going on to the user. *source* is a filename or file object containing XML data. *events* is a list of events to report back. If omitted, only ``end`` events are reported.

**Returns:**
A (event, elem) iterator.

**parse** *(source[, parser])*
Parses an XML section into an element tree. *source* is a filename or file object containing XML data. *parser* is an optional parser instance. If not given, the standard XMLTreeBuilder parser is used.

**Returns:**
An ElementTree instance

**ProcessingInstruction** *(target[, text])*
PI element factory. This factory function creates a special element that will be serialized as an XML processing instruction. *target* is a string containing the PI target. *text* is a string containing the PI contents, if given.

**Returns:**
An element instance, representing a PI.

**SubElement** *(parent, tag[, attrib] [, **extra])*
Subelement factory. This function creates an element instance, and appends it to an existing element.
The element name, attribute names, and attribute values can be either 8-bit ASCII strings or Unicode strings. `parent` is the parent element. `tag` is the subelement name. `attrib` is an optional dictionary, containing element attributes. `extra` contains additional attributes, given as keyword arguments.

**Returns:**
An element instance.

`tostring(element[, encoding])`
Generates a string representation of an XML element, including all subelements. `element` is an Element instance. `encoding` is the output encoding (default is US-ASCII).

**Returns:**
An encoded string containing the XML data.

`XML(text)`
Parses an XML section from a string constant. This function can be used to embed ``XML literals'' in Python code. `text` is a string containing XML data.

**Returns:**
An Element instance.

`XMLID(text)`
Parses an XML section from a string constant, and also returns a dictionary which maps from element id:s to elements. `text` is a string containing XML data.

**Returns:**
A tuple containing an Element instance and a dictionary.
8.13.2 ElementTree Objects

class ElementTree( [element,] [file])

ElementTree wrapper class. This class represents an entire element hierarchy, and adds some extra support for serialization to and from standard XML.

element is the root element. The tree is initialized with the contents of the XML file if given.

_setroot( element)

Replaces the root element for this tree. This discards the current contents of the tree, and replaces it with the given element. Use with care. element is an element instance.

find( path)

Finds the first toplevel element with given tag. Same as getroot().find(path).

path is the element to look for.

Returns:
The first matching element, or None if no element was found.

findall( path)

Finds all toplevel elements with the given tag. Same as getroot().findall(path).

path is the element to look for.

Returns:
A list or iterator containing all matching elements, in section order.

findtext( path[, default])

Finds the element text for the first toplevel element with given tag. Same as getroot().findtext(path).

path is the toplevel element to look for. default is the value to return if the element was not found.

Returns:
The text content of the first matching element, or the default value no
element was found. Note that if the element has is found, but has no text content, this method returns an empty string.

**getiterator**([*tag]*)

Creates a tree iterator for the root element. The iterator loops over all elements in this tree, in section order. *tag* is the tag to look for (default is to return all elements)

**Returns:**
An iterator.

**getroot()**

Gets the root element for this tree.

**Returns:**
An element instance.

**parse**([*source*, *parser]*)

Loads an external XML section into this element tree. *source* is a file name or file object. *parser* is an optional parser instance. If not given, the standard XMLTreeBuilder parser is used.

**Returns:**
The section root element.

**write**([*file*, *encoding]*)

Writes the element tree to a file, as XML. *file* is a file name, or a file object opened for writing. *encoding* is the output encoding (default is US-ASCII).
8.13.3 QName Objects

```python
class QName(text_or_uri[, tag])
```

QName wrapper. This can be used to wrap a QName attribute value, in order to get proper namespace handling on output. `text_or_uri` is a string containing the QName value, in the form `{uri}local`, or, if the tag argument is given, the URI part of a QName. If `tag` is given, the first argument is interpreted as an URI, and this argument is interpreted as a local name.

**Returns:**
An opaque object, representing the QName.
8.13.4 TreeBuilder Objects

class TreeBuilder([element_factory])

Generic element structure builder. This builder converts a sequence of start, data, and end method calls to a well-formed element structure. You can use this class to build an element structure using a custom XML parser, or a parser for some other XML-like format. The element_factory is called to create new Element instances when given.

close()
Flushes the parser buffers, and returns the toplevel document element.

Returns:
An Element instance.

data(data)
Adds text to the current element. data is a string. This should be either an 8-bit string containing ASCII text, or a Unicode string.

data(data)

end(tag)
Closes the current element. tag is the element name.

Returns:
The closed element.

start(tag, attrs)
Opens a new element. tag is the element name. attrs is a dictionary containing element attributes.

Returns:
The opened element.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
8.13.5 XMLTreeBuilder Objects

```python
class XMLTreeBuilder([html], [target])
```
Element structure builder for XML source data, based on the expat parser. `html` are predefined HTML entities. This flag is not supported by the current implementation. `target` is the target object. If omitted, the builder uses an instance of the standard TreeBuilder class.

```python
close()
```
Finishes feeding data to the parser.

**Returns:**
- An element structure.

```python
doctype(name, pubid, system)
```
Handles a doctype declaration. `name` is the doctype name. `pubid` is the public identifier. `system` is the system identifier.

```python
feed(data)
```
Feeds data to the parser.

`data` is encoded data.
9. File Formats

The modules described in this chapter parse various miscellaneous file formats that aren't markup languages or are related to e-mail.

- **csv**: Write and read tabular data to and from delimited files.
- **ConfigParser**: Configuration file parser.
- **robotparser**: Loads a `robots.txt` file and answers questions about fetchability of other URLs.
- **netrc**: Loading of `.netrc` files.
- **xdrlib**: Encoders and decoders for the External Data Representation (XDR).

---

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
# 9.1 csv -- CSV File Reading and Writing

New in version 2.3.

The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. There is no ``CSV standard'', so the format is operationally defined by the many applications which read and write it. The lack of a standard means that subtle differences often exist in the data produced and consumed by different applications. These differences can make it annoying to process CSV files from multiple sources. Still, while the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

The csv module implements classes to read and write tabular data in CSV format. It allows programmers to say, ``write this data in the format preferred by Excel,'' or ``read data from this file which was generated by Excel,'' without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

The csv module's reader and writer objects read and write sequences. Programmers can also read and write data in dictionary form using the DictReader and DictWriter classes.

---

**Note:** This version of the csv module doesn't support Unicode input. Also, there are currently some issues regarding ASCII NUL characters. Accordingly, all input should be UTF-8 or printable ASCII to be safe; see the examples in section [9.1.5](#). These restrictions will be removed in the future.

---

**See Also:**
PEP 305, CSV File API

The Python Enhancement Proposal which proposed this addition to Python.

Subsections

- 9.1.1 Module Contents
- 9.1.2 Dialects and Formatting Parameters
- 9.1.3 Reader Objects
- 9.1.4 Writer Objects
- 9.1.5 Examples

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
9.1.1 Module Contents

The csv module defines the following functions:

**reader** (*csvfile*, *dialect='excel'*, *fmtparam*)

Return a reader object which will iterate over lines in the given *csvfile*. *csvfile* can be any object which supports the iterator protocol and returns a string each time its *next* method is called -- file objects and list objects are both suitable. If *csvfile* is a file object, it must be opened with the 'b' flag on platforms where that makes a difference. An optional *dialect* parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the *Dialect* class or one of the strings returned by the *list_dialects* function. The other optional *fmtparam* keyword arguments can be given to override individual formatting parameters in the current dialect. For more information about the dialect and formatting parameters, see section 9.1.2, "Dialects and Formatting Parameters" for details of these parameters.

All data read are returned as strings. No automatic data type conversion is performed.

Changed in version 2.5: The parser is now stricter with respect to multi-line quoted fields. Previously, if a line ended within a quoted field without a terminating newline character, a newline would be inserted into the returned field. This behavior caused problems when reading files which contained carriage return characters within fields. The behavior was changed to return the field without inserting newlines. As a consequence, if newlines embedded within fields are important, the input should be split into lines in a manner which preserves the newline characters.

**writer** (*csvfile*, *dialect='excel'*, *fmtparam*)

Return a writer object responsible for converting the user's data into delimited strings on the given file-like object. *csvfile* can be any object with a *write* method. If *csvfile* is a file object, it must be opened with the 'b' flag on platforms where that makes a difference. An optional *dialect* parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the
**Dialect** class or one of the strings returned by the `list_dialects` function. The other optional `fmtparam` keyword arguments can be given to override individual formatting parameters in the current dialect. For more information about the dialect and formatting parameters, see section 9.1.2, ``Dialects and Formatting Parameters'' for details of these parameters. To make it as easy as possible to interface with modules which implement the DB API, the value `None` is written as the empty string. While this isn't a reversible transformation, it makes it easier to dump SQL NULL data values to CSV files without preprocessing the data returned from a `cursor.fetch*()` call. All other non-string data are stringified with `str()` before being written.

```python
register_dialect(name[, dialect][, fmtparam])
```

Associate dialect with `name`. `name` must be a string or Unicode object. The dialect can be specified either by passing a sub-class of `Dialect`, or by `fmtparam` keyword arguments, or both, with keyword arguments overriding parameters of the dialect. For more information about the dialect and formatting parameters, see section 9.1.2, ``Dialects and Formatting Parameters'' for details of these parameters.

```python
unregister_dialect(name)
```

Delete the dialect associated with `name` from the dialect registry. An `Error` is raised if `name` is not a registered dialect name.

```python
get_dialect(name)
```

Return the dialect associated with `name`. An `Error` is raised if `name` is not a registered dialect name.

```python
list_dialects()
```

Return the names of all registered dialects.

```python
field_size_limit([new_limit])
```

Returns the current maximum field size allowed by the parser. If `new_limit` is given, this becomes the new limit. New in version 2.5.

The `csv` module defines the following classes:
Create an object which operates like a regular reader but maps the information read into a dict whose keys are given by the optional fieldnames parameter. If the fieldnames parameter is omitted, the values in the first row of the csvfile will be used as the fieldnames. If the row read has fewer fields than the fieldnames sequence, the value of restval will be used as the default value. If the row read has more fields than the fieldnames sequence, the remaining data is added as a sequence keyed by the value of restkey. If the row read has fewer fields than the fieldnames sequence, the remaining keys take the value of the optional restval parameter. Any other optional or keyword arguments are passed to the underlying reader instance.

Create an object which operates like a regular writer but maps dictionaries onto output rows. The fieldnames parameter identifies the order in which values in the dictionary passed to the writerow() method are written to the csvfile. The optional restval parameter specifies the value to be written if the dictionary is missing a key in fieldnames. If the dictionary passed to the writerow() method contains a key not found in fieldnames, the optional extrasaction parameter indicates what action to take. If it is set to 'raise' a ValueError is raised. If it is set to 'ignore', extra values in the dictionary are ignored. Any other optional or keyword arguments are passed to the underlying writer instance.

Note that unlike the DictReader class, the fieldnames parameter of the DictWriter is not optional. Since Python's dict objects are not ordered, there is not enough information available to deduce the order in which the row should be written to the csvfile.

The Dialect class is a container class relied on primarily for its attributes, which are used to define the parameters for a specific reader or writer.
writer instance.

**class excel()**

The `excel` class defines the usual properties of an Excel-generated CSV file.

**class excel_tab()**

The `excel_tab` class defines the usual properties of an Excel-generated TAB-delimited file.

**class Sniffer()**

The `Sniffer` class is used to deduce the format of a CSV file.

The `Sniffer` class provides two methods:

**sniff(sample[, delimiters=None])**

Analyze the given `sample` and return a `Dialect` subclass reflecting the parameters found. If the optional `delimiters` parameter is given, it is interpreted as a string containing possible valid delimiter characters.

**has_header(sample)**

Analyze the sample text (presumed to be in CSV format) and return `True` if the first row appears to be a series of column headers.

The `csv` module defines the following constants:

**QUOTE_ALL**

Instructs `writer` objects to quote all fields.

**QUOTE_MINIMAL**

Instructs `writer` objects to only quote those fields which contain special characters such as `delimiter`, `quotechar` or any of the characters in `lineterminator`.

**QUOTE_NONNUMERIC**

Instructs `writer` objects to quote all non-numeric fields.

Instructs the reader to convert all non-quoted fields to type `float`. 
**QUOTE_NONE**

Instructs writer objects to never quote fields. When the current delimiter occurs in output data it is preceded by the current escapechar character. If escapechar is not set, the writer will raise Error if any characters that require escaping are encountered.

Instructs reader to perform no special processing of quote characters.

The csv module defines the following exception:

**exception Error**

Raised by any of the functions when an error is detected.
9.1.2 Dialects and Formatting Parameters

To make it easier to specify the format of input and output records, specific formatting parameters are grouped together into dialects. A dialect is a subclass of the Dialect class having a set of specific methods and a single validate() method. When creating reader or writer objects, the programmer can specify a string or a subclass of the Dialect class as the dialect parameter. In addition to, or instead of, the dialect parameter, the programmer can also specify individual formatting parameters, which have the same names as the attributes defined below for the Dialect class.

Dialects support the following attributes:

**delimiter**
A one-character string used to separate fields. It defaults to ', '.

**doublequote**
Controls how instances of quotecchar appearing inside a field should be themselves be quoted. When True, the character is doubled. When False, the escapechar is used as a prefix to the quotecchar. It defaults to True.

On output, if doublequote is False and no escapechar is set, Error is raised if a quotecchar is found in a field.

**escapechar**
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotecchar if doublequote is False. On reading, the escapechar removes any special meaning from the following character. It defaults to None, which disables escaping.

**lineterminator**
The string used to terminate lines produced by the writer. It defaults to '\r\n'.

**Note:** The reader is hard-coded to recognise either '\r' or '\n' as end-of-line, and ignores lineterminator. This behavior may change in the
future.

quotechar
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '\"'.

quoting
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section 9.1.1) and defaults to QUOTE_MINIMAL.

skipinitialspace
When True, whitespace immediately following the delimiter is ignored. The default is False.
9.1.3 Reader Objects

Reader objects (DictReader instances and objects returned by the reader() function) have the following public methods:

next()
    Return the next row of the reader's iterable object as a list, parsed according to the current dialect.

Reader objects have the following public attributes:

dialect
    A read-only description of the dialect in use by the parser.

line_num
    The number of lines read from the source iterator. This is not the same as the number of records returned, as records can span multiple lines.
9.1.4 Writer Objects

Writer objects (DictWriter instances and objects returned by the writer() function) have the following public methods. A row must be a sequence of strings or numbers for Writer objects and a dictionary mapping fieldnames to strings or numbers (by passing them through str() first) for DictWriter objects. Note that complex numbers are written out surrounded by parens. This may cause some problems for other programs which read CSV files (assuming they support complex numbers at all).

**writerow(row)**
Write the row parameter to the writer's file object, formatted according to the current dialect.

**writerows(rows)**
Write all the rows parameters (a list of row objects as described above) to the writer's file object, formatted according to the current dialect.

Writer objects have the following public attribute:

**dialect**
A read-only description of the dialect in use by the writer.
9.1.5 Examples

The simplest example of reading a CSV file:

```python
import csv
reader = csv.reader(open("some.csv", "rb"))
for row in reader:
    print row
```

Reading a file with an alternate format:

```python
import csv
reader = csv.reader(open("passwd", "rb"), delimiter=':', quoting=csv.QUOTE_NONE)
for row in reader:
    print row
```

The corresponding simplest possible writing example is:

```python
import csv
writer = csv.writer(open("some.csv", "wb"))
writer.writerows(someiterable)
```

Registering a new dialect:

```python
import csv

csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
reader = csv.reader(open("passwd", "rb"), 'unixpwd')
```

A slightly more advanced use of the reader -- catching and reporting errors:

```python
import csv, sys
filename = "some.csv"
reader = csv.reader(open(filename, "rb"))
try:
    for row in reader:
        print row
except csv.Error, e:
    sys.exit('file %s, line %d: %s' % (filename, reader.line_num, e))
```

And while the module doesn't directly support parsing strings, it can easily be done:

```python
import csv
```
for row in csv.reader(["one,two,three"]):
    print row

The **csv** module doesn't directly support reading and writing Unicode, but it is 8-bit-clean save for some problems with ASCII NUL characters. So you can write functions or classes that handle the encoding and decoding for you as long as you avoid encodings like UTF-16 that use NULs. UTF-8 is recommended.

**unicode_csv_reader** below is a generator that wraps **csv.reader** to handle Unicode CSV data (a list of Unicode strings). **utf_8_encoder** is a generator that encodes the Unicode strings as UTF-8, one string (or row) at a time. The encoded strings are parsed by the CSV reader, and **unicode_csv_reader** decodes the UTF-8-encoded cells back into Unicode:

```python
import csv
def unicode_csv_reader(unicode_csv_data, dialect=csv.excel, **kwargs):
    # csv.py doesn't do Unicode; encode temporarily as UTF-8:
    csv_reader = csv.reader(utf_8_encoder(unicode_csv_data),
                            dialect=dialect, **kwargs)
    for row in csv_reader:
        # decode UTF-8 back to Unicode, cell by cell:
        yield [unicode(cell, 'utf-8') for cell in row]

def utf_8_encoder(unicode_csv_data):
    for line in unicode_csv_data:
        yield line.encode('utf-8')
```

For all other encodings the following **UnicodeReader** and **UnicodeWriter** classes can be used. They take an additional *encoding* parameter in their constructor and make sure that the data passes the real reader or writer encoded as UTF-8:

```python
import csv, codecs, cStringIO
class UTF8Recoder:
    """
    Iterator that reads an encoded stream and reencodes the input to """
    def __init__(self, f, encoding):
        self.reader = codecs.getreader(encoding)(f)

    def __iter__(self):
        return self

    def next(self):
        return self.reader.next().encode("utf-8")
```
class UnicodeReader:
    ""
    A CSV reader which will iterate over lines in the CSV file "f",
    which is encoded in the given encoding.
    ""

def __init__(self, f, dialect=csv.excel, encoding="utf-8", **kwd)
    f = UTF8Recoder(f, encoding)
    self.reader = csv.reader(f, dialect=dialect, **kwd)

def next(self):
    row = self.reader.next()
    return [unicode(s, "utf-8") for s in row]

def __iter__(self):
    return self

class UnicodeWriter:
    ""
    A CSV writer which will write rows to CSV file "f",
    which is encoded in the given encoding.
    ""

def __init__(self, f, dialect=csv.excel, encoding="utf-8", **kwd)
    # Redirect output to a queue
    self.queue = cStringIO.StringIO()
    self.writer = csv.writer(self.queue, dialect=dialect, **kwd)
    self.stream = f
    self.encoder = codecs.getincrementalencoder(encoding)()

def writerow(self, row):
    self.writer.writerow([s.encode("utf-8") for s in row])
    # Fetch UTF-8 output from the queue ...
    data = self.queue.getvalue()
    data = data.decode("utf-8")
    # ... and reencode it into the target encoding
    data = self.encoder.encode(data)
    # write to the target stream
    self.stream.write(data)
    # empty queue
    self.queue.truncate(0)

def writerows(self, rows):
    for row in rows:
        self.writerow(row)
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
9.2 ConfigParser -- Configuration file parser

This module defines the class ConfigParser. The ConfigParser class implements a basic configuration file parser language which provides a structure similar to what you would find on Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

Warning: This library does not interpret or write the value-type prefixes used in the Windows Registry extended version of INI syntax.

The configuration file consists of sections, led by a "[section]" header and followed by "name: value" entries, with continuations in the style of RFC 822; "name=value" is also accepted. Note that leading whitespace is removed from values. The optional values can contain format strings which refer to other values in the same section, or values in a special DEFAULT section. Additional defaults can be provided on initialization and retrieval. Lines beginning with "#" or ";;" are ignored and may be used to provide comments.

For example:

```
[My Section]
foodir: %(dir)s/whatever
dir=frob
```

would resolve the "%(dir)s" to the value of "dir" ("frob" in this case). All reference expansions are done on demand.

Default values can be specified by passing them into the ConfigParser constructor as a dictionary. Additional defaults may be passed into the get() method which will override all others.

```
class RawConfigParser([defaults])
```
The basic configuration object. When defaults is given, it is initialized into the dictionary of intrinsic defaults. This class does not support the magical interpolation behavior. New in version 2.3.

**class ConfigParser([defaults])**

Derived class of RawConfigParser that implements the magical interpolation feature and adds optional arguments to the get() and items() methods. The values in defaults must be appropriate for the "% ( )s" string interpolation. Note that __name__ is an intrinsic default; its value is the section name, and will override any value provided in defaults.

All option names used in interpolation will be passed through the optionxform() method just like any other option name reference. For example, using the default implementation of optionxform() (which converts option names to lower case), the values "foo % (bar)s" and "foo % (BAR)s" are equivalent.

**class SafeConfigParser([defaults])**

Derived class of ConfigParser that implements a more-sane variant of the magical interpolation feature. This implementation is more predictable as well. New applications should prefer this version if they don't need to be compatible with older versions of Python. New in version 2.3.

**exception NoSectionError**

Exception raised when a specified section is not found.

**exception DuplicateSectionError**

Exception raised if add_section() is called with the name of a section that is already present.

**exception NoOptionError**

Exception raised when a specified option is not found in the specified section.

**exception InterpolationError**

Base class for exceptions raised when problems occur performing string interpolation.
exception InterpolationDepthError
   Exception raised when string interpolation cannot be completed because the number of iterations exceeds \texttt{MAX\_INTERPOLATION\_DEPTH}. Subclass of \texttt{InterpolationError}.

exception InterpolationMissingOptionError
   Exception raised when an option referenced from a value does not exist. Subclass of \texttt{InterpolationError}. New in version 2.3.

exception InterpolationSyntaxError
   Exception raised when the source text into which substitutions are made does not conform to the required syntax. Subclass of \texttt{InterpolationError}. New in version 2.3.

exception MissingSectionHeaderError
   Exception raised when attempting to parse a file which has no section headers.

exception ParsingError
   Exception raised when errors occur attempting to parse a file.

\texttt{MAX\_INTERPOLATION\_DEPTH}
   The maximum depth for recursive interpolation for \texttt{get()} when the \texttt{raw} parameter is false. This is relevant only for the \texttt{ConfigParser} class.

\textbf{See Also:}

Module \texttt{shlex}: Support for a creating UNIX shell-like mini-languages which can be used as an alternate format for application configuration files.

\textbf{Subsections}

- \texttt{9.2.1 RawConfigParser Objects}
- \texttt{9.2.2 ConfigParser Objects}
- \texttt{9.2.3 SafeConfigParser Objects}
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
### 9.2.1 RawConfigParser Objects

RawConfigParser instances have the following methods:

**defaults()**
- Return a dictionary containing the instance-wide defaults.

**sections()**
- Return a list of the sections available; DEFAULT is not included in the list.

**add_section(section)**
- Add a section named `section` to the instance. If a section by the given name already exists, DuplicateSectionError is raised.

**has_section(section)**
- Indicates whether the named section is present in the configuration. The DEFAULT section is not acknowledged.

**options(section)**
- Returns a list of options available in the specified `section`.

**has_option(section, option)**
- If the given section exists, and contains the given option, return `True`; otherwise return `False`. New in version 1.6.

**read(filenames)**
- Attempt to read and parse a list of filenames, returning a list of filenames which were successfully parsed. If `filenames` is a string or Unicode string, it is treated as a single filename. If a file named in `filenames` cannot be opened, that file will be ignored. This is designed so that you can specify a list of potential configuration file locations (for example, the current directory, the user's home directory, and some system-wide directory), and all existing configuration files in the list will be read. If none of the named files exist, the ConfigParser instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using `readfp()` before calling `read()` for any
optional files:

```python
import ConfigParser, os
config = ConfigParser.ConfigParser()
config.readfp(open('defaults.cfg'))
config.read(["site.cfg", os.path.expanduser('~/.myapp.cfg')])
```

Changed in version 2.4: Returns list of successfully parsed filenames.

**readfp**(fp[, filename])

Read and parse configuration data from the file or file-like object in fp (only the **readline()** method is used). If filename is omitted and fp has a name attribute, that is used for filename; the default is "<????>".

**get**(section, option)

Get an option value for the named section.

**getint**(section, option)

A convenience method which coerces the option in the specified section to an integer.

**getfloat**(section, option)

A convenience method which coerces the option in the specified section to a floating point number.

**getboolean**(section, option)

A convenience method which coerces the option in the specified section to a Boolean value. Note that the accepted values for the option are "1", "yes", "true", and "on", which cause this method to return True, and "0", "no", "false", and "off", which cause it to return False. These string values are checked in a case-insensitive manner. Any other value will cause it to raise ValueError.

**items**(section)

Return a list of (name, value) pairs for each option in the given section.

**set**(section, option, value)

If the given section exists, set the given option to the specified value;
otherwise raise NoSectionError. While it is possible to use RawConfigParser (or ConfigParser with raw parameters set to true) for internal storage of non-string values, full functionality (including interpolation and output to files) can only be achieved using string values. New in version 1.6.

**write(fileobject)**
Write a representation of the configuration to the specified file object. This representation can be parsed by a future read() call. New in version 1.6.

**remove_option(section, option)**
Remove the specified option from the specified section. If the section does not exist, raise NoSectionError. If the option existed to be removed, return True; otherwise return False. New in version 1.6.

**remove_section(section)**
Remove the specified section from the configuration. If the section in fact existed, return True. Otherwise return False.

**optionxform(option)**
Transforms the option name option as found in an input file or as passed in by client code to the form that should be used in the internal structures. The default implementation returns a lower-case version of option; subclasses may override this or client code can set an attribute of this name on instances to affect this behavior. Setting this to str(), for example, would make option names case sensitive.
9.2.2 ConfigParser Objects

The ConfigParser class extends some methods of the RawConfigParser interface, adding some optional arguments.

get( section, option[, raw[, vars]])

Get an option value for the named section. All the "%" interpolations are expanded in the return values, based on the defaults passed into the constructor, as well as the options vars provided, unless the raw argument is true.

items( section[, raw[, vars]])

Return a list of (name, value) pairs for each option in the given section. Optional arguments have the same meaning as for the get() method. New in version 2.3.
9.2.3 SafeConfigParser Objects

The SafeConfigParser class implements the same extended interface as ConfigParser, with the following addition:

set( section, option, value)

If the given section exists, set the given option to the specified value; otherwise raise NoSectionError. value must be a string (str or unicode); if not, TypeError is raised. New in version 2.4.
9.3 robotparser -- Parser for robots.txt

This module provides a single class, RobotFileParser, which answers questions about whether or not a particular user agent can fetch a URL on the Web site that published the robots.txt file. For more details on the structure of robots.txt files, see http://www.robotstxt.org/wc/norobots.html.

class RobotFileParser()

This class provides a set of methods to read, parse and answer questions about a single robots.txt file.

set_url(url)
Sets the URL referring to a robots.txt file.

read()
Reads the robots.txt URL and feeds it to the parser.

parse(lines)
Parses the lines argument.

can_fetch(useragent, url)
Returns True if the useragent is allowed to fetch the url according to the rules contained in the parsed robots.txt file.

mtime()
Returns the time the robots.txt file was last fetched. This is useful for long-running web spiders that need to check for new robots.txt files periodically.

modified()
Sets the time the robots.txt file was last fetched to the current time.
The following example demonstrates basic use of the RobotFileParser class.

```python
>>> import robotparser
>>> rp = robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```
9.4 netrc -- netrc file processing

New in version 1.5.2.

The netrc class parses and encapsulates the netrc file format used by the UNIX ftp program and other FTP clients.

class netrc([file])

A netrc instance or subclass instance encapsulates data from a netrc file. The initialization argument, if present, specifies the file to parse. If no argument is given, the file .netrc in the user's home directory will be read. Parse errors will raise NetrcParseError with diagnostic information including the file name, line number, and terminating token.

def parse(self, line):
    """""

exception NetrcParseError

Exception raised by the netrc class when syntactical errors are encountered in source text. Instances of this exception provide three interesting attributes: msg is a textual explanation of the error, filename is the name of the source file, and lineno gives the line number on which the error was found.

Subsections

- 9.4.1 netrc Objects
9.4.1 netrc Objects

A `netrc` instance has the following methods:

**authenticators( host)**
Return a 3-tuple `(login, account, password)` of authenticators for `host`. If the netrc file did not contain an entry for the given host, return the tuple associated with the `default' entry. If neither matching host nor default entry is available, return `None`.

**__repr__**( )
Dump the class data as a string in the format of a netrc file. (This discards comments and may reorder the entries.)

Instances of `netrc` have public instance variables:

**hosts**
Dictionary mapping host names to `(login, account, password)` tuples. The `default' entry, if any, is represented as a pseudo-host by that name.

**macros**
Dictionary mapping macro names to string lists.

**Note:** Passwords are limited to a subset of the ASCII character set. Versions of this module prior to 2.3 were extremely limited. Starting with 2.3, all ASCII punctuation is allowed in passwords. However, note that whitespace and non-printable characters are not allowed in passwords. This is a limitation of the way the .netrc file is parsed and may be removed in the future.
9.5 xdrlib -- Encode and decode XDR data

The xdrlib module supports the External Data Representation Standard as described in RFC 1014, written by Sun Microsystems, Inc. June 1987. It supports most of the data types described in the RFC.

The xdrlib module defines two classes, one for packing variables into XDR representation, and another for unpacking from XDR representation. There are also two exception classes.

**class Packer()**

Packer is the class for packing data into XDR representation. The Packer class is instantiated with no arguments.

**class Unpacker(data)**

Unpacker is the complementary class which unpacks XDR data values from a string buffer. The input buffer is given as data.

---

**See Also:**

[RFC 1014, XDR: External Data Representation Standard](#)

This RFC defined the encoding of data which was XDR at the time this module was originally written. It has apparently been obsoleted by RFC 1832.

[RFC 1832, XDR: External Data Representation Standard](#)

Newer RFC that provides a revised definition of XDR.

---

**Subsections**

- 9.5.1 Packer Objects
- 9.5.2 Unpacker Objects
9.5.3 Exceptions

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
9.5.1 Packer Objects

Packer instances have the following methods:

**get_buffer()**
Returns the current pack buffer as a string.

**reset()**
Resets the pack buffer to the empty string.

In general, you can pack any of the most common XDR data types by calling the appropriate `pack_type()` method. Each method takes a single argument, the value to pack. The following simple data type packing methods are supported: `pack_uint()`, `pack_int()`, `pack_enum()`, `pack_bool()`, `pack_uhyper()`, and `pack_hyper()`.

**pack_float(value)**
Packs the single-precision floating point number `value`.

**pack_double(value)**
Packs the double-precision floating point number `value`.

The following methods support packing strings, bytes, and opaque data:

**pack_fstring(n, s)**
Packs a fixed length string, `s`. `n` is the length of the string but it is *not* packed into the data buffer. The string is padded with null bytes if necessary to guaranteed 4 byte alignment.

**pack_fopaque(n, data)**
Packs a fixed length opaque data stream, similarly to `pack_fstring()`.

**pack_string(s)**
Packs a variable length string, `s`. The length of the string is first packed as an unsigned integer, then the string data is packed with `pack_fstring()`.
pack_opaque( data)
   Packs a variable length opaque data string, similarly to pack_string().

pack_bytes( bytes)
   Packs a variable length byte stream, similarly to pack_string().

The following methods support packing arrays and lists:

pack_list( list, pack_item)
   Packs a list of homogeneous items. This method is useful for lists with an
   indeterminate size; i.e. the size is not available until the entire list has been
   walked. For each item in the list, an unsigned integer 1 is packed first,
   followed by the data value from the list. pack_item is the function that is
   called to pack the individual item. At the end of the list, an unsigned integer
   0 is packed.

   For example, to pack a list of integers, the code might appear like this:

   import xdrlib
   p = xdrlib.Packer()
   p.pack_list([1, 2, 3], p.pack_int)

pack_farray( n, array, pack_item)
   Packs a fixed length list (array) of homogeneous items. n is the length of
   the list; it is not packed into the buffer, but a ValueError exception is
   raised if len(array) is not equal to n. As above, pack_item is the function
   used to pack each element.

pack_array( list, pack_item)
   Packs a variable length list of homogeneous items. First, the length of the
   list is packed as an unsigned integer, then each element is packed as in
   pack_farray() above.
9.5.2 Unpacker Objects

The Unpacker class offers the following methods:

**reset**( data )
Resets the string buffer with the given *data*.

**get_position**()
Returns the current unpack position in the data buffer.

**set_position**( position )
Sets the data buffer unpack position to *position*. You should be careful about using **get_position**() and **set_position**().

**get_buffer**()
Returns the current unpack data buffer as a string.

**done**()
Indicates unpack completion. Raises an *Error* exception if all of the data has not been unpacked.

In addition, every data type that can be packed with a Packer, can be unpacked with an Unpacker. Unpacking methods are of the form **unpack_type**(), and take no arguments. They return the unpacked object.

**unpack_float**( )
Unpacks a single-precision floating point number.

**unpack_double**( )
Unpacks a double-precision floating point number, similarly to **unpack_float**( ).

In addition, the following methods unpack strings, bytes, and opaque data:

**unpack_fstring**( n )
Unpacks and returns a fixed length string. *n* is the number of characters
expected. Padding with null bytes to guaranteed 4 byte alignment is assumed.

**unpack_fopaque**(*n*)
Unpacks and returns a fixed length opaque data stream, similarly to `unpack_fstring()`.

**unpack_string()**
Unpacks and returns a variable length string. The length of the string is first unpacked as an unsigned integer, then the string data is unpacked with `unpack_fstring()`.

**unpack_opaque()**
Unpacks and returns a variable length opaque data string, similarly to `unpack_string()`.

**unpack_bytes()**
Unpacks and returns a variable length byte stream, similarly to `unpack_string()`.

The following methods support unpacking arrays and lists:

**unpack_list**(unpack_item)
Unpacks and returns a list of homogeneous items. The list is unpacked one element at a time by first unpacking an unsigned integer flag. If the flag is 1, then the item is unpacked and appended to the list. A flag of 0 indicates the end of the list. `unpack_item` is the function that is called to unpack the items.

**unpack_farray**(n, unpack_item)
Unpacks and returns (as a list) a fixed length array of homogeneous items. *n* is number of list elements to expect in the buffer. As above, `unpack_item` is the function used to unpack each element.

**unpack_array**(unpack_item)
Unpacks and returns a variable length list of homogeneous items. First, the length of the list is unpacked as an unsigned integer, then each element is
unpacked as in `unpack_farray()` above.
9.5.3 Exceptions

Exceptions in this module are coded as class instances:

exception Error
   The base exception class. Error has a single public data member msg
   containing the description of the error.

exception ConversionError
   Class derived from Error. Contains no additional instance variables.

Here is an example of how you would catch one of these exceptions:

```python
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError, instance:
    print 'packing the double failed:', instance.msg
```
10. Cryptographic Services

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation. Here's an overview:

- **hashlib**: Secure hash and message digest algorithms.
- **hmac**: Keyed-Hashing for Message Authentication (HMAC) implementation for Python.
- **md5**: RSA's MD5 message digest algorithm.
- **sha**: NIST's secure hash algorithm, SHA.

Hardcore cypherpunks will probably find the cryptographic modules written by A.M. Kuchling of further interest; the package contains modules for various encryption algorithms, most notably AES. These modules are not distributed with Python but available separately. See the URL [http://www.amk.ca/python/code/crypto.html](http://www.amk.ca/python/code/crypto.html) for more information.
10.1 hashlib -- Secure hashes and message digests

New in version 2.5.

This module implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512 (defined in FIPS 180-2) as well as RSA's MD5 algorithm (defined in Internet RFC 1321). The terms secure hash and message digest are interchangeable. Older algorithms were called message digests. The modern term is secure hash.

**Warning:** Some algorithms have known hash collision weaknesses, see the FAQ at the end.

There is one constructor method named for each type of hash. All return a hash object with the same simple interface. For example: use sha1() to create a SHA1 hash object. You can now feed this object with arbitrary strings using the update() method. At any point you can ask it for the digest of the concatenation of the strings fed to it so far using the digest() or hexdigest() methods.

Constructors for hash algorithms that are always present in this module are md5(), sha1(), sha224(), sha256(), sha384(), and sha512(). Additional algorithms may also be available depending upon the OpenSSL library that Python uses on your platform.

For example, to obtain the digest of the string 'Nobody inspects the spammish repetition':

```python
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
```

More condensed:
A generic `new()` constructor that takes the string name of the desired algorithm as its first parameter also exists to allow access to the above listed hashes as well as any other algorithms that your OpenSSL library may offer. The named constructors are much faster than `new()` and should be preferred.

Using `new()` with an algorithm provided by OpenSSL:

```python
c >>> h = hashlib.new('ripemd160')
c >>> h.update("Nobody inspects the spammish repetition")
c >>> h.hexdigest()
c 'cc4a5ce1b3df48aeec5d22d1f16b894a0b894ecc'
```

The following values are provided as constant attributes of the hash objects returned by the constructors:

- `digest_size`
  The size of the resulting digest in bytes.

A hash object has the following methods:

- `update(arg)`
  Update the hash object with the string `arg`. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`.

- `digest()`
  Return the digest of the strings passed to the `update()` method so far. This is a string of `digest_size` bytes which may contain non-ASCII characters, including null bytes.

- `hexdigest()`
  Like `digest()` except the digest is returned as a string of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

- `copy()`
  Return a copy ("clone") of the hash object. This can be used to efficiently
compute the digests of strings that share a common initial substring.

See Also:

Module **hmac**:  
A module to generate message authentication codes using hashes.

Module **base64**:  
Another way to encode binary hashes for non-binary environments.

The FIPS 180-2 publication on Secure Hash Algorithms.

http://www.cryptography.com/cnews/hash.html  
Hash Collision FAQ with information on which algorithms have known issues and what that means regarding their use.
10.2 hmac -- Keyed-Hashing for Message Authentication

New in version 2.2.

This module implements the HMAC algorithm as described by RFC 2104.

new( key[, msg[, digestmod]])

Return a new hmac object. If msg is present, the method call update(msg) is made. digestmod is the digest constructor or module for the HMAC object to use. It defaults to the hashlib.md5 constructor. 

Note: The md5 hash has known weaknesses but remains the default for backwards compatibility. Choose a better one for your application.

An HMAC object has the following methods:

update( msg)

Update the hmac object with the string msg. Repeated calls are equivalent to a single call with the concatenation of all the arguments:

m.update(a); m.update(b) is equivalent to m.update(a + b).

digest()

Return the digest of the strings passed to the update() method so far. This string will be the same length as the digest_size of the digest given to the constructor. It may contain non-ASCII characters, including NUL bytes.

hexdigest()

Like digest() except the digest is returned as a string twice the length containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

copy()

Return a copy (``clone``) of the hmac object. This can be used to efficiently compute the digests of strings that share a common initial substring.
See Also:

Module **hashlib**:
The python module providing secure hash functions.
10.3 md5 -- MD5 message digest algorithm

Deprecated since release 2.5. Use the hashlib module instead.

This module implements the interface to RSA's MD5 message digest algorithm (see also Internet RFC 1321). Its use is quite straightforward: use new() to create an md5 object. You can now feed this object with arbitrary strings using the update() method, and at any point you can ask it for the digest (a strong kind of 128-bit checksum, a.k.a. "fingerprint") of the concatenation of the strings fed to it so far using the digest() method.

For example, to obtain the digest of the string 'Nobody inspects the spammish repetition':

```python
>>> import md5
>>> m = md5.new()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\xe1\xe5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
```

More condensed:

```python
>>> md5.new("Nobody inspects the spammish repetition").digest()
'\xbbd\x9c\x83\xdd\xe1\xe5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
```

The following values are provided as constants in the module and as attributes of the md5 objects returned by new():

**digest_size**

The size of the resulting digest in bytes. This is always 16.

The md5 module provides the following functions:

**new([arg])**

Return a new md5 object. If arg is present, the method call update(arg) is made.
md5([arg])

For backward compatibility reasons, this is an alternative name for the new() function.

An md5 object has the following methods:

**update(arg)**

Update the md5 object with the string arg. Repeated calls are equivalent to a single call with the concatenation of all the arguments: m.update(a); m.update(b) is equivalent to m.update(a+b).

**digest()**

Return the digest of the strings passed to the update() method so far. This is a 16-byte string which may contain non-ASCII characters, including null bytes.

**hexdigest()**

Like digest() except the digest is returned as a string of length 32, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

**copy()**

Return a copy ("clone") of the md5 object. This can be used to efficiently compute the digests of strings that share a common initial substring.

---

See Also:

Module [sha]:

Similar module implementing the Secure Hash Algorithm (SHA). The SHA algorithm is considered a more secure hash.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
10.4 sha -- SHA-1 message digest algorithm

Deprecated since release 2.5. Use the hashlib module instead.

This module implements the interface to NIST's secure hash algorithm, known as SHA-1. SHA-1 is an improved version of the original SHA hash algorithm. It is used in the same way as the md5 module: use new() to create an sha object, then feed this object with arbitrary strings using the update() method, and at any point you can ask it for the digest of the concatenation of the strings fed to it so far. SHA-1 digests are 160 bits instead of MD5's 128 bits.

new([string])
   Return a new sha object. If string is present, the method call update(string) is made.

The following values are provided as constants in the module and as attributes of the sha objects returned by new():

blocksize
   Size of the blocks fed into the hash function; this is always 1. This size is used to allow an arbitrary string to be hashed.

digest_size
   The size of the resulting digest in bytes. This is always 20.

An sha object has the same methods as md5 objects:

update(arg)
   Update the sha object with the string arg. Repeated calls are equivalent to a single call with the concatenation of all the arguments: m.update(a); m.update(b) is equivalent to m.update(a+b).

digest()
   Return the digest of the strings passed to the update() method so far.
This is a 20-byte string which may contain non-ASCII characters, including null bytes.

**hexdigest()**
Like `digest()` except the digest is returned as a string of length 40, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

**copy()**
Return a copy (``clone``) of the sha object. This can be used to efficiently compute the digests of strings that share a common initial substring.

---

**See Also:**

- **Secure Hash Standard**

- **Cryptographic Toolkit (Secure Hashing)**
  Links from NIST to various information on secure hashing.
11. File and Directory Access

The modules described in this chapter deal with disk files and directories. For example, there are modules for reading the properties of files, manipulating paths in a portable way, and creating temporary files. The full list of modules in this chapter is:

- **os.path**: Common pathname manipulations.
- **fileinput**: Perl-like iteration over lines from multiple input streams, with "save in place" capability.
- **stat**: Utilities for interpreting the results of `os.stat()`, `os.lstat()` and `os.fstat()`.
- **statvfs**: Constants for interpreting the result of `os.statvfs()`.
- **filecmp**: Compare files efficiently.
- **tempfile**: Generate temporary files and directories.
- **glob**: UNIX shell style pathname pattern expansion.
- **fnmatch**: UNIX shell style filename pattern matching.
- **linecache**: This module provides random access to individual lines from text files.
- **shutil**: High-level file operations, including copying.
- **dircache**: Return directory listing, with cache mechanism.

Also see section 3.9 for a description of Python's built-in file objects.

**See Also:**

Module **os**: Operating system interfaces, including functions to work with files at a lower level than the built-in file object.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.1 os.path -- Common pathname manipulations

This module implements some useful functions on pathnames.

**Warning:** On Windows, many of these functions do not properly support UNC pathnames. `splitunc()` and `ismount()` do handle them correctly.

**abspath**(path)
Return a normalized absolutized version of the pathname `path`. On most platforms, this is equivalent to `normpath(join(os.getcwd(), path))`. New in version 1.5.2.

**basename**(path)
Return the base name of pathname `path`. This is the second half of the pair returned by `split(path)`. Note that the result of this function is different from the UNIX `basename` program; where `basename` for '/foo/bar/' returns 'bar', the `basename()` function returns an empty string ('').

**commonprefix**(list)
Return the longest path prefix (taken character-by-character) that is a prefix of all paths in `list`. If `list` is empty, return the empty string (''). Note that this may return invalid paths because it works a character at a time.

**dirname**(path)
Return the directory name of pathname `path`. This is the first half of the pair returned by `split(path)`.

**exists**(path)
Return `True` if `path` refers to an existing path. Returns `False` for broken symbolic links. On some platforms, this function may return `False` if permission is not granted to execute `os.stat()` on the requested file, even if the `path` physically exists.
lexists(path)

Return True if path refers to an existing path. Returns True for broken symbolic links. Equivalent to exists() on platforms lacking os.lstat(). New in version 2.4.

expanduser(path)

On UNIX, return the argument with an initial component of "~" or "~user" replaced by that user's home directory. An initial "~" is replaced by the environment variable HOME if it is set; otherwise the current user's home directory is looked up in the password directory through the built-in module pwd. An initial "~user" is looked up directly in the password directory.

On Windows, only "~" is supported; it is replaced by the environment variable HOME or by a combination of HOMEDRIVE and HOMEPATH.

If the expansion fails or if the path does not begin with a tilde, the path is returned unchanged.

expandvars(path)

Return the argument with environment variables expanded. Substrings of the form "$name" or "${name}" are replaced by the value of environment variable name. Malformed variable names and references to non-existing variables are left unchanged.

getatime(path)

Return the time of last access of path. The return value is a number giving the number of seconds since the epoch (see the time module). Raise os.error if the file does not exist or is inaccessible. New in version 1.5.2. Changed in version 2.3: If os.stat_float_times() returns True, the result is a floating point number.

gmtime(path)

Return the time of last modification of path. The return value is a number giving the number of seconds since the epoch (see the time module). Raise os.error if the file does not exist or is inaccessible. New in version 1.5.2. Changed in version 2.3: If os.stat_float_times() returns True, the result is a floating point number.
getctime(path)
    Return the system's ctime which, on some systems (like UNIX) is the time of
    the last change, and, on others (like Windows), is the creation time for path.
    The return value is a number giving the number of seconds since the epoch
    (see the time module). Raise os.error if the file does not exist or is
    inaccessible. New in version 2.3.

getsize(path)
    Return the size, in bytes, of path. Raise os.error if the file does not exist
    or is inaccessible. New in version 1.5.2.

isabs(path)
    Return True if path is an absolute pathname (begins with a slash).

isfile(path)
    Return True if path is an existing regular file. This follows symbolic links,
    so both islink() and isfile() can be true for the same path.

isdir(path)
    Return True if path is an existing directory. This follows symbolic links,
    so both islink() and isdir() can be true for the same path.

islink(path)
    Return True if path refers to a directory entry that is a symbolic link.
    Always False if symbolic links are not supported.

ismount(path)
    Return True if pathname path is a mount point: a point in a file system
    where a different file system has been mounted. The function checks
    whether path's parent, path/.., is on a different device than path, or whether
    path/.. and path point to the same i-node on the same device -- this should
detect mount points for all UNIX and POSIX variants.

join(path1[, path2[, ...]])
    Join one or more path components intelligently. If any component is an
    absolute path, all previous components (on Windows, including the
    previous drive letter, if there was one) are thrown away, and joining
continues. The return value is the concatenation of path1, and optionally path2, etc., with exactly one directory separator (os.sep) inserted between components, unless path2 is empty. Note that on Windows, since there is a current directory for each drive, os.path.join("c:", "foo") represents a path relative to the current directory on drive C: (c:foo), not c:\foo.

**normcase(path)**

Normalize the case of a pathname. On UNIX, this returns the path unchanged; on case-insensitive filesystems, it converts the path to lowercase. On Windows, it also converts forward slashes to backward slashes.

**normpath(path)**

Normalize a pathname. This collapses redundant separators and up-level references so that A//B, A/.B and A/foo/../B all become A/B. It does not normalize the case (use normcase() for that). On Windows, it converts forward slashes to backward slashes. It should be understood that this may change the meaning of the path if it contains symbolic links!

**realpath(path)**

Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path (if they are supported by the operating system). New in version 2.2.

**samefile(path1, path2)**

Return True if both pathname arguments refer to the same file or directory (as indicated by device number and i-node number). Raise an exception if a os.stat() call on either pathname fails. Availability: Macintosh, UNIX.

**sameopenfile(fp1, fp2)**

Return True if the file descriptors fp1 and fp2 refer to the same file. Availability: Macintosh, UNIX.

**samestat(stat1, stat2)**

Return True if the stat tuples stat1 and stat2 refer to the same file. These structures may have been returned by fstat(), lstat(), or stat().
This function implements the underlying comparison used by `samefile()` and `sameopenfile()`. Availability: Macintosh, UNIX.

**split(path)**

Split the pathname `path` into a pair, `(head, tail)` where `tail` is the last pathname component and `head` is everything leading up to that. The `tail` part will never contain a slash; if `path` ends in a slash, `tail` will be empty. If there is no slash in `path`, `head` will be empty. If `path` is empty, both `head` and `tail` are empty. Trailing slashes are stripped from `head` unless it is the root (one or more slashes only). In nearly all cases, `join(head, tail)` equals `path` (the only exception being when there were multiple slashes separating `head` from `tail`).

**splitdrive(path)**

Split the pathname `path` into a pair `(drive, tail)` where `drive` is either a drive specification or the empty string. On systems which do not use drive specifications, `drive` will always be the empty string. In all cases, `drive + tail` will be the same as `path`. New in version 1.3.

**splitext(path)**

Split the pathname `path` into a pair `(root, ext)` such that `root + ext == path`, and `ext` is empty or begins with a period and contains at most one period.

**splitunc(path)**

Split the pathname `path` into a pair `(unc, rest)` so that `unc` is the UNC mount point (such as `r'\host\mount'`), if present, and `rest` the rest of the path (such as `r'\path\file.ext'`). For paths containing drive letters, `unc` will always be the empty string. Availability: Windows.

**walk(path, visit, arg)**

Calls the function `visit` with arguments `(arg, dirname, names)` for each directory in the directory tree rooted at `path` (including `path` itself, if it is a directory). The argument `dirname` specifies the visited directory, the argument `names` lists the files in the directory (gotten from `os.listdir(dirname)`). The `visit` function may modify `names` to influence the set of directories visited below `dirname`, e.g. to avoid visiting
certain parts of the tree. (The object referred to by *names* must be modified in place, using *del* or slice assignment.)

**Note:** Symbolic links to directories are not treated as subdirectories, and that *walk()* therefore will not visit them. To visit linked directories you must identify them with *os.path.islink(file)* and *os.path.isdir(file)*, and invoke *walk()* as necessary.

**Note:** The newer *os.walk()* generator supplies similar functionality and can be easier to use.

**supports_unicode_filenames**

True if arbitrary Unicode strings can be used as file names (within limitations imposed by the file system), and if *os.listdir()* returns Unicode strings for a Unicode argument. New in version 2.3.
11.2 fileinput -- Iterate over lines from multiple input streams

This module implements a helper class and functions to quickly write a loop over standard input or a list of files.

The typical use is:

```python
import fileinput
for line in fileinput.input():
    process(line)
```

This iterates over the lines of all files listed in `sys.argv[1:]`, defaulting to `sys.stdin` if the list is empty. If a filename is '-' , it is also replaced by `sys.stdin`. To specify an alternative list of filenames, pass it as the first argument to `input()`. A single file name is also allowed.

All files are opened in text mode by default, but you can override this by specifying the `mode` parameter in the call to `input()` or `FileInput()`. If an I/O error occurs during opening or reading a file, `IOError` is raised.

If `sys.stdin` is used more than once, the second and further use will return no lines, except perhaps for interactive use, or if it has been explicitly reset (e.g. using `sys.stdin.seek(0)`).

Empty files are opened and immediately closed; the only time their presence in the list of filenames is noticeable at all is when the last file opened is empty.

It is possible that the last line of a file does not end in a newline character; lines are returned including the trailing newline when it is present.

You can control how files are opened by providing an opening hook via the `openhook` parameter to `input()` or `FileInput()`. The hook must be a function that takes two arguments, `filename` and `mode`, and returns an accordingly opened file-like object. Two useful hooks are already provided by this module.
The following function is the primary interface of this module:

```
input([files[, inplace[, backup[, mode[, openhook]]]]])
```

Create an instance of the `FileInput` class. The instance will be used as global state for the functions of this module, and is also returned to use during iteration. The parameters to this function will be passed along to the constructor of the `FileInput` class.

Changed in version 2.5: Added the `mode` and `openhook` parameters.

The following functions use the global state created by `input()`; if there is no active state, `RuntimeError` is raised.

```
filename()
```

Return the name of the file currently being read. Before the first line has been read, returns `None`.

```
fileno()
```

Return the integer `"file descriptor"` for the current file. When no file is opened (before the first line and between files), returns `-1`. New in version 2.5.

```
lineno()
```

Return the cumulative line number of the line that has just been read. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line.

```
filelineno()
```

Return the line number in the current file. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line within the file.

```
isfirstline()
```

Returns true if the line just read is the first line of its file, otherwise returns false.

```
isstdin()
```
Returns true if the last line was read from `sys.stdin`, otherwise returns false.

**nextfile()**
Close the current file so that the next iteration will read the first line from the next file (if any); lines not read from the file will not count towards the cumulative line count. The filename is not changed until after the first line of the next file has been read. Before the first line has been read, this function has no effect; it cannot be used to skip the first file. After the last line of the last file has been read, this function has no effect.

**close()**
Close the sequence.

The class which implements the sequence behavior provided by the module is available for subclassing as well:

```python
class FileInput([files, inplace, backup, mode, openhook])
```

Class `FileInput` is the implementation; its methods `filename()`, `fileno()`, `lineno()`, `fileline()`, `isfirstline()`, `isstdin()`, `nextfile()` and `close()` correspond to the functions of the same name in the module. In addition it has a `readline()` method which returns the next input line, and a `__getitem__()` method which implements the sequence behavior. The sequence must be accessed in strictly sequential order; random access and `readline()` cannot be mixed.

With `mode` you can specify which file mode will be passed to `open()`. It must be one of `'r'`, `'rU'`, `'U'` and `'rb'`.

The `openhook`, when given, must be a function that takes two arguments, `filename` and `mode`, and returns an accordingly opened file-like object. You cannot use `inplace` and `openhook` together.

Changed in version 2.5: Added the `mode` and `openhook` parameters.

**Optional in-place filtering:** if the keyword argument `inplace=1` is passed to `input()` or to the `FileInput` constructor, the file is moved to a backup file
and standard output is directed to the input file (if a file of the same name as the backup file already exists, it will be replaced silently). This makes it possible to write a filter that rewrites its input file in place. If the keyword argument
backup='.<some extension>' is also given, it specifies the extension for the backup file, and the backup file remains around; by default, the extension is '.bak' and it is deleted when the output file is closed. In-place filtering is disabled when standard input is read.

**Caveat:** The current implementation does not work for MS-DOS 8+3 filesystems.

The two following opening hooks are provided by this module:

**hook_compressed((filename, mode))**

  Transparently opens files compressed with gzip and bzip2 (recognized by the extensions '.gz' and '.bz2') using the gzip and bz2 modules. If the filename extension is not '.gz' or '.bz2', the file is opened normally (ie, using open() without any decompression).

  Usage example: "fi =
  fileinput.FileInput(openhook=fileinput.hook_compressed)

  New in version 2.5.

**hook_encoded((encoding))**

  Returns a hook which opens each file with codecs.open(), using the given encoding to read the file.

  Usage example: "fi =
  fileinput.FileInput(openhook=fileinput.hook_encoded('iso-8859-1'))"

  **Note:** With this hook, FileInput might return Unicode strings depending on the specified encoding. New in version 2.5.
Release 2.5, documentation updated on 19th September, 2006. 
See About this document... for information on suggesting changes.
11.3 stat -- Interpreting stat() results

The stat module defines constants and functions for interpreting the results of os.stat(), os.fstat() and os.lstat() (if they exist). For complete details about the stat(), fstat() and lstat() calls, consult the documentation for your system.

The stat module defines the following functions to test for specific file types:

**S_ISDIR( mode)**  
Return non-zero if the mode is from a directory.

**S_ISCHR( mode)**  
Return non-zero if the mode is from a character special device file.

**S_ISBLK( mode)**  
Return non-zero if the mode is from a block special device file.

**S_ISREG( mode)**  
Return non-zero if the mode is from a regular file.

**S_ISFIFO( mode)**  
Return non-zero if the mode is from a FIFO (named pipe).

**S_ISLNK( mode)**  
Return non-zero if the mode is from a symbolic link.

**S_ISSOCK( mode)**  
Return non-zero if the mode is from a socket.

Two additional functions are defined for more general manipulation of the file's mode:
S_IMODE( mode)
Return the portion of the file's mode that can be set by os.chmod()—that
is, the file's permission bits, plus the sticky bit, set-group-id, and set-user-id
bits (on systems that support them).

S_IFMT( mode)
Return the portion of the file's mode that describes the file type (used by the
S_IS*() functions above).

Normally, you would use the os.path.is*() functions for testing the type of
a file; the functions here are useful when you are doing multiple tests of the
same file and wish to avoid the overhead of the stat() system call for each
test. These are also useful when checking for information about a file that isn't
handled by os.path, like the tests for block and character devices.

All the variables below are simply symbolic indexes into the 10-tuple returned
by os.stat(), os.fstat() or os.lstat().

ST_MODE
Inode protection mode.

ST_INO
Inode number.

ST_DEV
Device inode resides on.

ST_NLINK
Number of links to the inode.

ST_UID
User id of the owner.

ST_GID
Group id of the owner.

ST_SIZE
Size in bytes of a plain file; amount of data waiting on some special files.
**ST_ATIME**
Time of last access.

**ST_MTIME**
Time of last modification.

**ST_CTIME**
The ``ctime'' as reported by the operating system. On some systems (like *UNIX*) is the time of the last metadata change, and, on others (like Windows), is the creation time (see platform documentation for details).

The interpretation of ``file size'' changes according to the file type. For plain files this is the size of the file in bytes. For FIFOs and sockets under most flavors of *UNIX* (including Linux in particular), the ``size'' is the number of bytes waiting to be read at the time of the call to `os.stat()`, `os.fstat()`, or `os.lstat()`; this can sometimes be useful, especially for polling one of these special files after a non-blocking open. The meaning of the size field for other character and block devices varies more, depending on the implementation of the underlying system call.

Example:

```python
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top, calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname)[ST_MODE]
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print 'Skipping %s' % pathname

    def visitfile(file):
        print 'visiting', file
```
if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
11.4 statvfs -- Constants used with os.statvfs()

The `statvfs` module defines constants so interpreting the result if `os.statvfs()`, which returns a tuple, can be made without remembering "magic numbers." Each of the constants defined in this module is the index of the entry in the tuple returned by `os.statvfs()` that contains the specified information.

**F_BSIZE**
Preferred file system block size.

**F_FRSIZE**
Fundamental file system block size.

**F_BLOCKS**
Total number of blocks in the filesystem.

**F_BFREE**
Total number of free blocks.

**F_FAVAIL**
Free blocks available to non-super user.

**F_FILES**
Total number of file nodes.

**F_FFREE**
Total number of free file nodes.

**F_FAVAIL**
Free nodes available to non-super user.

**F_FLAG**
Flags. System dependent: see `statvfs()` man page.

**F_NAMEMAX**
Maximum file name length.
11.5 filecmp -- File and Directory Comparisons

The filecmp module defines functions to compare files and directories, with various optional time/correctness trade-offs.

The filecmp module defines the following functions:

```
cmp(f1, f2[, shallow])
```

Compare the files named `f1` and `f2`, returning `True` if they seem equal, `False` otherwise.

Unless `shallow` is given and is false, files with identical `os.stat()` signatures are taken to be equal.

Files that were compared using this function will not be compared again unless their `os.stat()` signature changes.

Note that no external programs are called from this function, giving it portability and efficiency.

```
cmpfiles(dir1, dir2, common[, shallow])
```

Returns three lists of file names: `match`, `mismatch`, `errors`. `match` contains the list of files match in both directories, `mismatch` includes the names of those that don't, and `errors` lists the names of files which could not be compared. Files may be listed in `errors` because the user may lack permission to read them or many other reasons, but always that the comparison could not be done for some reason.

The `common` parameter is a list of file names found in both directories. The `shallow` parameter has the same meaning and default value as for `filecmp.cmp()`.

Example:

```
>>> import filecmp
```
11.5.1 The dircmp class

dircmp instances are built using this constructor:

```python
class dircmp(a, b[, ignore[, hide]])
```

Construct a new directory comparison object, to compare the directories `a` and `b`. `ignore` is a list of names to ignore, and defaults to `['RCS', 'CVS', 'tags']`. `hide` is a list of names to hide, and defaults to `[os.curdir, os.pardir]`.

The dircmp class provides the following methods:

- `report()`
  Print (to `sys.stdout`) a comparison between `a` and `b`.

- `report_partial_closure()`
  Print a comparison between `a` and `b` and common immediate subdirectories.

- `report_full_closure()`
  Print a comparison between `a` and `b` and common subdirectories (recursively).

The dircmp offers a number of interesting attributes that may be used to get various bits of information about the directory trees being compared.

Note that via `__getattr__()` hooks, all attributes are computed lazily, so there is no speed penalty if only those attributes which are lightweight to compute are used.

- `left_list`
  Files and subdirectories in `a`, filtered by `hide` and `ignore`.

- `right_list`
  Files and subdirectories in `b`, filtered by `hide` and `ignore`.

- `common`
  Files and subdirectories in both `a` and `b`. 
left_only
   Files and subdirectories only in $a$.

right_only
   Files and subdirectories only in $b$.

common_dirs
   Subdirectories in both $a$ and $b$.

common_files
   Files in both $a$ and $b$

common_funny
   Names in both $a$ and $b$, such that the type differs between the directories, or names for which `os.stat()` reports an error.

same_files
   Files which are identical in both $a$ and $b$.

diff_files
   Files which are in both $a$ and $b$, whose contents differ.

funny_files
   Files which are in both $a$ and $b$, but could not be compared.

subdirs
   A dictionary mapping names in common_dirs to dircmp objects.
11.6 tempfile -- Generate temporary files and directories

This module generates temporary files and directories. It works on all supported platforms.

In version 2.3 of Python, this module was overhauled for enhanced security. It now provides three new functions, NamedTemporaryFile(), mkstemp(), and mkdtemp(), which should eliminate all remaining need to use the insecure mktemp() function. Temporary file names created by this module no longer contain the process ID; instead a string of six random characters is used.

Also, all the user-callable functions now take additional arguments which allow direct control over the location and name of temporary files. It is no longer necessary to use the global tempdir and template variables. To maintain backward compatibility, the argument order is somewhat odd; it is recommended to use keyword arguments for clarity.

The module defines the following user-callable functions:

**TemporaryFile**([mode='w+b', bufsize=-1, suffix, prefix, dir])
Return a file (or file-like) object that can be used as a temporary storage area. The file is created using mkstemp. It will be destroyed as soon as it is closed (including an implicit close when the object is garbage collected). Under UNIX, the directory entry for the file is removed immediately after the file is created. Other platforms do not support this; your code should not rely on a temporary file created using this function having or not having a visible name in the file system.

The mode parameter defaults to 'w+b' so that the file created can be read and written without being closed. Binary mode is used so that it behaves consistently on all platforms without regard for the data that is stored. bufsize defaults to -1, meaning that the operating system default is used.

The dir, prefix and suffix parameters are passed to mkstemp().
NamedTemporaryFile([mode='w+b', bufsize=-1, suffix[, prefix[, dir]]]])

This function operates exactly as TemporaryFile() does, except that the file is guaranteed to have a visible name in the file system (on UNIX, the directory entry is not unlinked). That name can be retrieved from the name member of the file object. Whether the name can be used to open the file a second time, while the named temporary file is still open, varies across platforms (it can be so used on UNIX; it cannot on Windows NT or later). New in version 2.3.

mkstemp([suffix[, prefix[, dir[, text]]]])

Creates a temporary file in the most secure manner possible. There are no race conditions in the file's creation, assuming that the platform properly implements the O_EXCL flag for os.open(). The file is readable and writable only by the creating user ID. If the platform uses permission bits to indicate whether a file is executable, the file is executable by no one. The file descriptor is not inherited by child processes.

Unlike TemporaryFile(), the user of mkstemp() is responsible for deleting the temporary file when done with it.

If suffix is specified, the file name will end with that suffix, otherwise there will be no suffix. mkstemp() does not put a dot between the file name and the suffix; if you need one, put it at the beginning of suffix.

If prefix is specified, the file name will begin with that prefix; otherwise, a default prefix is used.

If dir is specified, the file will be created in that directory; otherwise, a default directory is used.

If text is specified, it indicates whether to open the file in binary mode (the default) or text mode. On some platforms, this makes no difference.

mkstemp() returns a tuple containing an OS-level handle to an open file (as would be returned by os.open()) and the absolute pathname of that file, in that order. New in version 2.3.
**`mkdtemp([suffix[, prefix[, dir]]])`**

Creates a temporary directory in the most secure manner possible. There are no race conditions in the directory's creation. The directory is readable, writable, and searchable only by the creating user ID.

The user of `mkdtemp()` is responsible for deleting the temporary directory and its contents when done with it.

The `prefix`, `suffix`, and `dir` arguments are the same as for `mkstemp()`.

`mkdtemp()` returns the absolute pathname of the new directory. New in version 2.3.

**`mktemp([suffix[, prefix[, dir]]])`**

*Deprecated since release 2.3.* Use `mkstemp()` instead.

Return an absolute pathname of a file that did not exist at the time the call is made. The `prefix`, `suffix`, and `dir` arguments are the same as for `mkstemp()`.

**Warning:** Use of this function may introduce a security hole in your program. By the time you get around to doing anything with the file name it returns, someone else may have beaten you to the punch.

The module uses two global variables that tell it how to construct a temporary name. They are initialized at the first call to any of the functions above. The caller may change them, but this is discouraged; use the appropriate function arguments, instead.

**`tempdir`**

When set to a value other than `None`, this variable defines the default value for the `dir` argument to all the functions defined in this module.

If `tempdir` is unset or `None` at any call to any of the above functions, Python searches a standard list of directories and sets `tempdir` to the first one which the calling user can create files in. The list is:

1. The directory named by the TMPDIR environment variable.
2. The directory named by the TEMP environment variable.
3. The directory named by the TMP environment variable.
4. A platform-specific location:
   - On RiscOS, the directory named by the Wimp$ScrapDir environment variable.
   - On Windows, the directories C:\TEMP, C:\TMP, \TEMP, and \TMP, in that order.
   - On all other platforms, the directories /tmp, /var/tmp, and /usr/tmp, in that order.
5. As a last resort, the current working directory.

gettempdir()
Return the directory currently selected to create temporary files in. If tempdir is not None, this simply returns its contents; otherwise, the search described above is performed, and the result returned.

template
Deprecated since release 2.0. Use gettempprefix() instead.

When set to a value other than None, this variable defines the prefix of the final component of the filenames returned by mktemp(). A string of six random letters and digits is appended to the prefix to make the filename unique. On Windows, the default prefix is ~T; on all other systems it is tmp.

Older versions of this module used to require that template be set to None after a call to os.fork(); this has not been necessary since version 1.5.2.

gettempprefix()
Return the filename prefix used to create temporary files. This does not contain the directory component. Using this function is preferred over reading the template variable directly. New in version 1.5.2.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.7 glob -- UNIX style pathname pattern expansion

The glob module finds all the pathnames matching a specified pattern according to the rules used by the UNIX shell. No tilde expansion is done, but *, ?, and character ranges expressed with [ ] will be correctly matched. This is done by using the os.listdir() and fnmatch.fnmatch() functions in concert, and not by actually invoking a subshell. (For tilde and shell variable expansion, use os.path.expanduser() and os.path.expandvars().)

**glob**( pathname )

Return a possibly-empty list of path names that match *pathname*, which must be a string containing a path specification. *pathname* can be either absolute (like /usr/src/Python-1.5/Makefile) or relative (like ../../Tools/*/*.gif), and can contain shell-style wildcards. Broken symlinks are included in the results (as in the shell).

**iglob**( pathname )

Return an iterator which yields the same values as glob() without actually storing them all simultaneously. New in version 2.5.

For example, consider a directory containing only the following files: 1.gif, 2.txt, and card.gif. glob() will produce the following results. Notice how any leading components of the path are preserved.

```python
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
```

**See Also:**
Module **fnmatch**:
Shell-style filename (not path) expansion.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.8 fnmatch -- UNIX filename pattern matching

This module provides support for UNIX shell-style wildcards, which are not the same as regular expressions (which are documented in the re module). The special characters used in shell-style wildcards are:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>matches everything</td>
</tr>
<tr>
<td>?</td>
<td>matches any single character</td>
</tr>
<tr>
<td>[seq]</td>
<td>matches any character in seq</td>
</tr>
<tr>
<td>[!seq]</td>
<td>matches any character not in seq</td>
</tr>
</tbody>
</table>

Note that the filename separator (/ on UNIX) is not special to this module. See module glob for pathname expansion (glob uses fnmatch() to match pathname segments). Similarly, filenames starting with a period are not special for this module, and are matched by the * and ? patterns.

fnmatch( filename, pattern)
Test whether the filename string matches the pattern string, returning true or false. If the operating system is case-insensitive, then both parameters will be normalized to all lower- or upper-case before the comparison is performed. If you require a case-sensitive comparison regardless of whether that's standard for your operating system, use fnmatchcase() instead.

fnmatchcase( filename, pattern)
Test whether filename matches pattern, returning true or false; the comparison is case-sensitive.

filter( names, pattern)
Return the subset of the list of names that match pattern. It is the same as [n for n in names if fnmatch(n, pattern)], but implemented more efficiently. New in version 2.2.
See Also:

Module **glob**:  
UNIX shell-style path expansion.
11.9 linecache -- Random access to text lines

The linecache module allows one to get any line from any file, while attempting to optimize internally, using a cache, the common case where many lines are read from a single file. This is used by the traceback module to retrieve source lines for inclusion in the formatted traceback.

The linecache module defines the following functions:

**getline**(filename, lineno[, module_globals])
Get line lineno from file named filename. This function will never throw an exception -- it will return '' on errors (the terminating newline character will be included for lines that are found).

If a file named filename is not found, the function will look for it in the module search path, sys.path, after first checking for a PEP 302 __loader__ in module_globals, in case the module was imported from a zipfile or other non-filesystem import source.

New in version 2.5: The module_globals parameter was added.

**clearcache()**
Clear the cache. Use this function if you no longer need lines from files previously read using getline().

**checkcache**(filename)
Check the cache for validity. Use this function if files in the cache may have changed on disk, and you require the updated version. If filename is omitted, it will check all the entries in the cache.

Example:

```python
>>> import linecache
>>> linecache.getline('/etc/passwd', 4)
'sys:x:3:3:sys:/dev:/bin/sh\n'
```
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.10 shutil -- High-level file operations

The `shutil` module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal.

**Caveat:** On MacOS, the resource fork and other metadata are not used. For file copies, this means that resources will be lost and file type and creator codes will not be correct.

`copyfile( src, dst)`
Copy the contents of the file named `src` to a file named `dst`. The destination location must be writable; otherwise, an `IOError` exception will be raised. If `dst` already exists, it will be replaced. Special files such as character or block devices and pipes cannot be copied with this function. `src` and `dst` are path names given as strings.

`copyfileobj( fsrcc, fdst[, length])`
Copy the contents of the file-like object `fsrcc` to the file-like object `fdst`. The integer `length`, if given, is the buffer size. In particular, a negative `length` value means to copy the data without looping over the source data in chunks; by default the data is read in chunks to avoid uncontrolled memory consumption.

`copymode( src, dst)`
Copy the permission bits from `src` to `dst`. The file contents, owner, and group are unaffected. `src` and `dst` are path names given as strings.

`copystat( src, dst)`
Copy the permission bits, last access time, and last modification time from `src` to `dst`. The file contents, owner, and group are unaffected. `src` and `dst` are path names given as strings.

`copy( src, dst)`
Copy the file src to the file or directory dst. If dst is a directory, a file with the same basename as src is created (or overwritten) in the directory specified. Permission bits are copied. src and dst are path names given as strings.

**copy2( src, dst)**

Similar to copy(), but last access time and last modification time are copied as well. This is similar to the UNIX command `cp -p`.

**copytree( src, dst[, symlinks])**

Recursively copy an entire directory tree rooted at src. The destination directory, named by dst, must not already exist; it will be created as well as missing parent directories. Permissions and times of directories are copied with copystat(), individual files are copied using copy2(). If symlinks is true, symbolic links in the source tree are represented as symbolic links in the new tree; if false or omitted, the contents of the linked files are copied to the new tree. If exception(s) occur, an Error is raised with a list of reasons.

The source code for this should be considered an example rather than a tool.

Changed in version 2.3: Error is raised if any exceptions occur during copying, rather than printing a message.

Changed in version 2.5: Create intermediate directories needed to create dst, rather than raising an error. Copy permissions and times of directories using copystat().

**rmtree( path[, ignore_errors[, onerror]])**

Delete an entire directory tree. If ignore_errors is true, errors resulting from failed removals will be ignored; if false or omitted, such errors are handled by calling a handler specified by onerror or, if that is omitted, they raise an exception.

If onerror is provided, it must be a callable that accepts three parameters: function, path, and excinfo. The first parameter, function, is the function which raised the exception; it will be `os.listdir()`, `os.remove()` or `os.rmdir()`. The second parameter, path, will be the path name passed
to function. The third parameter, excinfo, will be the exception information return by sys.exc_info(). Exceptions raised by onerror will not be caught.

**move**( src, dst)

Recursively move a file or directory to another location.

If the destination is on our current filesystem, then simply use rename. Otherwise, copy src to the dst and then remove src.

New in version 2.3.

**exception Error**

This exception collects exceptions that raised during a mult-file operation. For copytree, the exception argument is a list of 3-tuples (srcname, dstname, exception).

New in version 2.3.

---

**Subsections**

- [11.10.1 Example](#)
11.10.1 Example

This example is the implementation of the `copytree()` function, described above, with the docstring omitted. It demonstrates many of the other functions provided by this module.

```python
def copytree(src, dst, symlinks=0):
    names = os.listdir(src)
    os.mkdir(dst)
    for name in names:
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks)
            else:
                copy2(srcname, dstname)
        except (IOError, os.error), why:
            print "Can't copy %s to %s: %s" % (`srcname`, `dstname`,
```

Python Library Reference

Previous: 11.10 shutil  Up: 11.10 shutil  Next: 11.11 dircache

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
The `dircache` module defines a function for reading directory listing using a cache, and cache invalidation using the `mtime` of the directory. Additionally, it defines a function to annotate directories by appending a slash.

The `dircache` module defines the following functions:

**reset()**
- Resets the directory cache.

**listdir(path)**
- Return a directory listing of `path`, as gotten from `os.listdir()`. Note that unless `path` changes, further call to `listdir()` will not re-read the directory structure.

Note that the list returned should be regarded as read-only. (Perhaps a future version should change it to return a tuple?)

**opendir(path)**
- Same as `listdir()`. Defined for backwards compatibility.

**annotate(head, list)**
- Assume `list` is a list of paths relative to `head`, and append, in place, a "/" to each path which points to a directory.

```python
>>> import dircache
>>> a = dircache.listdir('/')
>>> a = a[:] # Copy the return value so we can change 'a'
>>> a
['bin', 'boot', 'cdrom', 'dev', 'etc', 'floppy', 'home', 'initrd', 'found', 'mnt', 'proc', 'root', 'sbin', 'tmp', 'usr', 'var', 'vmlinuz
>>> dircache.annotate('/', a)
>>> a
['bin/', 'boot/', 'cdrom/', 'dev/', 'etc/', 'floppy/', 'home/', 'initrd', 'found', 'mnt/', 'proc/', 'root/', 'sbin/', 'tmp/', 'usr/', 'vmlinuz']
```
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
12. Data Compression and Archiving

The modules described in this chapter support data compression with the zlib, gzip, and bzip2 algorithms, and the creation of ZIP- and tar-format archives.

**zlib**
Low-level interface to compression and decompression routines compatible with gzip.

**gzip**
Interfaces for gzip compression and decompression using file objects.

**bz2**
Interface to compression and decompression routines compatible with bzip2.

**zipfile**
Read and write ZIP-format archive files.

**tarfile**
Read and write tar-format archive files.
12.1 zlib -- Compression compatible with gzip

For applications that require data compression, the functions in this module allow compression and decompression, using the zlib library. The zlib library has its own home page at http://www.zlib.net. There are known incompatibilities between the Python module and versions of the zlib library earlier than 1.1.3; 1.1.3 has a security vulnerability, so we recommend using 1.1.4 or later.

zlib's functions have many options and often need to be used in a particular order. This documentation doesn't attempt to cover all of the permutations; consult the zlib manual at http://www.zlib.net/manual.html for authoritative information.

The available exception and functions in this module are:

**exception error**
Exception raised on compression and decompression errors.

**adler32( string[, value])**
Computes a Adler-32 checksum of string. (An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much more quickly.) If value is present, it is used as the starting value of the checksum; otherwise, a fixed default value is used. This allows computing a running checksum over the concatenation of several input strings. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

**compress( string[, level])**
Compresses the data in string, returning a string contained compressed data. level is an integer from 1 to 9 controlling the level of compression; 1 is fastest and produces the least compression, 9 is slowest and produces the most. The default value is 6. Raises the error exception if any error occurs.
**compressobj([level])**

Returns a compression object, to be used for compressing data streams that won't fit into memory at once. *level* is an integer from 1 to 9 controlling the level of compression; 1 is fastest and produces the least compression, 9 is slowest and produces the most. The default value is 6.

**crc32(string[, value])**

Computes a CRC (Cyclic Redundancy Check) checksum of *string*. If *value* is present, it is used as the starting value of the checksum; otherwise, a fixed default value is used. This allows computing a running checksum over the concatenation of several input strings. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

**decompress(string[, wbits[, bufsize]])**

Decompresses the data in *string*, returning a string containing the uncompressed data. The *wbits* parameter controls the size of the window buffer. If *bufsize* is given, it is used as the initial size of the output buffer. Raises the error exception if any error occurs.

The absolute value of *wbits* is the base two logarithm of the size of the history buffer (the ``window size'') used when compressing data. Its absolute value should be between 8 and 15 for the most recent versions of the zlib library, larger values resulting in better compression at the expense of greater memory usage. The default value is 15. When *wbits* is negative, the standard gzip header is suppressed; this is an undocumented feature of the zlib library, used for compatibility with unzip's compression file format.

*bufsize* is the initial size of the buffer used to hold decompressed data. If more space is required, the buffer size will be increased as needed, so you don't have to get this value exactly right; tuning it will only save a few calls to malloc(). The default size is 16384.

**decompressobj([wbits])**

Returns a decompression object, to be used for decompressing data streams that won't fit into memory at once. The *wbits* parameter controls the size of
the window buffer.

Compression objects support the following methods:

**compress**(string)

Compress string, returning a string containing compressed data for at least part of the data in string. This data should be concatenated to the output produced by any preceding calls to the compress() method. Some input may be kept in internal buffers for later processing.

**flush**(mode)

All pending input is processed, and a string containing the remaining compressed output is returned. mode can be selected from the constants `Z_SYNC_FLUSH`, `Z_FULL_FLUSH`, or `Z_FINISH`, defaulting to `Z_FINISH`. `Z_SYNC_FLUSH` and `Z_FULL_FLUSH` allow compressing further strings of data, while `Z_FINISH` finishes the compressed stream and prevents compressing any more data. After calling flush() with mode set to `Z_FINISH`, the compress() method cannot be called again; the only realistic action is to delete the object.

**copy()**

Returns a copy of the compression object. This can be used to efficiently compress a set of data that share a common initial prefix. New in version 2.5.

Decompression objects support the following methods, and two attributes:

**unused_data**

A string which contains any bytes past the end of the compressed data. That is, this remains "" until the last byte that contains compression data is available. If the whole string turned out to contain compressed data, this is "", the empty string.

The only way to determine where a string of compressed data ends is by actually decompressing it. This means that when compressed data is contained part of a larger file, you can only find the end of it by reading data and feeding it followed by some non-empty string into a decompression object's decompress method until the unused_data
attribute is no longer the empty string.

**unconsumed_tail**
A string that contains any data that was not consumed by the last decompress call because it exceeded the limit for the uncompressed data buffer. This data has not yet been seen by the zlib machinery, so you must feed it (possibly with further data concatenated to it) back to a subsequent decompress method call in order to get correct output.

**decompress**(*string*, *max_length*)
Decompress *string*, returning a string containing the uncompressed data corresponding to at least part of the data in *string*. This data should be concatenated to the output produced by any preceding calls to the decompress() method. Some of the input data may be preserved in internal buffers for later processing.

If the optional parameter *max_length* is supplied then the return value will be no longer than *max_length*. This may mean that not all of the compressed input can be processed; and unconsumed data will be stored in the attribute unconsumed_tail. This string must be passed to a subsequent call to decompress() if decompression is to continue. If *max_length* is not supplied then the whole input is decompressed, and unconsumed_tail is an empty string.

**flush**(*length*)
All pending input is processed, and a string containing the remaining uncompressed output is returned. After calling flush(), the decompress() method cannot be called again; the only realistic action is to delete the object.

The optional parameter *length* sets the initial size of the output buffer.

**copy**()
Returns a copy of the decompression object. This can be used to save the state of the decompressor midway through the data stream in order to speed up random seeks into the stream at a future point. New in version 2.5.
See Also:

Module **gzip**: Reading and writing gzip-format files.

http://www.zlib.net
The zlib library home page.

http://www.zlib.net/manual.html
The zlib manual explains the semantics and usage of the library's many functions.
12.2 gzip -- Support for gzip files

The data compression provided by the zlib module is compatible with that used by the GNU compression program gzip. Accordingly, the gzip module provides the GzipFile class to read and write gzip-format files, automatically compressing or decompressing the data so it looks like an ordinary file object. Note that additional file formats which can be decompressed by the gzip and gunzip programs, such as those produced by compress and pack, are not supported by this module.

The module defines the following items:

class GzipFile([filename[, mode[, compresslevel[, fileobj]]]])

Constructor for the GzipFile class, which simulates most of the methods of a file object, with the exception of the readinto() and truncate() methods. At least one of fileobj and filename must be given a non-trivial value.

The new class instance is based on fileobj, which can be a regular file, a StringIO object, or any other object which simulates a file. It defaults to None, in which case filename is opened to provide a file object.

When fileobj is not None, the filename argument is only used to be included in the gzip file header, which may includes the original filename of the uncompressed file. It defaults to the filename of fileobj, if discernible; otherwise, it defaults to the empty string, and in this case the original filename is not included in the header.

The mode argument can be any of 'r', 'rb', 'a', 'ab', 'w', or 'wb', depending on whether the file will be read or written. The default is the mode of fileobj if discernible; otherwise, the default is 'rb'. If not given, the 'b' flag will be added to the mode to ensure the file is opened in binary mode for cross-platform portability.

The compresslevel argument is an integer from 1 to 9 controlling the level of compression; 1 is fastest and produces the least compression, and 9 is
slowest and produces the most compression. The default is 9.

Calling a GzipFile object's close() method does not close fileobj, since you might wish to append more material after the compressed data. This also allows you to pass a StringIO object opened for writing as fileobj, and retrieve the resulting memory buffer using the StringIO object's getvalue() method.

open( filename[, mode[, compresslevel]])
This is a shorthand for GzipFile(filename, mode, compresslevel). The filename argument is required; mode defaults to 'rb' and compresslevel defaults to 9.

See Also:
Module zlib:
The basic data compression module needed to support the gzip file format.
12.2 gzip  Up: 12. Data Compression and  Next: 12.3.1 (De)compression of files
12.3 bz2 -- Compression compatible with bzip2

New in version 2.3.

This module provides a comprehensive interface for the bz2 compression library. It implements a complete file interface, one-shot (de)compression functions, and types for sequential (de)compression.

Here is a resume of the features offered by the bz2 module:

- **BZ2File** class implements a complete file interface, including `readline()`, `readlines()`, `writelines()`, `seek()`, etc;
- **BZ2File** class implements emulated `seek()` support;
- **BZ2File** class implements universal newline support;
- **BZ2File** class offers an optimized line iteration using the readahead algorithm borrowed from file objects;
- Sequential (de)compression supported by **BZ2Compressor** and **BZ2Decompressor** classes;
- One-shot (de)compression supported by `compress()` and `decompress()` functions;
- Thread safety uses individual locking mechanism;
- Complete inline documentation;

Subsections

- [12.3.1 (De)compression of files](#)
- [12.3.2 Sequential (de)compression](#)
- [12.3.3 One-shot (de)compression](#)
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
12.3.1 (De)compression of files

Handling of compressed files is offered by the BZ2File class.

```python
class BZ2File( filename[, mode[, buffering[, compresslevel]]])
```

Open a bz2 file. Mode can be either 'r' or 'w', for reading (default) or writing. When opened for writing, the file will be created if it doesn't exist, and truncated otherwise. If buffering is given, 0 means unbuffered, and larger numbers specify the buffer size; the default is 0. If compresslevel is given, it must be a number between 1 and 9; the default is 9. Add a "U" to mode to open the file for input with universal newline support. Any line ending in the input file will be seen as a "\n" in Python. Also, a file so opened gains the attribute newlines; the value for this attribute is one of None (no newline read yet), '\r', '\n', '\r\n' or a tuple containing all the newline types seen. Universal newlines are available only when reading. Instances support iteration in the same way as normal file instances.

close()
Close the file. Sets data attribute closed to true. A closed file cannot be used for further I/O operations. close() may be called more than once without error.

read([size])
Read at most size uncompressed bytes, returned as a string. If the size argument is negative or omitted, read until EOF is reached.

readline([size])
Return the next line from the file, as a string, retaining newline. A non-negative size argument limits the maximum number of bytes to return (an incomplete line may be returned then). Return an empty string at EOF.

readlines([size])
Return a list of lines read. The optional size argument, if given, is an approximate bound on the total number of bytes in the lines returned.
**xreadlines()**

For backward compatibility. *BZ2File* objects now include the performance optimizations previously implemented in the *xreadlines* module.

**Deprecated since release 2.3.** This exists only for compatibility with the method by this name on *file* objects, which is deprecated. Use `for line in file` instead.

**seek( offset[, whence])**

Move to new file position. Argument *offset* is a byte count. Optional argument *whence* defaults to 0 (offset from start of file, *offset* should be $\geq 0$); other values are 1 (move relative to current position, positive or negative), and 2 (move relative to end of file, usually negative, although many platforms allow seeking beyond the end of a file).

Note that seeking of bz2 files is emulated, and depending on the parameters the operation may be extremely slow.

**tell()**

Return the current file position, an integer (may be a long integer).

**write( data)**

Write string *data* to file. Note that due to buffering, `close()` may be needed before the file on disk reflects the data written.

**writelines( sequence_of_strings)**

Write the sequence of strings to the file. Note that newlines are not added. The sequence can be any iterable object producing strings. This is equivalent to calling `write()` for each string.
12.3.2 Sequential (de)compression

Sequential compression and decompression is done using the classes BZ2Compressor and BZ2Decompressor.

class BZ2Compressor([compresslevel])

Create a new compressor object. This object may be used to compress data sequentially. If you want to compress data in one shot, use the `compress()` function instead. The `compresslevel` parameter, if given, must be a number between 1 and 9; the default is 9.

compress(data)

Provide more data to the compressor object. It will return chunks of compressed data whenever possible. When you've finished providing data to compress, call the `flush()` method to finish the compression process, and return what is left in internal buffers.

flush()

Finish the compression process and return what is left in internal buffers. You must not use the compressor object after calling this method.

class BZ2Decompressor()

Create a new decompressor object. This object may be used to decompress data sequentially. If you want to decompress data in one shot, use the `decompress()` function instead.

decompress(data)

Provide more data to the decompressor object. It will return chunks of decompressed data whenever possible. If you try to decompress data after the end of stream is found, `EOFError` will be raised. If any data was found after the end of stream, it'll be ignored and saved in `unused_data` attribute.
(de)compression

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
12.3.3 One-shot (de)compression

One-shot compression and decompression is provided through the `compress()` and `decompress()` functions.

`compress(data[, compresslevel])`
Compress `data` in one shot. If you want to compress data sequentially, use an instance of `BZ2Compressor` instead. The `compresslevel` parameter, if given, must be a number between 1 and 9; the default is 9.

`decompress(data)`
Decompress `data` in one shot. If you want to decompress data sequentially, use an instance of `BZ2Decompressor` instead.

---

Python Library Reference

Previous: 12.3.2 Sequential (de)compression  Up: 12.3 bz2  Next: 12.4 zipfile

License

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
12.4 zipfile -- Work with ZIP archives

New in version 1.6.

The ZIP file format is a common archive and compression standard. This module provides tools to create, read, write, append, and list a ZIP file. Any advanced use of this module will require an understanding of the format, as defined in PKZIP Application Note.

This module does not currently handle ZIP files which have appended comments, or multi-disk ZIP files. It can handle ZIP files that use the ZIP64 extensions (that is ZIP files that are more than 4 GByte in size).

The available attributes of this module are:

**exception error**
The error raised for bad ZIP files.

**exception LargeZipFile**
The error raised when a ZIP file would require ZIP64 functionality but that has not been enabled.

**class ZipFile**
The class for reading and writing ZIP files. See `ZipFile Objects` (section [12.4.1]) for constructor details.

**class PyZipFile**
Class for creating ZIP archives containing Python libraries.

**class ZipInfo([filename[, date_time]])**
Class used to represent information about a member of an archive. Instances of this class are returned by the getinfo() and infolist() methods of ZipFile objects. Most users of the zipfile module will not need to create these, but only use those created by this module. filename should be the full name of the archive member, and date_time should be a
tuple containing six fields which describe the time of the last modification
to the file; the fields are described in section 12.4.3, "ZipInfo Objects."

is_zipfile( filename )
Returns True if filename is a valid ZIP file based on its magic number,
otherwise returns False. This module does not currently handle ZIP files
which have appended comments.

ZIP_STORED
The numeric constant for an uncompressed archive member.

ZIP_DEFLATED
The numeric constant for the usual ZIP compression method. This requires
the zlib module. No other compression methods are currently supported.

See Also:

PKZIP Application Note
Documentation on the ZIP file format by Phil Katz, the creator of the
format and algorithms used.

Info-ZIP Home Page
Information about the Info-ZIP project's ZIP archive programs and
development libraries.

Subsections

• 12.4.1 ZipFile Objects
• 12.4.2 PyZipFile Objects
• 12.4.3 ZipInfo Objects
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
12.4.1 ZipFile Objects

```python
class ZipFile(file[, mode[, compression[, allowZip64]]])
```

Open a ZIP file, where `file` can be either a path to a file (a string) or a file-like object. The `mode` parameter should be 'r' to read an existing file, 'w' to truncate and write a new file, or 'a' to append to an existing file. For `mode` is 'a' and `file` refers to an existing ZIP file, then additional files are added to it. If `file` does not refer to a ZIP file, then a new ZIP archive is appended to the file. This is meant for adding a ZIP archive to another file, such as `python.exe`. Using

```
cat myzip.zip >> python.exe
```

also works, and at least **WinZip** can read such files. `compression` is the ZIP compression method to use when writing the archive, and should be `ZIP_STORED` or `ZIP_DEFLATED`; unrecognized values will cause `RuntimeError` to be raised. If `ZIP_DEFLATED` is specified but the `zlib` module is not available, `RuntimeError` is also raised. The default is `ZIP_STORED`. If `allowZip64` is `True` `zipfile` will create ZIP files that use the ZIP64 extensions when the zipfile is larger than 2 GB. If it is false (the default) `zipfile` will raise an exception when the ZIP file would require ZIP64 extensions. ZIP64 extensions are disabled by default because the default `zip` and `unzip` commands on **UNIX** (the InfoZIP utilities) don't support these extensions.

```python
close()
```

Close the archive file. You must call `close()` before exiting your program or essential records will not be written.

```python
cINFO = getinfo(name)
```

Return a `ZipInfo` object with information about the archive member `name`.

```python
infolist()
```

Return a list containing a `ZipInfo` object for each member of the archive. The objects are in the same order as their entries in the actual ZIP file on
disk if an existing archive was opened.

**namelist()**

Return a list of archive members by name.

**printdir()**

Print a table of contents for the archive to `sys.stdout`.

**read(name)**

Return the bytes of the file in the archive. The archive must be open for read or append.

**testzip()**

Read all the files in the archive and check their CRC's and file headers. Return the name of the first bad file, or else return `None`.

**write(filename[, arcname[, compress_type]])**

Write the file named `filename` to the archive, giving it the archive name `arcname` (by default, this will be the same as `filename`, but without a drive letter and with leading path separators removed). If given, `compress_type` overrides the value given for the `compression` parameter to the constructor for the new entry. The archive must be open with mode 'w' or 'a'.

**Note:** There is no official file name encoding for ZIP files. If you have unicode file names, please convert them to byte strings in your desired encoding before passing them to `write()`. WinZip interprets all file names as encoded in CP437, also known as DOS Latin.

**Note:** Archive names should be relative to the archive root, that is, they should not start with a path separator.

**writestr(zinfo_or_arcname, bytes)**

Write the string `bytes` to the archive; `zinfo_or_arcname` is either the file name it will be given in the archive, or a `ZipInfo` instance. If it's an instance, at least the filename, date, and time must be given. If it's a name, the date and time is set to the current date and time. The archive must be opened with mode 'w' or 'a'.

The following data attribute is also available:

**debug**

The level of debug output to use. This may be set from 0 (the default, no output) to 3 (the most output). Debugging information is written to `sys.stdout`.
12.4.2 PyZipFile Objects

The PyZipFile constructor takes the same parameters as the ZipFile constructor. Instances have one method in addition to those of ZipFile objects.

\texttt{writepy(pathname[, basename])}

Search for files *.py and add the corresponding file to the archive. The corresponding file is a *.pyo file if available, else a *.pyc file, compiling if necessary. If the pathname is a file, the filename must end with .py, and just the (corresponding *.py[co]) file is added at the top level (no path information). If it is a directory, and the directory is not a package directory, then all the files *.py[co] are added at the top level. If the directory is a package directory, then all *.py[oc] are added under the package name as a file path, and if any subdirectories are package directories, all of these are added recursively. \texttt{basename} is intended for internal use only. The \texttt{writepy()} method makes archives with file names like this:

- string.pyc # Top level name
- test/__init__.pyc # Package directory
- test/testall.pyc # Module test.test
- test/bogus/__init__.pyc # Subpackage directory
- test/bogus/myfile.pyc # Submodule test.

 RELEASE 2.5, documentation updated on 19th September, 2006. See \texttt{About this document...} for information on suggesting changes.
12.4.3 ZipInfo Objects

Instances of the ZipInfo class are returned by the getinfo() and infolist() methods of ZipFile objects. Each object stores information about a single member of the ZIP archive.

Instances have the following attributes:

filename
   Name of the file in the archive.

date_time
   The time and date of the last modification to the archive member. This is a tuple of six values:

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Year</td>
</tr>
<tr>
<td>1</td>
<td>Month (one-based)</td>
</tr>
<tr>
<td>2</td>
<td>Day of month (one-based)</td>
</tr>
<tr>
<td>3</td>
<td>Hours (zero-based)</td>
</tr>
<tr>
<td>4</td>
<td>Minutes (zero-based)</td>
</tr>
<tr>
<td>5</td>
<td>Seconds (zero-based)</td>
</tr>
</tbody>
</table>

compress_type
   Type of compression for the archive member.

comment
   Comment for the individual archive member.

extra
   Expansion field data. The PKZIP Application Note contains some comments on the internal structure of the data contained in this string.

create_system
   System which created ZIP archive.

create_version
PKZIP version which created ZIP archive.

**extract_version**
PKZIP version needed to extract archive.

**reserved**
Must be zero.

**flag_bits**
ZIP flag bits.

**volume**
Volume number of file header.

**internal_attr**
Internal attributes.

**external_attr**
External file attributes.

**header_offset**
Byte offset to the file header.

**CRC**
CRC-32 of the uncompressed file.

**compress_size**
Size of the compressed data.

**file_size**
Size of the uncompressed file.
12.5 tarfile -- Read and write tar archive files

New in version 2.3.

The tarfile module makes it possible to read and create tar archives. Some facts and figures:

- reads and writes gzip and bzip2 compressed archives.
- creates POSIX 1003.1-1990 compliant or GNU tar compatible archives.
- reads GNU tar extensions longname, longlink and sparse.
- stores pathnames of unlimited length using GNU tar extensions.
- handles directories, regular files, hardlinks, symbolic links, fifos, character devices and block devices and is able to acquire and restore file information like timestamp, access permissions and owner.
- can handle tape devices.

open([name[, mode [, fileobj[, bufsize]]]])

Return a TarFile object for the pathname name. For detailed information on TarFile objects, see TarFile Objects (section 12.5.1).

mode has to be a string of the form 'filemode[:compression]', it defaults to 'r'. Here is a full list of mode combinations:

<table>
<thead>
<tr>
<th>mode</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r' or</td>
<td>Open for reading with transparent compression</td>
</tr>
<tr>
<td>'r:*'</td>
<td>(recommended).</td>
</tr>
<tr>
<td>'r:'</td>
<td>Open for reading exclusively without compression.</td>
</tr>
<tr>
<td>'r:gz'</td>
<td>Open for reading with gzip compression.</td>
</tr>
<tr>
<td>'r:bz2'</td>
<td>Open for reading with bzip2 compression.</td>
</tr>
<tr>
<td>'a' or</td>
<td>Open for appending with no compression.</td>
</tr>
<tr>
<td>'a:'</td>
<td></td>
</tr>
<tr>
<td>'w' or</td>
<td>Open for uncompressed writing.</td>
</tr>
<tr>
<td>'w:'</td>
<td></td>
</tr>
</tbody>
</table>
'w:gz' | Open for gzip compressed writing.
--|---
'w:bz2' | Open for bzip2 compressed writing.

Note that 'a:gz' or 'a:bz2' is not possible. If mode is not suitable to open a certain (compressed) file for reading, ReadError is raised. Use mode 'r' to avoid this. If a compression method is not supported, CompressionError is raised.

If fileobj is specified, it is used as an alternative to a file object opened for name.

For special purposes, there is a second format for mode: 'filemode|compression'. open() will return a TarFile object that processes its data as a stream of blocks. No random seeking will be done on the file. If given, fileobj may be any object that has a read() or write() method (depending on the mode). bufsize specifies the blocksize and defaults to 20 * 512 bytes. Use this variant in combination with e.g. sys.stdin, a socket file object or a tape device. However, such a TarFile object is limited in that it does not allow to be accessed randomly, see ``Examples'' (section 12.5.3). The currently possible modes:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r</td>
<td>*'</td>
</tr>
<tr>
<td>'r</td>
<td>'</td>
</tr>
<tr>
<td>'r</td>
<td>gz'</td>
</tr>
<tr>
<td>'r</td>
<td>bz2'</td>
</tr>
<tr>
<td>'w</td>
<td>'</td>
</tr>
<tr>
<td>'w</td>
<td>gz'</td>
</tr>
<tr>
<td>'w</td>
<td>bz2'</td>
</tr>
</tbody>
</table>

class TarFile

Class for reading and writing tar archives. Do not use this class directly, better use open() instead. See ``TarFile Objects'' (section 12.5.1).

is_tarfile(name)

Return True if name is a tar archive file, that the tarfile module can
read.

**class TarFileCompat**(filename[, mode[, compression]])

Class for limited access to tar archives with a zipfile-like interface. Please consult the documentation of the zipfile module for more details. `compression` must be one of the following constants:

**TAR_PLAIN**

Constant for an uncompressed tar archive.

**TAR_GZIPPED**

Constant for a gzip compressed tar archive.

**exception TarError**

Base class for all tarfile exceptions.

**exception ReadError**

Is raised when a tar archive is opened, that either cannot be handled by the tarfile module or is somehow invalid.

**exception CompressionError**

Is raised when a compression method is not supported or when the data cannot be decoded properly.

**exception StreamError**

Is raised for the limitations that are typical for stream-like TarFile objects.

**exception ExtractError**

Is raised for non-fatal errors when using `extract()`, but only if `TarFile.errorlevel == 2`.

---

**See Also:**

Module **zipfile**:

Documentation of the zipfile standard module.

**GNU tar manual, Basic Tar Format**

Documentation for tar archive files, including GNU tar extensions.
Subsections

- 12.5.1 TarFile Objects
- 12.5.2 TarInfo Objects
- 12.5.3 Examples

Release 2.5, documentation updated on 19th September, 2006.  
See About this document... for information on suggesting changes.
12.5.1 TarFile Objects

The `TarFile` object provides an interface to a tar archive. A tar archive is a sequence of blocks. An archive member (a stored file) is made up of a header block followed by data blocks. It is possible, to store a file in a tar archive several times. Each archive member is represented by a `TarInfo` object, see `TarInfo Objects` (section 12.5.2) for details.

**class TarFile([name [, mode[, fileobj]]])**

Open an *(uncompressed)* tar archive `name`. *mode* is either 'r' to read from an existing archive, 'a' to append data to an existing file or 'w' to create a new file overwriting an existing one. *mode* defaults to 'r'.

If *fileobj* is given, it is used for reading or writing data. If it can be determined, *mode* is overridden by *fileobj*'s mode.

**Note:** *fileobj* is not closed, when `TarFile` is closed.

**open(...)**

Alternative constructor. The `open()` function on module level is actually a shortcut to this classmethod. See section 12.5 for details.

**getmember(name)**

Return a `TarInfo` object for member `name`. If `name` can not be found in the archive, `KeyError` is raised.

**Note:** If a member occurs more than once in the archive, its last occurrence is assumed to be the most up-to-date version.

**getmembers()**

Return the members of the archive as a list of `TarInfo` objects. The list has the same order as the members in the archive.
getnames()
Return the members as a list of their names. It has the same order as the list returned by getmembers().

list( verbose=True)
Print a table of contents to sys.stdout. If verbose is False, only the names of the members are printed. If it is True, output similar to that of ls -l is produced.

next()
Return the next member of the archive as a TarInfo object, when TarFile is opened for reading. Return None if there is no more available.

extractall([path[, members]])
Extract all members from the archive to the current working directory or directory path. If optional members is given, it must be a subset of the list returned by getmembers(). Directory informations like owner, modification time and permissions are set after all members have been extracted. This is done to work around two problems: A directory's modification time is reset each time a file is created in it. And, if a directory's permissions do not allow writing, extracting files to it will fail. New in version 2.5.

extract(member[, path])
Extract a member from the archive to the current working directory, using its full name. Its file information is extracted as accurately as possible. member may be a filename or a TarInfo object. You can specify a different directory using path.

Note: Because the extract() method allows random access to a tar archive there are some issues you must take care of yourself. See the description for extractall() above.

extractfile(member)
Extract a member from the archive as a file object. member may be a
filename or a TarInfo object. If member is a regular file, a file-like object is returned. If member is a link, a file-like object is constructed from the link’s target. If member is none of the above, None is returned.

**Note:** The file-like object is read-only and provides the following methods: read(), readline(), readlines(), seek(), tell().

```python
add(name[, arcname[, recursive]])
Add the file name to the archive. name may be any type of file (directory, fifo, symbolic link, etc.). If given, arcname specifies an alternative name for the file in the archive. Directories are added recursively by default. This can be avoided by setting recursive to False; the default is True.
```

```python
addfile(tarinfo[, fileobj])
Add the TarInfo object tarinfo to the archive. If fileobj is given, tarinfo.size bytes are read from it and added to the archive. You can create TarInfo objects using gettarinfo().

**Note:** On Windows platforms, fileobj should always be opened with mode 'rb' to avoid irritation about the file size.
```

```python
gettarinfo([name[, arcname[, fileobj]]])
Create a TarInfo object for either the file name or the file object fileobj (using os.fstat() on its file descriptor). You can modify some of the TarInfo’s attributes before you add it using addfile(). If given, arcname specifies an alternative name for the file in the archive.
```

```python
close()
Close the TarFile. In write mode, two finishing zero blocks are appended to the archive.
```

```python
posix
If true, create a POSIX 1003.1-1990 compliant archive. GNU extensions
are not used, because they are not part of the POSIX standard. This limits the length of filenames to at most 256, link names to 100 characters and the maximum file size to 8 gigabytes. A ValueError is raised if a file exceeds this limit. If false, create a GNU tar compatible archive. It will not be POSIX compliant, but can store files without any of the above restrictions. Changed in version 2.4: posix defaults to False.

dereference
If false, add symbolic and hard links to archive. If true, add the content of the target files to the archive. This has no effect on systems that do not support symbolic links.

ignore_zeros
If false, treat an empty block as the end of the archive. If true, skip empty (and invalid) blocks and try to get as many members as possible. This is only useful for concatenated or damaged archives.

default=0
To be set from 0 (no debug messages; the default) up to 3 (all debug messages). The messages are written to sys.stderr.

errorlevel
If 0 (the default), all errors are ignored when using extract(). Nevertheless, they appear as error messages in the debug output, when debugging is enabled. If 1, all fatal errors are raised as OSError or IOError exceptions. If 2, all non-fatal errors are raised as TarError exceptions as well.
### 12.5.2 TarInfo Objects

A TarInfo object represents one member in a TarFile. Aside from storing all required attributes of a file (like file type, size, time, permissions, owner etc.), it provides some useful methods to determine its type. It does not contain the file's data itself.

TarInfo objects are returned by TarFile's methods getmember(), getmembers() and gettarinfo().

```python
class TarInfo([name])
    Create a TarInfo object.

frombuf()
    Create and return a TarInfo object from a string buffer.

tobuf(posix)
    Create a string buffer from a TarInfo object. See TarFile's posix attribute for information on the posix argument. It defaults to False.
```

New in version 2.5: The posix parameter.

A TarInfo object has the following public data attributes:

- **name**
  - Name of the archive member.

- **size**
  - Size in bytes.

- **mtime**
  - Time of last modification.

- **mode**
  - Permission bits.

- **type**
File type. *type* is usually one of these constants: REGTYPE, AREGTYPE, LNKTYPE, SYMTYPE, DIRTYPE, FIFOTYPE, CONNTYPE, CHRTYPE, BLKTYPE, GNUTYPE_SPARSE. To determine the type of a *TarInfo* object more conveniently, use the `is_*()` methods below.

**linkname**
Name of the target file name, which is only present in *TarInfo* objects of type LNKTYPE and SYMTYPE.

**uid**
User ID of the user who originally stored this member.

**gid**
Group ID of the user who originally stored this member.

**uname**
User name.

**gname**
Group name.

A *TarInfo* object also provides some convenient query methods:

**isfile()**
Return `True` if the *Tarinfo* object is a regular file.

**isreg()**
Same as `isfile()`.

**isdir()**
Return `True` if it is a directory.

**issym()**
Return `True` if it is a symbolic link.

**islnk()**
Return `True` if it is a hard link.
ischr()
Return True if it is a character device.

isblk()
Return True if it is a block device.

isfifo()
Return True if it is a FIFO.

isdev()
Return True if it is one of character device, block device or FIFO.
12.5.3 Examples

How to extract an entire tar archive to the current working directory:

```python
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

How to create an uncompressed tar archive from a list of filenames:

```python
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

How to read a gzip compressed tar archive and display some member information:

```python
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print tarinfo.name, "is", tarinfo.size, "bytes in size and is",
    if tarinfo.isreg():
        print "a regular file."
    elif tarinfo.isdir():
        print "a directory."
    else:
        print "something else."
tar.close()
```

How to create a tar archive with faked information:

```python
import tarfile
tar = tarfile.open("sample.tar.gz", "w:gz")
for name in namelist:
    tarinfo = tar.gettarinfo(name, "fakeproj-1.0/" + name)
    tarinfo.uid = 123
    tarinfo.gid = 456
    tarinfo.uname = "johndoe"
    tarinfo.gname = "fake"
    tar.addfile(tarinfo, file(name))
tar.close()
```

The only way to extract an uncompressed tar stream from `sys.stdin`:
import sys
import tarfile
tar = tarfile.open(mode="r|", fileobj=sys.stdin)
for tarinfo in tar:
    tar.extract(tarinfo)
tar.close()
13. Data Persistence

The modules described in this chapter support storing Python data in a persistent form on disk. The pickle and marshal modules can turn many Python data types into a stream of bytes and then recreate the objects from the bytes. The various DBM-related modules support a family of hash-based file formats that store a mapping of strings to other strings. The bsddb module also provides such disk-based string-to-string mappings based on hashing, and also supports B-Tree and record-based formats.

The list of modules described in this chapter is:

- **pickle**  Convert Python objects to streams of bytes and back.
- **cPickle**  Faster version of pickle, but not subclassable.
- **copy_reg**  Register pickle support functions.
- **shelve**  Python object persistence.
- **marshal**  Convert Python objects to streams of bytes and back (with different constraints).
- **anydbm**  Generic interface to DBM-style database modules.
- **whichdb**  Guess which DBM-style module created a given database.
- **db**  The standard ``database'' interface, based on ndbm.
- **gdbm**  GNU's reinterpretation of dbm.
- **dbhash**  DBM-style interface to the BSD database library.
- **bsddb**  Interface to Berkeley DB database library
- **dumbdbm**  Portable implementation of the simple DBM interface.
- **sqlite3**  A DB-API 2.0 implementation using SQLite 3.x.
Relationship to other
13.1 pickle -- Python object serialization

The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as "serialization", "marshalling,"[^13.1] or "flattening", however, to avoid confusion, the terms used here are "pickling" and "unpickling".

This documentation describes both the pickle module and the cPickle module.

Footnotes

... ``marshalling,"[^13.1]
  Don't confuse this with the marshal module

Subsections

- 13.1.1 Relationship to other Python modules
- 13.1.2 Data stream format
- 13.1.3 Usage
- 13.1.4 What can be pickled and unpickled?
- 13.1.5 The pickle protocol
  - 13.1.5.1 Pickling and unpickling normal class instances
  - 13.1.5.2 Pickling and unpickling extension types
  - 13.1.5.3 Pickling and unpickling external objects
- 13.1.6 Subclassing Unpicklers
- 13.1.7 Example
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
13.1.1 Relationship to other Python modules

The pickle module has an optimized cousin called the cPickle module. As its name implies, cPickle is written in C, so it can be up to 1000 times faster than pickle. However it does not support subclassing of the Pickler() and Unpickler() classes, because in cPickle these are functions, not classes. Most applications have no need for this functionality, and can benefit from the improved performance of cPickle. Other than that, the interfaces of the two modules are nearly identical; the common interface is described in this manual and differences are pointed out where necessary. In the following discussions, we use the term ``pickle'' to collectively describe the pickle and cPickle modules.

The data streams the two modules produce are guaranteed to be interchangeable.

Python has a more primitive serialization module called marshal, but in general pickle should always be the preferred way to serialize Python objects. marshal exists primarily to support Python's .pyc files.

The pickle module differs from marshal several significant ways:

- The pickle module keeps track of the objects it has already serialized, so that later references to the same object won't be serialized again. marshal doesn't do this.

  This has implications both for recursive objects and object sharing. Recursive objects are objects that contain references to themselves. These are not handled by marshal, and in fact, attempting to marshal recursive objects will crash your Python interpreter. Object sharing happens when there are multiple references to the same object in different places in the object hierarchy being serialized. pickle stores such objects only once, and ensures that all other references point to the master copy. Shared objects remain shared, which can be very important for mutable objects.

- marshal cannot be used to serialize user-defined classes and their instances. pickle can save and restore class instances transparently, however the class definition must be importable and live in the same
module as when the object was stored.

- The marshal serialization format is not guaranteed to be portable across Python versions. Because its primary job in life is to support .pyc files, the Python implementers reserve the right to change the serialization format in non-backwards compatible ways should the need arise. The pickle serialization format is guaranteed to be backwards compatible across Python releases.

**Warning:** The pickle module is not intended to be secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

Note that serialization is a more primitive notion than persistence; although pickle reads and writes file objects, it does not handle the issue of naming persistent objects, nor the (even more complicated) issue of concurrent access to persistent objects. The pickle module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure. Perhaps the most obvious thing to do with these byte streams is to write them onto a file, but it is also conceivable to send them across a network or store them in a database. The module shelve provides a simple interface to pickle and unpickle objects on DBM-style database files.
13.1.2 Data stream format

The data format used by pickle is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as XDR (which can't represent pointer sharing); however it means that non-Python programs may not be able to reconstruct pickled Python objects.

By default, the pickle data format uses a printable ASCII representation. This is slightly more voluminous than a binary representation. The big advantage of using printable ASCII (and of some other characteristics of pickle's representation) is that for debugging or recovery purposes it is possible for a human to read the pickled file with a standard text editor.

There are currently 3 different protocols which can be used for pickling.

- Protocol version 0 is the original ASCII protocol and is backwards compatible with earlier versions of Python.
- Protocol version 1 is the old binary format which is also compatible with earlier versions of Python.
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of new-style classes.

Refer to PEP 307 for more information.

If a protocol is not specified, protocol 0 is used. If protocol is specified as a negative value or HIGHEST_PROTOCOL, the highest protocol version available will be used.

Changed in version 2.3: Introduced the protocol parameter.

A binary format, which is slightly more efficient, can be chosen by specifying a protocol version >= 1.
13.1.3 Usage

To serialize an object hierarchy, you first create a pickler, then you call the pickler's `dump()` method. To de-serialize a data stream, you first create an unpickler, then you call the unpickler's `load()` method. The pickle module provides the following constant:

**HIGHEST_PROTOCOL**

The highest protocol version available. This value can be passed as a `protocol` value. New in version 2.3.

**Note:** Be sure to always open pickle files created with protocols >= 1 in binary mode. For the old ASCII-based pickle protocol 0 you can use either text mode or binary mode as long as you stay consistent.

A pickle file written with protocol 0 in binary mode will contain lone linefeeds as line terminators and therefore will look ```funny``` when viewed in Notepad or other editors which do not support this format.

The pickle module provides the following functions to make the pickling process more convenient:

```python
dump(obj, file[, protocol])
```

Write a pickled representation of `obj` to the open file object `file`. This is equivalent to `Pickler(file, protocol).dump(obj)`.

If the `protocol` parameter is omitted, protocol 0 is used. If `protocol` is specified as a negative value or `HIGHEST_PROTOCOL`, the highest protocol version will be used.

Changed in version 2.3: Introduced the `protocol` parameter.

`file` must have a `write()` method that accepts a single string argument. It can thus be a file object opened for writing, a `StringIO` object, or any other custom object that meets this interface.

```python
load(file)
```
Read a string from the open file object `file` and interpret it as a pickle data stream, reconstructing and returning the original object hierarchy. This is equivalent to `Unpickler(file).load()`.

`file` must have two methods, a `read()` method that takes an integer argument, and a `readline()` method that requires no arguments. Both methods should return a string. Thus `file` can be a file object opened for reading, a `StringIO` object, or any other custom object that meets this interface.

This function automatically determines whether the data stream was written in binary mode or not.

**dumps** *(obj[, protocol])*

Return the pickled representation of the object as a string, instead of writing it to a file.

If the `protocol` parameter is omitted, protocol 0 is used. If `protocol` is specified as a negative value or `HIGHEST_PROTOCOL`, the highest protocol version will be used.

Changed in version 2.3: The `protocol` parameter was added.

**loads** *(string)*

Read a pickled object hierarchy from a string. Characters in the string past the pickled object's representation are ignored.

The `pickle` module also defines three exceptions:

**exception PickleError**

A common base class for the other exceptions defined below. This inherits from `Exception`.

**exception PicklingError**

This exception is raised when an unpicklable object is passed to the `dump()` method.

**exception UnpicklingError**

This exception is raised when there is a problem unpickling an object. Note
that other exceptions may also be raised during unpickling, including (but not necessarily limited to) AttributeError, EOFError, ImportError, and IndexError.

The pickle module also exports two callables \(^{132}\), Pickler and Unpickler:

\[ \text{class Pickler} \left( \text{file}, \text{protocol} \right) \]

This takes a file-like object to which it will write a pickle data stream.

If the protocol parameter is omitted, protocol 0 is used. If protocol is specified as a negative value, the highest protocol version will be used.

Changed in version 2.3: Introduced the protocol parameter.

file must have a write() method that accepts a single string argument. It can thus be an open file object, a StringIO object, or any other custom object that meets this interface.

Pickler objects define one (or two) public methods:

\[ \text{dump} \left( \text{obj} \right) \]

Write a pickled representation of obj to the open file object given in the constructor. Either the binary or ASCII format will be used, depending on the value of the protocol argument passed to the constructor.

\[ \text{clear_memo}() \]

Clears the pickler's ``memo''. The memo is the data structure that remembers which objects the pickler has already seen, so that shared or recursive objects pickled by reference and not by value. This method is useful when re-using picklers.

\[ \text{Note: Prior to Python 2.3, clear_memo}() \text{ was only available on the picklers created by \texttt{cpickle}. In the pickle module, picklers have an instance variable called memo which is a Python dictionary. So to clear the memo for a pickle module pickler, you could do the following:} \]
It is possible to make multiple calls to the `dump()` method of the same `Pickler` instance. These must then be matched to the same number of calls to the `load()` method of the corresponding `Unpickler` instance. If the same object is pickled by multiple `dump()` calls, the `load()` will all yield references to the same object. \[13.3\]

**Unpickler** objects are defined as:

```python
class Unpickler(file)
    This takes a file-like object from which it will read a pickle data stream. This class automatically determines whether the data stream was written in binary mode or not, so it does not need a flag as in the `Pickler` factory.

    file must have two methods, a `read()` method that takes an integer argument, and a `readline()` method that requires no arguments. Both methods should return a string. Thus `file` can be a file object opened for reading, a `StringIO` object, or any other custom object that meets this interface.
```

**Unpickler** objects have one (or two) public methods:

- `load()`
  Read a pickled object representation from the open file object given in the constructor, and return the reconstituted object hierarchy specified therein.

- `noload()`
  This is just like `load()` except that it doesn't actually create any objects. This is useful primarily for finding what's called "persistent ids" that may be referenced in a pickle data stream. See section \[13.1.5\] below for more details.

**Note:** the `noload()` method is currently only available on Unpickler
objects created with the cPickle module. pickle module
Unpicklers do not have the noload() method.

Footnotes

... callables 13.2

In the pickle module these callables are classes, which you could
subclass to customize the behavior. However, in the cPickle module
these callables are factory functions and so cannot be subclassed. One
common reason to subclass is to control what objects can actually be
unpickled. See section 13.1.6 for more details.

... object. 13.3

Warning: this is intended for pickling multiple objects without intervening
modifications to the objects or their parts. If you modify an object and then
pickle it again using the same Pickler instance, the object is not pickled
again -- a reference to it is pickled and the Unpickler will return the old
value, not the modified one. There are two problems here: (1) detecting
changes, and (2) marshalling a minimal set of changes. Garbage Collection
may also become a problem here.
13.1.4 What can be pickled and unpickled?

The following types can be pickled:

- None, True, and False
- integers, long integers, floating point numbers, complex numbers
- normal and Unicode strings
- tuples, lists, sets, and dictionaries containing only picklable objects
- functions defined at the top level of a module
- built-in functions defined at the top level of a module
- classes that are defined at the top level of a module
- instances of such classes whose `__dict__` or `__setstate__()` is picklable (see section 13.1.5 for details)

Attempts to pickle unpicklable objects will raise the `PicklingError` exception; when this happens, an unspecified number of bytes may have already been written to the underlying file. Trying to pickle a highly recursive data structure may exceed the maximum recursion depth, a `RuntimeError` will be raised in this case. You can carefully raise this limit with `sys.setrecursionlimit()`.

Note that functions (built-in and user-defined) are pickled by ```fully qualified``` name reference, not by value. This means that only the function name is pickled, along with the name of module the function is defined in. Neither the function's code, nor any of its function attributes are pickled. Thus the defining module must be importable in the unpickling environment, and the module must contain the named object, otherwise an exception will be raised.13.4

Similarly, classes are pickled by named reference, so the same restrictions in the unpickling environment apply. Note that none of the class's code or data is pickled, so in the following example the class attribute `attr` is not restored in
the unpickling environment:

```python
class Foo:
    attr = 'a class attr'

picklestring = pickle.dumps(Foo)
```

These restrictions are why picklable functions and classes must be defined in the top level of a module.

Similarly, when class instances are pickled, their class's code and data are not pickled along with them. Only the instance data are pickled. This is done on purpose, so you can fix bugs in a class or add methods to the class and still load objects that were created with an earlier version of the class. If you plan to have long-lived objects that will see many versions of a class, it may be worthwhile to put a version number in the objects so that suitable conversions can be made by the class's `__setstate__()` method.

---

**Footnotes**

... raised.\(^{13.4}\)

The exception raised will likely be an `ImportError` or an `AttributeError` but it could be something else.
13.1.5 The pickle protocol

This section describes the ``pickling protocol'' that defines the interface between the pickler/unpickler and the objects that are being serialized. This protocol provides a standard way for you to define, customize, and control how your objects are serialized and de-serialized. The description in this section doesn't cover specific customizations that you can employ to make the unpickling environment slightly safer from untrusted pickle data streams; see section 13.1.6 for more details.

Subsections

- 13.1.5.1 Pickling and unpickling normal class instances
- 13.1.5.2 Pickling and unpickling extension types
- 13.1.5.3 Pickling and unpickling external objects
13.1.6 Subclassing Unpicklers

By default, unpickling will import any class that it finds in the pickle data. You can control exactly what gets unpickled and what gets called by customizing your unpickler. Unfortunately, exactly how you do this is different depending on whether you're using pickle or cPickle.\footnote{13.9}

In the pickle module, you need to derive a subclass from Unpickler, overriding the load_global() method. load_global() should read two lines from the pickle data stream where the first line will be the name of the module containing the class and the second line will be the name of the instance's class. It then looks up the class, possibly importing the module and digging out the attribute, then it appends what it finds to the unpickler's stack. Later on, this class will be assigned to the __class__ attribute of an empty class, as a way of magically creating an instance without calling its class's __init__(). Your job (should you choose to accept it), would be to have load_global() push onto the unpickler's stack, a known safe version of any class you deem safe to unpickle. It is up to you to produce such a class. Or you could raise an error if you want to disallow all unpickling of instances. If this sounds like a hack, you're right. Refer to the source code to make this work.

Things are a little cleaner with cPickle, but not by much. To control what gets unpickled, you can set the unpickler's find_global attribute to a function or None. If it is None then any attempts to unpickle instances will raise an UnpicklingError. If it is a function, then it should accept a module name and a class name, and return the corresponding class object. It is responsible for looking up the class and performing any necessary imports, and it may raise an error to prevent instances of the class from being unpickled.

The moral of the story is that you should be really careful about the source of the strings your application unpickles.

Footnotes
A word of caution: the mechanisms described here use internal attributes and methods, which are subject to change in future versions of Python. We intend to someday provide a common interface for controlling this behavior, which will work in either pickle or cPickle.
13.1.7 Example

For the simplest code, use the `dump()` and `load()` functions. Note that a self-referencing list is pickled and restored correctly.

```python
import pickle

data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}

selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.
pickle.dump(selfref_list, output, -1)

output.close()
```

The following example reads the resulting pickled data. When reading a pickle-containing file, you should open the file in binary mode because you can't be sure if the ASCII or binary format was used.

```python
import pprint, pickle

pkl_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl_file)
pprint.pprint(data1)

data2 = pickle.load(pkl_file)
pprint.pprint(data2)

pkl_file.close()
```

Here's a larger example that shows how to modify pickling behavior for a class. The TextReader class opens a text file, and returns the line number and line contents each time its `readline()` method is called. If a TextReader instance is pickled, all attributes except the file object member are saved. When the instance is unpickled, the file is reopened, and reading resumes from the last
location. The __setstate__() and __getstate__() methods are used to implement this behavior.

class TextReader:
    """Print and number lines in a text file."""
    def __init__(self, file):
        self.file = file
        self.fh = open(file)
        self.lineno = 0

    def readline(self):
        self.lineno = self.lineno + 1
        line = self.fh.readline()
        if not line:
            return None
        if line.endswith("\n") :
            line = line[:-1]
        return "%d: %s" % (self.lineno, line)

    def __getstate__(self):
        odict = self.__dict__.copy() # copy the dict since we change
        del odict['fh'] # remove filehandle entry
        return odict

    def __setstate__(self,dict):
        fh = open(dict['file']) # reopen file
        count = dict['lineno'] # read from file...
        while count:
            fh.readline() # until line count is restored
            count = count - 1
        self.__dict__.update(dict) # update attributes
        self.fh = fh # save the file object

A sample usage might be something like this:

    >>> import TextReader
    >>> obj = TextReader.TextReader("TextReader.py")
    >>> obj.readline()
    '1: #!/usr/local/bin/python'
    >>> # (more invocations of obj.readline() here)
    ... obj.readline()
    '7: class TextReader:'
    >>> import pickle
    >>> pickle.dump(obj,open('save.p','w'))

If you want to see that pickle works across Python processes, start another
Python session, before continuing. What follows can happen from either the
same process or a new process.
>>> import pickle
>>> reader = pickle.load(open('save.p'))
>>> reader.readline()
'8:    "Print and number lines in a text file."

See Also:

Module **copy_reg**:
   Pickle interface constructor registration for extension types.

Module **shelve**:
   Indexed databases of objects; uses **pickle**.

Module **copy**:
   Shallow and deep object copying.

Module **marshal**:
   High-performance serialization of built-in types.
13.2 cPickle -- A faster pickle

The cPickle module supports serialization and de-serialization of Python objects, providing an interface and functionality nearly identical to the pickle module. There are several differences, the most important being performance and subclassability.

First, cPickle can be up to 1000 times faster than pickle because the former is implemented in C. Second, in the cPickle module the callables Pickler() and Unpickler() are functions, not classes. This means that you cannot use them to derive custom pickling and unpickling subclasses. Most applications have no need for this functionality and should benefit from the greatly improved performance of the cPickle module.

The pickle data stream produced by pickle and cPickle are identical, so it is possible to use pickle and cPickle interchangeably with existing pickles.13.10

There are additional minor differences in API between cPickle and pickle, however for most applications, they are interchangeable. More documentation is provided in the pickle module documentation, which includes a list of the documented differences.

---

Footnotes

... pickles.13.10

Since the pickle data format is actually a tiny stack-oriented programming language, and some freedom is taken in the encodings of certain objects, it is possible that the two modules produce different data streams for the same input objects. However it is guaranteed that they will always be able to read each other's data streams.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
13.3 copy_reg -- Register pickle support functions

The copy_reg module provides support for the pickle and cPickle modules. The copy module is likely to use this in the future as well. It provides configuration information about object constructors which are not classes. Such constructors may be factory functions or class instances.

**constructor**( *object*)

Declares *object* to be a valid constructor. If *object* is not callable (and hence not valid as a constructor), raises TypeError.

**pickle**( *type*, *function*[ [, *constructor*] ] )

Declares that *function* should be used as a "reduction" function for objects of type *type*; *type* must not be a "classic" class object. (Classic classes are handled differently; see the documentation for the pickle module for details.) *function* should return either a string or a tuple containing two or three elements.

The optional *constructor* parameter, if provided, is a callable object which can be used to reconstruct the object when called with the tuple of arguments returned by *function* at pickling time. TypeError will be raised if *object* is a class or *constructor* is not callable.

See the pickle module for more details on the interface expected of *function* and *constructor*.

Python Library Reference


Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
13.4 `shelve` -- Python object persistence

A "`shelf`" is a persistent, dictionary-like object. The difference with "`dbm`" databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects -- anything that the `pickle` module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

```
open( filename[, flag='c'[, protocol=None[, writeback=False]]])
```

Open a persistent dictionary. The filename specified is the base filename for the underlying database. As a side-effect, an extension may be added to the filename and more than one file may be created. By default, the underlying database file is opened for reading and writing. The optional `flag` parameter has the same interpretation as the `flag` parameter of `anydbm.open`.

By default, version 0 pickles are used to serialize values. The version of the pickle protocol can be specified with the `protocol` parameter. Changed in version 2.3: The `protocol` parameter was added.

By default, mutations to persistent-dictionary mutable entries are not automatically written back. If the optional `writeback` parameter is set to `True`, all entries accessed are cached in memory, and written back at close time; this can make it handier to mutate mutable entries in the persistent dictionary, but, if many entries are accessed, it can consume vast amounts of memory for the cache, and it can make the close operation very slow since all accessed entries are written back (there is no way to determine which accessed entries are mutable, nor which ones were actually mutated).

Shelve objects support all methods supported by dictionaries. This eases the transition from dictionary based scripts to those requiring persistent storage.

One additional method is supported:

```
sync()
```
Write back all entries in the cache if the shelf was opened with writeback set to True. Also empty the cache and synchronize the persistent dictionary on disk, if feasible. This is called automatically when the shelf is closed with close().

Subsections

- 13.4.1 Restrictions
- 13.4.2 Example

Python Library Reference

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
13.4.1 Restrictions

- The choice of which database package will be used (such as `dbm`, `gdbm` or `bsddb`) depends on which interface is available. Therefore it is not safe to open the database directly using `dbm`. The database is also (unfortunately) subject to the limitations of `dbm`, if it is used -- this means that (the pickled representation of) the objects stored in the database should be fairly small, and in rare cases key collisions may cause the database to refuse updates.

- Depending on the implementation, closing a persistent dictionary may or may not be necessary to flush changes to disk. The `__del__` method of the `Shelf` class calls the `close` method, so the programmer generally need not do this explicitly.

- The `shelve` module does not support concurrent read/write access to shelved objects. (Multiple simultaneous read accesses are safe.) When a program has a shelf open for writing, no other program should have it open for reading or writing. Unix file locking can be used to solve this, but this differs across Unix versions and requires knowledge about the database implementation used.

```python
class Shelf(dict, protocol=None, writeback=False)
```

A subclass of `UserDict.DictMixin` which stores pickled values in the `dict` object.

By default, version 0 pickles are used to serialize values. The version of the pickle protocol can be specified with the `protocol` parameter. See the `pickle` documentation for a discussion of the pickle protocols. Changed in version 2.3: The `protocol` parameter was added.

If the `writeback` parameter is `True`, the object will hold a cache of all entries accessed and write them back to the `dict` at sync and close times. This allows natural operations on mutable entries, but can consume much more memory and make sync and close take a long time.

```python
class BsdDbShelf(dict, protocol=None, writeback=False)
```
A subclass of Shelf which exposes first, next, previous, last and set_location which are available in the bsddb module but not in other database modules. The dict object passed to the constructor must support those methods. This is generally accomplished by calling one of bsddb.hashopen, bsddb.btopen or bsddb.rnopen. The optional protocol and writeback parameters have the same interpretation as for the Shelf class.

```python
class DbfilenameShelf(filename[, flag='c'[, protocol=None[, writeback=False]]])
```

A subclass of Shelf which accepts a filename instead of a dict-like object. The underlying file will be opened using anydbm.open. By default, the file will be created and opened for both read and write. The optional flag parameter has the same interpretation as for the open function. The optional protocol and writeback parameters have the same interpretation as for the Shelf class.
13.4.2 Example

To summarize the interface (key is a string, data is an arbitrary object):

```python
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by low
# library

d[key] = data  # store data at key (overwrites old data if
# using an existing key)
data = d[key]  # retrieve a COPY of data at key (raise KeyError if
# such key)
del d[key]  # delete data stored at key (raises KeyError
# if no such key)
flag = d.has_key(key)  # true if the key exists
klist = d.keys()  # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = range(4)  # this works as expected, but...
d['xx'].append(5)  # *this doesn't!* -- d['xx'] is STILL range(4)!!

# having opened d without writeback=True, you need to code carefully
temp = d['xx']  # extracts the copy
temp.append(5)  # mutates the copy
d['xx'] = temp  # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.
d.close()  # close it
```

See Also:

Module **anydbm**:  
Generic interface to dbm-style databases.

Module **bsddb**:  
BSD db database interface.

Module **dbhash**:  
Thin layer around the bsddb which provides an open function like
the other database modules.

Module **dbm**: Standard UNIX database interface.

Module **dumbdbm**: Portable implementation of the dbm interface.

Module **gdbm**: GNU database interface, based on the dbm interface.

Module **pickle**: Object serialization used by shelve.

Module **cPickle**: High-performance version of pickle.
13.5 marshal -- Internal Python object serialization

This module contains functions that can read and write Python values in a binary format. The format is specific to Python, but independent of machine architecture issues (e.g., you can write a Python value to a file on a PC, transport the file to a Sun, and read it back there). Details of the format are undocumented on purpose; it may change between Python versions (although it rarely does).

This is not a general "persistence" module. For general persistence and transfer of Python objects through RPC calls, see the modules pickle and shelve. The marshal module exists mainly to support reading and writing the "pseudo-compiled" code for Python modules of .pyc files. Therefore, the Python maintainers reserve the right to modify the marshal format in backward incompatible ways should the need arise. If you're serializing and de-serializing Python objects, use the pickle module instead.

**Warning:** The marshal module is not intended to be secure against erroneous or maliciously constructed data. Never unmarshal data received from an untrusted or unauthenticated source.

Not all Python object types are supported; in general, only objects whose value is independent from a particular invocation of Python can be written and read by this module. The following types are supported: None, integers, long integers, floating point numbers, strings, Unicode objects, tuples, lists, dictionaries, and code objects, where it should be understood that tuples, lists and dictionaries are only supported as long as the values contained therein are themselves supported; and recursive lists and dictionaries should not be written (they will cause infinite loops).

**Caveat:** On machines where C's `long int` type has more than 32 bits (such as the DEC Alpha), it is possible to create plain Python integers that are longer than
32 bits. If such an integer is marshaled and read back in on a machine where C's \texttt{long int} type has only 32 bits, a Python long integer object is returned instead. While of a different type, the numeric value is the same. (This behavior is new in Python 2.2. In earlier versions, all but the least-significant 32 bits of the value were lost, and a warning message was printed.)

There are functions that read/write files as well as functions operating on strings.

The module defines these functions:

\texttt{dump(value, file[, version])}

Write the value on the open file. The value must be a supported type. The file must be an open file object such as \texttt{sys.stdout} or returned by \texttt{open()} or \texttt{posix.popen()}. It must be opened in binary mode ('\texttt{wb}'' or '\texttt{w+b}').

If the value has (or contains an object that has) an unsupported type, a \texttt{ValueError} exception is raised -- but garbage data will also be written to the file. The object will not be properly read back by \texttt{load()}. New in version 2.4: The \texttt{version} argument indicates the data format that \texttt{dump} should use (see below).

\texttt{load(file)}

Read one value from the open file and return it. If no valid value is read, raise \texttt{EOFError, ValueError} or \texttt{TypeError}. The file must be an open file object opened in binary mode ('\texttt{rb}'' or '\texttt{r+b}').

\textbf{Warning:} If an object containing an unsupported type was marshalled with \texttt{dump()}, \texttt{load()} will substitute \texttt{None} for the unmarshallable type.

\texttt{dumps(value[, version])}

Return the string that would be written to a file by \texttt{dump(value, file)}. The value must be a supported type. Raise a \texttt{ValueError} exception if value has (or contains an object that has) an unsupported type.

New in version 2.4: The \texttt{version} argument indicates the data format that \texttt{dumps} should use (see below).
loads( string)

Convert the string to a value. If no valid value is found, raise EOFError, ValueError or TypeError. Extra characters in the string are ignored.

In addition, the following constants are defined:

version

Indicates the format that the module uses. Version 0 is the historical format, version 1 (added in Python 2.4) shares interned strings and version 2 (added in Python 2.5) uses a binary format for floating point numbers. The current version is 2.

New in version 2.4.

Footnotes

... does). 13.11

The name of this module stems from a bit of terminology used by the designers of Modula-3 (amongst others), who use the term ``marshalling'' for shipping of data around in a self-contained form. Strictly speaking, ``to marshal'' means to convert some data from internal to external form (in an RPC buffer for instance) and ``unmarshalling'' for the reverse process.
13.6 anydbm -- Generic access to DBM-style databases

anydbm is a generic interface to variants of the DBM database -- dbhash (requires bsddb), gdbm, or dbm. If none of these modules is installed, the slow-but-simple implementation in module dumbdbm will be used.

**open( filename[, flag[, mode]])**

Open the database file `filename` and return a corresponding object.

If the database file already exists, the `whichdb` module is used to determine its type and the appropriate module is used; if it does not exist, the first module listed above that can be imported is used.

The optional `flag` argument can be 'r' to open an existing database for reading only, 'w' to open an existing database for reading and writing, 'c' to create the database if it doesn't exist, or 'n', which will always create a new empty database. If not specified, the default value is 'r'.

The optional `mode` argument is the UNIX mode of the file, used only when the database has to be created. It defaults to octal 0666 (and will be modified by the prevailing umask).

**exception error**

A tuple containing the exceptions that can be raised by each of the supported modules, with a unique exception `anydbm.error` as the first item -- the latter is used when `anydbm.error` is raised.

The object returned by `open()` supports most of the same functionality as dictionaries; keys and their corresponding values can be stored, retrieved, and deleted, and the `has_key()` and `keys()` methods are available. Keys and values must always be strings.

The following example records some hostnames and a corresponding title, and then prints out the contents of the database:
import anydbm

# Open database, creating it if necessary.
db = anydbm.open('cache', 'c')

# Record some values
db['www.python.org'] = 'Python Website'
db['www.cnn.com'] = 'Cable News Network'

# Loop through contents. Other dictionary methods
# such as .keys(), .values() also work.
for k, v in db.iteritems():
    print k, '\t', v

# Storing a non-string key or value will raise an exception (most
# likely a TypeError).
db['www.yahoo.com'] = 4

# Close when done.
db.close()

See Also:

Module **dbhash**: BSD db database interface.

Module **dbm**: Standard UNIX database interface.

Module **dumbdbm**: Portable implementation of the dbm interface.

Module **gdbm**: GNU database interface, based on the dbm interface.

Module **shelve**: General object persistence built on top of the Python dbm interface.

Module **whichdb**: Utility module used to determine the type of an existing database.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
13.7 `whichdb` -- Guess which DBM module created a database

The single function in this module attempts to guess which of the several simple database modules available-`dbm`, `gdbm`, or `dbhash`-should be used to open a given file.

`whichdb`(filename)

Returns one of the following values: `None` if the file can't be opened because it's unreadable or doesn't exist; the empty string (``) if the file's format can't be guessed; or a string containing the required module name, such as 'dbm' or 'gdbm'.
13.8 dbm -- Simple ``database'' interface

Availability: Unix.

The dbm module provides an interface to the UNIX (n)dbm library. Dbm objects behave like mappings (dictionaries), except that keys and values are always strings. Printing a dbm object doesn't print the keys and values, and the items() and values() methods are not supported.

This module can be used with the `classic' ndbm interface, the BSD DB compatibility interface, or the GNU GDBM compatibility interface. On UNIX, the configure script will attempt to locate the appropriate header file to simplify building this module.

The module defines the following:

**exception error**

Raised on dbm-specific errors, such as I/O errors. KeyError is raised for general mapping errors like specifying an incorrect key.

**library**

Name of the ndbm implementation library used.

**open(filename[, flag[, mode]])**

Open a dbm database and return a dbm object. The filename argument is the name of the database file (without the .dir or .pag extensions; note that the BSD DB implementation of the interface will append the extension .db and only create one file).

The optional flag argument must be one of these values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r'</td>
<td>Open existing database for reading only (default)</td>
</tr>
<tr>
<td>'w'</td>
<td>Open existing database for reading and writing</td>
</tr>
</tbody>
</table>
The optional `mode` argument is the UNIX mode of the file, used only when the database has to be created. It defaults to octal `0666`.

**See Also:**

Module **anydbm**:  
Generic interface to dbm-style databases.

Module **gdbm**:  
Similar interface to the GNU GDBM library.

Module **whichdb**:  
Utility module used to determine the type of an existing database.
13.9 gdbm -- GNU's reimplementation of dbm

Availability: Unix.

This module is quite similar to the dbm module, but uses gdbm instead to provide some additional functionality. Please note that the file formats created by gdbm and dbm are incompatible.

The gdbm module provides an interface to the GNU DBM library. gdbm objects behave like mappings (dictionaries), except that keys and values are always strings. Printing a gdbm object doesn't print the keys and values, and the items() and values() methods are not supported.

The module defines the following constant and functions:

**exception error**

Raised on gdbm-specific errors, such as I/O errors. KeyError is raised for general mapping errors like specifying an incorrect key.

**open**(*filename*, *flag*, *mode*)

Open a gdbm database and return a gdbm object. The filename argument is the name of the database file.

The optional flag argument can be 'r' (to open an existing database for reading only -- default), 'w' (to open an existing database for reading and writing), 'c' (which creates the database if it doesn't exist), or 'n' (which always creates a new empty database).

The following additional characters may be appended to the flag to control how the database is opened:

- 'f' -- Open the database in fast mode. Writes to the database will not be synchronized.
- 's' -- Synchronized mode. This will cause changes to the database will be immediately written to the file.
- 'u' -- Do not lock database.

Not all flags are valid for all versions of \texttt{gdbm}. The module constant \texttt{open\_flags} is a string of supported flag characters. The exception \texttt{error} is raised if an invalid flag is specified.

The optional \texttt{mode} argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0666.

In addition to the dictionary-like methods, \texttt{gdbm} objects have the following methods:

**firstkey()**

It's possible to loop over every key in the database using this method and the \texttt{nextkey()} method. The traversal is ordered by \texttt{gdbm}'s internal hash values, and won't be sorted by the key values. This method returns the starting key.

**nextkey(key)**

Returns the key that follows \texttt{key} in the traversal. The following code prints every key in the database \texttt{db}, without having to create a list in memory that contains them all:

```python
k = db.firstkey()
while k != None:
    print k
    k = db.nextkey(k)
```

**reorganize()**

If you have carried out a lot of deletions and would like to shrink the space used by the \texttt{gdbm} file, this routine will reorganize the database. \texttt{gdbm} will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

**sync()**

When the database has been opened in fast mode, this method forces any unwritten data to be written to the disk.
See Also:

Module **anydbm**:
- Generic interface to dbm-style databases.

Module **whichdb**:
- Utility module used to determine the type of an existing database.

---

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
13.10 dbhash -- DBM-style interface to the BSD database library

Availability: Unix, Windows.

The dbhash module provides a function to open databases using the BSD db library. This module mirrors the interface of the other Python database modules that provide access to DBM-style databases. The bsddb module is required to use dbhash.

This module provides an exception and a function:

**exception error**

Exception raised on database errors other than KeyError. It is a synonym for bsddb.error.

**open**(path[, flag[, mode]])

Open a db database and return the database object. The path argument is the name of the database file.

The flag argument can be 'r' (the default), 'w', 'c' (which creates the database if it doesn't exist), or 'n' (which always creates a new empty database). For platforms on which the BSD db library supports locking, an "l" can be appended to indicate that locking should be used.

The optional mode parameter is used to indicate the UNIX permission bits that should be set if a new database must be created; this will be masked by the current umask value for the process.

See Also:

Module anydbm: Generic interface to dbm-style databases.
Module **bsddb**: Lower-level interface to the BSD db library.

Module **whichdb**: Utility module used to determine the type of an existing database.

---

**Subsections**

- [13.10.1 Database Objects](#)

---

**Python Library Reference**

Previous: [13.9 gdbm](#)  Up: [13. Data Persistence](#)  Next: [13.10.1 Database Objects](#)

*Release 2.5, documentation updated on 19th September, 2006.  See [About this document...](#) for information on suggesting changes.*
13.10.1 Database Objects

The database objects returned by `open()` provide the methods common to all the DBM-style databases and mapping objects. The following methods are available in addition to the standard methods.

**first()**

It's possible to loop over every key/value pair in the database using this method and the `next()` method. The traversal is ordered by the databases internal hash values, and won't be sorted by the key values. This method returns the starting key.

**last()**

Return the last key/value pair in a database traversal. This may be used to begin a reverse-order traversal; see `previous()`.

**next()**

Returns the key next key/value pair in a database traversal. The following code prints every key in the database `db`, without having to create a list in memory that contains them all:

```python
print db.first()
for i in xrange(1, len(db)):
    print db.next()
```

**previous()**

Returns the previous key/value pair in a forward-traversal of the database. In conjunction with `last()`, this may be used to implement a reverse-order traversal.

**sync()**

This method forces any unwritten data to be written to the disk.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
13.11 bsddb -- Interface to Berkeley DB library

Availability: **Unix, Windows.**

The `bsddb` module provides an interface to the Berkeley DB library. Users can create hash, btree or record based library files using the appropriate open call. Bsddb objects behave generally like dictionaries. Keys and values must be strings, however, so to use other objects as keys or to store other kinds of objects the user must serialize them somehow, typically using `marshal.dumps()` or `pickle.dumps()`.

The `bsddb` module requires a Berkeley DB library version from 3.3 thru 4.4.

**See Also:**

  The website with documentation for the `bsddb.db` python Berkeley DB interface that closely mirrors the Sleepycat object oriented interface provided in Berkeley DB 3 and 4.

- [http://www.sleepycat.com/](http://www.sleepycat.com/)
  Sleepycat Software produces the Berkeley DB library.

A more modern DB, DBEnv and DBSequence object interface is available in the `bsddb.db` module which closely matches the Sleepycat Berkeley DB C API documented at the above URLs. Additional features provided by the `bsddb.db` API include fine tuning, transactions, logging, and multiprocess concurrent database access.

The following is a description of the legacy `bsddb` interface compatible with the old python `bsddb` module. Starting in Python 2.5 this interface should be safe for multithreaded access. The `bsddb.db` API is recommended for threading users as it provides better control.
The bsddb module defines the following functions that create objects that access the appropriate type of Berkeley DB file. The first two arguments of each function are the same. For ease of portability, only the first two arguments should be used in most instances.

```c
hashopen(filename[, flag[, mode[, bsize[, ffactor[, nelem[, cachesize[, hash[, lorder]]]]]]]])
```

Open the hash format file named `filename`. Files never intended to be preserved on disk may be created by passing `None` as the `filename`. The optional `flag` identifies the mode used to open the file. It may be "r" (read only), "w" (read-write), "c" (read-write - create if necessary; the default) or "n" (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level `dbopen()` function. Consult the Berkeley DB documentation for their use and interpretation.

```c
btopen(filename[, flag[, mode[, btflags[, cachesize[, maxkeypage[, minkeypage[, pgsize[, lorder]]]]]]]])
```

Open the btree format file named `filename`. Files never intended to be preserved on disk may be created by passing `None` as the `filename`. The optional `flag` identifies the mode used to open the file. It may be "r" (read only), "w" (read-write), "c" (read-write - create if necessary; the default) or "n" (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level `dbopen` function. Consult the Berkeley DB documentation for their use and interpretation.

```c
rnopen(filename[, flag[, mode[, rnflags[, cachesize[, pgsize[, lorder[, reclen[, bval[, bfname]]]]]]]])
```

Open a DB record format file named `filename`. Files never intended to be preserved on disk may be created by passing `None` as the `filename`. The optional `flag` identifies the mode used to open the file. It may be "r" (read only), "w" (read-write), "c" (read-write - create if necessary; the default) or "n" (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level `dbopen` function. Consult the Berkeley DB documentation for their use and interpretation.
**Note:** Beginning in 2.3 some UNIX versions of Python may have a `bsddb185` module. This is present *only* to allow backwards compatibility with systems which ship with the old Berkeley DB 1.85 database library. The `bsddb185` module should never be used directly in new code.

**See Also:**

Module **dbhash**:
DBM-style interface to the `bsddb`.

---

**Subsections**

- 13.11.1 Hash, BTree and Record Objects

---

**Python Library Reference**

Previous: 13.10.1 Database Objects  Up: 13. Data Persistence  Next: 13.11.1 Hash, BTree and

*Release 2.5, documentation updated on 19th September, 2006.*

See [About this document...](#) for information on suggesting changes.
13.11.1 Hash, BTree and Record Objects

Once instantiated, hash, btree and record objects support the same methods as dictionaries. In addition, they support the methods listed below. Changed in version 2.3.1: Added dictionary methods.

**close()**
Close the underlying file. The object can no longer be accessed. Since there is no open open method for these objects, to open the file again a new bsddb module open function must be called.

**keys()**
Return the list of keys contained in the DB file. The order of the list is unspecified and should not be relied on. In particular, the order of the list returned is different for different file formats.

**has_key(key)**
Return 1 if the DB file contains the argument as a key.

**set_location(key)**
Set the cursor to the item indicated by key and return a tuple containing the key and its value. For binary tree databases (opened using btopen()), if key does not actually exist in the database, the cursor will point to the next item in sorted order and return that key and value. For other databases, KeyError will be raised if key is not found in the database.

**first()**
Set the cursor to the first item in the DB file and return it. The order of keys in the file is unspecified, except in the case of B-Tree databases. This method raises bsddb.error if the database is empty.

**next()**
Set the cursor to the next item in the DB file and return it. The order of keys in the file is unspecified, except in the case of B-Tree databases.

**previous()**
Set the cursor to the previous item in the DB file and return it. The order of keys in the file is unspecified, except in the case of B-Tree databases. This is not supported on hashtable databases (those opened with `hashopen()`).

`last()`

Set the cursor to the last item in the DB file and return it. The order of keys in the file is unspecified. This is not supported on hashtable databases (those opened with `hashopen()`). This method raises `bsddb.error` if the database is empty.

`sync()`

Synchronize the database on disk.

Example:

```python
>>> import bsddb
>>> db = bsddb.btopen('/tmp/spam.db', 'c')
>>> for i in range(10): db['%d'%i] = '%d' % (i*i)
...
>>> db['3']
'9'
>>> db.keys()
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> db.first()
('0', '0')
>>> db.next()
('1', '1')
>>> db.last()
('9', '81')
>>> db.set_location('2')
('2', '4')
>>> db.previous()
('1', '1')
>>> for k, v in db.iteritems():
...    print k, v
0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
>>> '8' in db
True
```
>>> db.sync()
0

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
13.12 dumbdbm -- Portable DBM implementation

**Note:** The dumbdbm module is intended as a last resort fallback for the anydbm module when no more robust module is available. The dumbdbm module is not written for speed and is not nearly as heavily used as the other database modules.

The dumbdbm module provides a persistent dictionary-like interface which is written entirely in Python. Unlike other modules such as gdbm and bsddb, no external library is required. As with other persistent mappings, the keys and values must always be strings.

The module defines the following:

**exception error**

Raised on dumbdbm-specific errors, such as I/O errors. KeyError is raised for general mapping errors like specifying an incorrect key.

**open**(*filename[, flag[, mode]]*)

Open a dumbdbm database and return a dumbdbm object. The filename argument is the basename of the database file (without any specific extensions). When a dumbdbm database is created, files with .dat and .dir extensions are created.

The optional flag argument is currently ignored; the database is always opened for update, and will be created if it does not exist.

The optional mode argument is the UNIX mode of the file, used only when the database has to be created. It defaults to octal 0666 (and will be modified by the prevailing umask). Changed in version 2.2: The mode argument was ignored in earlier versions.
See Also:

Module **anydbm**:  
Generic interface to dbm-style databases.

Module **dbm**:  
Similar interface to the DBM/NDBM library.

Module **gdbm**:  
Similar interface to the GNU GDBM library.

Module **shelve**:  
Persistence module which stores non-string data.

Module **whichdb**:  
Utility module used to determine the type of an existing database.

Subsections

- [13.12.1 Dumbdbm Objects](#)
13.12.1 Dumbdbm Objects

In addition to the methods provided by the UserDict.DictMixin class, dumbdbm objects provide the following methods.

**sync()**

Synchronize the on-disk directory and data files. This method is called by the sync method of Shelve objects.
13.13 sqlite3 -- DB-API 2.0 interface for SQLite databases

New in version 2.5.

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

pysqlite was written by Gerhard Häring and provides a SQL interface compliant with the DB-API 2.0 specification described by PEP 249.

To use the module, you must first create a Connection object that represents the database. Here the data will be stored in the /tmp/example file:

```python
conn = sqlite3.connect('/tmp/example')
```

You can also supply the special name "':memory:'" to create a database in RAM.

Once you have a Connection, you can create a Cursor object and call its execute() method to perform SQL commands:

```python
    c = conn.cursor()

    # Create table
    c.execute('''create table stocks
                   (date text, trans text, symbol text,
                    qty real, price real)''')

    # Insert a row of data
    c.execute("insert into stocks
                values ('2006-01-05','BUY','RHAT',100,35.14)"")
```

Usually your SQL operations will need to use values from Python variables. You shouldn't assemble your query using Python's string operations because doing so
is insecure; it makes your program vulnerable to an SQL injection attack.

Instead, use the DB-API's parameter substitution. Put "?" as a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor's `execute()` method. (Other database modules may use a different placeholder, such as "%s" or ":1".) For example:

```python
# Never do this -- insecure!
symbol = 'IBM'
c.execute("... where symbol = '%s'" % symbol)

# Do this instead
t = (symbol,)
c.execute('select * from stocks where symbol=?', t)

# Larger example
for t in (('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
          ('2006-04-05', 'BUY', 'MSOFT', 1000, 72.00),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
          ):
    c.execute('insert into stocks values (?, ?, ?, ?, ?)', t)
```

To retrieve data after executing a SELECT statement, you can either treat the cursor as an iterator, call the cursor's `fetchone()` method to retrieve a single matching row, or call `fetchall()` to get a list of the matching rows.

This example uses the iterator form:

```python
>>> c = conn.cursor()
>>> c.execute('select * from stocks order by price')
>>>
```

See Also:

[http://www.pysqlite.org](http://www.pysqlite.org)
The pysqlite web page.
http://www.sqlite.org

The SQLite web page; the documentation describes the syntax and the available data types for the supported SQL dialect.

PEP 249, Database API Specification 2.0

PEP written by Marc-André Lemburg.

Subsections

- 13.13.1 Module functions and constants
- 13.13.2 Connection Objects
- 13.13.3 Cursor Objects
- 13.13.4 SQLite and Python types
  - 13.13.4.1 Introduction
  - 13.13.4.2 Using adapters to store additional Python types in SQLite databases
    - 13.13.4.2.1 Letting your object adapt itself
    - 13.13.4.2.2 Registering an adapter callable
  - 13.13.4.3 Converting SQLite values to custom Python types
  - 13.13.4.4 Default adapters and converters
- 13.13.5 Controlling Transactions
- 13.13.6 Using pysqlite efficiently
  - 13.13.6.1 Using shortcut methods
  - 13.13.6.2 Accessing columns by name instead of by index
13.13.1 Module functions and constants

PARSE_DECLTYPES
This constant is meant to be used with the detect_types parameter of the connect function.

Setting it makes the sqlite3 module parse the declared type for each column it returns. It will parse out the first word of the declared type, i. e. for "integer primary key", it will parse out "integer". Then for that column, it will look into the converters dictionary and use the converter function registered for that type there. Converter names are case-sensitive!

PARSE_COLNAMES
This constant is meant to be used with the detect_types parameter of the connect function.

Setting this makes the SQLite interface parse the column name for each column it returns. It will look for a string formed [mytype] in there, and then decide that 'mytype' is the type of the column. It will try to find an entry of 'mytype' in the converters dictionary and then use the converter function found there to return the value. The column name found in cursor.description is only the first word of the column name, i. e. if you use something like 'as "x [datetime]"' in your SQL, then we will parse out everything until the first blank for the column name: the column name would simply be "x".

connect(database[, timeout, isolation_level, detect_types, factory])
Opens a connection to the SQLite database file database. You can use " :memory :" to open a database connection to a database that resides in RAM instead of on disk.

When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The timeout parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds).
For the *isolation_level* parameter, please see the *isolation_level* property of *Connection* objects in section 13.13.2.

SQLite natively supports only the types TEXT, INTEGER, FLOAT, BLOB and NULL. If you want to use other types, like you have to add support for them yourself. The *detect_types* parameter and the using custom *converters* registered with the module-level *register_converter* function allow you to easily do that.

*detect_types* defaults to 0 (i.e. off, no type detection), you can set it to any combination of *PARSE_DECLTYPES* and *PARSE_COLNAMES* to turn type detection on.

By default, the *sqlite3* module uses its *Connection* class for the connect call. You can, however, subclass the *Connection* class and make *connect* use your class instead by providing your class for the *factory* parameter.

Consult the section 13.13.4 of this manual for details.

The *sqlite3* module internally uses a statement cache to avoid SQL parsing overhead. If you want to explicitly set the number of statements that are cached for the connection, you can set the *cached_statements* parameter. The currently implemented default is to cache 100 statements.

**register_converter**( *typename*, *callable*)

 Registers a callable to convert a bytestring from the database into a custom Python type. The callable will be invoked for all database values that are of the type *typename*. Confer the parameter *detect_types* of the *connect* function for how the type detection works. Note that the case of *typename* and the name of the type in your query must match!

**register_adapter**( *type*, *callable*)

 Registers a callable to convert the custom Python type *type* into one of SQLite's supported types. The callable *callable* accepts as single parameter the Python value, and must return a value of the following types: int, long, float, str (UTF-8 encoded), unicode or buffer.

**complete_statement**( *sql*)
Returns `True` if the string `sql` one or more complete SQL statements terminated by semicolons. It does not verify if the SQL is syntactically correct, only if there are no unclosed string literals and if the statement is terminated by a semicolon.

This can be used to build a shell for SQLite, like in the following example:

```python
# A minimal SQLite shell for experiments
import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print "Enter your SQL commands to execute in sqlite3."
print "Enter a blank line to exit."

while True:
    line = raw_input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)

            if buffer.lstrip().upper().startswith("SELECT"):
                print cur.fetchall()
        except sqlite3.Error, e:
            print "An error occurred: ", e.args[0]
        buffer = ""

con.close()
```

Download as text (original file name: sqlite3/complete_statement.py).

**enable_callback_tracebacks(flag)**

By default you will not get any tracebacks in user-defined functions, aggregates, converters, authorizer callbacks etc. If you want to debug them, you can call this function with `flag` as True. Afterwards, you will get tracebacks from callbacks on `sys.stderr`. Use `False` to disable the feature again.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
13.13.2 Connection Objects

A Connection instance has the following attributes and methods:

**isolation_level**
Get or set the current isolation level. None for autocommit mode or one of "DEFERRED", "IMMEDIATE" or "EXCLUSIVE". See "Controlling Transactions", section 13.13.5, for a more detailed explanation.

**cursor**([cursorClass])
The cursor method accepts a single optional parameter cursorClass. If supplied, this must be a custom cursor class that extends sqlite3.Cursor.

**execute**(sql, [parameters])
This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's execute method with the parameters given.

**executemany**(sql, [parameters])
This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's executemany method with the parameters given.

**executescript**(sql_script)
This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's executescript method with the parameters given.

**create_function**(name, num_params, func)
Creates a user-defined function that you can later use from within SQL statements under the function name name. num_params is the number of parameters the function accepts, and func is a Python callable that is called as the SQL function.
The function can return any of the types supported by SQLite: unicode, str, int, long, float, buffer and None.

Example:

```python
import sqlite3
import md5

def md5sum(t):
    return md5.md5(t).hexdigest()

con = sqlite3.connect(":memory:"
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", ("foo",))
print cur.fetchone()[0]
```

create_aggregate(name, num_params, aggregate_class)

Creates a user-defined aggregate function.

The aggregate class must implement a step method, which accepts the number of parameters num_params, and a finalize method which will return the final result of the aggregate.

The finalize method can return any of the types supported by SQLite: unicode, str, int, long, float, buffer and None.

Example:

```python
import sqlite3

class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:"
con.create_aggregate("mysum", 1, MySum)
```
cur = con.cursor()
cur.execute("create table test(i)"")
cur.execute("insert into test(i) values (1)"")
cur.execute("insert into test(i) values (2)"")
cur.execute("select mysum(i) from test")
print cur.fetchone()[0]

create_collation( name, callable)

Creates a collation with the specified name and callable. The callable will be passed two string arguments. It should return -1 if the first is ordered lower than the second, 0 if they are ordered equal and 1 if the first is ordered higher than the second. Note that this controls sorting (ORDER BY in SQL) so your comparisons don't affect other SQL operations.

Note that the callable will get its parameters as Python bytestrings, which will normally be encoded in UTF-8.

The following example shows a custom collation that sorts "the wrong way":

import sqlite3
def collate_reverse(string1, string2):
    return -cmp(string1, string2)

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.cursor()
cur.execute("create table test(x)"")
cur.executemany("insert into test(x) values (?)", ["a",], ["b",])
cur.execute("select x from test order by x collate reverse")
for row in cur:
    print row
con.close()

To remove a collation, call create_collation with None as callable:

con.create_collation("reverse", None)
**interrupt()**

You can call this method from a different thread to abort any queries that might be executing on the connection. The query will then abort and the caller will get an exception.

**set_authorizer( authorizer_callback)**

This routine registers a callback. The callback is invoked for each attempt to access a column of a table in the database. The callback should return SQLITE_OK if access is allowed, SQLITE_DENY if the entire SQL statement should be aborted with an error and SQLITE_IGNORE if the column should be treated as a NULL value. These constants are available in the sqlite3 module.

The first argument to the callback signifies what kind of operation is to be authorized. The second and third argument will be arguments or None depending on the first argument. The 4th argument is the name of the database ("main", "temp", etc.) if applicable. The 5th argument is the name of the inner-most trigger or view that is responsible for the access attempt or None if this access attempt is directly from input SQL code.

Please consult the SQLite documentation about the possible values for the first argument and the meaning of the second and third argument depending on the first one. All necessary constants are available in the sqlite3 module.

**row_factory**

You can change this attribute to a callable that accepts the cursor and the original row as a tuple and will return the real result row. This way, you can implement more advanced ways of returning results, such as returning an object that can also access columns by name.

Example:

```python
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
```

return d

con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print cur.fetchone()['a']

Download as text (original file name: sqlite3/row_factory.py).

If returning a tuple doesn't suffice and you want name-based access to columns, you should consider setting `row_factory` to the highly-optimized `sqlite3.Row` type. `Row` provides both index-based and case-insensitive name-based access to columns with almost no memory overhead. It will probably be better than your own custom dictionary-based approach or even a `db_row` based solution.

**text_factory**

Using this attribute you can control what objects are returned for the TEXT data type. By default, this attribute is set to `unicode` and the `sqlite3` module will return Unicode objects for TEXT. If you want to return bytestrings instead, you can set it to `str`.

For efficiency reasons, there's also a way to return Unicode objects only for non-ASCII data, and bytestrings otherwise. To activate it, set this attribute to `sqlite3.OptimizedUnicode`.

You can also set it to any other callable that accepts a single bytestring parameter and returns the resulting object.

See the following example code for illustration:

```python
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()

# Create the table
con.execute("create table person(lastname, firstname)"

AUSTRIA = u"\xd6sterreich"

# by default, rows are returned as Unicode
cur.execute("select ?, (AUSTRIA,)
row = cur.fetchone()
```
assert row[0] == AUSTRIA

# but we can make pysqlite always return bytestrings ...
con.text_factory = str
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) == str
# the bytestrings will be encoded in UTF-8, unless you stored garbage in database ...
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that will ignore Unicode characters that cannot be decoded from UTF-8
con.text_factory = lambda x: unicode(x, "utf-8", "ignore")
cur.execute("select ?", ("this is latin1 and would normally create errors" + u"€äö".encode("latin1"),))
row = cur.fetchone()
assert type(row[0]) == unicode

# pysqlite offers a builtin optimized text_factory that will return bytestring objects, if the data is in ASCII only, and otherwise return unicode objects
con.text_factory = sqlite3.OptimizedUnicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) == unicode

cur.execute("select ?", ("Germany",))
row = cur.fetchone()
assert type(row[0]) == str

Download as text (original file name: sqlite3/text_factory.py).

total_changes

Returns the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.
13.13.3 Cursor Objects

A Cursor instance has the following attributes and methods:

**execute** *(sql, [parameters])*  
Executes a SQL statement. The SQL statement may be parametrized (i.e., placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks (qmark style) and named placeholders (named style).

This example shows how to use parameters with qmark style:

```
import sqlite3
con = sqlite3.connect("mydb")
cur = con.cursor()
who = "Yeltsin"
age = 72

cur.execute("select name_last, age from people where name_last=? and age=?", (who, age))
print cur.fetchone()
```

Download as text (original file name: sqlite3/execute_1.py).

This example shows how to use the named style:

```
import sqlite3
con = sqlite3.connect("mydb")
cur = con.cursor()
who = "Yeltsin"
age = 72

cur.execute("select name_last, age from people where name_last=:who and age=:age",
            {"who": who, "age": age})
print cur.fetchone()
```

Download as text (original file name: sqlite3/execute_2.py).
execute() will only execute a single SQL statement. If you try to execute more than one statement with it, it will raise a Warning. Use executescript() if you want to execute multiple SQL statements with one call.

executemany(
    sql, seq_of_parameters
)
Executes a SQL command against all parameter sequences or mappings found in the sequence sql. The sqlite3 module also allows using an iterator yielding parameters instead of a sequence.

```python
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def next(self):  # sometimes next is used instead of __next__
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),)  # this is a 1-tuple

con = sqlite3.connect(':memory:)
cur = con.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

cur.execute("select c from characters")
print cur.fetchall()
```

Here's a shorter example using a generator:

```python
import sqlite3

def char_generator():
    import string
    for c in string.letters[:26]:
        yield (c,

con = sqlite3.connect(':memory:)
```
```
cur = con.cursor()
cur.execute("create table characters(c)")
cur.executemany("insert into characters(c) values (?)", char_generator())
cur.execute("select c from characters")
print cur.fetchall()
```

**executescript( sql_script)**

This is a nonstandard convenience method for executing multiple SQL statements at once. It issues a COMMIT statement first, then executes the SQL script it gets as a parameter.

*sql_script* can be a bytestring or a Unicode string.

Example:

```python
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript(""
    create table person(
        firstname,
        lastname,
        age
    );

    create table book(
        title,
        author,
        published
    );

    "")
```

Download as text (original file name: sqlite3/executescript.py).
rowcount

Although the Cursor class of the sqlite3 module implements this attribute, the database engine's own support for the determination of "rows affected"/"rows selected" is quirky.

For SELECT statements, rowcount is always None because we cannot determine the number of rows a query produced until all rows were fetched.

For DELETE statements, SQLite reports rowcount as 0 if you make a DELETE FROM table without any condition.

For executemany statements, the number of modifications are summed up into rowcount.

As required by the Python DB API Spec, the rowcount attribute "is -1 in case no executeXX() has been performed on the cursor or the rowcount of the last operation is not determinable by the interface".

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
13.13.4 SQLite and Python types

Subsections

- 13.13.4.1 Introduction
- 13.13.4.2 Using adapters to store additional Python types in SQLite databases
  - 13.13.4.2.1 Letting your object adapt itself
  - 13.13.4.2.2 Registering an adapter callable
- 13.13.4.3 Converting SQLite values to custom Python types
- 13.13.4.4 Default adapters and converters

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
13.13.5 Controlling Transactions

By default, the sqlite3 module opens transactions implicitly before a Data Modification Language (DML) statement (i.e. INSERT/UPDATE/DELETE/REPLACE), and commits transactions implicitly before a non-DML, non-query statement (i.e. anything other than SELECT/INSERT/UPDATE/DELETE/REPLACE).

So if you are within a transaction and issue a command like CREATE TABLE ..., VACUUM, PRAGMA, the sqlite3 module will commit implicitly before executing that command. There are two reasons for doing that. The first is that some of these commands don't work within transactions. The other reason is that pysqlite needs to keep track of the transaction state (if a transaction is active or not).

You can control which kind of "BEGIN" statements pysqlite implicitly executes (or none at all) via the isolation_level parameter to the connect call, or via the isolation_level property of connections.

If you want autocommit mode, then set isolation_level to None.

Otherwise leave it at its default, which will result in a plain "BEGIN" statement, or set it to one of SQLite's supported isolation levels: DEFERRED, IMMEDIATE or EXCLUSIVE.

As the sqlite3 module needs to keep track of the transaction state, you should not use OR ROLLBACK or ON CONFLICT ROLLBACK in your SQL. Instead, catch the IntegrityError and call the rollback method of the connection yourself.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
Using shortcut methods
13.13.6 Using pysqlite efficiently

Subsections

- 13.13.6.1 Using shortcut methods
- 13.13.6.2 Accessing columns by name instead of by index

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14. Generic Operating System Services

The modules described in this chapter provide interfaces to operating system features that are available on (almost) all operating systems, such as files and a clock. The interfaces are generally modeled after the UNIX or C interfaces, but they are available on most other systems as well. Here's an overview:

- **os**
  - Miscellaneous operating system interfaces.
- **time**
  - Time access and conversions.
- **optparse**
  - More convenient, flexible, and powerful command-line parsing library.
- **getopt**
  - Portable parser for command line options; support both short and long option names.
- **logging**
  - Logging module for Python based on PEP 282.
- **getpass**
  - Portable reading of passwords and retrieval of the userid.
- **curses**
  - An interface to the curses library, providing portable terminal handling.
- **curses.textpad**
  - Emacs-like input editing in a curses window.
- **curses.wrapper**
  - Terminal configuration wrapper for curses programs.
- **curses.ascii**
  - Constants and set-membership functions for ASCII characters.
- **curses.panel**
  - A panel stack extension that adds depth to curses windows.
- **platform**
  - Retrieves as much platform identifying data as possible.
- **errno**
  - Standard errno system symbols.
- **ctypes**
  - A foreign function library for Python.
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
14.1 os -- Miscellaneous operating system interfaces

This module provides a more portable way of using operating system dependent functionality than importing a operating system dependent built-in module like posix or nt.

This module searches for an operating system dependent built-in module like mac or posix and exports the same functions and data as found there. The design of all Python's built-in operating system dependent modules is such that as long as the same functionality is available, it uses the same interface; for example, the function os.stat(path) returns stat information about path in the same format (which happens to have originated with the POSIX interface).

Extensions peculiar to a particular operating system are also available through the os module, but using them is of course a threat to portability!

Note that after the first time os is imported, there is no performance penalty in using functions from os instead of directly from the operating system dependent built-in module, so there should be no reason not to use os!

The os module contains many functions and data values. The items below and in the following sub-sections are all available directly from the os module.

exception error
This exception is raised when a function returns a system-related error (not for illegal argument types or other incidental errors). This is also known as the built-in exception OSError. The accompanying value is a pair containing the numeric error code from errno and the corresponding string, as would be printed by the C function perror(). See the module errno, which contains names for the error codes defined by the underlying operating system.

When exceptions are classes, this exception carries two attributes, errno and strerror. The first holds the value of the C errno variable, and the latter holds the corresponding error message from strerror(). For
exceptions that involve a file system path (such as chdir() or unlink()), the exception instance will contain a third attribute, filename, which is the file name passed to the function.

**name**
The name of the operating system dependent module imported. The following names have currently been registered: 'posix', 'nt', 'mac', 'os2', 'ce', 'java', 'riscos'.

**path**
The corresponding operating system dependent standard module for pathname operations, such as posixpath or macpath. Thus, given the proper imports, os.path.split(file) is equivalent to but more portable than posixpath.split(file). Note that this is also an importable module: it may be imported directly as os.path.

---

**Subsections**

- 14.1.1 Process Parameters
- 14.1.2 File Object Creation
- 14.1.3 File Descriptor Operations
- 14.1.4 Files and Directories
- 14.1.5 Process Management
- 14.1.6 Miscellaneous System Information
- 14.1.7 Miscellaneous Functions

---

**Python Library Reference**


*Release 2.5, documentation updated on 19th September, 2006.*

*See About this document... for information on suggesting changes.*
14.1.1 Process Parameters

These functions and data items provide information and operate on the current process and user.

environ

A mapping object representing the string environment. For example, `environ['HOME']` is the pathname of your home directory (on some platforms), and is equivalent to `getenv("HOME")` in C.

This mapping is captured the first time the os module is imported, typically during Python startup as part of processing `site.py`. Changes to the environment made after this time are not reflected in `os.environ`, except for changes made by modifying `os.environ` directly.

If the platform supports the `putenv()` function, this mapping may be used to modify the environment as well as query the environment. `putenv()` will be called automatically when the mapping is modified. **Note:** Calling `putenv()` directly does not change `os.environ`, so it's better to modify `os.environ`. **Note:** On some platforms, including FreeBSD and Mac OS X, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv()`.

If `putenv()` is not provided, a modified copy of this mapping may be passed to the appropriate process-creation functions to cause child processes to use a modified environment.

If the platform supports the `unsetenv()` function, you can delete items in this mapping to unset environment variables. `unsetenv()` will be called automatically when an item is deleted from `os.environ`.

chdir(path)
fchdir(fd)
getcwd()

These functions are described in ``Files and Directories'' (section 14.1.4).
ctermsid()  
Return the filename corresponding to the controlling terminal of the process. Availability: UNIX.

gtegid()  
Return the effective group id of the current process. This corresponds to the `set id' bit on the file being executed in the current process. Availability: UNIX.

gteuido()  
Return the current process' effective user id. Availability: UNIX.

gtegid()  
Return the real group id of the current process. Availability: UNIX.

gtegroups()  
Return list of supplemental group ids associated with the current process. Availability: UNIX.

gtelogin()  
Return the name of the user logged in on the controlling terminal of the process. For most purposes, it is more useful to use the environment variable LOGNAME to find out who the user is, or 
pwd.getpwuid(os.getuid())[0] to get the login name of the currently effective user ID. Availability: UNIX.

gtepgid(pid)  
Return the process group id of the process with process id pid. If pid is 0, the process group id of the current process is returned. Availability: UNIX. New in version 2.3.

gtepggrp()  
Return the id of the current process group. Availability: UNIX.

gtepid()  
Return the current process id. Availability: UNIX, Windows.
getppid()
Return the parent's process id. Availability: UNIX.

getuid()
Return the current process' user id. Availability: UNIX.

getenv( varname[, value])
Return the value of the environment variable varname if it exists, or value if it doesn't. value defaults to None. Availability: most flavors of UNIX, Windows.

putenv( varname, value)
Set the environment variable named varname to the string value. Such changes to the environment affect subprocesses started with os.system(), popen() or fork() and execv(). Availability: most flavors of UNIX, Windows.

Note: On some platforms, including FreeBSD and Mac OS X, setting environ may cause memory leaks. Refer to the system documentation for putenv.

When putenv() is supported, assignments to items in os.environ are automatically translated into corresponding calls to putenv(); however, calls to putenv() don't update os.environ, so it is actually preferable to assign to items of os.environ.

setegid( egid)
Set the current process's effective group id. Availability: UNIX.

seteuid( euid)
Set the current process's effective user id. Availability: UNIX.

setgid( gid)
Set the current process' group id. Availability: UNIX.

setgroups( groups)
Set the list of supplemental group ids associated with the current process to
groups. groups must be a sequence, and each element must be an integer identifying a group. This operation is typical available only to the superuser. Availability: UNIX. New in version 2.2.

setpgrp()
Calls the system call setpgrp() or setpgrp(0, 0) depending on which version is implemented (if any). See the UNIX manual for the semantics. Availability: UNIX.

setpgid(pid, pgrp)
Calls the system call setpgid() to set the process group id of the process with id pid to the process group with id pgrp. See the UNIX manual for the semantics. Availability: UNIX.

setreuid(ruid, euid)
Set the current process's real and effective user ids. Availability: UNIX.

setregid(rgid, egid)
Set the current process's real and effective group ids. Availability: UNIX.

getsid(pid)
Calls the system call getsid(). See the UNIX manual for the semantics. Availability: UNIX. New in version 2.4.

setsid()
Calls the system call setsid(). See the UNIX manual for the semantics. Availability: UNIX.

setuid(uid)
Set the current process' user id. Availability: UNIX.

strerror(code)
Return the error message corresponding to the error code in code. Availability: UNIX, Windows.

umask(mask)
Set the current numeric umask and returns the previous umask. Availability:
UNIX, Windows.

uname()
Return a 5-tuple containing information identifying the current operating system. The tuple contains 5 strings: (sysname, nodename, release, version, machine). Some systems truncate the nodename to 8 characters or to the leading component; a better way to get the hostname is socket.gethostname() or even socket.gethostbyaddr(socket.gethostname()). Availability: recent flavors of UNIX.

unsetenv( varname)
Unset (delete) the environment variable named varname. Such changes to the environment affect subprocesses started with os.system(), popen() or fork() and execv(). Availability: most flavors of UNIX, Windows.

When unsetenv() is supported, deletion of items in os.environ is automatically translated into a corresponding call to unsetenv(); however, calls to unsetenv() don't update os.environ, so it is actually preferable to delete items of os.environ.
14.1.2 File Object Creation

These functions create new file objects.

\textbf{fdopen(} \textit{fd[, mode[, bufsize]]})

Return an open file object connected to the file descriptor \textit{fd}. The \textit{mode} and \textit{bufsize} arguments have the same meaning as the corresponding arguments to the built-in \texttt{open( )} function. Availability: Macintosh, UNIX, Windows.

Changed in version 2.3: When specified, the \textit{mode} argument must now start with one of the letters "r", "w", or "a", otherwise a \texttt{ValueError} is raised. Changed in version 2.5: On UNIX, when the \textit{mode} argument starts with "a", the \texttt{O_APPEND} flag is set on the file descriptor (which the \texttt{fdopen( )} implementation already does on most platforms).

\textbf{popen(} \textit{command[, mode[, bufsize]]})

Open a pipe to or from \textit{command}. The return value is an open file object connected to the pipe, which can be read or written depending on whether \textit{mode} is 'r' (default) or 'w'. The \textit{bufsize} argument has the same meaning as the corresponding argument to the built-in \texttt{open( )} function. The exit status of the command (encoded in the format specified for \texttt{wait( )}) is available as the return value of the \texttt{close( )} method of the file object, except that when the exit status is zero (termination without errors), \texttt{None} is returned. Availability: Macintosh, UNIX, Windows.

Changed in version 2.0: This function worked unreliably under Windows in earlier versions of Python. This was due to the use of the \texttt{_popen( )} function from the libraries provided with Windows. Newer versions of Python do not use the broken implementation from the Windows libraries.

\textbf{tmpfile( )}

Return a new file object opened in update mode ("w+b"). The file has no directory entries associated with it and will be automatically deleted once there are no file descriptors for the file. Availability: Macintosh, UNIX, Windows.
For each of the following `popen()` variants, if *bufsize* is specified, it specifies
the buffer size for the I/O pipes. *mode*, if provided, should be the string 'b' or
't'; on Windows this is needed to determine whether the file objects should be
opened in binary or text mode. The default value for *mode* is 't'.

Also, for each of these variants, on UNIX, *cmd* may be a sequence, in which case
arguments will be passed directly to the program without shell intervention (as
with `os.spawnv()`). If *cmd* is a string it will be passed to the shell (as with
`os.system()`).

These methods do not make it possible to retrieve the exit status from the child
processes. The only way to control the input and output streams and also retrieve
the return codes is to use the `Popen3` and `Popen4` classes from the `popen2`
module; these are only available on UNIX.

For a discussion of possible deadlock conditions related to the use of these
functions, see ``Flow Control Issues'' (section 17.4.2).

**popen2**: `popen2(cmd[, mode[, bufsize]])`


**popen3**: `popen3(cmd[, mode[, bufsize]])`


**popen4**: `popen4(cmd[, mode[, bufsize]])`


(Note that *child_stdin*, *child_stdout*, and *child_stderr* are named from the
point of view of the child process, so *child_stdin* is the child's standard input.)

This functionality is also available in the `popen2` module using functions of the
same names, but the return values of those functions have a different order.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.1.3 File Descriptor Operations

These functions operate on I/O streams referenced using file descriptors.

File descriptors are small integers corresponding to a file that has been opened by the current process. For example, standard input is usually file descriptor 0, standard output is 1, and standard error is 2. Further files opened by a process will then be assigned 3, 4, 5, and so forth. The name "file descriptor" is slightly deceptive; on UNIX platforms, sockets and pipes are also referenced by file descriptors.

**close** *(fd)*


**Note:** This function is intended for low-level I/O and must be applied to a file descriptor as returned by open() or pipe(). To close a "file object" returned by the built-in function open() or by popen() or fdopen(), use its close() method.

**dup** *(fd)*

Return a duplicate of file descriptor *fd*. Availability: Macintosh, UNIX, Windows.

**dup2** *(fd, fd2)*

Duplicate file descriptor *fd* to *fd2*, closing the latter first if necessary. Availability: Macintosh, UNIX, Windows.

**fdatasync** *(fd)*

Force write of file with file descriptor *fd* to disk. Does not force update of metadata. Availability: UNIX.

**fpathconf** *(fd, name)*

Return system configuration information relevant to an open file. *name* specifies the configuration value to retrieve; it may be a string which is the
name of a defined system value; these names are specified in a number of standards (POSIX.1, UNIX 95, UNIX 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the pathconf_names dictionary. For configuration variables not included in that mapping, passing an integer for name is also accepted. Availability: Macintosh, UNIX.

If name is a string and is not known, ValueError is raised. If a specific value for name is not supported by the host system, even if it is included in pathconf_names, an OSError is raised with errno.EINVAL for the error number.

fstat(fd)
Return status for file descriptor fd, like stat(). Availability: Macintosh, UNIX, Windows.

fstatvfs(fd)
Return information about the filesystem containing the file associated with file descriptor fd, like statvfs(). Availability: UNIX.

fsync(fd)
Force write of file with file descriptor fd to disk. On UNIX, this calls the native fsync() function; on Windows, the MS _commit() function.

If you're starting with a Python file object f, first do f.flush(), and then do os.fsync(f.fileno()), to ensure that all internal buffers associated with f are written to disk. Availability: Macintosh, UNIX, and Windows starting in 2.2.3.

ftruncate(fd, length)
Truncate the file corresponding to file descriptor fd, so that it is at most length bytes in size. Availability: Macintosh, UNIX.

isatty(fd)
Return True if the file descriptor fd is open and connected to a tty(-like) device, else False. Availability: Macintosh, UNIX.

lseek(fd, pos, how)
Set the current position of file descriptor \textit{fd} to position \textit{pos}, modified by \textit{how}: 0 to set the position relative to the beginning of the file; 1 to set it relative to the current position; 2 to set it relative to the end of the file. Availability: Macintosh, UNIX, Windows.

\textbf{open(} file, flags[, mode]\textbf{)}

Open the file \textit{file} and set various flags according to \textit{flags} and possibly its mode according to \textit{mode}. The default \textit{mode} is 0777 (octal), and the current umask value is first masked out. Return the file descriptor for the newly opened file. Availability: Macintosh, UNIX, Windows.

For a description of the flag and mode values, see the C run-time documentation; flag constants (like O_RDONLY and O_WRONLY) are defined in this module too (see below).

\textbf{Note:} This function is intended for low-level I/O. For normal usage, use the built-in function \texttt{open()}, which returns a "file object" with \texttt{read()} and \texttt{write()} methods (and many more). To wrap a file descriptor in a "file object", use \texttt{fdopen()}.

\textbf{openpty(\textbf{)}}

Open a new pseudo-terminal pair. Return a pair of file descriptors (\textit{master}, \textit{slave}) for the pty and the tty, respectively. For a (slightly) more portable approach, use the \texttt{pty} module. Availability: Macintosh, Some flavors of UNIX.

\textbf{pipe()} 

Create a pipe. Return a pair of file descriptors (\textit{r}, \textit{w}) usable for reading and writing, respectively. Availability: Macintosh, UNIX, Windows.

\textbf{read(} fd, n\textbf{)}

Read at most \textit{n} bytes from file descriptor \textit{fd}. Return a string containing the bytes read. If the end of the file referred to by \textit{fd} has been reached, an empty string is returned. Availability: Macintosh, UNIX, Windows.
Note: This function is intended for low-level I/O and must be applied to a file descriptor as returned by open() or pipe(). To read a ``file object'' returned by the built-in function open() or by popen() or fdopen(), or sys.stdin, use its read() or readline() methods.

tcgetpgrp(fd)
Return the process group associated with the terminal given by fd (an open file descriptor as returned by open()). Availability: Macintosh, UNIX.

tcsetpgrp(fd, pg)
Set the process group associated with the terminal given by fd (an open file descriptor as returned by open()) to pg. Availability: Macintosh, UNIX.

ttyname(fd)
Return a string which specifies the terminal device associated with file-descriptor fd. If fd is not associated with a terminal device, an exception is raised. Availability: Macintosh, UNIX.

write(fd, str)
Write the string str to file descriptor fd. Return the number of bytes actually written. Availability: Macintosh, UNIX, Windows.

Note: This function is intended for low-level I/O and must be applied to a file descriptor as returned by open() or pipe(). To write a ``file object'' returned by the built-in function open() or by popen() or fdopen(), or sys.stdout or sys.stderr, use its write() method.

The following data items are available for use in constructing the flags parameter to the open() function. Some items will not be available on all platforms. For descriptions of their availability and use, consult open(2).

O_RDONLY
O_WRONLY
0_RDONLY
0_APPEND
0_CREAT
0_EXCL
0_TRUNC

Options for the flag argument to the open() function. These can be bit-wise OR'd together. Availability: Macintosh, UNIX, Windows.

0_DSYNC
0_RSYNC
0_SYNC
0_NDELAY
0_NONBLOCK
0_NOCTTY
0_SHLOCK
0_EXLOCK

More options for the flag argument to the open() function. Availability: Macintosh, UNIX.

0_BINARY

Option for the flag argument to the open() function. This can be bit-wise OR'd together with those listed above. Availability: Windows.

0_NOINHERIT
0_SHORT_LIVED
0_TEMPORARY
0_RANDOM
0_SEQUENTIAL
0_TEXT

Options for the flag argument to the open() function. These can be bit-wise OR'd together. Availability: Windows.

SEEK_SET
SEEK_CUR
SEEK_END

Parameters to the lseek() function. Their values are 0, 1, and 2, respectively. Availability: Windows, Macintosh, UNIX. New in version 2.5.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.1.4 Files and Directories

**access( path, mode)**
Use the real uid/gid to test for access to path. Note that most operations will use the effective uid/gid, therefore this routine can be used in a suid/sgid environment to test if the invoking user has the specified access to path.

*mode* should be **F_OK** to test the existence of path, or it can be the inclusive OR of one or more of **R_OK**, **W_OK**, and **X_OK** to test permissions. Return **True** if access is allowed, **False** if not. See the UNIX man page *access*(2) for more information. Availability: Macintosh, UNIX, Windows.

**Note:** Using *access()* to check if a user is authorized to e.g. open a file before actually doing so using *open()* creates a security hole, because the user might exploit the short time interval between checking and opening the file to manipulate it.

**Note:** I/O operations may fail even when *access()* indicates that they would succeed, particularly for operations on network filesystems which may have permissions semantics beyond the usual POSIX permission-bit model.

**F_OK**
Value to pass as the *mode* parameter of *access()* to test the existence of path.

**R_OK**
Value to include in the *mode* parameter of *access()* to test the readability of path.

**W_OK**
Value to include in the *mode* parameter of *access()* to test the writability of path.

**X_OK**
Value to include in the *mode* parameter of *access()* to determine if path can be executed.
chdir(*path*)
Change the current working directory to *path*. Availability: Macintosh, *UNIX*, Windows.

fchdir(*fd*)
Change the current working directory to the directory represented by the file descriptor *fd*. The descriptor must refer to an opened directory, not an open file. Availability: *UNIX*. New in version 2.3.

getcwd()

getcwd()

chroot(*path*)

chmod(*path*, *mode*)
Change the mode of *path* to the numeric *mode*. *mode* may take one of the following values (as defined in the *stat* module) or bitwise or-ed combinations of them:
- S_ISUID
- S_ISGID
- S_ENFMT
- S_ISVTX
- S_IREAD
- S_IWRITE
- S_IEXEC
- S_IRWXU
- S_IRUSR
- S_IWUSR
- S_IXUSR
- S_IRWXG
• S_IRGRP
• S_IWGRP
• S_IXGRP
• S_IRWXO
• S_IROTH
• S_IWOTH
• S_IXOTH

Availability: Macintosh, UNIX, Windows.

Note: Although Windows supports chmod(), you can only set the file's read-only flag with it (via the S_IWRITE and S_IREAD constants or a corresponding integer value). All other bits are ignored.

chown(path, uid, gid)
Change the owner and group id of path to the numeric uid and gid. To leave one of the ids unchanged, set it to -1. Availability: Macintosh, UNIX.

lchown(path, uid, gid)
Change the owner and group id of path to the numeric uid and gid. This function will not follow symbolic links. Availability: Macintosh, UNIX. New in version 2.3.

link(src, dst)
Create a hard link pointing to src named dst. Availability: Macintosh, UNIX.

listdir(path)
Return a list containing the names of the entries in the directory. The list is in arbitrary order. It does not include the special entries '.' and '..' even if they are present in the directory. Availability: Macintosh, UNIX, Windows.

Changed in version 2.3: On Windows NT/2k/XP and UNIX, if path is a Unicode object, the result will be a list of Unicode objects.

lstat(path)
Like stat(), but do not follow symbolic links. Availability: Macintosh, UNIX.
**mknod**( filename[, mode=0600, device])

Create a filesystem node (file, device special file or named pipe) named filename. mode specifies both the permissions to use and the type of node to be created, being combined (bitwise OR) with one of S_IFREG, S_IFCHR, S_IFBLK, and S_IFIFO (those constants are available in stat). For S_IFCHR and S_IFBLK, device defines the newly created device special file (probably using os.makedev()), otherwise it is ignored. New in version 2.3.

**major**( device)

Extracts the device major number from a raw device number (usually the st_dev or st_rdev field from stat). New in version 2.3.

**minor**( device)

Extracts the device minor number from a raw device number (usually the st_dev or st_rdev field from stat). New in version 2.3.

**makedev**( major, minor)

Composes a raw device number from the major and minor device numbers. New in version 2.3.

**mkdir**( path[, mode])

Create a directory named path with numeric mode mode. The default mode is 0777 (octal). On some systems, mode is ignored. Where it is used, the current umask value is first masked out. Availability: Macintosh, UNIX.

**mkfifo**( path[, mode])

Create a FIFO (a named pipe) named path with numeric mode mode. The default mode is 0666 (octal). The current umask value is first masked out from the mode. Availability: Macintosh, UNIX.
Windows.

\texttt{makedirs(path[, mode])}

Recursive directory creation function. Like \texttt{mkdir()}, but makes all intermediate-level directories needed to contain the leaf directory. Throws an \texttt{error} exception if the leaf directory already exists or cannot be created. The default \texttt{mode} is \texttt{0777} (octal). On some systems, \texttt{mode} is ignored. Where it is used, the current umask value is first masked out. \textbf{Note:} \texttt{makedirs()} will become confused if the path elements to create include \texttt{os.pardir}. New in version 1.5.2. Changed in version 2.3: This function now handles UNC paths correctly.

\texttt{pathconf(path, name)}

Return system configuration information relevant to a named file. \texttt{name} specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, \texttt{UNIX} 95, \texttt{UNIX} 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the \texttt{pathconf_names} dictionary. For configuration variables not included in that mapping, passing an integer for \texttt{name} is also accepted. Availability: Macintosh, \texttt{UNIX}.

If \texttt{name} is a string and is not known, \texttt{ValueError} is raised. If a specific value for \texttt{name} is not supported by the host system, even if it is included in \texttt{pathconf_names}, an \texttt{OSError} is raised with \texttt{errno.EINVAL} for the error number.

\texttt{pathconf_names}

Dictionary mapping names accepted by \texttt{pathconf()} and \texttt{fpathconf()} to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system. Availability: Macintosh, \texttt{UNIX}.

\texttt{readlink(path)}

Return a string representing the path to which the symbolic link points. The result may be either an absolute or relative pathname; if it is relative, it may be converted to an absolute pathname using \texttt{os.path.join(os.path.dirname(path), result)}. Availability:
Macintosh, UNIX.

remove(path)
Remove the file path. If path is a directory, OSError is raised; see rmdir() below to remove a directory. This is identical to the unlink() function documented below. On Windows, attempting to remove a file that is in use causes an exception to be raised; on UNIX, the directory entry is removed but the storage allocated to the file is not made available until the original file is no longer in use. Availability: Macintosh, UNIX, Windows.

removedirs(path)
Removes directories recursively. Works like rmdir() except that, if the leaf directory is successfully removed, removedirs() tries to successively remove every parent directory mentioned in path until an error is raised (which is ignored, because it generally means that a parent directory is not empty). For example, "os.removedirs('foo/bar/baz')" will first remove the directory "'foo/bar/baz'", and then remove "'foo/bar'"and "'foo'" if they are empty. Raises OSError if the leaf directory could not be successfully removed. New in version 1.5.2.

rename(src, dst)
Rename the file or directory src to dst. If dst is a directory, OSError will be raised. On UNIX, if dst exists and is a file, it will be removed silently if the user has permission. The operation may fail on some UNIX flavors if src and dst are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement). On Windows, if dst already exists, OSError will be raised even if it is a file; there may be no way to implement an atomic rename when dst names an existing file. Availability: Macintosh, UNIX, Windows.

renames(old, new)
Recursive directory or file renaming function. Works like rename(), except creation of any intermediate directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be pruned away using removedirs(). New in version 1.5.2.
**Note:** This function can fail with the new directory structure made if you lack permissions needed to remove the leaf directory or file.

`rmdir(path)`
Remove the directory path. Availability: Macintosh, UNIX, Windows.

`stat(path)`
Perform a `stat()` system call on the given path. The return value is an object whose attributes correspond to the members of the `stat` structure, namely: `st_mode` (protection bits), `st_ino` (inode number), `st_dev` (device), `st_nlink` (number of hard links), `st_uid` (user ID of owner), `st_gid` (group ID of owner), `st_size` (size of file, in bytes), `st_atime` (time of most recent access), `st_mtime` (time of most recent content modification), `st_ctime` (platform dependent; time of most recent metadata change on UNIX, or the time of creation on Windows):

```python
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
(33188, 422511L, 769L, 1, 1032, 100, 926L, 1105022698, 1105022732,
  1105022732)
>>> statinfo.st_size
926L
```

Changed in version 2.3: If `stat_float_times` returns true, the time values are floats, measuring seconds. Fractions of a second may be reported if the system supports that. On Mac OS, the times are always floats. See `stat_float_times` for further discussion.

On some UNIX systems (such as Linux), the following attributes may also be available: `st_blocks` (number of blocks allocated for file), `st_blksize` (filesystem blocksize), `st_rdev` (type of device if an inode device). `st_flags` (user defined flags for file).

On other UNIX systems (such as FreeBSD), the following attributes may be available (but may be only filled out if root tries to use them): `st_gen` (file generation number), `st_birthtime` (time of file creation).
On Mac OS systems, the following attributes may also be available:
*st_rsize*, *st_creator*, *st_type*.

On RISCOS systems, the following attributes are also available:
*st_ftype* (file type), *st_attr* (attributes), *st_obtype* (object type).

For backward compatibility, the return value of *stat()* is also accessible as a tuple of at least 10 integers giving the most important (and portable) members of the *stat* structure, in the order *st_mode*, *st_ino*,
*st_dev*, *st_nlink*, *st_uid*, *st_gid*, *st_size*, *st_atime*,
*st_mtime*, *st_ctime*. More items may be added at the end by some implementations. The standard module *stat* defines functions and constants that are useful for extracting information from a *stat* structure. (On Windows, some items are filled with dummy values.)

**Note:** The exact meaning and resolution of the *st_atime*, *st_mtime*, and *st_ctime* members depends on the operating system and the file system. For example, on Windows systems using the FAT or FAT32 file systems, *st_mtime* has 2-second resolution, and *st_atime* has only 1-day resolution. See your operating system documentation for details.

Availability: Macintosh, UNIX, Windows.

Changed in version 2.2: Added access to values as attributes of the returned object. Changed in version 2.5: Added *st_gen*, *st_birthtime*.

**stat_float_times**([*newvalue]*)

Determine whether *stat_result* represents time stamps as float objects. If *newvalue* is *True*, future calls to *stat()* return floats, if it is *False*, future calls return ints. If *newvalue* is omitted, return the current setting.

For compatibility with older Python versions, accessing *stat_result* as a tuple always returns integers.

Changed in version 2.5: Python now returns float values by default. Applications which do not work correctly with floating point time stamps can use this function to restore the old behaviour.

The resolution of the timestamps (that is the smallest possible fraction)
depends on the system. Some systems only support second resolution; on these systems, the fraction will always be zero.

It is recommended that this setting is only changed at program startup time in the `__main__` module; libraries should never change this setting. If an application uses a library that works incorrectly if floating point time stamps are processed, this application should turn the feature off until the library has been corrected.

**statvfs**( path)

Perform a `statvfs()` system call on the given path. The return value is an object whose attributes describe the filesystem on the given path, and correspond to the members of the `statvfs` structure, namely: `f_bsize`, `f_frsize`, `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`. Availability: UNIX.

For backward compatibility, the return value is also accessible as a tuple whose values correspond to the attributes, in the order given above. The standard module `statvfs` defines constants that are useful for extracting information from a `statvfs` structure when accessing it as a sequence; this remains useful when writing code that needs to work with versions of Python that don't support accessing the fields as attributes.

Changed in version 2.2: Added access to values as attributes of the returned object.

**symlink**( src, dst)

Create a symbolic link pointing to `src` named `dst`. Availability: UNIX.

**tempnam**( [dir[, prefix]])

Return a unique path name that is reasonable for creating a temporary file. This will be an absolute path that names a potential directory entry in the directory `dir` or a common location for temporary files if `dir` is omitted or `None`. If given and not `None`, `prefix` is used to provide a short prefix to the filename. Applications are responsible for properly creating and managing files created using paths returned by `tempnam()`; no automatic cleanup is provided. On UNIX, the environment variable TMPDIR overrides `dir`, while on Windows the TMP is used. The specific behavior of this function
depends on the C library implementation; some aspects are underspecified in system documentation. **Warning:** Use of `tempnam()` is vulnerable to symlink attacks; consider using `tmpfile()` (section 14.1.2) instead. Availability: Macintosh, UNIX, Windows.

`tmpnam()`

Return a unique path name that is reasonable for creating a temporary file. This will be an absolute path that names a potential directory entry in a common location for temporary files. Applications are responsible for properly creating and managing files created using paths returned by `tmpnam()`; no automatic cleanup is provided. **Warning:** Use of `tmpnam()` is vulnerable to symlink attacks; consider using `tmpfile()` (section 14.1.2) instead. Availability: UNIX, Windows. This function probably shouldn't be used on Windows, though: Microsoft's implementation of `tmpnam()` always creates a name in the root directory of the current drive, and that's generally a poor location for a temp file (depending on privileges, you may not even be able to open a file using this name).

**TMP_MAX**

The maximum number of unique names that `tmpnam()` will generate before reusing names.

`unlink(path)`

Remove the file `path`. This is the same function as `remove()`; the `unlink()` name is its traditional UNIX name. Availability: Macintosh, UNIX, Windows.

`utime(path, times)`

Set the access and modified times of the file specified by `path`. If `times` is `None`, then the file's access and modified times are set to the current time. Otherwise, `times` must be a 2-tuple of numbers, of the form `(atime, mtime)` which is used to set the access and modified times, respectively. Whether a directory can be given for `path` depends on whether the operating system implements directories as files (for example, Windows does not). Note that the exact times you set here may not be returned by a subsequent `stat()` call, depending on the resolution with which your operating system records access and modification times; see `stat()`. Changed in
version 2.0: Added support for None for times. Availability: Macintosh, UNIX, Windows.

`walk(top[, topendown=True [, onerror=None]])`

`walk()` generates the file names in a directory tree, by walking the tree either top down or bottom up. For each directory in the tree rooted at directory `top` (including `top` itself), it yields a 3-tuple `(dirpath, dirnames, filenames)`.

`dirpath` is a string, the path to the directory. `dirnames` is a list of the names of the subdirectories in `dirpath` (excluding '. ' and '..'). `filenames` is a list of the names of the non-directory files in `dirpath`. Note that the names in the lists contain no path components. To get a full path (which begins with `top`) to a file or directory in `dirpath`, do `os.path.join(dirpath, name)`.

If optional argument `topdown` is true or not specified, the triple for a directory is generated before the triples for any of its subdirectories (directories are generated top down). If `topdown` is false, the triple for a directory is generated after the triples for all of its subdirectories (directories are generated bottom up).

When `topdown` is true, the caller can modify the `dirnames` list in-place (perhaps using `del` or slice assignment), and `walk()` will only recurse into the subdirectories whose names remain in `dirnames`; this can be used to prune the search, impose a specific order of visiting, or even to inform `walk()` about directories the caller creates or renames before it resumes `walk()` again. Modifying `dirnames` when `topdown` is false is ineffective, because in bottom-up mode the directories in `dirnames` are generated before `dirpath` itself is generated.

By default errors from the `os.listdir()` call are ignored. If optional argument `onerror` is specified, it should be a function; it will be called with one argument, an `OSError` instance. It can report the error to continue with the walk, or raise the exception to abort the walk. Note that the filename is available as the `filename` attribute of the exception object.

**Note:** If you pass a relative pathname, don't change the
current working directory between resumptions of `walk()`.

`walk()` never changes the current directory, and assumes that its caller doesn't either.

**Note:** On systems that support symbolic links, links to subdirectories appear in `dirnames` lists, but `walk()` will not visit them (infinite loops are hard to avoid when following symbolic links). To visit linked directories, you can identify them with `os.path.islink(path)`, and invoke `walk(path)` on each directly.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn't look under any CVS subdirectory:

```python
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print(root, 'consumes',
          print(sum(getsize(join(root, name)) for name in files),
          print('bytes in', len(files), 'non-directory files')
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

In the next example, walking the tree bottom up is essential: `rmdir()` doesn't allow deleting a directory before the directory is empty:

```python
# Delete everything reachable from the directory named in 'top',
# assuming there are no symbolic links.
# CAUTION: This is dangerous! For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

New in version 2.3.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.1.5 Process Management

These functions may be used to create and manage processes.

The various `exec*()` functions take a list of arguments for the new program loaded into the process. In each case, the first of these arguments is passed to the new program as its own name rather than as an argument a user may have typed on a command line. For the C programmer, this is the `argv[0]` passed to a program's `main()`. For example, `"os.execv('/bin/echo', ['foo', 'bar'])"` will only print "bar" on standard output; "foo" will seem to be ignored.

`abort()`
Generate a SIGABRT signal to the current process. On UNIX, the default behavior is to produce a core dump; on Windows, the process immediately returns an exit code of 3. Be aware that programs which use `signal.signal()` to register a handler for SIGABRT will behave differently. Availability: Macintosh, UNIX, Windows.

`exec1(path, arg0, arg1, ...)`
`execle(path, arg0, arg1, ..., env)`
`execlp(file, arg0, arg1, ...)`
`execlpe(file, arg0, arg1, ..., env)`
`execv(path, args)`
`execve(path, args, env)`
`execvp(file, args)`
`execvpe(file, args, env)`

These functions all execute a new program, replacing the current process; they do not return. On UNIX, the new executable is loaded into the current process, and will have the same process ID as the caller. Errors will be reported as `OSError` exceptions.

The "$l" and "$v" variants of the `exec*()` functions differ in how command-line arguments are passed. The "$l" variants are perhaps
easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the `exec1*()` functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the `args` parameter. In either case, the arguments to the child process should start with the name of the command being run, but this is not enforced.

The variants which include a "p" near the end (`exec1p()`, `exec1pe()`, `exec1p()`, and `exec1pe()`) will use the PATH environment variable to locate the program file. When the environment is being replaced (using one of the `exec*e()` variants, discussed in the next paragraph), the new environment is used as the source of the PATH variable. The other variants, `exec1()`, `exec1e()`, `exec1()`, and `exec1e()`, will not use the PATH variable to locate the executable; `path` must contain an appropriate absolute or relative path.

For `exec1e()`, `exec1pe()`, `exec1e()`, and `exec1pe()` (note that these all end in "e"), the `env` parameter must be a mapping which is used to define the environment variables for the new process; the `exec1()`, `exec1p()`, `exec1()`, and `exec1p()` all cause the new process to inherit the environment of the current process. Availability: Macintosh, UNIX, Windows.

`_exit(n)`
Exit to the system with status `n`, without calling cleanup handlers, flushing stdio buffers, etc. Availability: Macintosh, UNIX, Windows.

**Note:** The standard way to exit is `sys.exit(n)`. `_exit()` should normally only be used in the child process after a `fork()`.

The following exit codes are a defined, and can be used with `_exit()`, although they are not required. These are typically used for system programs written in Python, such as a mail server's external command delivery program. **Note:** Some of these may not be available on all UNIX platforms, since there is some variation. These constants are defined where they are defined by the underlying platform.
**EX_OK**
Exit code that means no error occurred. Availability: Macintosh, UNIX. New in version 2.3.

**EX_USAGE**
Exit code that means the command was used incorrectly, such as when the wrong number of arguments are given. Availability: Macintosh, UNIX. New in version 2.3.

**EX_DATAERR**
Exit code that means the input data was incorrect. Availability: Macintosh, UNIX. New in version 2.3.

**EX_NOSINPUT**
Exit code that means an input file did not exist or was not readable. Availability: Macintosh, UNIX. New in version 2.3.

**EX_NOUSER**
Exit code that means a specified user did not exist. Availability: Macintosh, UNIX. New in version 2.3.

**EX_NOHOST**
Exit code that means a specified host did not exist. Availability: Macintosh, UNIX. New in version 2.3.

**EX_UNAVAILABLE**
Exit code that means that a required service is unavailable. Availability: Macintosh, UNIX. New in version 2.3.

**EX_SOFTWARE**
Exit code that means an internal software error was detected. Availability: Macintosh, UNIX. New in version 2.3.

**EX_OSERR**
Exit code that means an operating system error was detected, such as the inability to fork or create a pipe. Availability: Macintosh, UNIX. New in version 2.3.

**EX_OSFILE**
Exit code that means some system file did not exist, could not be opened, or
had some other kind of error. Availability: Macintosh, UNIX. New in version 2.3.

**EX_CANTCREAT**
Exit code that means a user specified output file could not be created. Availability: Macintosh, UNIX. New in version 2.3.

**EX_IOERR**
Exit code that means that an error occurred while doing I/O on some file. Availability: Macintosh, UNIX. New in version 2.3.

**EX_TEMPFAIL**
Exit code that means a temporary failure occurred. This indicates something that may not really be an error, such as a network connection that couldn't be made during a retryable operation. Availability: Macintosh, UNIX. New in version 2.3.

**EX_PROTOCOL**
Exit code that means that a protocol exchange was illegal, invalid, or not understood. Availability: Macintosh, UNIX. New in version 2.3.

**EX_NOPERM**
Exit code that means that there were insufficient permissions to perform the operation (but not intended for file system problems). Availability: Macintosh, UNIX. New in version 2.3.

**EX_CONFIG**
Enter code that means that some kind of configuration error occurred. Availability: Macintosh, UNIX. New in version 2.3.

**EX_NOTFOUND**
Enter code that means something like ``an entry was not found". Availability: Macintosh, UNIX. New in version 2.3.

**fork()**
Fork a child process. Return 0 in the child, the child's process id in the parent. Availability: Macintosh, UNIX.

**forkpty()**
Fork a child process, using a new pseudo-terminal as the child's controlling terminal. Return a pair of \((\text{pid}, \text{fd})\), where \(\text{pid}\) is 0 in the child, the new child's process id in the parent, and \(\text{fd}\) is the file descriptor of the master end of the pseudo-terminal. For a more portable approach, use the \texttt{pty} module. Availability: Macintosh, Some flavors of \texttt{UNIX}.

\texttt{kill(\texttt{pid}, \texttt{sig})}

Send signal \texttt{sig} to the process \texttt{pid}. Constants for the specific signals available on the host platform are defined in the \texttt{signal} module. Availability: Macintosh, \texttt{UNIX}.

\texttt{killpg(\texttt{pgid}, \texttt{sig})}

Send the signal \texttt{sig} to the process group \texttt{pgid}. Availability: Macintosh, \texttt{UNIX}. New in version 2.3.

\texttt{nice(\texttt{increment})}

Add \texttt{increment} to the process's ```niceness```. Return the new niceness. Availability: Macintosh, \texttt{UNIX}.

\texttt{plock(\texttt{op})}

Lock program segments into memory. The value of \texttt{op} (defined in \texttt{<sys/lock.h>}) determines which segments are locked. Availability: Macintosh, \texttt{UNIX}.

\texttt{popen(...)}
\texttt{popen2(...)}
\texttt{popen3(...)}
\texttt{popen4(...)}

Run child processes, returning opened pipes for communications. These functions are described in section 14.1.2.

\texttt{spawnl(\texttt{mode, path, ...})}
\texttt{spawnle(\texttt{mode, path, ...}, \texttt{env})}
\texttt{spawnlp(\texttt{mode, file, ...})}
\texttt{spawnlpe(\texttt{mode, file, ...}, \texttt{env})}
spawnv( mode, path, args)
spawnve( mode, path, args, env)
spawnvp( mode, file, args)
spawnvpe( mode, file, args, env)

Execute the program path in a new process. If mode is P_NOWAIT, this function returns the process ID of the new process; if mode is P_WAIT, returns the process's exit code if it exits normally, or -signal, where signal is the signal that killed the process. On Windows, the process ID will actually be the process handle, so can be used with the waitpid() function.

The "l" and "v" variants of the spawn*() functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the spawnl*() functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the args parameter. In either case, the arguments to the child process must start with the name of the command being run.

The variants which include a second "p" near the end (spawnlp(), spawnlpe(), spawnvp(), and spawnvpe()) will use the PATH environment variable to locate the program file. When the environment is being replaced (using one of the spawn*e() variants, discussed in the next paragraph), the new environment is used as the source of the PATH variable. The other variants, spawnl(), spawnle(), spawnv(), and spawnve(), will not use the PATH variable to locate the executable; path must contain an appropriate absolute or relative path.

For spawnle(), spawnlpe(), spawnve(), and spawnvpe() (note that these all end in "e"), the env parameter must be a mapping which is used to define the environment variables for the new process; the spawnl(), spawnlp(), spawnv(), and spawnvp() all cause the new process to inherit the environment of the current process.

As an example, the following calls to spawnlp() and spawnvpe() are equivalent:
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)

Availability: 
UNIX, Windows. spawnlp(), spawnlpe(), spawnvp() and spawnvpe() are not available on Windows. New in version 1.6.

P_NOWAIT
P_NOWAITO
Possible values for the mode parameter to the spawn*() family of functions. If either of these values is given, the spawn*() functions will return as soon as the new process has been created, with the process ID as the return value. Availability: Macintosh, UNIX, Windows. New in version 1.6.

P_WAIT
Possible value for the mode parameter to the spawn*() family of functions. If this is given as mode, the spawn*() functions will not return until the new process has run to completion and will return the exit code of the process the run is successful, or -signal if a signal kills the process. Availability: Macintosh, UNIX, Windows. New in version 1.6.

P_DETACH
P_OVERLAY
Possible values for the mode parameter to the spawn*() family of functions. These are less portable than those listed above. P_DETACH is similar to P_NOWAIT, but the new process is detached from the console of the calling process. If P_OVERLAY is used, the current process will be replaced; the spawn*() function will not return. Availability: Windows. New in version 1.6.

startfile(path[, operation])
Start a file with its associated application.

When operation is not specified or 'open', this acts like double-clicking the file in Windows Explorer, or giving the file name as an argument to the start command from the interactive command shell: the file is opened with whatever application (if any) its extension is associated.
When another operation is given, it must be a "command verb" that specifies what should be done with the file. Common verbs documented by Microsoft are 'print' and 'edit' (to be used on files) as well as 'explore' and 'find' (to be used on directories).

`startfile()` returns as soon as the associated application is launched. There is no option to wait for the application to close, and no way to retrieve the application's exit status. The `path` parameter is relative to the current directory. If you want to use an absolute path, make sure the first character is not a slash ('/'); the underlying Win32 `ShellExecute()` function doesn't work if it is. Use the `os.path.normpath()` function to ensure that the path is properly encoded for Win32. Availability: Windows. New in version 2.0. New in version 2.5: The `operation` parameter.

`system(command)`

Execute the command (a string) in a subshell. This is implemented by calling the Standard C function `system()`, and has the same limitations. Changes to `posix.environ`, `sys.stdin`, etc. are not reflected in the environment of the executed command.

On UNIX, the return value is the exit status of the process encoded in the format specified for `wait()`. Note that POSIX does not specify the meaning of the return value of the C `system()` function, so the return value of the Python function is system-dependent.

On Windows, the return value is that returned by the system shell after running `command`, given by the Windows environment variable COMSPEC: on `command.com` systems (Windows 95, 98 and ME) this is always 0; on `cmd.exe` systems (Windows NT, 2000 and XP) this is the exit status of the command run; on systems using a non-native shell, consult your shell documentation.

Availability: Macintosh, UNIX, Windows.

times()

Return a 5-tuple of floating point numbers indicating accumulated (processor or other) times, in seconds. The items are: user time, system time, children's user time, children's system time, and elapsed real time
since a fixed point in the past, in that order. See the \texttt{UNIX} manual page \texttt{times(2)} or the corresponding \texttt{Windows} Platform API documentation. Availability: Macintosh, \texttt{UNIX}, Windows.

\texttt{wait()}
Wait for completion of a child process, and return a tuple containing its pid and exit status indication: a 16-bit number, whose low byte is the signal number that killed the process, and whose high byte is the exit status (if the signal number is zero); the high bit of the low byte is set if a core file was produced. Availability: Macintosh, \texttt{UNIX}.

\texttt{waitpid(pid, options)}
The details of this function differ on \texttt{UNIX} and \texttt{Windows}.

On \texttt{UNIX}: Wait for completion of a child process given by process id \texttt{pid}, and return a tuple containing its process id and exit status indication (encoded as for \texttt{wait()}). The semantics of the call are affected by the value of the integer \texttt{options}, which should be 0 for normal operation.

If \texttt{pid} is greater than 0, \texttt{waitpid()} requests status information for that specific process. If \texttt{pid} is 0, the request is for the status of any child in the process group of the current process. If \texttt{pid} is -1, the request pertains to any child of the current process. If \texttt{pid} is less than -1, status is requested for any process in the process group \texttt{-pid} (the absolute value of \texttt{pid}).

On \texttt{Windows}: Wait for completion of a process given by process handle \texttt{pid}, and return a tuple containing \texttt{pid}, and its exit status shifted left by 8 bits (shifting makes cross-platform use of the function easier). A \texttt{pid} less than or equal to 0 has no special meaning on \texttt{Windows}, and raises an exception. The value of integer \texttt{options} has no effect. \texttt{pid} can refer to any process whose id is known, not necessarily a child process. The \texttt{spawn()} functions called with \texttt{P\_NOWAIT} return suitable process handles.

\texttt{wait3([options])}
Similar to \texttt{waitpid()}, except no process id argument is given and a 3-element tuple containing the child's process id, exit status indication, and resource usage information is returned. Refer to \texttt{resource.getrusage()} for details on resource usage information. The
option argument is the same as that provided to `waitpid()` and `wait4()`. Availability: UNIX. New in version 2.5.

`wait4()`

Similar to `waitpid()`, except a 3-element tuple, containing the child's process id, exit status indication, and resource usage information is returned. Refer to `resource.getrusage()` for details on resource usage information. The arguments to `wait4()` are the same as those provided to `waitpid()`. Availability: UNIX. New in version 2.5.

**WNOHANG**

The option for `waitpid()` to return immediately if no child process status is available immediately. The function returns `(0, 0)` in this case. Availability: Macintosh, UNIX.

**WCONTINUED**

This option causes child processes to be reported if they have been continued from a job control stop since their status was last reported. Availability: Some UNIX systems. New in version 2.3.

**WUNTRACED**

This option causes child processes to be reported if they have been stopped but their current state has not been reported since they were stopped. Availability: Macintosh, UNIX. New in version 2.3.

The following functions take a process status code as returned by `system()`, `wait()`, or `waitpid()` as a parameter. They may be used to determine the disposition of a process.

**WCOREDUMP**(status)

Returns `True` if a core dump was generated for the process, otherwise it returns `False`. Availability: Macintosh, UNIX. New in version 2.3.

**WIFCONTINUED**(status)

Returns `True` if the process has been continued from a job control stop, otherwise it returns `False`. Availability: UNIX. New in version 2.3.

**WIFSTOPPED**(status)
Returns True if the process has been stopped, otherwise it returns False. Availability: UNIX.

WIFSIGNALED( status)
Returns True if the process exited due to a signal, otherwise it returns False. Availability: Macintosh, UNIX.

WIFEXITED( status)
Returns True if the process exited using the exit(2) system call, otherwise it returns False. Availability: Macintosh, UNIX.

WEXITSTATUS( status)
If WIFEXITED( status) is true, return the integer parameter to the exit(2) system call. Otherwise, the return value is meaningless. Availability: Macintosh, UNIX.

WSTOPSIG( status)
Return the signal which caused the process to stop. Availability: Macintosh, UNIX.

WTERMSIG( status)
Return the signal which caused the process to exit. Availability: Macintosh, UNIX.
Miscellaneous Functions
14.1.6 Miscellaneous System Information

confstr(name)
Return string-valued system configuration values. name specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX, UNIX 95, UNIX 98, and others). Some platforms define additional names as well. The names known to the host operating system are given as the keys of the confstr_names dictionary. For configuration variables not included in that mapping, passing an integer for name is also accepted. Availability: Macintosh, UNIX.

If the configuration value specified by name isn't defined, None is returned.

If name is a string and is not known, ValueError is raised. If a specific value for name is not supported by the host system, even if it is included in confstr_names, an OSError is raised with errno.EINVAL for the error number.

confstr_names
Dictionary mapping names accepted by confstr() to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system. Availability: Macintosh, UNIX.

getloadavg()
Return the number of processes in the system run queue averaged over the last 1, 5, and 15 minutes or raises OSError if the load average was unobtainable.

New in version 2.3.

sysconf(name)
Return integer-valued system configuration values. If the configuration value specified by name isn't defined, -1 is returned. The comments regarding the name parameter for confstr() apply here as well; the dictionary that provides information on the known names is given by
sysconf_names. Availability: Macintosh, UNIX.

**sysconf_names**

Dictionary mapping names accepted by `sysconf()` to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system. Availability: Macintosh, UNIX.

The follow data values are used to support path manipulation operations. These are defined for all platforms.

Higher-level operations on pathnames are defined in the `os.path` module.

**curdir**

The constant string used by the operating system to refer to the current directory. For example: `.' for POSIX or ':.' for Mac OS 9. Also available via `os.path`.

**pardir**

The constant string used by the operating system to refer to the parent directory. For example: `..` for POSIX or `::` for Mac OS 9. Also available via `os.path`.

**sep**

The character used by the operating system to separate pathname components, for example, `/` for POSIX or `:` for Mac OS 9. Note that knowing this is not sufficient to be able to parse or concatenate pathnames - - use `os.path.split()` and `os.path.join()` -- but it is occasionally useful. Also available via `os.path`.

**altsep**

An alternative character used by the operating system to separate pathname components, or `None` if only one separator character exists. This is set to `/` on Windows systems where `sep` is a backslash. Also available via `os.path`.

**extsep**

The character which separates the base filename from the extension; for example, the `"." in os.py. Also available via `os.path`. New in version
2.2.

**pathsep**
The character conventionally used by the operating system to separate search path components (as in PATH), such as ":" for POSIX or ";" for Windows. Also available via os.path.

**defpath**
The default search path used by exec*p*() and spawn*p*() if the environment doesn't have a 'PATH' key. Also available via os.path.

**linesep**
The string used to separate (or, rather, terminate) lines on the current platform. This may be a single character, such as '\n' for POSIX or '\r' for Mac OS, or multiple characters, for example, '\r\n' for Windows.

**devnull**
The file path of the null device. For example: '/dev/null' for POSIX or 'Dev:Nul' for Mac OS 9. Also available via os.path. New in version 2.4.
14.1.7 Miscellaneous Functions

urandom(\(n\))

Return a string of \(n\) random bytes suitable for cryptographic use.

This function returns random bytes from an OS-specific randomness source. The returned data should be unpredictable enough for cryptographic applications, though its exact quality depends on the OS implementation. On a UNIX-like system this will query /dev/urandom, and on Windows it will use CryptGenRandom. If a randomness source is not found, NotImplementedError will be raised. New in version 2.4.
14.2 time -- Time access and conversions

This module provides various time-related functions. It is always available, but not all functions are available on all platforms. Most of the functions defined in this module call platform C library functions with the same name. It may sometimes be helpful to consult the platform documentation, because the semantics of these functions varies among platforms.

An explanation of some terminology and conventions is in order.

- The *epoch* is the point where the time starts. On January 1st of that year, at 0 hours, the "time since the epoch" is zero. For UNIX, the epoch is 1970. To find out what the epoch is, look at `gmtime(0)`.

- The functions in this module do not handle dates and times before the epoch or far in the future. The cut-off point in the future is determined by the C library; for UNIX, it is typically in 2038.

- **Year 2000 (Y2K) issues**: Python depends on the platform's C library, which generally doesn't have year 2000 issues, since all dates and times are represented internally as seconds since the epoch. Functions accepting a `struct_time` (see below) generally require a 4-digit year. For backward compatibility, 2-digit years are supported if the module variable `accept2dyear` is a non-zero integer; this variable is initialized to 1 unless the environment variable `PYTHONY2K` is set to a non-empty string, in which case it is initialized to 0. Thus, you can set `PYTHONY2K` to a non-empty string in the environment to require 4-digit years for all year input. When 2-digit years are accepted, they are converted according to the POSIX or X/Open standard: values 69-99 are mapped to 1969-1999, and values 0-68 are mapped to 2000-2068. Values 100-1899 are always illegal. Note that this is new as of Python 1.5.2(a2); earlier versions, up to Python 1.5.1 and 1.5.2a1, would add 1900 to year values below 1900.

- UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT). The acronym UTC is not a mistake but a compromise.
between English and French.

- DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year. DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.

- The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most UNIX systems, the clock ``ticks'' only 50 or 100 times a second, and on the Mac, times are only accurate to whole seconds.

- On the other hand, the precision of `time()` and `sleep()` is better than their UNIX equivalents: times are expressed as floating point numbers, `time()` returns the most accurate time available (using UNIX `gettimeofday()` where available), and `sleep()` will accept a time with a nonzero fraction (UNIX `select()` is used to implement this, where available).

- The time value as returned by `gmtime()`, `localtime()`, and `strptime()`, and accepted by `asctime()`, `mktime()` and `strftime()`, is a sequence of 9 integers. The return values of `gmtime()`, `localtime()`, and `strptime()` also offer attribute names for individual fields.

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>tm_year</code></td>
<td>(for example, 1993)</td>
</tr>
<tr>
<td>1</td>
<td><code>tm_mon</code></td>
<td>range [1,12]</td>
</tr>
<tr>
<td>2</td>
<td><code>tm_mday</code></td>
<td>range [1,31]</td>
</tr>
<tr>
<td>3</td>
<td><code>tm_hour</code></td>
<td>range [0,23]</td>
</tr>
<tr>
<td>4</td>
<td><code>tm_min</code></td>
<td>range [0,59]</td>
</tr>
<tr>
<td>5</td>
<td><code>tm_sec</code></td>
<td>range [0,61]; see (1) in <code>strftime()</code> description</td>
</tr>
<tr>
<td>6</td>
<td><code>tm_wday</code></td>
<td>range [0,6], Monday is 0</td>
</tr>
<tr>
<td>7</td>
<td><code>tm_yday</code></td>
<td>range [1,366]</td>
</tr>
<tr>
<td>8</td>
<td><code>tm_isdst</code></td>
<td>0, 1 or -1; see below</td>
</tr>
</tbody>
</table>
Note that unlike the C structure, the month value is a range of 1-12, not 0-11. A year value will be handled as described under ```Year 2000 (Y2K) issues''` above. A -1 argument as the daylight savings flag, passed to \texttt{mktime()} will usually result in the correct daylight savings state to be filled in.

When a tuple with an incorrect length is passed to a function expecting a \texttt{struct\_time}, or having elements of the wrong type, a \texttt{TypeError} is raised.

Changed in version 2.2: The time value sequence was changed from a tuple to a \texttt{struct\_time}, with the addition of attribute names for the fields.

The module defines the following functions and data items:

\textbf{accept2dyear}

Boolean value indicating whether two-digit year values will be accepted. This is true by default, but will be set to false if the environment variable PYTHONY2K has been set to a non-empty string. It may also be modified at run time.

\textbf{altzone}

The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if \texttt{daylight} is nonzero.

\textbf{asctime([t])}

Convert a tuple or \texttt{struct\_time} representing a time as returned by \texttt{gmtime()} or \texttt{localtime()} to a 24-character string of the following form: 'Sun Jun 20 23:21:05 1993'. If \texttt{t} is not provided, the current time as returned by \texttt{localtime()} is used. Locale information is not used by \texttt{asctime()}. \textbf{Note:} Unlike the C function of the same name, there is no trailing newline. Changed in version 2.1: Allowed \texttt{t} to be omitted.

\textbf{clock()}

On UNIX, return the current processor time as a floating point number expressed in seconds. The precision, and in fact the very definition of the
meaning of ``processor time'', depends on that of the C function of the same name, but in any case, this is the function to use for benchmarking Python or timing algorithms.

On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number, based on the Win32 function `QueryPerformanceCounter()`. The resolution is typically better than one microsecond.

```python
cime([secs])
```
Convert a time expressed in seconds since the epoch to a string representing local time. If `secs` is not provided or `None`, the current time as returned by `time()` is used. `cime(secs)` is equivalent to `asctime(localtime(secs))`. Locale information is not used by `cime()`. Changed in version 2.1: Allowed `secs` to be omitted. Changed in version 2.4: If `secs` is `None`, the current time is used.

```python
daylight
```
Nonzero if a DST timezone is defined.

```python
gmtime([secs])
```
Convert a time expressed in seconds since the epoch to a `struct_time` in UTC in which the dst flag is always zero. If `secs` is not provided or `None`, the current time as returned by `time()` is used. Fractions of a second are ignored. See above for a description of the `struct_time` object. See `calendar.timegm()` for the inverse of this function. Changed in version 2.1: Allowed `secs` to be omitted. Changed in version 2.4: If `secs` is `None`, the current time is used.

```python
localtime([secs])
```
Like `gmtime()` but converts to local time. If `secs` is not provided or `None`, the current time as returned by `time()` is used. The dst flag is set to 1 when DST applies to the given time. Changed in version 2.1: Allowed `secs` to be omitted. Changed in version 2.4: If `secs` is `None`, the current time is used.

```python
mktime(t)
```
This is the inverse function of `localtime()`. Its argument is the `struct_time` or full 9-tuple (since the dst flag is needed; use `-1` as the dst flag if it is unknown) which expresses the time in `local` time, not UTC. It returns a floating point number, for compatibility with `time()`. If the input value cannot be represented as a valid time, either `OverflowError` or `ValueError` will be raised (which depends on whether the invalid value is caught by Python or the underlying C libraries). The earliest date for which it can generate a time is platform-dependent.

**sleep(secs)**
Suspend execution for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time. The actual suspension time may be less than that requested because any caught signal will terminate the `sleep()` following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount because of the scheduling of other activity in the system.

**strftime(format[, t])**
Convert a tuple or `struct_time` representing a time as returned by `gmtime()` or `localtime()` to a string as specified by the `format` argument. If `t` is not provided, the current time as returned by `localtime()` is used. `format` must be a string. `ValueError` is raised if any field in `t` is outside of the allowed range. Changed in version 2.1: Allowed `t` to be omitted. Changed in version 2.4: `ValueError` raised if a field in `t` is out of range. Changed in version 2.5: 0 is now a legal argument for any position in the time tuple; if it is normally illegal the value is forced to a correct one..

The following directives can be embedded in the `format` string. They are shown without the optional field width and precision specification, and are replaced by the indicated characters in the `strftime()` result:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Locale's abbreviated weekday name.</td>
<td></td>
</tr>
<tr>
<td>%A</td>
<td>Locale's full weekday name.</td>
<td></td>
</tr>
<tr>
<td>%b</td>
<td>Locale's abbreviated month name.</td>
<td></td>
</tr>
<tr>
<td>%B</td>
<td>Locale's full month name.</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>%c</td>
<td>Locale's appropriate date and time representation.</td>
<td></td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month as a decimal number [01,31].</td>
<td></td>
</tr>
<tr>
<td>%H</td>
<td>Hour (24-hour clock) as a decimal number [00,23].</td>
<td></td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) as a decimal number [01,12].</td>
<td></td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year as a decimal number [001,366].</td>
<td></td>
</tr>
<tr>
<td>%m</td>
<td>Month as a decimal number [01,12].</td>
<td></td>
</tr>
<tr>
<td>%M</td>
<td>Minute as a decimal number [00,59].</td>
<td></td>
</tr>
<tr>
<td>%p</td>
<td>Locale's equivalent of either AM or PM. (1)</td>
<td></td>
</tr>
<tr>
<td>%S</td>
<td>Second as a decimal number [00,61]. (2)</td>
<td></td>
</tr>
<tr>
<td>%u</td>
<td>Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0. (3)</td>
<td></td>
</tr>
<tr>
<td>%w</td>
<td>Weekday as a decimal number [0(Sunday),6].</td>
<td></td>
</tr>
<tr>
<td>%W</td>
<td>Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0. (3)</td>
<td></td>
</tr>
<tr>
<td>%x</td>
<td>Locale's appropriate date representation.</td>
<td></td>
</tr>
<tr>
<td>%X</td>
<td>Locale's appropriate time representation.</td>
<td></td>
</tr>
<tr>
<td>%y</td>
<td>Year without century as a decimal number [00,99].</td>
<td></td>
</tr>
<tr>
<td>%Y</td>
<td>Year with century as a decimal number.</td>
<td></td>
</tr>
<tr>
<td>%z</td>
<td>Time zone name (no characters if no time zone exists).</td>
<td></td>
</tr>
<tr>
<td>%%</td>
<td>A literal &quot;%&quot; character.</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1) When used with the `strptime()` function, the %p directive only affects the output hour field if the %I directive is used to parse the hour.

(2) The range really is 0 to 61; this accounts for leap seconds and the (very rare) double leap seconds.

(3)
When used with the `strptime()` function, `%U` and `%W` are only used in calculations when the day of the week and the year are specified.

Here is an example, a format for dates compatible with that specified in the [RFC 2822](https://tools.ietf.org/html/rfc2822) Internet email standard.

```python
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S %Z", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

Additional directives may be supported on certain platforms, but only the ones listed here have a meaning standardized by ANSI C.

On some platforms, an optional field width and precision specification can immediately follow the initial "%" of a directive in the following order; this is also not portable. The field width is normally 2 except for `%j` where it is 3.

`strptime(string[, format])`

Parse a string representing a time according to a format. The return value is a `struct_time` as returned by `gmtime()` or `localtime()`. The `format` parameter uses the same directives as those used by `strptime()`; it defaults to "%a %b %d %H:%M:%S %Y" which matches the formatting returned by `ctime()`. If `string` cannot be parsed according to `format`, `ValueError` is raised. If the string to be parsed has excess data after parsing, `ValueError` is raised. The default values used to fill in any missing data when more accurate values cannot be inferred are (1900, 1, 1, 0, 0, 0, 0, 1, -1).

Support for the `%Z` directive is based on the values contained in `tzname` and whether `daylight` is true. Because of this, it is platform-specific except for recognizing UTC and GMT which are always known (and are considered to be non-daylight savings timezones).

`struct_time`

The type of the time value sequence returned by `gmtime()`, `localtime()`, and `strptime()`. New in version 2.2.

`time()`
Return the time as a floating point number expressed in seconds since the epoch, in UTC. Note that even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second. While this function normally returns non-decreasing values, it can return a lower value than a previous call if the system clock has been set back between the two calls.

**timezone**

The offset of the local (non-DST) timezone, in seconds west of UTC (negative in most of Western Europe, positive in the US, zero in the UK).

**tzname**

A tuple of two strings: the first is the name of the local non-DST timezone, the second is the name of the local DST timezone. If no DST timezone is defined, the second string should not be used.

**tzset()**

Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done. New in version 2.3.

Availability: UNIX.

**Note:** Although in many cases, changing the TZ environment variable may affect the output of functions like `localtime` without calling `tzset`, this behavior should not be relied on.

The TZ environment variable should contain no whitespace.

The standard format of the TZ environment variable is: (whitespace added for clarity)

```plaintext
std offset [dst [offset] [.start[/time], end[/time]
 ]]```

Where:
std and dst  
Three or more alphanumerics giving the timezone abbreviations.  
These will be propagated into time.tzname

offset
The offset has the form: ± hh[:mm[:ss]]. This indicates the value added 
the local time to arrive at UTC. If preceded by a '-', the timezone is 
est east of the Prime Meridian; otherwise, it is west. If no offset follows 
dst, summer time is assumed to be one hour ahead of standard time.

start[/time],end[/time]
Indicates when to change to and back from DST. The format of the 
start and end dates are one of the following:

Jn
The Julian day n (1 <= n <= 365). Leap days are not counted, so 
in all years February 28 is day 59 and March 1 is day 60.

n
The zero-based Julian day (0 <= n <= 365). Leap days are 
counted, and it is possible to refer to February 29.

Mm.n.d
The d'th day (0 <= d <= 6) or week n of month m of the year (1 
<= n <= 5, 1 <= m <= 12, where week 5 means "the last d day in 
month m" which may occur in either the fourth or the fifth week). 
Week 1 is the first week in which the d'th day occurs. Day zero is 
Sunday.

time has the same format as offset except that no leading sign ('-' or '+') 
is allowed. The default, if time is not given, is 02:00:00.

>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'

On many UNIX systems (including *BSD, Linux, Solaris, and Darwin), it is
more convenient to use the system's zoneinfo *(tzfile(5))* database to specify the timezone rules. To do this, set the TZ environment variable to the path of the required timezone datafile, relative to the root of the systems 'zoneinfo' timezone database, usually located at */usr/share/zoneinfo*. For example, 'US/Eastern', 'Australia/Melbourne', 'Egypt' or 'Europe/Amsterdam'.

```python
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

See Also:

Module `datetime`:

More object-oriented interface to dates and times.

Module `locale`:

Internationalization services. The locale settings can affect the return values for some of the functions in the `time` module.

Module `calendar`:

General calendar-related functions. `timegm()` is the inverse of `gmtime()` from this module.

Footnotes

... standard.14.1

The use of `%Z` is now deprecated, but the `%Z` escape that expands to the preferred hour/minute offset is not supported by all ANSI C libraries. Also, a strict reading of the original 1982 RFC 822 standard calls for a two-digit year (%y rather than %Y), but practice moved to 4-digit years long before the year 2000. The 4-digit year has been mandated by RFC 2822, which
obsoletes [RFC 822].

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...] for information on suggesting changes.
14.3 optparse -- More powerful command line option parser

New in version 2.3.

optparse is a more convenient, flexible, and powerful library for parsing command-line options than getopt. optparse uses a more declarative style of command-line parsing: you create an instance of OptionParser, populate it with options, and parse the command line. optparse allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Here's an example of using optparse in a simple script:

```python
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                 help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                 action="store_false", dest="verbose", default=True,
                 help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

With these few lines of code, users of your script can now do the ``usual thing'' on the command-line, for example:

```bash
<yourscript> --file=outfile -q
```

As it parses the command line, optparse sets attributes of the options object returned by parse_args() based on user-supplied command-line values. When parse_args() returns from parsing this command line, options.filename will be "outfile" and options.VERBOSE will be False. optparse supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example:
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile

Additionally, users can run one of

<yourscript> -h
<yourscript> --help

and optparse will print out a brief summary of your script's options:

usage: <yourscript> [options]

options:
    -h, --help                      show this help message and exit
    -f FILE, --file=FILE           write report to FILE
    -q, --quiet                    don't print status messages to stdout

where the value of yourscript is determined at runtime (normally from sys.argv[0]).

Subsections

• 14.3.1 Background
  • 14.3.1.1 Terminology
  • 14.3.1.2 What are options for?
  • 14.3.1.3 What are positional arguments for?
• 14.3.2 Tutorial
  • 14.3.2.1 Understanding option actions
  • 14.3.2.2 The store action
  • 14.3.2.3 Handling boolean (flag) options
  • 14.3.2.4 Other actions
  • 14.3.2.5 Default values
  • 14.3.2.6 Generating help
  • 14.3.2.7 Printing a version string
  • 14.3.2.8 How optparse handles errors
  • 14.3.2.9 Putting it all together
• 14.3.3 Reference Guide
  • 14.3.3.1 Creating the parser
  • 14.3.3.2 Populating the parser
  • 14.3.3.3 Defining options
14.3.4 Option Callbacks
  14.3.4.1 Defining a callback option
  14.3.4.2 How callbacks are called
  14.3.4.3 Raising errors in a callback
  14.3.4.4 Callback example 1: trivial callback
  14.3.4.5 Callback example 2: check option order
  14.3.4.6 Callback example 3: check option order (generalized)
  14.3.4.7 Callback example 4: check arbitrary condition
  14.3.4.8 Callback example 5: fixed arguments
  14.3.4.9 Callback example 6: variable arguments

14.3.5 Extending optparse
  14.3.5.1 Adding new types
  14.3.5.2 Adding new actions

Previous: 14.2 time Up: 14. Generic Operating System Next: 14.3.1 Background

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.3.1 Background

`optparse` was explicitly designed to encourage the creation of programs with straightforward, conventional command-line interfaces. To that end, it supports only the most common command-line syntax and semantics conventionally used under UNIX. If you are unfamiliar with these conventions, read this section to acquaint yourself with them.

---

Subsections

- [14.3.1.1 Terminology](#)
- [14.3.1.2 What are options for?](#)
- [14.3.1.3 What are positional arguments for?](#)

---

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
14.3.2 Tutorial

While optparse is quite flexible and powerful, it's also straightforward to use in most cases. This section covers the code patterns that are common to any optparse-based program.

First, you need to import the OptionParser class; then, early in the main program, create an OptionParser instance:

```python
from optparse import OptionParser
[...]
parser = OptionParser()
```

Then you can start defining options. The basic syntax is:

```python
parser.add_option(opt_str, ..., attr=value, ...)
```

Each option has one or more option strings, such as "-f" or "--file", and several option attributes that tell optparse what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g.:

```python
parser.add_option("-f", "--file", ...)
```

You're free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to add_option() are effectively labels for the option defined by that call. For brevity, we will frequently refer to encountering an option on the command line; in reality, optparse encounters option strings and looks up options from them.

Once all of your options are defined, instruct optparse to parse your program's command line:

```python
(options, args) = parser.parse_args()
```
(If you like, you can pass a custom argument list to `parse_args()`, but that's rarely necessary: by default it uses `sys.argv[1:]`.)

`parse_args()` returns two values:

- **options**, an object containing values for all of your options--e.g. if "-file" takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option

- **args**, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes: `action`, `type`, `dest` (destination), and `help`. Of these, `action` is the most fundamental.

Subsections

- [14.3.2.1 Understanding option actions](#)
- [14.3.2.2 The store action](#)
- [14.3.2.3 Handling boolean (flag) options](#)
- [14.3.2.4 Other actions](#)
- [14.3.2.5 Default values](#)
- [14.3.2.6 Generating help](#)
- [14.3.2.7 Printing a version string](#)
- [14.3.2.8 How `optparse` handles errors](#)
- [14.3.2.9 Putting it all together](#)
14.3.3 Reference Guide

Subsections

- 14.3.3.1 Creating the parser
- 14.3.3.2 Populating the parser
- 14.3.3.3 Defining options
- 14.3.3.4 Standard option actions
- 14.3.3.5 Option attributes
- 14.3.3.6 Standard option types
- 14.3.3.7 Parsing arguments
- 14.3.3.8 Querying and manipulating your option parser
- 14.3.3.9 Conflicts between options
- 14.3.3.10 Cleanup
- 14.3.3.11 Other methods

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.3.4 Option Callbacks

When `optparse`'s built-in actions and types aren't quite enough for your needs, you have two choices: extend `optparse` or define a callback option. Extending `optparse` is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option:

- define the option itself using the `callback` action
- write the callback; this is a function (or method) that takes at least four arguments, as described below

Subsections

- [14.3.4.1 Defining a callback option](#)
- [14.3.4.2 How callbacks are called](#)
- [14.3.4.3 Raising errors in a callback](#)
- [14.3.4.4 Callback example 1: trivial callback](#)
- [14.3.4.5 Callback example 2: check option order](#)
- [14.3.4.6 Callback example 3: check option order (generalized)](#)
- [14.3.4.7 Callback example 4: check arbitrary condition](#)
- [14.3.4.8 Callback example 5: fixed arguments](#)
- [14.3.4.9 Callback example 6: variable arguments](#)
14.3.5 Extending optparse

Since the two major controlling factors in how optparse interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

Subsections

- 14.3.5.1 Adding new types
- 14.3.5.2 Adding new actions

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.4 getopt -- Parser for command line options

This module helps scripts to parse the command line arguments in `sys.argv`. It supports the same conventions as the UNIX `getopt()` function (including the special meanings of arguments of the form `-' and `--`). Long options similar to those supported by GNU software may be used as well via an optional third argument. This module provides a single function and an exception:

```
getopt( args, options[, long_options])
```

Parses command line options and parameter list. `args` is the argument list to be parsed, without the leading reference to the running program. Typically, this means "`sys.argv[1:]`". `options` is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (""; i.e., the same format that UNIX `getopt()` uses).

**Note:** Unlike GNU `getopt()`, after a non-option argument, all further arguments are considered also non-options. This is similar to the way non-GNU UNIX systems work.

`long_options`, if specified, must be a list of strings with the names of the long options which should be supported. The leading `--` characters should not be included in the option name. Long options which require an argument should be followed by an equal sign ("="). To accept only long options, `options` should be an empty string. Long options on the command line can be recognized so long as they provide a prefix of the option name that matches exactly one of the accepted options. For example, if `long_options` is `['foo', 'frob']`, the option `--fo` will match as `--foo`, but `--f` will not match uniquely, so `GetoptError` will be raised.

The return value consists of two elements: the first is a list of `(option, value)` pairs; the second is the list of program arguments left after the option list was stripped (this is a trailing slice of `args`). Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., `'-x'`) or two hyphens for long options (e.g.,
'--long-option'), and the option argument as its second element, or an empty string if the option has no argument. The options occur in the list in the same order in which they were found, thus allowing multiple occurrences. Long and short options may be mixed.

`gnu_getopt(args, options[, long_options])`

This function works like `getopt()`, except that GNU style scanning mode is used by default. This means that option and non-option arguments may be intermixed. The `getopt()` function stops processing options as soon as a non-option argument is encountered.

If the first character of the option string is `+`, or if the environment variable POSIXLY_CORRECT is set, then option processing stops as soon as a non-option argument is encountered.

New in version 2.3.

**exception GetoptError**

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none. The argument to the exception is a string indicating the cause of the error. For long options, an argument given to an option which does not require one will also cause this exception to be raised. The attributes `msg` and `opt` give the error message and related option; if there is no specific option to which the exception relates, `opt` is an empty string.

Changed in version 1.6: Introduced `GetoptError` as a synonym for `error`.

**exception error**

Alias for `GetoptError`; for backward compatibility.

An example using only UNIX style options:

```python
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
```
Using long option names is equally easy:

```python
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1 a2']
>>> optlist, args = getopt.getopt(args, 'x', ['condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def')]
>>> args
['a1', 'a2']
```

In a script, typical usage is something like this:

```python
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "o" "v"])
    except getopt.GetoptError:
        # print help information and exit:
        usage()
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        if o in ("-h", "--help"):
            usage()
            sys.exit()
        if o in ("-o", "--output"):
            output = a
        
    if __name__ == "__main__":
        main()
```

See Also:

Module **optparse**:

More object-oriented command line option parsing.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.5 logging -- Logging facility for Python

New in version 2.3. This module defines functions and classes which implement a flexible error logging system for applications.

Logging is performed by calling methods on instances of the `Logger` class (hereafter called `loggers`). Each instance has a name, and they are conceptually arranged in a name space hierarchy using dots (periods) as separators. For example, a logger named "scan" is the parent of loggers "scan.text", "scan.html" and "scan.pdf". Logger names can be anything you want, and indicate the area of an application in which a logged message originates.

Logged messages also have levels of importance associated with them. The default levels provided are DEBUG, INFO, WARNING, ERROR and CRITICAL. As a convenience, you indicate the importance of a logged message by calling an appropriate method of `Logger`. The methods are `debug()`, `info()`, `warning()`, `error()` and `critical()`, which mirror the default levels. You are not constrained to use these levels: you can specify your own and use a more general `Logger` method, `log()`, which takes an explicit level argument.

The numeric values of logging levels are given in the following table. These are primarily of interest if you want to define your own levels, and need them to have specific values relative to the predefined levels. If you define a level with the same numeric value, it overwrites the predefined value; the predefined name is lost.

<table>
<thead>
<tr>
<th>Level</th>
<th>Numeric value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRITICAL</td>
<td>50</td>
</tr>
<tr>
<td>ERROR</td>
<td>40</td>
</tr>
<tr>
<td>WARNING</td>
<td>30</td>
</tr>
<tr>
<td>INFO</td>
<td>20</td>
</tr>
<tr>
<td>DEBUG</td>
<td>10</td>
</tr>
<tr>
<td>NOTSET</td>
<td>0</td>
</tr>
</tbody>
</table>
Levels can also be associated with loggers, being set either by the developer or through loading a saved logging configuration. When a logging method is called on a logger, the logger compares its own level with the level associated with the method call. If the logger's level is higher than the method call's, no logging message is actually generated. This is the basic mechanism controlling the verbosity of logging output.

Logging messages are encoded as instances of the `LogRecord` class. When a logger decides to actually log an event, a `LogRecord` instance is created from the logging message.

Logging messages are subjected to a dispatch mechanism through the use of `handlers`, which are instances of subclasses of the `Handler` class. Handlers are responsible for ensuring that a logged message (in the form of a `LogRecord`) ends up in a particular location (or set of locations) which is useful for the target audience for that message (such as end users, support desk staff, system administrators, developers). Handlers are passed `LogRecord` instances intended for particular destinations. Each logger can have zero, one or more handlers associated with it (via the `addHandler()` method of `Logger`). In addition to any handlers directly associated with a logger, all handlers associated with all ancestors of the logger are called to dispatch the message.

Just as for loggers, handlers can have levels associated with them. A handler's level acts as a filter in the same way as a logger's level does. If a handler decides to actually dispatch an event, the `emit()` method is used to send the message to its destination. Most user-defined subclasses of `Handler` will need to override this `emit()`.

In addition to the base `Handler` class, many useful subclasses are provided:

1. `StreamHandler` instances send error messages to streams (file-like objects).
2. `FileHandler` instances send error messages to disk files.
3. `BaseRotatingHandler` is the base class for handlers that rotate log files at a certain point. It is not meant to be instantiated directly. Instead, use `RotatingFileHandler` or `TimedRotatingFileHandler`. 
4. `RotatingFileHandler` instances send error messages to disk files, with support for maximum log file sizes and log file rotation.

5. `TimedRotatingFileHandler` instances send error messages to disk files rotating the log file at certain timed intervals.

6. `SocketHandler` instances send error messages to TCP/IP sockets.

7. `DatagramHandler` instances send error messages to UDP sockets.

8. `SMTPHandler` instances send error messages to a designated email address.

9. `SysLogHandler` instances send error messages to a UNIX syslog daemon, possibly on a remote machine.

10. `NTEventLogHandler` instances send error messages to a Windows NT/2000/XP event log.

11. `MemoryHandler` instances send error messages to a buffer in memory, which is flushed whenever specific criteria are met.

12. `HTTPHandler` instances send error messages to an HTTP server using either "GET" or "POST" semantics.

The `StreamHandler` and `FileHandler` classes are defined in the core logging package. The other handlers are defined in a sub-module, `logging.handlers`. (There is also another sub-module, `logging.config`, for configuration functionality.)

Logged messages are formatted for presentation through instances of the `Formatter` class. They are initialized with a format string suitable for use with the `%` operator and a dictionary.

For formatting multiple messages in a batch, instances of `BufferingFormatter` can be used. In addition to the format string (which is applied to each message in the batch), there is provision for header and trailer format strings.

When filtering based on logger level and/or handler level is not enough,
instances of Filter can be added to both Logger and Handler instances (through their addFilter() method). Before deciding to process a message further, both loggers and handlers consult all their filters for permission. If any filter returns a false value, the message is not processed further.

The basic Filter functionality allows filtering by specific logger name. If this feature is used, messages sent to the named logger and its children are allowed through the filter, and all others dropped.

In addition to the classes described above, there are a number of module-level functions.

**getLogger([name])**

Return a logger with the specified name or, if no name is specified, return a logger which is the root logger of the hierarchy. If specified, the name is typically a dot-separated hierarchical name like "a", "a.b" or "a.b.c.d". Choice of these names is entirely up to the developer who is using logging.

All calls to this function with a given name return the same logger instance. This means that logger instances never need to be passed between different parts of an application.

**getLoggerClass()**

Return either the standard Logger class, or the last class passed to setLoggerClass(). This function may be called from within a new class definition, to ensure that installing a customised Logger class will not undo customisations already applied by other code. For example:

```python
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

**debug(msg[, *args[, **kwars]])**

Logs a message with level DEBUG on the root logger. The msg is the message format string, and the args are the arguments which are merged into msg using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

There are two keyword arguments in kwargs which are inspected: exc_info
which, if it does not evaluate as false, causes exception information to be added to the logging message. If an exception tuple (in the format returned by `sys.exc_info()`) is provided, it is used; otherwise, `sys.exc_info()` is called to get the exception information.

The other optional keyword argument is `extra` which can be used to pass a dictionary which is used to populate the `__dict__` of the LogRecord created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```python
FORMAT = "%(asctime)-15s (%(clientip)s %(user)-8s) %(message)s"
logging.basicConfig(format=FORMAT)
dict = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logging.warning("Protocol problem: %s", "connection reset", extra=dict)
```

would print something like

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs Protocol problem: connection reset
```

The keys in the dictionary passed in `extra` should not clash with the keys used by the logging system. (See the `Formatter` documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the `Formatter` has been set up with a format string which expects `clientip` and `user` in the attribute dictionary of the LogRecord. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the `extra` dictionary with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized `Formatters` would be used with particular `Handlers`.

Changed in version 2.5: `extra` was added.
info(msg[, *args[, **kwargs]])
Logs a message with level INFO on the root logger. The arguments are interpreted as for debug().

warning(msg[, *args[, **kwargs]])
Logs a message with level WARNING on the root logger. The arguments are interpreted as for debug().

critical(msg[, *args[, **kwargs]])
Logs a message with level CRITICAL on the root logger. The arguments are interpreted as for debug().

exception(msg[, *args])
Logs a message with level ERROR on the root logger. The arguments are interpreted as for debug(). Exception info is added to the logging message. This function should only be called from an exception handler.

log(level, msg[, *args[, **kwargs]])
Logs a message with level level on the root logger. The other arguments are interpreted as for debug().

disable(lvl)
Provides an overriding level lvl for all loggers which takes precedence over the logger's own level. When the need arises to temporarily throttle logging output down across the whole application, this function can be useful.

addLevelName(lvl, levelName)
Associates level lvl with text levelName in an internal dictionary, which is used to map numeric levels to a textual representation, for example when a Formatter formats a message. This function can also be used to define your own levels. The only constraints are that all levels used must be registered using this function, levels should be positive integers and they
should increase in increasing order of severity.

**getLevelName(lvl)**

Returns the textual representation of logging level *lvl*. If the level is one of the predefined levels CRITICAL, ERROR, WARNING, INFO or DEBUG then you get the corresponding string. If you have associated levels with names using `addLevelName()` then the name you have associated with *lvl* is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned. Otherwise, the string "Level %s" % lvl is returned.

**makeLogRecord(attrdict)**

Creates and returns a new LogRecord instance whose attributes are defined by *attrdict*. This function is useful for taking a pickled LogRecord attribute dictionary, sent over a socket, and reconstituting it as a LogRecord instance at the receiving end.

**basicConfig([**kwargs])**

Does basic configuration for the logging system by creating a StreamHandler with a default Formatter and adding it to the root logger. The functions debug(), info(), warning(), error() and critical() will call basicConfig() automatically if no handlers are defined for the root logger.

Changed in version 2.4: Formerly, basicConfig did not take any keyword arguments.

The following keyword arguments are supported.

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>Specifies that a FileHandler be created, using the specified filename, rather than a StreamHandler.</td>
</tr>
<tr>
<td>filemode</td>
<td>Specifies the mode to open the file, if filename is specified (if filemode is unspecified, it defaults to 'a').</td>
</tr>
<tr>
<td>format</td>
<td>Use the specified format string for the handler.</td>
</tr>
<tr>
<td>datefmt</td>
<td>Use the specified date/time format.</td>
</tr>
<tr>
<td>level</td>
<td>Set the root logger level to the specified level.</td>
</tr>
</tbody>
</table>
stream | Use the specified stream to initialize the StreamHandler. Note that this argument is incompatible with 'filename' - if both are present, 'stream' is ignored.

**shutdown()**
Informs the logging system to perform an orderly shutdown by flushing and closing all handlers.

**setLoggerClass((klass)**
Tells the logging system to use the class klass when instantiating a logger. The class should define __init__() such that only a name argument is required, and the __init__() should call Logger.__init__(). This function is typically called before any loggers are instantiated by applications which need to use custom logger behavior.

---

**See Also:**

[PEP 282, A Logging System](#)
The proposal which described this feature for inclusion in the Python standard library.

[Original Python logging package](#)
This is the original source for the logging package. The version of the package available from this site is suitable for use with Python 1.5.2, 2.1.x and 2.2.x, which do not include the logging package in the standard library.

---

**Subsections**

- [14.5.1 Logger Objects](#)
- [14.5.2 Basic example](#)
- [14.5.3 Logging to multiple destinations](#)
- [14.5.4 Sending and receiving logging events across a network](#)
- [14.5.5 Handler Objects](#)
  - [14.5.5.1 StreamHandler](#)
  - [14.5.5.2 FileHandler](#)
• **14.5.3** RotatingFileHandler
• **14.5.4** TimedRotatingFileHandler
• **14.5.5** SocketHandler
• **14.5.6** DatagramHandler
• **14.5.7** SysLogHandler
• **14.5.8** NTEventLogHandler
• **14.5.9** SMTPHandler
• **14.5.10** MemoryHandler
• **14.5.11** HTTPHandler

• **14.5.6** Formatter Objects
• **14.5.7** Filter Objects
• **14.5.8** LogRecord Objects
• **14.5.9** Thread Safety
• **14.5.10** Configuration
  • **14.5.10.1** Configuration functions
  • **14.5.10.2** Configuration file format

_Release 2.5, documentation updated on 19th September, 2006._
See [About this document...](...) for information on suggesting changes.
14.5.1 Logger Objects

Loggers have the following attributes and methods. Note that Loggers are never instantiated directly, but always through the module-level function logging.getLogger(name).

**propagate**
If this evaluates to false, logging messages are not passed by this logger or by child loggers to higher level (ancestor) loggers. The constructor sets this attribute to 1.

**setLevel**(lvl)
Sets the threshold for this logger to lvl. Logging messages which are less severe than lvl will be ignored. When a logger is created, the level is set to NOTSET (which causes all messages to be processed when the logger is the root logger, or delegation to the parent when the logger is a non-root logger). Note that the root logger is created with level WARNING.

The term "delegation to the parent" means that if a logger has a level of NOTSET, its chain of ancestor loggers is traversed until either an ancestor with a level other than NOTSET is found, or the root is reached.

If an ancestor is found with a level other than NOTSET, then that ancestor's level is treated as the effective level of the logger where the ancestor search began, and is used to determine how a logging event is handled.

If the root is reached, and it has a level of NOTSET, then all messages will be processed. Otherwise, the root's level will be used as the effective level.

**isEnabledFor**(lvl)
Indicates if a message of severity lvl would be processed by this logger. This method checks first the module-level level set by logging.disable(lvl) and then the logger's effective level as determined by getEffectiveLevel().

**getEffectiveLevel()**
Indicates the effective level for this logger. If a value other than NOTSET
has been set using \texttt{setLevel()}\texttt{, it is returned. Otherwise, the hierarchy is traversed towards the root until a value other than \texttt{NOTSET} is found, and that value is returned.}

\texttt{debug( msg[, \*args[, \**kwargs]])}

Logs a message with level \texttt{DEBUG} on this logger. The \texttt{msg} is the message format string, and the \texttt{args} are the arguments which are merged into \texttt{msg} using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

There are two keyword arguments in \texttt{kwargs} which are inspected: \texttt{exc_info} which, if it does not evaluate as false, causes exception information to be added to the logging message. If an exception tuple (in the format returned by \texttt{sys.exc_info()}) is provided, it is used; otherwise, \texttt{sys.exc_info()} is called to get the exception information.

The other optional keyword argument is \texttt{extra} which can be used to pass a dictionary which is used to populate the \texttt{__dict__} of the LogRecord created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```python
FORMAT = "%(asctime)-15s (%(clientip)s %(user)-8s %(message)s"
logging.basicConfig(format=FORMAT)
dict = {'clientip' : '192.168.0.1', 'user' : 'fbloggs'}
logger = logging.getLogger("tcpserver")
logger.warning("Protocol problem: %s", "connection reset", extra=dict)
```

would print something like

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs Protocol problem: 
```

The keys in the dictionary passed in \texttt{extra} should not clash with the keys used by the logging system. (See the \texttt{Formatter} documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the \texttt{Formatter} has been set up with a format string which expects 'clientip' and 'user' in the attribute dictionary of the LogRecord. If these are missing, the message will not be logged because a string formatting exception will occur. So in this
case, you always need to pass the extra dictionary with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized Formatters would be used with particular Handlers.

Changed in version 2.5: extra was added.

```
info( msg[, *args[, **kwargs]])

Logs a message with level INFO on this logger. The arguments are interpreted as for debug().
```

```
warning( msg[, *args[, **kwargs]])

Logs a message with level WARNING on this logger. The arguments are interpreted as for debug().
```

```
error( msg[, *args[, **kwargs]])

Logs a message with level ERROR on this logger. The arguments are interpreted as for debug().
```

```
critical( msg[, *args[, **kwargs]])

Logs a message with level CRITICAL on this logger. The arguments are interpreted as for debug().
```

```
log( lvl, msg[, *args[, **kwargs]])

Logs a message with integer level lvl on this logger. The other arguments are interpreted as for debug().
```

```
exception( msg[, *args])

Logs a message with level ERROR on this logger. The arguments are interpreted as for debug(). Exception info is added to the logging message. This method should only be called from an exception handler.
```
addFilter( filt)
    Adds the specified filter filt to this logger.

removeFilter( filt)
    Removes the specified filter filt from this logger.

filter( record)
    Applies this logger's filters to the record and returns a true value if the
    record is to be processed.

addHandler( hdlr)
    Adds the specified handler hdlr to this logger.

removeHandler( hdlr)
    Removes the specified handler hdlr from this logger.

findCaller()
    Finds the caller's source filename and line number. Returns the filename
    and line number as a 2-element tuple.

handle( record)
    Handles a record by passing it to all handlers associated with this logger
    and its ancestors (until a false value of propagate is found). This method is
    used for unpickled records received from a socket, as well as those created
    locally. Logger-level filtering is applied using filter().

makeRecord( name, lvl, fn, lno, msg, args, exc_info, func, extra)
    This is a factory method which can be overridden in subclasses to create
    specialized LogRecord instances. Changed in version 2.5: func and extra
    were added.
14.5.2 Basic example

Changed in version 2.4: formerly `basicConfig` did not take any keyword arguments.

The `logging` package provides a lot of flexibility, and its configuration can appear daunting. This section demonstrates that simple use of the logging package is possible.

The simplest example shows logging to the console:

```python
import logging
logging.debug('A debug message')
logging.info('Some information')
logging.warning('A shot across the bows')
```

If you run the above script, you'll see this:

```
WARNING:root:A shot across the bows
```

Because no particular logger was specified, the system used the root logger. The debug and info messages didn't appear because by default, the root logger is configured to only handle messages with a severity of WARNING or above. The message format is also a configuration default, as is the output destination of the messages - `sys.stderr`. The severity level, the message format and destination can be easily changed, as shown in the example below:

```python
import logging
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(levelname)s %(message)s',
                    filename='/tmp/myapp.log',
                    filemode='w')
logging.debug('A debug message')
logging.info('Some information')
logging.warning('A shot across the bows')
```

The `basicConfig()` method is used to change the configuration defaults, which results in output (written to `/tmp/myapp.log`) which should look something like the following:
This time, all messages with a severity of DEBUG or above were handled, and the format of the messages was also changed, and output went to the specified file rather than the console.

Formatting uses standard Python string formatting - see section 3.6.2. The format string takes the following common specifiers. For a complete list of specifiers, consult the `Formatter` documentation.

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%(name)s</td>
<td>Name of the logger (logging channel).</td>
</tr>
<tr>
<td>% (levelname)s</td>
<td>Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL').</td>
</tr>
<tr>
<td>%(asctime)s</td>
<td>Human-readable time when the LogRecord was created. By default this is of the form <code>2003-07-08 16:49:45,896</code> (the numbers after the comma are millisecond portion of the time).</td>
</tr>
<tr>
<td>%(message)s</td>
<td>The logged message.</td>
</tr>
</tbody>
</table>

To change the date/time format, you can pass an additional keyword parameter, `datefmt`, as in the following:

```python
import logging

logging.basicConfig(level=logging.DEBUG,
                     format='%(asctime)s %(levelname)-8s %(message)s',
                     datefmt='%a, %d %b %Y %H:%M:%S',
                     filename='/temp/myapp.log',
                     filemode='w')

logging.debug('A debug message')
logging.info('Some information')
logging.warning('A shot across the bows')
```

which would result in output like

```
Fri, 02 Jul 2004 13:06:18 DEBUG    A debug message
Fri, 02 Jul 2004 13:06:18 INFO      Some information
Fri, 02 Jul 2004 13:06:18 WARNING   A shot across the bows
```

The date format string follows the requirements of `strftime()` - see the
documentation for the \texttt{time} module.

If, instead of sending logging output to the console or a file, you'd rather use a file-like object which you have created separately, you can pass it to \texttt{basicConfig()} using the \textit{stream} keyword argument. Note that if both \textit{stream} and \textit{filename} keyword arguments are passed, the \textit{stream} argument is ignored.

Of course, you can put variable information in your output. To do this, simply have the message be a format string and pass in additional arguments containing the variable information, as in the following example:

\begin{verbatim}
import logging

logging.basicConfig(level=logging.DEBUG,
    format='%(asctime)s %(levelname)-8s %(message)s',
    datefmt='%a, %d %b %Y %H:%M:%S',
    filename='/temp/myapp.log',
    filemode='w')

logging.error('Pack my box with %d dozen %s', 5, 'liquor jugs')
\end{verbatim}

which would result in

\begin{verbatim}
Wed, 21 Jul 2004 15:35:16 ERROR Pack my box with 5 dozen liquor jugs
\end{verbatim}
14.5.3 Logging to multiple destinations

Let's say you want to log to console and file with different message formats and in differing circumstances. Say you want to log messages with levels of DEBUG and higher to file, and those messages at level INFO and higher to the console. Let's also assume that the file should contain timestamps, but the console messages should not. Here's how you can achieve this:

```python
import logging

# set up logging to file - see previous section for more details
logging.basicConfig(level=logging.DEBUG,  
                    format='%(asctime)s	%(name)-12s	%(levelname)-8s	%(message)s',  
                    datefmt='%m-%d	%H:%M',  
                    filename='/temp/myapp.log',  
                    filemode='w')

# define a Handler which writes INFO messages or higher to the sys.stderr
console = logging.StreamHandler()
console.setLevel(logging.INFO)

# set a format which is simpler for console use
formatter = logging.Formatter('%(name)-12s:	%(levelname)-8s	%(message)s')

# tell the handler to use this format
console.setFormatter(formatter)

# add the handler to the root logger
logging.getLogger('').addHandler(console)

# Now, we can log to the root logger, or any other logger. First the
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

When you run this, on the console you will see

```
root : INFO      Jackdaws love my big sphinx of quartz.
myapp.area1 : INFO  How quickly daft jumping zebras vex.
myapp.area2 : WARNING  Jail zesty vixen who grabbed pay from quack.
myapp.area2 : ERROR  The five boxing wizards jump quickly.
```
and in the file you will see something like

<table>
<thead>
<tr>
<th>Time</th>
<th>User</th>
<th>Level</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>10-22 22:19</td>
<td>root</td>
<td>INFO</td>
<td>Jackdaws love my big sphinx of quartz.</td>
</tr>
<tr>
<td>10-22 22:19</td>
<td>myapp.area1</td>
<td>DEBUG</td>
<td>Quick zephyrs blow, vexing daft Jim.</td>
</tr>
<tr>
<td>10-22 22:19</td>
<td>myapp.area1</td>
<td>INFO</td>
<td>How quickly daft jumping zebras vex.</td>
</tr>
<tr>
<td>10-22 22:19</td>
<td>myapp.area2</td>
<td>WARNING</td>
<td>Jail zesty vixen who grabbed pay f</td>
</tr>
<tr>
<td>10-22 22:19</td>
<td>myapp.area2</td>
<td>ERROR</td>
<td>The five boxing wizards jump quickly</td>
</tr>
</tbody>
</table>

As you can see, the DEBUG message only shows up in the file. The other messages are sent to both destinations.

This example uses console and file handlers, but you can use any number and combination of handlers you choose.
14.5.4 Sending and receiving logging events across a network

Let's say you want to send logging events across a network, and handle them at the receiving end. A simple way of doing this is attaching a `SocketHandler` instance to the root logger at the sending end:

```python
import logging, logging.handlers

rootLogger = logging.getLogger('')
rootLogger.setLevel(logging.DEBUG)
socketHandler = logging.handlers.SocketHandler('localhost',
                                           logging.handlers.DEFAULT_TCP_LOGGING_PORT)
# don't bother with a formatter, since a socket handler sends the ev
# an unformatted pickle
rootLogger.addHandler(socketHandler)

# Now, we can log to the root logger, or any other logger. First the
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas
# application:
logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')

At the receiving end, you can set up a receiver using the `SocketServer` module. Here is a basic working example:

```python
import cPickle
import logging
import logging.handlers
import SocketServer
import struct

class LogRecordStreamHandler(SocketServer.StreamRequestHandler):
    '''Handler for a streaming logging request.

    This basically logs the record using whatever logging policy is
    configured locally.''
```
def handle(self):
    """
    Handle multiple requests - each expected to be a 4-byte length
    followed by the LogRecord in pickle format. Logs the record
    according to whatever policy is configured locally.
    """
    while 1:
        chunk = self.connection.recv(4)
        if len(chunk) < 4:
            break
        slen = struct.unpack(">L", chunk)[0]
        chunk = self.connection.recv(slen)
        while len(chunk) < slen:
            chunk = chunk + self.connection.recv(slen - len(chunk))
        obj = self.unPickle(chunk)
        record = logging.makeLogRecord(obj)
        self.handleLogRecord(record)

def unPickle(self, data):
    return cPickle.loads(data)

def handleLogRecord(self, record):
    # if a name is specified, we use the named logger rather than
    # implied by the record.
    if self.server.logname is not None:
        name = self.server.logname
    else:
        name = record.name
    logger = logging.getLogger(name)
    # N.B. EVERY record gets logged. This is because Logger.handle
    # is normally called AFTER logger-level filtering. If you want
    # to do filtering, do it at the client end to save wasting
    # cycles and network bandwidth!
    logger.handle(record)

class LogRecordSocketReceiver(SocketServer.ThreadingTCPServer):
    """simple TCP socket-based logging receiver suitable for testing
    """
    allow_reuse_address = 1

    def __init__(self, host='localhost',
                 port=logging.handlers.DEFAULT_TCP_LOGGING_PORT,
                 handler=LogRecordStreamHandler):
        SocketServer.ThreadingTCPServer.__init__(self, (host, port),
                                                   handler)
        self.abort = 0
        self.timeout = 1
        self.logname = None

    def serve_until_stopped(self):
import select
abort = 0
while not abort:
    rd, wr, ex = select.select([self.socket.fileno()], [], [],
                                self.timeout)

    if rd:
        self.handle_request()
    abort = self.abort

def main():
    logging.basicConfig(
        format="%s
        tcpserver = LogRecordSocketReceiver()
        print "About to start TCP server..."
        tcpserver.serve_until_stopped()

        if __name__ == "__main__":
            main()

First run the server, and then the client. On the client side, nothing is printed on
the console; on the server side, you should see something like:

    About to start TCP server...
    59 root INFO  Jackdaws love my big sphinx of quartz
    59 myapp.area1 DEBUG Quick zephyrs blow, vexing daft Jim.
    69 myapp.area1 INFO  How quickly daft jumping zebras vex.
    69 myapp.area2 WARNING Jail zesty vixen who grabbed pay from
    69 myapp.area2 ERROR  The five boxing wizards jump quickly.
14.5.5 Handler Objects

Handlers have the following attributes and methods. Note that Handler is never instantiated directly; this class acts as a base for more useful subclasses. However, the `__init__()` method in subclasses needs to call Handler.__init__().

```python
__init__(level=NOTSET)
```
Initializes the Handler instance by setting its level, setting the list of filters to the empty list and creating a lock (using createLock()) for serializing access to an I/O mechanism.

```python
createLock()
```
Initializes a thread lock which can be used to serialize access to underlying I/O functionality which may not be threadsafe.

```python
acquire()
```
Acquires the thread lock created with createLock().

```python
release()
```
Releases the thread lock acquired with acquire().

```python
setLevel(lvl)
```
Sets the threshold for this handler to lvl. Logging messages which are less severe than lvl will be ignored. When a handler is created, the level is set to NOTSET (which causes all messages to be processed).

```python
setFormatter(form)
```
Sets the Formatter for this handler to form.

```python
addFilter(filt)
```
Adds the specified filter filt to this handler.

```python
removeFilter(filt)
```
Removes the specified filter filt from this handler.
**filter**( *record*)
Applies this handler's filters to the record and returns a true value if the record is to be processed.

**flush()**
Ensure all logging output has been flushed. This version does nothing and is intended to be implemented by subclasses.

**close()**
Tidy up any resources used by the handler. This version does nothing and is intended to be implemented by subclasses.

**handle**( *record*)
Conditionally emits the specified logging record, depending on filters which may have been added to the handler. Wraps the actual emission of the record with acquisition/release of the I/O thread lock.

**handleError**( *record*)
This method should be called from handlers when an exception is encountered during an `emit()` call. By default it does nothing, which means that exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The specified record is the one which was being processed when the exception occurred.

**format**( *record*)
Do formatting for a record - if a formatter is set, use it. Otherwise, use the default formatter for the module.

**emit**( *record*)
Do whatever it takes to actually log the specified logging record. This version is intended to be implemented by subclasses and so raises a `NotImplementedError`.
- 14.5.5.1 StreamHandler
- 14.5.5.2 FileHandler
- 14.5.5.3 RotatingFileHandler
- 14.5.5.4 TimedRotatingFileHandler
- 14.5.5.5 SocketHandler
- 14.5.5.6 DatagramHandler
- 14.5.5.7 SysLogHandler
- 14.5.5.8 NTEventLogHandler
- 14.5.5.9 SMTPHandler
- 14.5.5.10 MemoryHandler
- 14.5.5.11 HTTPHandler

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.5.6 Formatter Objects

Formatters have the following attributes and methods. They are responsible for converting a LogRecord to (usually) a string which can be interpreted by either a human or an external system. The base Formatter allows a formatting string to be specified. If none is supplied, the default value of '% (message)s' is used.

A Formatter can be initialized with a format string which makes use of knowledge of the LogRecord attributes - such as the default value mentioned above making use of the fact that the user's message and arguments are pre-formatted into a LogRecord's message attribute. This format string contains standard python %-style mapping keys. See section 3.6.2, "String Formatting Operations," for more information on string formatting.

Currently, the useful mapping keys in a LogRecord are:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%(name)s</td>
<td>Name of the logger (logging channel).</td>
</tr>
<tr>
<td>%(levelname)s</td>
<td>Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL).</td>
</tr>
<tr>
<td>%(pathname)s</td>
<td>Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL').</td>
</tr>
<tr>
<td>%(filename)s</td>
<td>Full pathname of the source file where the logging call was issued (if available).</td>
</tr>
<tr>
<td>%(module)s</td>
<td>Filename portion of pathname.</td>
</tr>
<tr>
<td>%(funcName)s</td>
<td>Module (name portion of filename).</td>
</tr>
<tr>
<td>%(lineno)d</td>
<td>Name of function containing the logging call.</td>
</tr>
<tr>
<td>%(created)f</td>
<td>Source line number where the logging call was issued (if available).</td>
</tr>
<tr>
<td>%(asctime)s</td>
<td>Time when the LogRecord was created (as returned by time.time()).</td>
</tr>
<tr>
<td>%(asctime)s</td>
<td>Human-readable time when the LogRecord was created. By default this is of the form ``2003-07-08 16:49:45,896'' (the numbers after the comma are</td>
</tr>
<tr>
<td>Format</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td>%(msecs)d</td>
<td>Millisecond portion of the time when the LogRecord was created.</td>
</tr>
<tr>
<td>%(thread)d</td>
<td>Thread ID (if available).</td>
</tr>
<tr>
<td>%(threadName)s</td>
<td>Thread name (if available).</td>
</tr>
<tr>
<td>%(process)d</td>
<td>Process ID (if available).</td>
</tr>
<tr>
<td>%(message)s</td>
<td>The logged message, computed as msg % args.</td>
</tr>
</tbody>
</table>

Changed in version 2.5: `funcName` was added.

**class Formatter**([`fmt`, `datefmt`])

Returns a new instance of the `Formatter` class. The instance is initialized with a format string for the message as a whole, as well as a format string for the date/time portion of a message. If no `fmt` is specified, '%(message)s' is used. If no `datefmt` is specified, the ISO8601 date format is used.

**format**( `record` )

The record's attribute dictionary is used as the operand to a string formatting operation. Returns the resulting string. Before formatting the dictionary, a couple of preparatory steps are carried out. The `message` attribute of the record is computed using `msg % args`. If the formatting string contains '(asctime)', `formatTime()` is called to format the event time. If there is exception information, it is formatted using `formatException()` and appended to the message.

**formatTime**( `record`, `datefmt` )

This method should be called from `format()` by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behavior is as follows: if `datefmt` (a string) is specified, it is used with `time.strftime()` to format the creation time of the record. Otherwise, the ISO8601 format is used. The resulting string is returned.

**formatException**( `exc_info` )
Formats the specified exception information (a standard exception tuple as returned by \texttt{sys.exc_info()}) as a string. This default implementation just uses \texttt{traceback.print_exception()}. The resulting string is returned.
14.5.7 Filter Objects

Filters can be used by Handlers and Loggers for more sophisticated filtering than is provided by levels. The base filter class only allows events which are below a certain point in the logger hierarchy. For example, a filter initialized with "A.B" will allow events logged by loggers "A.B", "A.B.C", "A.B.C.D", "A.B.D" etc. but not "A.BB", "B.A.B" etc. If initialized with the empty string, all events are passed.

class Filter([name])

Returns an instance of the Filter class. If name is specified, it names a logger which, together with its children, will have its events allowed through the filter. If no name is specified, allows every event.

filter(record)

Is the specified record to be logged? Returns zero for no, nonzero for yes. If deemed appropriate, the record may be modified in-place by this method.
14.5.8 LogRecord Objects

LogRecord instances are created every time something is logged. They contain all the information pertinent to the event being logged. The main information passed in is in msg and args, which are combined using msg % args to create the message field of the record. The record also includes information such as when the record was created, the source line where the logging call was made, and any exception information to be logged.

class LogRecord(name, lvl, pathname, lineno, msg, args, exc_info)

Returns an instance of LogRecord initialized with interesting information. The name is the logger name; lvl is the numeric level; pathname is the absolute pathname of the source file in which the logging call was made; lineno is the line number in that file where the logging call is found; msg is the user-supplied message (a format string); args is the tuple which, together with msg, makes up the user message; and exc_info is the exception tuple obtained by calling sys.exc_info() (or None, if no exception information is available).

getMessage()

Returns the message for this LogRecord instance after merging any user-supplied arguments with the message.
14.5.9 Thread Safety

The logging module is intended to be thread-safe without any special work needing to be done by its clients. It achieves this through using threading locks; there is one lock to serialize access to the module's shared data, and each handler also creates a lock to serialize access to its underlying I/O.
14.5.10 Configuration

Subsections

- 14.5.10.1 Configuration functions
- 14.5.10.2 Configuration file format

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.6 getpass -- Portable password input

The getpass module provides two functions:

**getpass** ([*prompt*, *stream]*)

Prompt the user for a password without echoing. The user is prompted using the string *prompt*, which defaults to 'Password: '. On UNIX, the prompt is written to the file-like object *stream*, which defaults to *sys.stdout* (this argument is ignored on Windows).

Availability: Macintosh, UNIX, Windows. Changed in version 2.5: The *stream* parameter was added.

**getuser()**

Return the ``login name`` of the user. Availability: UNIX, Windows.

This function checks the environment variables LOGNAME, USER, LNAME and USERNAME, in order, and returns the value of the first one which is set to a non-empty string. If none are set, the login name from the password database is returned on systems which support the *pwd* module, otherwise, an exception is raised.
14.7 curses -- Terminal handling for character-cell displays

Changed in version 1.6: Added support for the ncurses library and converted to a package.

The curses module provides an interface to the curses library, the de-facto standard for portable advanced terminal handling.

While curses is most widely used in the UNIX environment, versions are available for DOS, OS/2, and possibly other systems as well. This extension module is designed to match the API of ncurses, an open-source curses library hosted on Linux and the BSD variants of UNIX.

See Also:

Module curses.ascii: Utilities for working with ASCII characters, regardless of your locale settings.

Module curses.panel: A panel stack extension that adds depth to curses windows.

Module curses.textpad: Editable text widget for curses supporting Emacs-like bindings.

Module curses.wrapper: Convenience function to ensure proper terminal setup and resetting on application entry and exit.

Curses Programming with Python Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond, is available on the Python Web site.

The Demo/curses/ directory in the Python source distribution contains
some example programs using the curses bindings provided by this module.

Subsections

- **14.7.1 Functions**
- **14.7.2 Window Objects**
- **14.7.3 Constants**

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
14.7.1 Functions

The module `curses` defines the following exception:

**exception error**

Exception raised when a curses library function returns an error.

**Note:** Whenever $x$ or $y$ arguments to a function or a method are optional, they default to the current cursor location. Whenever `attr` is optional, it defaults to `A_NORMAL`.

The module `curses` defines the following functions:

**baudrate()**

Returns the output speed of the terminal in bits per second. On software terminal emulators it will have a fixed high value. Included for historical reasons; in former times, it was used to write output loops for time delays and occasionally to change interfaces depending on the line speed.

**beep()**

Emit a short attention sound.

**can_change_color()**

Returns true or false, depending on whether the programmer can change the colors displayed by the terminal.

**cbreak()**

Enter cbreak mode. In cbreak mode (sometimes called ``rare" mode) normal tty line buffering is turned off and characters are available to be read one by one. However, unlike raw mode, special characters (interrupt, quit, suspend, and flow control) retain their effects on the tty driver and calling program. Calling first `raw()` then `cbreak()` leaves the terminal in cbreak mode.

**color_content(color_number)**

Returns the intensity of the red, green, and blue (RGB) components in the color `color_number`, which must be between 0 and `COLORS`. A 3-tuple is
returned, containing the R,G,B values for the given color, which will be between 0 (no component) and 1000 (maximum amount of component).

**color_pair( color_number)**
Returns the attribute value for displaying text in the specified color. This attribute value can be combined with A_STANDOUT, A_REVERSE, and the other A_* attributes. pair_number() is the counterpart to this function.

**curs_set( visibility)**
Sets the cursor state. visibility can be set to 0, 1, or 2, for invisible, normal, or very visible. If the terminal supports the visibility requested, the previous cursor state is returned; otherwise, an exception is raised. On many terminals, the ``visible'' mode is an underline cursor and the ``very visible'' mode is a block cursor.

**def_prog_mode()**
Saves the current terminal mode as the ``program'' mode, the mode when the running program is using curses. (Its counterpart is the ``shell'' mode, for when the program is not in curses.) Subsequent calls to reset_prog_mode() will restore this mode.

**def_shell_mode()**
Saves the current terminal mode as the ``shell'' mode, the mode when the running program is not using curses. (Its counterpart is the ``program'' mode, when the program is using curses capabilities.) Subsequent calls to reset_shell_mode() will restore this mode.

**delay_output( ms)**
Inserts an ms millisecond pause in output.

**doupdate()**
Update the physical screen. The curses library keeps two data structures, one representing the current physical screen contents and a virtual screen representing the desired next state. The doupdate() ground updates the physical screen to match the virtual screen.

The virtual screen may be updated by a noutrefresh() call after write
operations such as `addstr()` have been performed on a window. The normal `refresh()` call is simply `noutrefresh()` followed by `doupdate();` if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing `noutrefresh()` calls on all windows, followed by a single `doupdate();`.

**echo()**

Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

**endwin()**

De-initialize the library, and return terminal to normal status.

**erasechar()**

Returns the user's current erase character. Under UNIX operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

**filter()**

The filter() routine, if used, must be called before `initscr()` is called. The effect is that, during those calls, LINES is set to 1; the capabilities clear, cup, cud, cud1, cuu1, cuu, vpa are disabled; and the home string is set to the value of cr. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

**flash()**

Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as `visible bell' to the audible attention signal produced by `beep();`.

**flushinp()**

Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

**getmouse()**
After `getch()` returns `KEY_MOUSE` to signal a mouse event, this method should be called to retrieve the queued mouse event, represented as a 5-tuple `(id, x, y, z, bstate)`. `id` is an ID value used to distinguish multiple devices, and `x, y, z` are the event’s coordinates. (`z` is currently unused). `bstate` is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of one or more of the following constants, where `n` is the button number from 1 to 4: `BUTTONn_PRESSED`, `BUTTONn_RELEASED`, `BUTTONn_CLICKED`, `BUTTONn_DOUBLE_CLICKED`, `BUTTONn_TRIPLE_CLICKED`, `BUTTON_SHIFT`, `BUTTON_CTRL`, `BUTTON_ALT`.

`getsyx()`

Returns the current coordinates of the virtual screen cursor in `y` and `x`. If `leaveok` is currently true, then `-1,-1` is returned.

`getwin(file)`

Reads window related data stored in the file by an earlier `putwin()` call. The routine then creates and initializes a new window using that data, returning the new window object.

`has_colors()`

Returns true if the terminal can display colors; otherwise, it returns false.

`has_ic()`

Returns true if the terminal has insert- and delete- character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

`has_il()`

Returns true if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

`has_key(ch)`

Takes a key value `ch`, and returns true if the current terminal type recognizes a key with that value.
**halfdelay** *(tenths)*

Used for half-delay mode, which is similar to cbreak mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, an exception is raised if nothing has been typed. The value of *tenths* must be a number between 1 and 255. Use **nocbreak()** to leave half-delay mode.

**init_color** *(color_number, r, g, b)*

Changes the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of *color_number* must be between 0 and COLORS. Each of *r*, *g*, *b*, must be a value between 0 and 1000. When **init_color()** is used, all occurrences of that color on the screen immediately change to the new definition. This function is a no-op on most terminals; it is active only if **can_change_color()** returns 1.

**init_pair** *(pair_number, fg, bg)*

Changes the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of *pair_number* must be between 1 and COLOR_PAIRS - 1 (the 0 color pair is wired to white on black and cannot be changed). The value of *fg* and *bg* arguments must be between 0 and COLORS. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

**initscr()**

Initialize the library. Returns a **WindowObject** which represents the whole screen. **Note:** If there is an error opening the terminal, the underlying curses library may cause the interpreter to exit.

**isendwin()**

Returns true if **endwin()** has been called (that is, the curses library has been deinitialized).

**keyname** *(k)*

Return the name of the key numbered *k*. The name of a key generating
printable ASCII character is the key's character. The name of a control-key combination is a two-character string consisting of a caret followed by the corresponding printable ASCII character. The name of an alt-key combination (128-255) is a string consisting of the prefix `M-' followed by the name of the corresponding ASCII character.

**killchar()**
- Returns the user's current line kill character. Under UNIX operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

**longname()**
- Returns a string containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to `initscr()`.

**meta(\textit{yes})**
- If \textit{yes} is 1, allow 8-bit characters to be input. If \textit{yes} is 0, allow only 7-bit chars.

**mouseinterval(\textit{interval})**
- Sets the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and returns the previous interval value. The default value is 200 msec, or one fifth of a second.

**mousemask(\textit{mousemask})**
- Sets the mouse events to be reported, and returns a tuple (\texttt{availmask}, \texttt{oldmask}). \texttt{availmask} indicates which of the specified mouse events can be reported; on complete failure it returns 0. \texttt{oldmask} is the previous value of the given window's mouse event mask. If this function is never called, no mouse events are ever reported.

**napms(\textit{ms})**
- Sleep for \textit{ms} milliseconds.

**newpad(\textit{nlines}, \textit{ncols})**
Creates and returns a pointer to a new pad data structure with the given number of lines and columns. A pad is returned as a window object.

A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (such as from scrolling or echoing of input) do not occur. The refresh() and noutrefresh() methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol; the p arguments refer to the upper left corner of the pad region to be displayed and the s arguments define a clipping box on the screen within which the pad region is to be displayed.

\texttt{newwin([nlines, ncols], begin_y, begin_x)}

Return a new window, whose left-upper corner is at \((begin_y, begin_x)\), and whose height/width is \(nlines/ncols\).

By default, the window will extend from the specified position to the lower right corner of the screen.

\texttt{nl()}

Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.

\texttt{nocbreak()}

Leave cbreak mode. Return to normal ``cooked'' mode with line buffering.

\texttt{noecho()}

Leave echo mode. Echoing of input characters is turned off.

\texttt{nonl()}

Leave newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of addch('\n'), which always does the equivalent of return and line feed on the virtual screen). With
translation off, curses can sometimes speed up vertical motion a little; also, it will be able to detect the return key on input.

**noqiflush()**

When the noqiflush routine is used, normal flush of input and output queues associated with the INTR, QUIT and SUSP characters will not be done. You may want to call noqiflush() in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

**noraw()**

Leave raw mode. Return to normal ```cooked`` mode with line buffering.

**pair_content(pair_number)**

Returns a tuple (fg, bg) containing the colors for the requested color pair. The value of pair_number must be between 1 and COLOR_PAIRS - 1.

**pair_number(attr)**

Returns the number of the color-pair set by the attribute value attr. color_pair() is the counterpart to this function.

**putp(string)**

Equivalent to tputs(str, 1, putchar); emits the value of a specified terminfo capability for the current terminal. Note that the output of putp always goes to standard output.

**qiflush([flag])**

If flag is false, the effect is the same as calling noqiflush(). If flag is true, or no argument is provided, the queues will be flushed when these control characters are read.

**raw()**

Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off; characters are presented to curses input functions one by one.

**reset_prog_mode()**
Restores the terminal to "program" mode, as previously saved by `def_prog_mode()`.

**reset_shell_mode()**
Restores the terminal to "shell" mode, as previously saved by `def_shell_mode()`.

**setsyx(y, x)**
Sets the virtual screen cursor to y, x. If y and x are both -1, then leaveok is set.

**setupterm([termstr, fd])**
Initializes the terminal. `termstr` is a string giving the terminal name; if omitted, the value of the TERM environment variable will be used. `fd` is the file descriptor to which any initialization sequences will be sent; if not supplied, the file descriptor for `sys.stdout` will be used.

**start_color()**
Must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after `initscr()`.

`start_color()` initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables in the `curses` module, `COLORS` and `COLOR_PAIRS`, containing the maximum number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the values they had when the terminal was just turned on.

**termattrs()**
Returns a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs complete control over the appearance of the screen.

**termname()**
Returns the value of the environment variable TERM, truncated to 14 characters.
**tigetflag**( *capname*)

Returns the value of the Boolean capability corresponding to the terminfo capability name *capname*. The value -1 is returned if *capname* is not a Boolean capability, or 0 if it is canceled or absent from the terminal description.

**tigetnum**( *capname*)

Returns the value of the numeric capability corresponding to the terminfo capability name *capname*. The value -2 is returned if *capname* is not a numeric capability, or -1 if it is canceled or absent from the terminal description.

**tigetstr**( *capname*)

Returns the value of the string capability corresponding to the terminfo capability name *capname*. None is returned if *capname* is not a string capability, or is canceled or absent from the terminal description.

**tparm**( *str*[,...])

Instantiates the string *str* with the supplied parameters, where *str* should be a parameterized string obtained from the terminfo database. E.g. tparm(tigetstr("cup"), 5, 3) could result in '\033[6;4H', the exact result depending on terminal type.

**typeahead**( *fd*)

Specifies that the file descriptor *fd* be used for typeahead checking. If *fd* is -1, then no typeahead checking is done.

The curses library does ``line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until refresh or doupdate is called again, allowing faster response to commands typed in advance. This function allows specifying a different file descriptor for typeahead checking.

**unctrl**( *ch*)

Returns a string which is a printable representation of the character *ch*. Control characters are displayed as a caret followed by the character, for
example as ^C. Printing characters are left as they are.

ungetch( ch)
Push ch so the next getch() will return it. Note: Only one ch can be pushed before getch() is called.

ungetmouse( id, x, y, z, bstate)
Push a KEY_MOUSE event onto the input queue, associating the given state data with it.

use_env( flag)
If used, this function should be called before initscr() or newterm are called. When flag is false, the values of lines and columns specified in the terminfo database will be used, even if environment variables LINES and COLUMNS (used by default) are set, or if curses is running in a window (in which case default behavior would be to use the window size if LINES and COLUMNS are not set).

use_default_colors()
Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application. The default color is assigned to the color number -1. After calling this function, init_pair(x, curses.COLOR_RED, -1) initializes, for instance, color pair x to a red foreground color on the default background.
14.7.2 Window Objects

Window objects, as returned by `initscr()` and `newwin()` above, have the following methods:

`addch([y, x], ch[, attr])`

Note: A character means a C character (an ASCII code), rather than a Python character (a string of length 1). (This note is true whenever the documentation mentions a character.) The built-in `ord()` is handy for conveying strings to codes.

Paint character `ch` at `(y, x)` with attributes `attr`, overwriting any character previously painter at that location. By default, the character position and attributes are the current settings for the window object.

`addnstr([y, x], str, n[, attr])`

Paint at most `n` characters of the string `str` at `(y, x)` with attributes `attr`, overwriting anything previously on the display.

`addstr([y, x], str[, attr])`

Paint the string `str` at `(y, x)` with attributes `attr`, overwriting anything previously on the display.

`attroff(attr)`

Remove attribute `attr` from the ``background'' set applied to all writes to the current window.

`attron(attr)`

Add attribute `attr` from the ``background'' set applied to all writes to the current window.

`attrset(attr)`

Set the ``background'' set of attributes to `attr`. This set is initially 0 (no attributes).

`bkgd()`
Sets the background property of the window to the character \texttt{ch}, with attributes \texttt{attr}. The change is then applied to every character position in that window:

- The attribute of every character in the window is changed to the new background attribute.
- Wherever the former background character appears, it is changed to the new background character.

\texttt{bkgdset( ch[, attr])}

Sets the window's background. A window's background consists of a character and any combination of attributes. The attribute part of the background is combined (OR'ed) with all non-blank characters that are written into the window. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations.

\texttt{border([ls[, rs[, ts[, bs[, tl[, tr[, bl[, br]]]]]]])}

Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border; see the table below for more details. The characters can be specified as integers or as one-character strings.

\textbf{Note:} A 0 value for any parameter will cause the default character to be used for that parameter. Keyword parameters can not be used. The defaults are listed in this table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ls}</td>
<td>Left side</td>
<td>\texttt{ACS_VLINE}</td>
</tr>
<tr>
<td>\texttt{rs}</td>
<td>Right side</td>
<td>\texttt{ACS_VLINE}</td>
</tr>
<tr>
<td>\texttt{ts}</td>
<td>Top</td>
<td>\texttt{ACS_HLINE}</td>
</tr>
<tr>
<td>\texttt{bs}</td>
<td>Bottom</td>
<td>\texttt{ACS_HLINE}</td>
</tr>
<tr>
<td>\texttt{tl}</td>
<td>Upper-left corner</td>
<td>\texttt{ACS_ULCORNER}</td>
</tr>
<tr>
<td>\texttt{tr}</td>
<td>Upper-right corner</td>
<td>\texttt{ACS_URCORNER}</td>
</tr>
<tr>
<td>\texttt{bl}</td>
<td>Bottom-left corner</td>
<td>\texttt{ACS_LLCORNER}</td>
</tr>
<tr>
<td>\texttt{br}</td>
<td>Bottom-right corner</td>
<td>\texttt{ACS_LRCORNER}</td>
</tr>
</tbody>
</table>
box([vertch, horch])
   Similar to border(), but both ls and rs are vertch and both ts and bs are
   horch. The default corner characters are always used by this function.

clear()
   Like erase(), but also causes the whole window to be repainted upon
   next call to refresh().

clearok(yes)
   If yes is 1, the next call to refresh() will clear the window completely.

clrtobot()
   Erase from cursor to the end of the window: all lines below the cursor are
   deleted, and then the equivalent of clrtoeol() is performed.

clrtoeol()
   Erase from cursor to the end of the line.
cursyncup()
   Updates the current cursor position of all the ancestors of the window to
   reflect the current cursor position of the window.
delch([y, x])
   Delete any character at (y, x).
deleteln()
   Delete the line under the cursor. All following lines are moved up by 1 line.
derwin([nlines, ncols], begin_y, begin_x)
   An abbreviation for ``derive window'', derwin() is the same as calling
   subwin(), except that begin_y and begin_x are relative to the origin of
   the window, rather than relative to the entire screen. Returns a window
   object for the derived window.
echochar(ch[, attr])
Add character \textit{ch} with attribute \textit{attr}, and immediately call \texttt{refresh()} on the window.

\textbf{enclose}(y, x)
Tests whether the given pair of screen-relative character-cell coordinates are enclosed by the given window, returning true or false. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

\textbf{erase()}
Clear the window.

\textbf{getbegyx()}
Return a tuple \((y, x)\) of co-ordinates of upper-left corner.

\textbf{getch()} \([y, x]\)
Get a character. Note that the integer returned does \textit{not} have to be in ASCII range: function keys, keypad keys and so on return numbers higher than 256. In no-delay mode, -1 is returned if there is no input.

\textbf{getkey()} \([y, x]\)
Get a character, returning a string instead of an integer, as \texttt{getch()} does. Function keys, keypad keys and so on return a multibyte string containing the key name. In no-delay mode, an exception is raised if there is no input.

\textbf{getmaxyx()}
Return a tuple \((y, x)\) of the height and width of the window.

\textbf{getparyx()}
Returns the beginning coordinates of this window relative to its parent window into two integer variables \(y\) and \(x\). Returns \(-1, -1\) if this window has no parent.

\textbf{getstr()} \([y, x]\)
Read a string from the user, with primitive line editing capacity.

\textbf{getyx()}

Return a tuple \((y, x)\) of current cursor position relative to the window's upper-left corner.

**hline**([\(y, x\], \(ch, n\)])
Display a horizontal line starting at \((y, x)\) with length \(n\) consisting of the character \(ch\).

**idcok**(*flag*)
If \(flag\) is false, curses no longer considers using the hardware insert/delete character feature of the terminal; if \(flag\) is true, use of character insertion and deletion is enabled. When curses is first initialized, use of character insert/delete is enabled by default.

**idlok**(*yes*)
If called with \(yes\) equal to 1, curses will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

**immedok**(*flag*)
If \(flag\) is true, any change in the window image automatically causes the window to be refreshed; you no longer have to call \(\text{refresh()}\) yourself. However, it may degrade performance considerably, due to repeated calls to \(\text{wrefresh}\). This option is disabled by default.

**inch**([\(y, x\)])
Return the character at the given position in the window. The bottom 8 bits are the character proper, and upper bits are the attributes.

**insch**([\(y, x\], \(ch[, attr]\)])
Paint character \(ch\) at \((y, x)\) with attributes \(attr\), moving the line from position \(x\) right by one character.

**insdelln**(*nlines*)
Inserts \(nlines\) lines into the specified window above the current line. The \(nlines\) bottom lines are lost. For negative \(nlines\), delete \(nlines\) lines starting with the one under the cursor, and move the remaining lines up. The bottom \(nlines\) lines are cleared. The current cursor position remains the same.
**insertln()**
Insert a blank line under the cursor. All following lines are moved down by 1 line.

**insnstr([y, x], str, n [, attr])**
Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to n characters. If n is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to y, x, if specified).

**instr([y, x] [, n])**
Returns a string of characters, extracted from the window starting at the current cursor position, or at y, x if specified. Attributes are stripped from the characters. If n is specified, instr() returns return a string at most n characters long (exclusive of the trailing NUL).

**is_linetouched(line)**
Returns true if the specified line was modified since the last call to refresh(); otherwise returns false. Raises a curses.error exception if line is not valid for the given window.

**is_wintouched()**
Returns true if the specified window was modified since the last call to refresh(); otherwise returns false.

**keypad(yes)**
If yes is 1, escape sequences generated by some keys (keypad, function keys) will be interpreted by curses. If yes is 0, escape sequences will be left as is in the input stream.
**leaveok**(yes)
If yes is 1, cursor is left where it is on update, instead of being at `"cursor position."` This reduces cursor movement where possible. If possible the cursor will be made invisible.

If yes is 0, cursor will always be at `"cursor position"` after an update.

**move**(new_y, new_x)
Move cursor to (new_y, new_x).

**mvderwin**(y, x)
Moves the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

**mvwin**(new_y, new_x)
Move the window so its upper-left corner is at (new_y, new_x).

**nodelay**(yes)
If yes is 1, getch() will be non-blocking.

**notimeout**(yes)
If yes is 1, escape sequences will not be timed out.

If yes is 0, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

**noutrefresh**( )
Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call **doupdate**( ).

**overlay**( destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])
Overlay the window on top of destwin. The windows need not be the same size, only the overlapping region is copied. This copy is non-destructive, which means that the current background character does not overwrite the old contents of destwin.
To get fine-grained control over the copied region, the second form of `overlay()` can be used. `sminrow` and `smincol` are the upper-left coordinates of the source window, and the other variables mark a rectangle in the destination window.

```
overwrite(destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])
```

Overwrite the window on top of `destwin`. The windows need not be the same size, in which case only the overlapping region is copied. This copy is destructive, which means that the current background character overwrites the old contents of `destwin`.

To get fine-grained control over the copied region, the second form of `overwrite()` can be used. `sminrow` and `smincol` are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

```
putwin(file)
```

Writes all data associated with the window into the provided file object. This information can be later retrieved using the `getwin()` function.

```
redrawln(beg, num)
```

Indicates that the `num` screen lines, starting at line `beg`, are corrupted and should be completely redrawn on the next `refresh()` call.

```
redrawwin()
```

Touches the entire window, causing it to be completely redrawn on the next `refresh()` call.

```
refresh([pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol])
```

Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window is a pad created with `newpad()`. The additional parameters are needed to indicate what part of the pad and screen are involved. `pminrow` and `pmincol` specify the upper left-hand corner of the rectangle to be displayed in the pad.
sminrow, smincol, smaxrow, and smaxcol specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of pminrow, pmincol, sminrow, or smincol are treated as if they were zero.

\[ \text{scroll}([lines = 1]) \]  
Scroll the screen or scrolling region upward by lines lines.

\[ \text{scrollok}(\text{flag}) \]  
Controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If flag is false, the cursor is left on the bottom line. If flag is true, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call idlok().

\[ \text{setscrreg}(\text{top, bottom}) \]  
Set the scrolling region from line top to line bottom. All scrolling actions will take place in this region.

\[ \text{standend}() \]  
Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

\[ \text{standout}() \]  
Turn on attribute A_STANDOUT.

\[ \text{subpad}([\text{nlines, ncols,} \text{ begin}_y, \text{ begin}_x]) \]  
Return a sub-window, whose upper-left corner is at (begin_y, begin_x), and whose width/height is ncols/nlines.

\[ \text{subwin}([\text{nlines, ncols,} \text{ begin}_y, \text{ begin}_x]) \]  
Return a sub-window, whose upper-left corner is at (begin_y, begin_x), and whose width/height is ncols/nlines.
By default, the sub-window will extend from the specified position to the lower right corner of the window.

**syncdown()**

Touches each location in the window that has been touched in any of its ancestor windows. This routine is called by `refresh()`, so it should almost never be necessary to call it manually.

**syncok(flag)**

If called with `flag` set to true, then `syncup()` is called automatically whenever there is a change in the window.

**syncup()**

Touches all locations in ancestors of the window that have been changed in the window.

**timeout(delay)**

Sets blocking or non-blocking read behavior for the window. If `delay` is negative, blocking read is used (which will wait indefinitely for input). If `delay` is zero, then non-blocking read is used, and -1 will be returned by `getch()` if no input is waiting. If `delay` is positive, then `getch()` will block for `delay` milliseconds, and return -1 if there is still no input at the end of that time.

**touchline(start, count)**

Pretend `count` lines have been changed, starting with line `start`.

**touchwin()**

Pretend the whole window has been changed, for purposes of drawing optimizations.

**untouchwin()**

Marks all lines in the window as unchanged since the last call to `refresh()`.

**vline([y, x], ch, n)**

Display a vertical line starting at `(y, x)` with length `n` consisting of the
character \textit{ch}.
14.7.3 Constants

The `curses` module defines the following data members:

**ERR**
Some `curses` routines that return an integer, such as `getch()`, return ERR upon failure.

**OK**
Some `curses` routines that return an integer, such as `napms()`, return OK upon success.

**version**
A string representing the current version of the module. Also available as `__version__`.

Several constants are available to specify character cell attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_ALTCHARSET</td>
<td>Alternate character set mode.</td>
</tr>
<tr>
<td>A_BLINK</td>
<td>Blink mode.</td>
</tr>
<tr>
<td>A_BOLD</td>
<td>Bold mode.</td>
</tr>
<tr>
<td>A_DIM</td>
<td>Dim mode.</td>
</tr>
<tr>
<td>A_NORMAL</td>
<td>Normal attribute.</td>
</tr>
<tr>
<td>A_STANDOUT</td>
<td>Standout mode.</td>
</tr>
<tr>
<td>A_UNDERLINE</td>
<td>Underline mode.</td>
</tr>
</tbody>
</table>

Keys are referred to by integer constants with names starting with "KEY_". The exact keycaps available are system dependent.

<table>
<thead>
<tr>
<th>Key constant</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_MIN</td>
<td>Minimum key value</td>
</tr>
<tr>
<td>KEY_BREAK</td>
<td>Break key (unreliable)</td>
</tr>
<tr>
<td>KEY_DOWN</td>
<td>Down-arrow</td>
</tr>
<tr>
<td>KEY_UP</td>
<td>Up-arrow</td>
</tr>
<tr>
<td>KEY_LEFT</td>
<td>Left-arrow</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>KEY_RIGHT</td>
<td>Right-arrow</td>
</tr>
<tr>
<td>KEY_HOME</td>
<td>Home key (upward+left arrow)</td>
</tr>
<tr>
<td>KEY_BACKSPACE</td>
<td>Backspace (unreliable)</td>
</tr>
<tr>
<td>KEY_F0</td>
<td>Function keys. Up to 64 function keys are supported.</td>
</tr>
<tr>
<td>KEY_Fn</td>
<td>Value of function key n</td>
</tr>
<tr>
<td>KEY_DL</td>
<td>Delete line</td>
</tr>
<tr>
<td>KEY_IL</td>
<td>Insert line</td>
</tr>
<tr>
<td>KEY_DC</td>
<td>Delete character</td>
</tr>
<tr>
<td>KEY_IC</td>
<td>Insert char or enter insert mode</td>
</tr>
<tr>
<td>KEY_EIC</td>
<td>Exit insert char mode</td>
</tr>
<tr>
<td>KEY_CLEAR</td>
<td>Clear screen</td>
</tr>
<tr>
<td>KEY_EOS</td>
<td>Clear to end of screen</td>
</tr>
<tr>
<td>KEY_EOL</td>
<td>Clear to end of line</td>
</tr>
<tr>
<td>KEY_SF</td>
<td>Scroll 1 line forward</td>
</tr>
<tr>
<td>KEY_SR</td>
<td>Scroll 1 line backward (reverse)</td>
</tr>
<tr>
<td>KEY_NPAGE</td>
<td>Next page</td>
</tr>
<tr>
<td>KEY_PPAGE</td>
<td>Previous page</td>
</tr>
<tr>
<td>KEY_STAB</td>
<td>Set tab</td>
</tr>
<tr>
<td>KEY_CTAB</td>
<td>Clear tab</td>
</tr>
<tr>
<td>KEY_CATAB</td>
<td>Clear all tabs</td>
</tr>
<tr>
<td>KEY_ENTER</td>
<td>Enter or send (unreliable)</td>
</tr>
<tr>
<td>KEY_SRESET</td>
<td>Soft (partial) reset (unreliable)</td>
</tr>
<tr>
<td>KEY_RESET</td>
<td>Reset or hard reset (unreliable)</td>
</tr>
<tr>
<td>KEY_PRINT</td>
<td>Print</td>
</tr>
<tr>
<td>KEY_LL</td>
<td>Home down or bottom (lower left)</td>
</tr>
<tr>
<td>KEY_A1</td>
<td>Upper left of keypad</td>
</tr>
<tr>
<td>KEY_A3</td>
<td>Upper right of keypad</td>
</tr>
<tr>
<td>KEY_B2</td>
<td>Center of keypad</td>
</tr>
<tr>
<td>KEY_C1</td>
<td>Lower left of keypad</td>
</tr>
<tr>
<td>KEY_C3</td>
<td>Lower right of keypad</td>
</tr>
<tr>
<td>KEY_BTAB</td>
<td>Back tab</td>
</tr>
<tr>
<td>KEY_BEG</td>
<td>Beg (beginning)</td>
</tr>
<tr>
<td>Key code</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>KEY_CANCEL</td>
<td>Cancel</td>
</tr>
<tr>
<td>KEY_CLOSE</td>
<td>Close</td>
</tr>
<tr>
<td>KEY_COMMAND</td>
<td>Cmd (command)</td>
</tr>
<tr>
<td>KEY_COPY</td>
<td>Copy</td>
</tr>
<tr>
<td>KEY_CREATE</td>
<td>Create</td>
</tr>
<tr>
<td>KEY_END</td>
<td>End</td>
</tr>
<tr>
<td>KEY_EXIT</td>
<td>Exit</td>
</tr>
<tr>
<td>KEY_FIND</td>
<td>Find</td>
</tr>
<tr>
<td>KEY_HELP</td>
<td>Help</td>
</tr>
<tr>
<td>KEY_MARK</td>
<td>Mark</td>
</tr>
<tr>
<td>KEY_MESSAGE</td>
<td>Message</td>
</tr>
<tr>
<td>KEY_MOVE</td>
<td>Move</td>
</tr>
<tr>
<td>KEY_NEXT</td>
<td>Next</td>
</tr>
<tr>
<td>KEY_OPEN</td>
<td>Open</td>
</tr>
<tr>
<td>KEY_OPTIONS</td>
<td>Options</td>
</tr>
<tr>
<td>KEY_PREVIOUS</td>
<td>Prev (previous)</td>
</tr>
<tr>
<td>KEY_REDO</td>
<td>Redo</td>
</tr>
<tr>
<td>KEY_REFERENCE</td>
<td>Ref (reference)</td>
</tr>
<tr>
<td>KEY_REFRESH</td>
<td>Refresh</td>
</tr>
<tr>
<td>KEY_REPLACE</td>
<td>Replace</td>
</tr>
<tr>
<td>KEY_RESTART</td>
<td>Restart</td>
</tr>
<tr>
<td>KEY_RESUME</td>
<td>Resume</td>
</tr>
<tr>
<td>KEY_SAVE</td>
<td>Save</td>
</tr>
<tr>
<td>KEY_SBEG</td>
<td>Shifted Beg (beginning)</td>
</tr>
<tr>
<td>KEY_SCANCEL</td>
<td>Shifted Cancel</td>
</tr>
<tr>
<td>KEY_SCOMMAND</td>
<td>Shifted Command</td>
</tr>
<tr>
<td>KEY_SCOPY</td>
<td>Shifted Copy</td>
</tr>
<tr>
<td>KEY_SCREATE</td>
<td>Shifted Create</td>
</tr>
<tr>
<td>KEY_SDC</td>
<td>Shifted Delete char</td>
</tr>
<tr>
<td>KEY(SDL)</td>
<td>Shifted Delete line</td>
</tr>
<tr>
<td>KEY_SELECT</td>
<td>Select</td>
</tr>
<tr>
<td>KEY_SEND</td>
<td>Shifted End</td>
</tr>
<tr>
<td>KEY_SEOL</td>
<td>Shifted Clear line</td>
</tr>
<tr>
<td>Key Code</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>KEY_SEXIT</td>
<td>Shifted Dxit</td>
</tr>
<tr>
<td>KEY_SFIND</td>
<td>Shifted Find</td>
</tr>
<tr>
<td>KEY_SHELP</td>
<td>Shifted Help</td>
</tr>
<tr>
<td>KEY_SHOME</td>
<td>Shifted Home</td>
</tr>
<tr>
<td>KEY_SIC</td>
<td>Shifted Input</td>
</tr>
<tr>
<td>KEY_SLEFT</td>
<td>Shifted Left arrow</td>
</tr>
<tr>
<td>KEY_SMESSAGE</td>
<td>Shifted Message</td>
</tr>
<tr>
<td>KEY_SMOVE</td>
<td>Shifted Move</td>
</tr>
<tr>
<td>KEY_SNEXT</td>
<td>Shifted Next</td>
</tr>
<tr>
<td>KEY_SOPTIONS</td>
<td>Shifted Options</td>
</tr>
<tr>
<td>KEY_SPREVIOUS</td>
<td>Shifted Prev</td>
</tr>
<tr>
<td>KEY_SPRINT</td>
<td>Shifted Print</td>
</tr>
<tr>
<td>KEY_SREDO</td>
<td>Shifted Redo</td>
</tr>
<tr>
<td>KEY_SREPLACE</td>
<td>Shifted Replace</td>
</tr>
<tr>
<td>KEY_SRIGHT</td>
<td>Shifted Right arrow</td>
</tr>
<tr>
<td>KEY_SRSUME</td>
<td>Shifted Resume</td>
</tr>
<tr>
<td>KEY_SSAVE</td>
<td>Shifted Save</td>
</tr>
<tr>
<td>KEY_SSUSPEND</td>
<td>Shifted Suspend</td>
</tr>
<tr>
<td>KEY_SUNDO</td>
<td>Shifted Undo</td>
</tr>
<tr>
<td>KEY_SUSPEND</td>
<td>Suspend</td>
</tr>
<tr>
<td>KEY_UNDO</td>
<td>undo</td>
</tr>
<tr>
<td>KEY_MOUSE</td>
<td>Mouse event has occurred</td>
</tr>
<tr>
<td>KEY_RESIZE</td>
<td>Terminal resize event</td>
</tr>
<tr>
<td>KEY_MAX</td>
<td>Maximum key value</td>
</tr>
</tbody>
</table>

On VT100s and their software emulations, such as X terminal emulators, there are normally at least four function keys (KEY_F1, KEY_F2, KEY_F3, KEY_F4) available, and the arrow keys mapped to KEY_UP, KEY_DOWN, KEY_LEFT and KEY_RIGHT in the obvious way. If your machine has a PC keyboard, it is safe to expect arrow keys and twelve function keys (older PC keyboards may have only ten function keys); also, the following keypad mappings are standard:
The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation. **Note:** These are available only after `initscr()` has been called.

<table>
<thead>
<tr>
<th>ACS code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACS_BBSS</td>
<td>alternate name for upper right corner</td>
</tr>
<tr>
<td>ACS_BLOCK</td>
<td>solid square block</td>
</tr>
<tr>
<td>ACS_BOARD</td>
<td>board of squares</td>
</tr>
<tr>
<td>ACS_BSBS</td>
<td>alternate name for horizontal line</td>
</tr>
<tr>
<td>ACS_BSSB</td>
<td>alternate name for upper left corner</td>
</tr>
<tr>
<td>ACS_BSSS</td>
<td>alternate name for top tee</td>
</tr>
<tr>
<td>ACS_BTEE</td>
<td>bottom tee</td>
</tr>
<tr>
<td>ACS_BULLET</td>
<td>bullet</td>
</tr>
<tr>
<td>ACS_CKBOARD</td>
<td>checker board (stipple)</td>
</tr>
<tr>
<td>ACS_DARROW</td>
<td>arrow pointing down</td>
</tr>
<tr>
<td>ACS_DEGREE</td>
<td>degree symbol</td>
</tr>
<tr>
<td>ACS_DIAMOND</td>
<td>diamond</td>
</tr>
<tr>
<td>ACS_GEQUAL</td>
<td>greater-than-or-equal-to</td>
</tr>
<tr>
<td>ACS_HLINE</td>
<td>horizontal line</td>
</tr>
<tr>
<td>ACS_LANTERN</td>
<td>lantern symbol</td>
</tr>
<tr>
<td>ACS_LARROW</td>
<td>left arrow</td>
</tr>
<tr>
<td>ACS_LEQUAL</td>
<td>less-than-or-equal-to</td>
</tr>
<tr>
<td>ACS_LLCORNER</td>
<td>lower left-hand corner</td>
</tr>
<tr>
<td>ACS_LRCORNER</td>
<td>lower right-hand corner</td>
</tr>
<tr>
<td>ACS_LTEE</td>
<td>left tee</td>
</tr>
<tr>
<td>Constant</td>
<td>Color</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>ACS_NEQUAL</td>
<td>not-equal sign</td>
</tr>
<tr>
<td>ACS_PI</td>
<td>letter pi</td>
</tr>
<tr>
<td>ACS_PLMINUS</td>
<td>plus-or-minus sign</td>
</tr>
<tr>
<td>ACS_PLUS</td>
<td>big plus sign</td>
</tr>
<tr>
<td>ACS_RARROW</td>
<td>right arrow</td>
</tr>
<tr>
<td>ACS_RTEE</td>
<td>right tee</td>
</tr>
<tr>
<td>ACS_S1</td>
<td>scan line 1</td>
</tr>
<tr>
<td>ACS_S3</td>
<td>scan line 3</td>
</tr>
<tr>
<td>ACS_S7</td>
<td>scan line 7</td>
</tr>
<tr>
<td>ACS_S9</td>
<td>scan line 9</td>
</tr>
<tr>
<td>ACS_SBBS</td>
<td>alternate name for lower right corner</td>
</tr>
<tr>
<td>ACS_SBSB</td>
<td>alternate name for vertical line</td>
</tr>
<tr>
<td>ACS_SBSS</td>
<td>alternate name for right tee</td>
</tr>
<tr>
<td>ACS_SSBB</td>
<td>alternate name for lower left corner</td>
</tr>
<tr>
<td>ACS_SSBS</td>
<td>alternate name for bottom tee</td>
</tr>
<tr>
<td>ACS_SSSB</td>
<td>alternate name for left tee</td>
</tr>
<tr>
<td>ACS_SSSS</td>
<td>alternate name for crossover or big plus</td>
</tr>
<tr>
<td>ACS_STERLING</td>
<td>pound sterling</td>
</tr>
<tr>
<td>ACS_TTEE</td>
<td>top tee</td>
</tr>
<tr>
<td>ACS_UARROW</td>
<td>up arrow</td>
</tr>
<tr>
<td>ACS_ULCORNER</td>
<td>upper left corner</td>
</tr>
<tr>
<td>ACS_URCORNER</td>
<td>upper right corner</td>
</tr>
<tr>
<td>ACS_VLINE</td>
<td>vertical line</td>
</tr>
</tbody>
</table>

The following table lists the predefined colors:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR_BLACK</td>
<td>Black</td>
</tr>
<tr>
<td>COLOR_BLUE</td>
<td>Blue</td>
</tr>
<tr>
<td>COLOR_CYAN</td>
<td>Cyan (light greenish blue)</td>
</tr>
<tr>
<td>COLOR_GREEN</td>
<td>Green</td>
</tr>
<tr>
<td>COLOR_MAGENTA</td>
<td>Magenta (purplish red)</td>
</tr>
<tr>
<td>COLOR_RED</td>
<td>Red</td>
</tr>
<tr>
<td>COLOR_WHITE</td>
<td>White</td>
</tr>
</tbody>
</table>
14.8 curses.textpad -- Text input widget for curses programs

New in version 1.6.

The curses.textpad module provides a Textbox class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BBedit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

The module curses.textpad defines the following function:

**rectangle** *(win, uly, ulx, lry, lrx)*

Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.

---

**Subsections**

- 14.8.1 Textbox objects

---

Python Library Reference


Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.8.1 Textbox objects

You can instantiate a Textbox object as follows:

```python
class Textbox( win)
    Return a textbox widget object. The win argument should be a curses WindowObject in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates (0, 0). The instance's stripspaces flag is initially on.
```

Textbox objects have the following methods:

```python
edit([validator])
    This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If validator is supplied, it must be a function. It will be called for each keystroke entered with the keystroke as a parameter; command dispatch is done on the result. This method returns the window contents as a string; whether blanks in the window are included is affected by the stripspaces member.
```

```python
do_command( ch)
    Process a single command keystroke. Here are the supported special keystrokes:
```

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control-A</td>
<td>Go to left edge of window.</td>
</tr>
<tr>
<td>Control-B</td>
<td>Cursor left, wrapping to previous line if appropriate.</td>
</tr>
<tr>
<td>Control-D</td>
<td>Delete character under cursor.</td>
</tr>
<tr>
<td>Control-E</td>
<td>Go to right edge (stripspaces off) or end of line (stripspaces on).</td>
</tr>
<tr>
<td>Control-F</td>
<td>Cursor right, wrapping to next line when appropriate.</td>
</tr>
</tbody>
</table>
Control-G | Terminate, returning the window contents.

Control-H | Delete character backward.

Control-J | Terminate if the window is 1 line, otherwise insert newline.

Control-K | If line is blank, delete it, otherwise clear to end of line.

Control-L | Refresh screen.

Control-N | Cursor down; move down one line.

Control-O | Insert a blank line at cursor location.

Control-P | Cursor up; move up one line.

Move operations do nothing if the cursor is at an edge where the movement is not possible. The following synonyms are supported where possible:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Keystroke</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_LEFT</td>
<td>Control-B</td>
</tr>
<tr>
<td>KEY_RIGHT</td>
<td>Control-F</td>
</tr>
<tr>
<td>KEY_UP</td>
<td>Control-P</td>
</tr>
<tr>
<td>KEY_DOWN</td>
<td>Control-N</td>
</tr>
<tr>
<td>KEY_BACKSPACE</td>
<td>Control-h</td>
</tr>
</tbody>
</table>

All other keystrokes are treated as a command to insert the given character and move right (with line wrapping).

**gather()**
This method returns the window contents as a string; whether blanks in the window are included is affected by the `stripspaces` member.

**stripspaces**
This data member is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks on each line are ignored; any cursor
motion that would land the cursor on a trailing blank goes to the end of that line instead, and trailing blanks are stripped when the window contents are gathered.
14.9 curses.wrapper -- Terminal handler for curses programs

New in version 1.6.

This module supplies one function, \texttt{wrapper()}, which runs another function which should be the rest of your curses-using application. If the application raises an exception, \texttt{wrapper()} will restore the terminal to a sane state before re-raising the exception and generating a traceback.

\texttt{wrapper(func, ...)}

Wrapper function that initializes curses and calls another function, \texttt{func}, restoring normal keyboard/screen behavior on error. The callable object \texttt{func} is then passed the main window 'stdscr' as its first argument, followed by any other arguments passed to \texttt{wrapper()}.

Before calling the hook function, \texttt{wrapper()} turns on cbreak mode, turns off echo, enables the terminal keypad, and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on echo, and disables the terminal keypad.
14.10 curses.ascii -- Utilities for ASCII characters

New in version 1.6.

The `curses.ascii` module supplies name constants for ASCII characters and functions to test membership in various ASCII character classes. The constants supplied are names for control characters as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>Start of heading, console interrupt</td>
</tr>
<tr>
<td>SOH</td>
<td>Start of text</td>
</tr>
<tr>
<td>STX</td>
<td>End of text</td>
</tr>
<tr>
<td>ETX</td>
<td>End of transmission</td>
</tr>
<tr>
<td>EOT</td>
<td>Enquiry, goes with ACK flow control</td>
</tr>
<tr>
<td>ENQ</td>
<td>Bell</td>
</tr>
<tr>
<td>BS</td>
<td>Backspace</td>
</tr>
<tr>
<td>HT</td>
<td>Alias for TAB: <code>Horizontal tab</code></td>
</tr>
<tr>
<td>LF</td>
<td>Line feed</td>
</tr>
<tr>
<td>NL</td>
<td>Alias for LF: `New line&quot;</td>
</tr>
<tr>
<td>VT</td>
<td>Vertical tab</td>
</tr>
<tr>
<td>FF</td>
<td>Form feed</td>
</tr>
<tr>
<td>CR</td>
<td>Carriage return</td>
</tr>
<tr>
<td>SO</td>
<td>Shift-out, begin alternate character set</td>
</tr>
<tr>
<td>SI</td>
<td>Shift-in, resume default character set</td>
</tr>
<tr>
<td>DLE</td>
<td>Data-link escape</td>
</tr>
<tr>
<td>DC1</td>
<td>XON, for flow control</td>
</tr>
<tr>
<td>DC2</td>
<td>Device control 2, block-mode flow control</td>
</tr>
<tr>
<td>DC3</td>
<td>XOFF, for flow control</td>
</tr>
<tr>
<td>DC4</td>
<td>Device control 4</td>
</tr>
<tr>
<td>-------</td>
<td>------------------</td>
</tr>
<tr>
<td>NAK</td>
<td>Negative acknowledgement</td>
</tr>
<tr>
<td>SYN</td>
<td>Synchronous idle</td>
</tr>
<tr>
<td>ETB</td>
<td>End transmission block</td>
</tr>
<tr>
<td>CAN</td>
<td>Cancel</td>
</tr>
<tr>
<td>EM</td>
<td>End of medium</td>
</tr>
<tr>
<td>SUB</td>
<td>Substitute</td>
</tr>
<tr>
<td>ESC</td>
<td>Escape</td>
</tr>
<tr>
<td>FS</td>
<td>File separator</td>
</tr>
<tr>
<td>GS</td>
<td>Group separator</td>
</tr>
<tr>
<td>RS</td>
<td>Record separator, block-mode terminator</td>
</tr>
<tr>
<td>US</td>
<td>Unit separator</td>
</tr>
<tr>
<td>SP</td>
<td>Space</td>
</tr>
<tr>
<td>DEL</td>
<td>Delete</td>
</tr>
</tbody>
</table>

Note that many of these have little practical significance in modern usage. The mnemonics derive from teleprinter conventions that predate digital computers.

The module supplies the following functions, patterned on those in the standard C library:

**isalnum**(c)
Checks for an ASCII alphanumeric character; it is equivalent to 
"isalpha(c) or isdigit(c)".

**isalpha**(c)
Checks for an ASCII alphabetic character; it is equivalent to 
"isupper(c) or islower(c)".

**isascii**(c)
Checks for a character value that fits in the 7-bit ASCII set.

**isblank**(c)
Checks for an ASCII whitespace character.

**iscntrl**(c)
Checks for an ASCII control character (in the range 0x00 to 0x1f).

**isdigit(c)**
Checks for an ASCII decimal digit, "0" through "9". This is equivalent to "c in string.digits".

**isgraph(c)**
Checks for ASCII any printable character except space.

**islower(c)**
Checks for an ASCII lower-case character.

**isprint(c)**
Checks for any ASCII printable character including space.

**ispunct(c)**
Checks for any printable ASCII character which is not a space or an alphanumeric character.

**isspace(c)**
Checks for ASCII white-space characters; space, line feed, carriage return, form feed, horizontal tab, vertical tab.

**isupper(c)**
Checks for an ASCII uppercase letter.

**isxdigit(c)**
Checks for an ASCII hexadecimal digit. This is equivalent to "c in string.hexdigits".

**isctrl(c)**
Checks for an ASCII control character (ordinal values 0 to 31).

**ismeta(c)**
Checks for a non-ASCII character (ordinal values 0x80 and above).

These functions accept either integers or strings; when the argument is a string, it
is first converted using the built-in function `ord()`.

Note that all these functions check ordinal bit values derived from the first character of the string you pass in; they do not actually know anything about the host machine's character encoding. For functions that know about the character encoding (and handle internationalization properly) see the `string` module.

The following two functions take either a single-character string or integer byte value; they return a value of the same type.

\[ \text{ascii}(c) \]
Return the ASCII value corresponding to the low 7 bits of \( c \).

\[ \text{ctrl}(c) \]
Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

\[ \text{alt}(c) \]
Return the 8-bit character corresponding to the given ASCII character (the character bit value is bitwise-ored with 0x80).

The following function takes either a single-character string or integer value; it returns a string.

\[ \text{unctrl}(c) \]
Return a string representation of the ASCII character \( c \). If \( c \) is printable, this string is the character itself. If the character is a control character (0x00-0x1f) the string consists of a caret ("^") followed by the corresponding uppercase letter. If the character is an ASCII delete (0x7f) the string is '^?'. If the character has its meta bit (0x80) set, the meta bit is stripped, the preceding rules applied, and "!" prepended to the result.

\[ \text{controlnames} \]
A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from 0 (NUL) to 0x1f (US), in order, plus the mnemonic "SP" for the space character.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.11 curses.panel -- A panel stack extension for curses.

Panels are windows with the added feature of depth, so they can be stacked on top of each other, and only the visible portions of each window will be displayed. Panels can be added, moved up or down in the stack, and removed.

Subsections

- 14.11.1 Functions
- 14.11.2 Panel Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.11.1 Functions

The module `curses.panel` defines the following functions:

**bottom_panel()**
Returns the bottom panel in the panel stack.

**new_panel( win)**
Returns a panel object, associating it with the given window `win`. Be aware that you need to keep the returned panel object referenced explicitly. If you don't, the panel object is garbage collected and removed from the panel stack.

**top_panel()**
Returns the top panel in the panel stack.

**update_panels()**
Updates the virtual screen after changes in the panel stack. This does not call `curses.doupdate()`, so you'll have to do this yourself.
### 14.11.2 Panel Objects

Panel objects, as returned by `new_panel()` above, are windows with a stacking order. There's always a window associated with a panel which determines the content, while the panel methods are responsible for the window's depth in the panel stack.

Panel objects have the following methods:

- **above()**
  
  Returns the panel above the current panel.

- **below()**
  
  Returns the panel below the current panel.

- **bottom()**
  
  Push the panel to the bottom of the stack.

- **hidden()**
  
  Returns true if the panel is hidden (not visible), false otherwise.

- **hide()**
  
  Hide the panel. This does not delete the object, it just makes the window on screen invisible.

- **move(y, x)**
  
  Move the panel to the screen coordinates \((y, x)\).

- **replace(win)**
  
  Change the window associated with the panel to the window `win`.

- **set_userptr(obj)**
  
  Set the panel's user pointer to `obj`. This is used to associate an arbitrary piece of data with the panel, and can be any Python object.

- **show()**
Display the panel (which might have been hidden).

**top()**
Push panel to the top of the stack.

**userptr()**
Returns the user pointer for the panel. This might be any Python object.

**window()**
Returns the window object associated with the panel.
14.12 platform -- Access to underlying platform's identifying data.

New in version 2.3.

Note: Specific platforms listed alphabetically, with Linux included in the UNIX section.

Subsections

- 14.12.1 Cross Platform
- 14.12.2 Java Platform
- 14.12.3 Windows Platform
  - 14.12.3.1 Win95/98 specific
- 14.12.4 Mac OS Platform
- 14.12.5 UNIX Platforms

Python Library Reference


Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
14.12.1 Cross Platform

architecture(\texttt{executable}=\texttt{sys.executable}, \texttt{bits}='', \texttt{linkage}='')

Queries the given executable (defaults to the Python interpreter binary) for various architecture information.

Returns a tuple \((\texttt{bits}, \texttt{linkage})\) which contain information about the bit architecture and the linkage format used for the executable. Both values are returned as strings.

Values that cannot be determined are returned as given by the parameter presets. If \texttt{bits} is given as '', the \texttt{sizeof(pointer)} (or \texttt{sizeof(long)} on Python version $< 1.5.2$) is used as indicator for the supported pointer size.

The function relies on the system's file command to do the actual work. This is available on most if not all UNIX platforms and some non-UNIX platforms and then only if the executable points to the Python interpreter. Reasonable defaults are used when the above needs are not met.

machine()

Returns the machine type, e.g. 'i386'. An empty string is returned if the value cannot be determined.

node()

Returns the computer's network name (may not be fully qualified!). An empty string is returned if the value cannot be determined.

platform(\texttt{aliased}=0, \texttt{terse}=0)

Returns a single string identifying the underlying platform with as much useful information as possible.

The output is intended to be \textit{human readable} rather than machine parseable. It may look different on different platforms and this is intended.

If \texttt{aliased} is true, the function will use aliases for various platforms that report system names which differ from their common names, for example
SunOS will be reported as Solaris. The system_alias() function is used to implement this.

Setting terse to true causes the function to return only the absolute minimum information needed to identify the platform.

**processor()**

Returns the (real) processor name, e.g. 'amdk6'.

An empty string is returned if the value cannot be determined. Note that many platforms do not provide this information or simply return the same value as for machine(). NetBSD does this.

**python_build()**

Returns a tuple (buildno, builddate) stating the Python build number and date as strings.

**python_compiler()**

Returns a string identifying the compiler used for compiling Python.

**python_version()**

Returns the Python version as string 'major.minor.patchlevel'

Note that unlike the Python sys.version, the returned value will always include the patchlevel (it defaults to 0).

**python_version_tuple()**

Returns the Python version as tuple (major, minor, patchlevel) of strings.

Note that unlike the Python sys.version, the returned value will always include the patchlevel (it defaults to '0').

**release()**

Returns the system's release, e.g. '2.2.0' or 'NT' An empty string is returned if the value cannot be determined.

**system()**
Returns the system/OS name, e.g. 'Linux', 'Windows', or 'Java'. An empty string is returned if the value cannot be determined.

**system_alias**(*system, release, version*)

Returns (*system, release, version*) aliased to common marketing names used for some systems. It also does some reordering of the information in some cases where it would otherwise cause confusion.

**version()**

Returns the system's release version, e.g. '#3 on degas'. An empty string is returned if the value cannot be determined.

**uname()**

Fairly portable uname interface. Returns a tuple of strings (*system, node, release, version, machine, processor*) identifying the underlying platform.

Note that unlike the os.uname() function this also returns possible processor information as additional tuple entry.

Entries which cannot be determined are set to ' '.
14.12.2 Java Platform

`java_ver( release='', vendor='', vminfo=('', '', ''), osinfo=('', '', ''))`

Version interface for JPython.

Returns a tuple `(release, vendor, vminfo, osinfo)` with `vminfo` being a tuple `(vm_name, vm_release, vm_vendor)` and `osinfo` being a tuple `(os_name, os_version, os_arch)`. Values which cannot be determined are set to the defaults given as parameters (which all default to ' ').
14.12.3 Windows Platform

`win32_ver(release='', version='', csd='', ptype='')`

Get additional version information from the Windows Registry and return a tuple (`version`, `csd`, `ptype`) referring to version number, CSD level and OS type (multi/single processor).

As a hint: `ptype` is 'Uniprocessor Free' on single processor NT machines and 'Multiprocessor Free' on multi processor machines. The 'Free' refers to the OS version being free of debugging code. It could also state 'Checked' which means the OS version uses debugging code, i.e. code that checks arguments, ranges, etc.

**Note:** This function only works if Mark Hammond's `win32all` package is installed and (obviously) only runs on Win32 compatible platforms.
14.12.4 Mac OS Platform

`mac_ver( release='', versioninfo=('', '', ''), machine='')`

Get Mac OS version information and return it as tuple (`release`, `versioninfo`, `machine`) with `versioninfo` being a tuple (`version`, `dev_stage`, `non_release_version`).

Entries which cannot be determined are set to ''. All tuple entries are strings.

Documentation for the underlying `gestalt()` API is available online at [http://www.rgaros.nl/gestalt/](http://www.rgaros.nl/gestalt/).
14.12.5 UNIX Platforms

\texttt{dist}(\texttt{distname='', version='', id='', supported_dists=}
\texttt{('SuSE','debian','redhat','mandrake'))}

Tries to determine the name of the OS distribution name
Returns a tuple \texttt{(distname, version, id)} which defaults to the args given as parameters.

\texttt{libc_ver}(\texttt{executable=sys.executable, lib='', version='', chunksize=2048})

Tries to determine the libc version against which the file executable
(defaults to the Python interpreter) is linked. Returns a tuple of strings
\texttt{(lib, version)} which default to the given parameters in case the lookup fails.

Note that this function has intimate knowledge of how different libc
versions add symbols to the executable is probably only useable for
executables compiled using \texttt{gcc}.

The file is read and scanned in chunks of \texttt{chunksize} bytes.
14.13 errno -- Standard errno system symbols

This module makes available standard errno system symbols. The value of each symbol is the corresponding integer value. The names and descriptions are borrowed from linux/include/errno.h, which should be pretty all-inclusive.

**errorcode**

Dictionary providing a mapping from the errno value to the string name in the underlying system. For instance, `errno.errorcode[errno.EPERM]` maps to 'EPERM'.

To translate a numeric error code to an error message, use `os.strerror()`.

Of the following list, symbols that are not used on the current platform are not defined by the module. The specific list of defined symbols is available as `errno.errorcode.keys()`. Symbols available can include:

**EPERM**

Operation not permitted

**ENOENT**

No such file or directory

**ESRCH**

No such process

**EINTR**

Interrupted system call

**EIO**

I/O error

**ENXIO**

No such device or address

**E2BIG**
Arg list too long

**ENOEXEC**
Exec format error

**EBADF**
Bad file number

**ECHILD**
No child processes

**EAGAIN**
Try again

**ENOMEM**
Out of memory

**EACCEES**
Permission denied

**EFAULT**
Bad address

**ENOTBLK**
Block device required

**EBUSY**
Device or resource busy

**EXISTS**
File exists

**EXDEV**
Cross-device link

**ENODEV**
No such device

**ENOTDIR**
Not a directory
**EISDIR**
  Is a directory

**EINVAL**
  Invalid argument

**ENFILE**
  File table overflow

**EMFILE**
  Too many open files

**ENOTTY**
  Not a typewriter

**ETXTBSY**
  Text file busy

**EFBIG**
  File too large

**ENOSPC**
  No space left on device

**ESPIPE**
  Illegal seek

**EROFS**
  Read-only file system

**EMLINK**
  Too many links

**EPIPE**
  Broken pipe

**EDOM**
  Math argument out of domain of func

**ERANGE**
Math result not representable

**EDEADLK**
Resource deadlock would occur

**ENAMETOOLONG**
File name too long

**ENOLCK**
No record locks available

**ENOSYS**
Function not implemented

**ENOTEMPTY**
Directory not empty

**ELOOP**
Too many symbolic links encountered

**EWOULDBLOCK**
Operation would block

**ENOMSG**
No message of desired type

**EIDRM**
Identifier removed

**ECHRNG**
Channel number out of range

**EL2NSYNC**
Level 2 not synchronized

**EL3HLT**
Level 3 halted

**EL3RST**
Level 3 reset
ELNRNG
  Link number out of range

EUNATCH
  Protocol driver not attached

ENOCSI
  No CSI structure available

EL2HLT
  Level 2 halted

EBADE
  Invalid exchange

EBADR
  Invalid request descriptor

EXFULL
  Exchange full

ENOANO
  No anode

EBADRQC
  Invalid request code

EBADSLT
  Invalid slot

EDEADLOCK
  File locking deadlock error

EBFONT
  Bad font file format

ENOSTR
  Device not a stream

ENODATA
No data available

**ETIME**
Timer expired

**ENOSR**
Out of streams resources

**ENONET**
Machine is not on the network

**ENOPKG**
Package not installed

**EREMOTE**
Object is remote

**ENOLINK**
Link has been severed

**EADV**
Advertise error

**ESRMNT**
Srmount error

**ECOMM**
Communication error on send

**EPROTO**
Protocol error

**EMULTIHOP**
Multihop attempted

**EDOTDOT**
RFS specific error

**EBADMSG**
Not a data message
**EOVERFLOW**
Value too large for defined data type

**ENOTUNIQ**
Name not unique on network

**EBADFD**
File descriptor in bad state

**EREMCHG**
Remote address changed

**ELIBACC**
Can not access a needed shared library

**ELIBBAD**
Accessing a corrupted shared library

**ELIBSCN**
.lib section in a.out corrupted

**ELIBMAX**
Attempting to link in too many shared libraries

**ELIBEXEC**
Cannot exec a shared library directly

**EILSEQ**
Illegal byte sequence

**ERESTART**
Interrupted system call should be restarted

**ESTRPIPE**
Streams pipe error

**EUSERS**
Too many users

**ENOTSOCK**
Socket operation on non-socket

**EDESTADDRREQ**
Destination address required

**EMSGSIZE**
Message too long

**EPROTOTYPE**
Protocol wrong type for socket

**ENOPROTOOPT**
Protocol not available

**EPROTONOSUPPORT**
Protocol not supported

**ESOCKTNOSUPPORT**
Socket type not supported

**EOPNOTSUPP**
Operation not supported on transport endpoint

**EPFNOSUPPORT**
Protocol family not supported

**EAFNOSUPPORT**
Address family not supported by protocol

**EADDRINUSE**
Address already in use

**EADDRNOTAVAIL**
Cannot assign requested address

**ENETDOWN**
Network is down

**ENETUNREACH**
Network is unreachable
ENETRESET
    Network dropped connection because of reset

ECONNABORTED
    Software caused connection abort

ECONNRESET
    Connection reset by peer

ENOBUFFS
    No buffer space available

EISCONN
    Transport endpoint is already connected

ENOTCONN
    Transport endpoint is not connected

ESHUTDOWN
    Cannot send after transport endpoint shutdown

ETOOMANYREFS
    Too many references: cannot splice

ETIMEDOUT
    Connection timed out

ECONNREFUSED
    Connection refused

EHOSTDOWN
    Host is down

EHOSTUNREACH
    No route to host

EALREADY
    Operation already in progress

EINPROGRESS
Operation now in progress

**ESTALE**
Stale NFS file handle

**EUCLEAN**
Structure needs cleaning

**ENOTNAM**
Not a XENIX named type file

**ENAVAIL**
No XENIX semaphores available

**EISNAM**
Is a named type file

**EREMOTEIO**
Remote I/O error

**EDQUOT**
Quota exceeded
14.14 ctypes -- A foreign function library for Python.

New in version 2.5.

cypes is a foreign function library for Python. It provides C compatible data types, and allows to call functions in dlls/shared libraries. It can be used to wrap these libraries in pure Python.

Subsections

- 14.14.1 ctypes tutorial
  - 14.14.1.1 Loading dynamic link libraries
  - 14.14.1.2 Accessing functions from loaded dlls
  - 14.14.1.3 Calling functions
  - 14.14.1.4 Fundamental data types
  - 14.14.1.5 Calling functions, continued
  - 14.14.1.6 Calling functions with your own custom data types
  - 14.14.1.7 Specifying the required argument types (function prototypes)
  - 14.14.1.8 Return types
  - 14.14.1.9 Passing pointers (or: passing parameters by reference)
  - 14.14.1.10 Structures and unions
  - 14.14.1.11 Structure/union alignment and byte order
  - 14.14.1.12 Bit fields in structures and unions
  - 14.14.1.13 Arrays
  - 14.14.1.14 Pointers
  - 14.14.1.15 Type conversions
  - 14.14.1.16 Incomplete Types
  - 14.14.1.17 Callback functions
  - 14.14.1.18 Accessing values exported from dlls
  - 14.14.1.19 Surprises
  - 14.14.1.20 Variable-sized data types
  - 14.14.1.21 Bugs, ToDo and non-implemented things

- 14.14.2 ctypes reference
  - 14.14.2.1 Finding shared libraries
14.14.2.2 Loading shared libraries
14.14.2.3 Foreign functions
14.14.2.4 Function prototypes
14.14.2.5 Utility functions
14.14.2.6 Data types
14.14.2.7 Fundamental data types
14.14.2.8 Structured data types
14.14.2.9 Arrays and pointers

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.14.1 ctypes tutorial

Note: The code samples in this tutorial uses doctest to make sure that they actually work. Since some code samples behave differently under Linux, Windows, or Mac OS X, they contain doctest directives in comments.

Note: Quite some code samples references the ctypes c_int type. This type is an alias to the c_long type on 32-bit systems. So, you should not be confused if c_long is printed if you would expect c_int - they are actually the same type.

Subsections

- 14.14.1.1 Loading dynamic link libraries
- 14.14.1.2 Accessing functions from loaded dlls
- 14.14.1.3 Calling functions
- 14.14.1.4 Fundamental data types
- 14.14.1.5 Calling functions, continued
- 14.14.1.6 Calling functions with your own custom data types
- 14.14.1.7 Specifying the required argument types (function prototypes)
- 14.14.1.8 Return types
- 14.14.1.9 Passing pointers (or: passing parameters by reference)
- 14.14.1.10 Structures and unions
- 14.14.1.11 Structure/union alignment and byte order
- 14.14.1.12 Bit fields in structures and unions
- 14.14.1.13 Arrays
- 14.14.1.14 Pointers
- 14.14.1.15 Type conversions
- 14.14.1.16 Incomplete Types
- 14.14.1.17 Callback functions
- 14.14.1.18 Accessing values exported from dlls
- 14.14.1.19 Surprises
- 14.14.1.20 Variable-sized data types
- 14.14.1.21 Bugs, ToDo and non-implemented things
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.14.2 ctypes reference

Subsections

- 14.14.2.1 Finding shared libraries
- 14.14.2.2 Loading shared libraries
- 14.14.2.3 Foreign functions
- 14.14.2.4 Function prototypes
- 14.14.2.5 Utility functions
- 14.14.2.6 Data types
- 14.14.2.7 Fundamental data types
- 14.14.2.8 Structured data types
- 14.14.2.9 Arrays and pointers

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
15. Optional Operating System Services

The modules described in this chapter provide interfaces to operating system features that are available on selected operating systems only. The interfaces are generally modeled after the UNIX or C interfaces but they are available on some other systems as well (e.g. Windows or NT). Here's an overview:

- **select**: Wait for I/O completion on multiple streams.
- **thread**: Create multiple threads of control within one interpreter.
- **threading**: Higher-level threading interface.
- **dummy_thread**: Drop-in replacement for the `thread` module.
- **dummy_threading**: Drop-in replacement for the `threading` module.
- **mmap**: Interface to memory-mapped files for UNIX and Windows.
- **readline**: GNU readline support for Python.
- **rlcompleter**: Python identifier completion, suitable for the GNU readline library.
15.1 select -- Waiting for I/O completion

This module provides access to the `select()` and `poll()` functions available in most operating systems. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on UNIX, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

The module defines the following:

**exception error**

The exception raised when an error occurs. The accompanying value is a pair containing the numeric error code from `errno` and the corresponding string, as would be printed by the C function `perror()`.

**poll()**

(Not supported by all operating systems.) Returns a polling object, which supports registering and unregistering file descriptors, and then polling them for I/O events; see section 15.1.1 below for the methods supported by polling objects.

**select( iwtd, owtd, ewtd[, timeout])**

This is a straightforward interface to the UNIX `select()` system call. The first three arguments are sequences of `waitable objects`: either integers representing file descriptors or objects with a parameterless method named `fileno()` returning such an integer. The three sequences of waitable objects are for input, output and `exceptional conditions`, respectively. Empty sequences are allowed, but acceptance of three empty sequences is platform-dependent. (It is known to work on UNIX but not on Windows.) The optional `timeout` argument specifies a time-out as a floating point number in seconds. When the `timeout` argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.
The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Among the acceptable object types in the sequences are Python file objects (e.g. `sys.stdin`, or objects returned by `open()` or `os.popen()`), socket objects returned by `socket.socket()`. You may also define a wrapper class yourself, as long as it has an appropriate `fileno()` method (that really returns a file descriptor, not just a random integer). **Note:** File objects on Windows are not acceptable, but sockets are. On Windows, the underlying `select()` function is provided by the WinSock library, and does not handle file descriptors that don't originate from WinSock.

---

**Subsections**

- [15.1.1 Polling Objects](#)

---

*Release 2.5, documentation updated on 19th September, 2006.*

*See [About this document...](#) for information on suggesting changes.*
15.1.1 Polling Objects

The `poll()` system call, supported on most UNIX systems, provides better scalability for network servers that service many, many clients at the same time. `poll()` scales better because the system call only requires listing the file descriptors of interest, while `select()` builds a bitmap, turns on bits for the fds of interest, and then afterward the whole bitmap has to be linearly scanned again. `select()` is $O(\text{highest file descriptor})$, while `poll()` is $O(\text{number of file descriptors})$.

`register(fd[, eventmask])`

Register a file descriptor with the polling object. Future calls to the `poll()` method will then check whether the file descriptor has any pending I/O events. `fd` can be either an integer, or an object with a `fileno()` method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

`eventmask` is an optional bitmask describing the type of events you want to check for, and can be a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`, described in the table below. If not specified, the default value used will check for all 3 types of events.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLLIN</td>
<td>There is data to read</td>
</tr>
<tr>
<td>POLLPRI</td>
<td>There is urgent data to read</td>
</tr>
<tr>
<td>POLLOUT</td>
<td>Ready for output: writing will not block</td>
</tr>
<tr>
<td>POLLERR</td>
<td>Error condition of some sort</td>
</tr>
<tr>
<td>POLLHUP</td>
<td>Hung up</td>
</tr>
<tr>
<td>POLLNVAL</td>
<td>Invalid request: descriptor not open</td>
</tr>
</tbody>
</table>

Registering a file descriptor that's already registered is not an error, and has the same effect as registering the descriptor exactly once.

`unregister(fd)`

Remove a file descriptor being tracked by a polling object. Just like the
register() method, \textit{fd} can be an integer or an object with a \texttt{fileno()} method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a \texttt{KeyError} exception to be raised.

\texttt{poll([\textit{timeout}])}

Polls the set of registered file descriptors, and returns a possibly-empty list containing \((\textit{fd}, \textit{event})\) 2-tuples for the descriptors that have events or errors to report. \textit{fd} is the file descriptor, and \textit{event} is a bitmask with bits set for the reported events for that descriptor -- \texttt{POLLIN} for waiting input, \texttt{POLLOUT} to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If \texttt{timeout} is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If \texttt{timeout} is omitted, negative, or \texttt{None}, the call will block until there is an event for this poll object.
15.2 thread -- Multiple threads of control

This module provides low-level primitives for working with multiple threads (a.k.a. light-weight processes or tasks) -- multiple threads of control sharing their global data space. For synchronization, simple locks (a.k.a. mutexes or binary semaphores) are provided.

The module is optional. It is supported on Windows, Linux, SGI IRIX, Solaris 2.x, as well as on systems that have a POSIX thread (a.k.a. ``pthread'') implementation. For systems lacking the thread module, the dummy_thread module is available. It duplicates this module's interface and can be used as a drop-in replacement.

It defines the following constant and functions:

**exception error**
Raised on thread-specific errors.

**LockType**
This is the type of lock objects.

**start_new_thread** *(function, args[, kwargs]*)
Start a new thread and return its identifier. The thread executes the function *function* with the argument list *args* (which must be a tuple). The optional *kwargs* argument specifies a dictionary of keyword arguments. When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

**interrupt_main()**
Raise a KeyboardInterrupt exception in the main thread. A subthread can use this function to interrupt the main thread. New in version 2.3.

**exit()**
Raise the `SystemExit` exception. When not caught, this will cause the thread to exit silently.

`allocate_lock()`

Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

`get_ident()`

Return the `thread identifier` of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

`stack_size([size])`

Return the thread stack size used when creating new threads. The optional `size` argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32kB). If changing the thread stack size is unsupported, a `ThreadError` is raised. If the specified stack size is invalid, a `ValueError` is raised and the stack size is unmodified. 32kB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32kB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4kB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information). Availability: Windows, systems with POSIX threads. New in version 2.5.

Lock objects have the following methods:

`acquire([waitflag])`

Without the optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock -- that's their reason for existence). If the integer `waitflag` argument is present, the action depends on its value: if it is zero, the lock is only acquired if it can be acquired
immediately without waiting, while if it is nonzero, the lock is acquired unconditionally as before. The return value is True if the lock is acquired successfully, False if not.

**release()**
Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

**locked()**
Return the status of the lock: True if it has been acquired by some thread, False if not.

In addition to these methods, lock objects can also be used via the with statement, e.g.:

```python
from __future__ import with_statement
import thread

a_lock = thread.allocate_lock()

with a_lock:
    print "a_lock is locked while this executes"
```

**Caveats:**

- Threads interact strangely with interrupts: the KeyboardInterrupt exception will be received by an arbitrary thread. (When the signal module is available, interrupts always go to the main thread.)

- Calling sys.exit() or raising the SystemExit exception is equivalent to calling exit().

- Not all built-in functions that may block waiting for I/O allow other threads to run. (The most popular ones (time.sleep(), file.read(), select.select()) work as expected.)

- It is not possible to interrupt the acquire() method on a lock -- the KeyboardInterrupt exception will happen after the lock has been acquired.

- When the main thread exits, it is system defined whether the other threads
survive. On SGI IRIX using the native thread implementation, they survive. On most other systems, they are killed without executing try ... finally clauses or executing object destructors.

- When the main thread exits, it does not do any of its usual cleanup (except that try ... finally clauses are honored), and the standard I/O files are not flushed.
15.3 threading -- Higher-level threading interface

This module constructs higher-level threading interfaces on top of the lower level `thread` module.

The `dummy_threading` module is provided for situations where `threading` cannot be used because `thread` is missing.

This module defines the following functions and objects:

**activeCount()**
Return the number of currently active `Thread` objects. The returned count is equal to the length of the list returned by `enumerate()`. A function that returns the number of currently active threads.

**Condition()**
A factory function that returns a new condition variable object. A condition variable allows one or more threads to wait until they are notified by another thread.

**currentThread()**
Return the current `Thread` object, corresponding to the caller's thread of control. If the caller's thread of control was not created through the `threading` module, a dummy thread object with limited functionality is returned.

**enumerate()**
Return a list of all currently active `Thread` objects. The list includes daemonic threads, dummy thread objects created by `currentThread()`, and the main thread. It excludes terminated threads and threads that have not yet been started.

**Event()**
A factory function that returns a new event object. An event manages a flag
that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

**class local**
A class that represents thread-local data. Thread-local data are data whose values are thread specific. To manage thread-local data, just create an instance of `local` (or a subclass) and store attributes on it:

```python
mydata = threading.local()
mydata.x = 1
```

The instance's values will be different for separate threads.

For more details and extensive examples, see the documentation string of the `threading_local` module.

New in version 2.4.

**Lock()**
A factory function that returns a new primitive lock object. Once a thread has acquired it, subsequent attempts to acquire it block, until it is released; any thread may release it.

**RLock()**
A factory function that returns a new reentrant lock object. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

**Semaphore([value])**
A factory function that returns a new semaphore object. A semaphore manages a counter representing the number of `release()` calls minus the number of `acquire()` calls, plus an initial value. The `acquire()` method blocks if necessary until it can return without making the counter negative. If not given, `value` defaults to 1.

**BoundedSemaphore([value])**
A factory function that returns a new bounded semaphore object. A bounded semaphore checks to make sure its current value doesn't exceed its
initial value. If it does, ValueError is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it's a sign of a bug. If not given, value defaults to 1.

class Thread
A class that represents a thread of control. This class can be safely subclassed in a limited fashion.

class Timer
A thread that executes a function after a specified interval has passed.

settrace(func)
Set a trace function for all threads started from the threading module. The func will be passed to sys.settrace() for each thread, before its run() method is called. New in version 2.3.

setprofile(func)
Set a profile function for all threads started from the threading module. The func will be passed to sys.setprofile() for each thread, before its run() method is called. New in version 2.3.

stack_size([size])
Return the thread stack size used when creating new threads. The optional size argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32kB). If changing the thread stack size is unsupported, a ThreadError is raised. If the specified stack size is invalid, a ValueError is raised and the stack size is unmodified. 32kB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32kB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4kB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information). Availability: Windows, systems with POSIX threads. New in version 2.5.
Detailed interfaces for the objects are documented below.

The design of this module is loosely based on Java's threading model. However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python. Python's `Thread` class supports a subset of the behavior of Java's `Thread` class; currently, there are no priorities, no thread groups, and threads cannot be destroyed, stopped, suspended, resumed, or interrupted. The static methods of Java's `Thread` class, when implemented, are mapped to module-level functions.

All of the methods described below are executed atomically.

Subsections

- 15.3.1 Lock Objects
- 15.3.2 RLock Objects
- 15.3.3 Condition Objects
- 15.3.4 Semaphore Objects
  - 15.3.4.1 Semaphore Example
- 15.3.5 Event Objects
- 15.3.6 Thread Objects
- 15.3.7 Timer Objects
- 15.3.8 Using locks, conditions, and semaphores in the `with` statement

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
15.3.1 Lock Objects

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is currently the lowest level synchronization primitive available, implemented directly by the `thread` extension module.

A primitive lock is in one of two states, ``locked" or "unlocked". It is created in the unlocked state. It has two basic methods, `acquire()` and `release()`. When the state is unlocked, `acquire()` changes the state to locked and returns immediately. When the state is locked, `acquire()` blocks until a call to `release()` in another thread changes it to unlocked, then the `acquire()` call resets it to locked and returns. The `release()` method should only be called in the locked state; it changes the state to unlocked and returns immediately. When more than one thread is blocked in `acquire()` waiting for the state to turn to unlocked, only one thread proceeds when a `release()` call resets the state to unlocked; which one of the waiting threads proceeds is not defined, and may vary across implementations.

All methods are executed atomically.

`acquire([blocking = 1])`

Acquire a lock, blocking or non-blocking.

When invoked without arguments, block until the lock is unlocked, then set it to locked, and return true.

When invoked with the `blocking` argument set to true, do the same thing as when called without arguments, and return true.

When invoked with the `blocking` argument set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

`release()`

Release a lock.

When the lock is locked, reset it to unlocked, and return. If any other
threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

Do not call this method when the lock is unlocked.

There is no return value.
15.3.2 RLock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of "owning thread" and "recursion level" in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.

To lock the lock, a thread calls its acquire() method; this returns once the thread owns the lock. To unlock the lock, a thread calls its release() method. acquire()/release() call pairs may be nested; only the final release() (the release() of the outermost pair) resets the lock to unlocked and allows another thread blocked in acquire() to proceed.

acquire([blocking = 1])

Acquire a lock, blocking or non-blocking.

When invoked without arguments: if this thread already owns the lock, increment the recursion level by one, and return immediately. Otherwise, if another thread owns the lock, block until the lock is unlocked. Once the lock is unlocked (not owned by any thread), then grab ownership, set the recursion level to one, and return. If more than one thread is blocked waiting until the lock is unlocked, only one at a time will be able to grab ownership of the lock. There is no return value in this case.

When invoked with the blocking argument set to true, do the same thing as when called without arguments, and return true.

When invoked with the blocking argument set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

release()

Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked (not owned by any thread), and if any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still
nonzero, the lock remains locked and owned by the calling thread.

Only call this method when the calling thread owns the lock. Do not call this method when the lock is unlocked.

There is no return value.
15.3.3 Condition Objects

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. (Passing one in is useful when several condition variables must share the same lock.)

A condition variable has `acquire()` and `release()` methods that call the corresponding methods of the associated lock. It also has a `wait()` method, and `notify()` and `notifyAll()` methods. These three must only be called when the calling thread has acquired the lock.

The `wait()` method releases the lock, and then blocks until it is awakened by a `notify()` or `notifyAll()` call for the same condition variable in another thread. Once awakened, it re-acquires the lock and returns. It is also possible to specify a timeout.

The `notify()` method wakes up one of the threads waiting for the condition variable, if any are waiting. The `notifyAll()` method wakes up all threads waiting for the condition variable.

Note: the `notify()` and `notifyAll()` methods don't release the lock; this means that the thread or threads awakened will not return from their `wait()` call immediately, but only when the thread that called `notify()` or `notifyAll()` finally relinquishes ownership of the lock.

Tip: the typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are interested in a particular change of state call `wait()` repeatedly until they see the desired state, while threads that modify the state call `notify()` or `notifyAll()` when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity:

```java
# Consume one item
cv.acquire()
while not an_item_is_available():
    cv.wait()
get_an_available_item()
cv.release()
```
# Produce one item

cv.acquire()
make_an_item_available()
cv.notify()
cv.release()

To choose between `notify()` and `notifyAll()`, consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

**class Condition([lock])**

  If the `lock` argument is given and not `None`, it must be a `Lock` or `RLock` object, and it is used as the underlying lock. Otherwise, a new `RLock` object is created and used as the underlying lock.

**acquire(** *args**)

  Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

**release()**

  Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

**wait([timeout])**

  Wait until notified or until a timeout occurs. This must only be called when the calling thread has acquired the lock.

  This method releases the underlying lock, and then blocks until it is awakened by a `notify()` or `notifyAll()` call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

  When the `timeout` argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

  When the underlying lock is an `RLock`, it is not released using its
release() method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the RLock class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

**notify()**

Wake up a thread waiting on this condition, if any. This must only be called when the calling thread has acquired the lock.

This method wakes up one of the threads waiting for the condition variable, if any are waiting; it is a no-op if no threads are waiting.

The current implementation wakes up exactly one thread, if any are waiting. However, it's not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than one thread.

Note: the awakened thread does not actually return from its wait() call until it can reacquire the lock. Since notify() does not release the lock, its caller should.

**notifyAll()**

Wake up all threads waiting on this condition. This method acts like notify(), but wakes up all waiting threads instead of one.
15.3.4 Semaphore Objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used \( P() \) and \( V() \) instead of \texttt{acquire()} and \texttt{release()}).

A semaphore manages an internal counter which is decremented by each \texttt{acquire()} call and incremented by each \texttt{release()} call. The counter can never go below zero; when \texttt{acquire()} finds that it is zero, it blocks, waiting until some other thread calls \texttt{release()}.

\begin{verbatim}

class Semaphore([value])

    The optional argument gives the initial value for the internal counter; it
defaults to 1.

acquire([blocking])

    Acquire a semaphore.

    When invoked without arguments: if the internal counter is larger than zero
    on entry, decrement it by one and return immediately. If it is zero on entry,
    block, waiting until some other thread has called \texttt{release()} to make it
    larger than zero. This is done with proper interlocking so that if multiple
    \texttt{acquire()} calls are blocked, \texttt{release()} will wake exactly one of
    them up. The implementation may pick one at random, so the order in
    which blocked threads are awakened should not be relied on. There is no
    return value in this case.

    When invoked with \texttt{blocking} set to true, do the same thing as when called
    without arguments, and return true.

    When invoked with \texttt{blocking} set to false, do not block. If a call without an
    argument would block, return false immediately; otherwise, do the same
    thing as when called without arguments, and return true.

release()

    Release a semaphore, incrementing the internal counter by one. When it
    was zero on entry and another thread is waiting for it to become larger than
\end{verbatim}
zero again, wake up that thread.

Subsections

- 15.3.4.1 Semaphore Example

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
15.3.5 Event Objects

This is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.

An event object manages an internal flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

```python
class Event():
    The internal flag is initially false.

    isSet()
    Return true if and only if the internal flag is true.

    set()
    Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call `wait()` once the flag is true will not block at all.

    clear()
    Reset the internal flag to false. Subsequently, threads calling `wait()` will block until `set()` is called to set the internal flag to true again.

    wait([timeout])
    Block until the internal flag is true. If the internal flag is true on entry, return immediately. Otherwise, block until another thread calls `set()` to set the flag to true, or until the optional timeout occurs.

    When the timeout argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).
```
Thread Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
15.3.6 Thread Objects

This class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the `run()` method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, *only* override the `__init__() and run()` methods of this class.

Once a thread object is created, its activity must be started by calling the thread's `start()` method. This invokes the `run()` method in a separate thread of control.

Once the thread's activity is started, the thread is considered 'alive' and 'active' (these concepts are almost, but not quite exactly, the same; their definition is intentionally somewhat vague). It stops being alive and active when its `run()` method terminates - either normally, or by raising an unhandled exception. The `isAlive()` method tests whether the thread is alive.

Other threads can call a thread's `join()` method. This blocks the calling thread until the thread whose `join()` method is called is terminated.

A thread has a name. The name can be passed to the constructor, set with the `setName()` method, and retrieved with the `getName()` method.

A thread can be flagged as a "daemon thread". The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set with the `setDaemon()` method and retrieved with the `isDaemon()` method.

There is a "main thread" object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread.

There is the possibility that "dummy thread objects" are created. These are thread objects corresponding to "alien threads". These are threads of control started outside the threading module, such as directly from C code. Dummy thread objects have limited functionality; they are always considered alive, active, and daemonic, and cannot be `join()`ed. They are never deleted, since it is impossible to detect the termination of alien threads.
**class Thread**

(group=None, target=None, name=None, args=(), kwags={})

This constructor should always be called with keyword arguments.

Arguments are:

- **group** should be **None**; reserved for future extension when a ThreadGroup class is implemented.

- **target** is the callable object to be invoked by the `run()` method. Defaults to **None**, meaning nothing is called.

- **name** is the thread name. By default, a unique name is constructed of the form ``Thread-N'' where N is a small decimal number.

- **args** is the argument tuple for the target invocation. Defaults to `()`.  

- **kwargs** is a dictionary of keyword arguments for the target invocation. Defaults to `{}`.

If the subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

**start()**

Start the thread's activity.

This must be called at most once per thread object. It arranges for the object's `run()` method to be invoked in a separate thread of control.

**run()**

Method representing the thread's activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

**join([timeout])**

Wait until the thread terminates. This blocks the calling thread until the
thread whose \texttt{join()} method is called terminates - either normally or through an unhandled exception - or until the optional timeout occurs.

When the \texttt{timeout} argument is present and not \texttt{None}, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As \texttt{join()} always returns \texttt{None}, you must call \texttt{isAlive()} to decide whether a timeout happened.

When the \texttt{timeout} argument is not present or \texttt{None}, the operation will block until the thread terminates.

A thread can be \texttt{join()}ed many times.

A thread cannot join itself because this would cause a deadlock.

It is an error to attempt to \texttt{join()} a thread before it has been started.

\texttt{getName()}
Return the thread's name.

\texttt{setName(name)}
Set the thread's name.

The name is a string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

\texttt{isAlive()}
Return whether the thread is alive.

Roughly, a thread is alive from the moment the \texttt{start()} method returns until its \texttt{run()} method terminates.

\texttt{isDaemon()}
Return the thread's daemon flag.

\texttt{setDaemon(daemonic)}
Set the thread's daemon flag to the Boolean value \texttt{daemonic}. This must be
called before `start()` is called.

The initial value is inherited from the creating thread.

The entire Python program exits when no active non-daemon threads are left.
15.3.7 Timer Objects

This class represents an action that should be run only after a certain amount of time has passed -- a timer. Timer is a subclass of Thread and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their start() method. The timer can be stopped (before its action has begun) by calling the cancel() method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

For example:

```python
def hello():
    print "hello, world"

    t = Timer(30.0, hello)
    t.start()  # after 30 seconds, "hello, world" will be printed
```

```python
class Timer(interval, function, args=[], kwargs={})
    Create a timer that will run function with arguments args and keyword arguments kwargs, after interval seconds have passed.
```

cancel()

Stop the timer, and cancel the execution of the timer's action. This will only work if the timer is still in its waiting stage.
15.3.8 Using locks, conditions, and semaphores in the with statement

All of the objects provided by this module that have acquire() and release() methods can be used as context managers for a with statement. The acquire() method will be called when the block is entered, and release() will be called when the block is exited.

Currently, Lock, RLock, Condition, Semaphore, and BoundedSemaphore objects may be used as with statement context managers. For example:

```python
from __future__ import with_statement
import threading

some_rlock = threading.RLock()

with some_rlock:
    print "some_rlock is locked while this executes"
```

---

Python Library Reference

Previous: 15.3.7 Timer Objects  Up: 15.3 threading  Next: 15.4 dummy_thread

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
15.4 dummy_thread -- Drop-in replacement for the thread module

This module provides a duplicate interface to the thread module. It is meant to be imported when the thread module is not provided on a platform.

Suggested usage is:

```python
try:
    import thread as _thread
except ImportError:
    import dummy_thread as _thread
```

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.
15.5 dummy_threading -- Drop-in replacement for the threading module

This module provides a duplicate interface to the `threading` module. It is meant to be imported when the `thread` module is not provided on a platform.

Suggested usage is:

```python
try:
    import threading as _threading
except ImportError:
    import dummy_threading as _threading
```

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.
15.6 mmap -- Memory-mapped file support

Memory-mapped file objects behave like both strings and like file objects. Unlike normal string objects, however, these are mutable. You can use mmap objects in most places where strings are expected; for example, you can use the `re` module to search through a memory-mapped file. Since they're mutable, you can change a single character by doing `obj[index] = 'a'`, or change a substring by assigning to a slice: `obj[i1:i2] = '...'`. You can also read and write data starting at the current file position, and `seek()` through the file to different positions.

A memory-mapped file is created by the `mmap()` function, which is different on UNIX and on Windows. In either case you must provide a file descriptor for a file opened for update. If you wish to map an existing Python file object, use its `fileno()` method to obtain the correct value for the `fileno` parameter. Otherwise, you can open the file using the `os.open()` function, which returns a file descriptor directly (the file still needs to be closed when done).

For both the UNIX and Windows versions of the function, `access` may be specified as an optional keyword parameter. `access` accepts one of three values: `ACCESS_READ`, `ACCESS_WRITE`, or `ACCESS_COPY` to specify readonly, write-through or copy-on-write memory respectively. `access` can be used on both UNIX and Windows. If `access` is not specified, Windows mmap returns a write-through mapping. The initial memory values for all three access types are taken from the specified file. Assignment to an `ACCESS_READ` memory map raises a `TypeError` exception. Assignment to an `ACCESS_WRITE` memory map affects both memory and the underlying file. Assignment to an `ACCESS_COPY` memory map affects memory but does not update the underlying file. Changed in version 2.5: To map anonymous memory, -1 should be passed as the `fileno` along with the length.

\[ \texttt{mmap}( \texttt{fileno, length[, tagname][, access]]) \]

(Windows version) Maps `length` bytes from the file specified by the file handle `fileno`, and returns a mmap object. If `length` is larger than the current
size of the file, the file is extended to contain length bytes. If length is 0, the maximum length of the map is the current size of the file, except that if the file is empty Windows raises an exception (you cannot create an empty mapping on Windows).

tagname, if specified and not None, is a string giving a tag name for the mapping. Windows allows you to have many different mappings against the same file. If you specify the name of an existing tag, that tag is opened, otherwise a new tag of this name is created. If this parameter is omitted or None, the mapping is created without a name. Avoiding the use of the tag parameter will assist in keeping your code portable between UNIX and Windows.

mmap( fileno, length[, flags[, prot[, access]]]])

(Unix version) Maps length bytes from the file specified by the file descriptor fileno, and returns a mmap object. If length is 0, the maximum length of the map will be the current size of the file when mmap( ) is called.

flags specifies the nature of the mapping. MAP_PRIVATE creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and MAP_SHARED creates a mapping that's shared with all other processes mapping the same areas of the file. The default value is MAP_SHARED.

prot, if specified, gives the desired memory protection; the two most useful values are PROT_READ and PROT_WRITE, to specify that the pages may be read or written. prot defaults to PROT_READ | PROT_WRITE.

access may be specified in lieu of flags and prot as an optional keyword parameter. It is an error to specify both flags, prot and access. See the description of access above for information on how to use this parameter.

Memory-mapped file objects support the following methods:

close()

Close the file. Subsequent calls to other methods of the object will result in an exception being raised.
**find**(string[, start])

Returns the lowest index in the object where the substring `string` is found. Returns -1 on failure. `start` is the index at which the search begins, and defaults to zero.

**flush**(offset, size)

Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If `offset` and `size` are specified, only changes to the given range of bytes will be flushed to disk; otherwise, the whole extent of the mapping is flushed.

**move**(dest, src, count)

Copy the `count` bytes starting at offset `src` to the destination index `dest`. If the mmap was created with ACCESS_READ, then calls to move will throw a TypeError exception.

**read**(num)

Return a string containing up to `num` bytes starting from the current file position; the file position is updated to point after the bytes that were returned.

**read_byte**()

Returns a string of length 1 containing the character at the current file position, and advances the file position by 1.

**readline**()

Returns a single line, starting at the current file position and up to the next newline.

**resize**(newsize)

Resizes the map and the underlying file, if any. If the mmap was created with ACCESS_READ or ACCESS_COPY, resizing the map will throw a TypeError exception.

**seek**(pos[, whence])
Set the file's current position. whence argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file's end).

**size()**

Return the length of the file, which can be larger than the size of the memory-mapped area.

**tell()**

Returns the current position of the file pointer.

**write**( `string`)  
Write the bytes in string into memory at the current position of the file pointer; the file position is updated to point after the bytes that were written. If the mmap was created with ACCESS_READ, then writing to it will throw a TypeError exception.

**write_byte**( `byte`)  
Write the single-character string byte into memory at the current position of the file pointer; the file position is advanced by 1. If the mmap was created with ACCESS_READ, then writing to it will throw a TypeError exception.
15.7 readline -- GNU readline interface

Availability: Unix.

The readline module defines a number of functions to facilitate completion and reading/writing of history files from the Python interpreter. This module can be used directly or via the rlcompleter module. Settings made using this module affect the behaviour of both the interpreter's interactive prompt and the prompts offered by the raw_input() and input() built-in functions.

The readline module defines the following functions:

**parse_and_bind**( *string*)
Parse and execute single line of a readline init file.

**get_line_buffer**( )
Return the current contents of the line buffer.

**insert_text**( *string*)
Insert text into the command line.

**read_init_file**( *[filename]*)
Parse a readline initialization file. The default filename is the last filename used.

**read_history_file**( *[filename]*)
Load a readline history file. The default filename is ~/.history.

**write_history_file**( *[filename]*)
Save a readline history file. The default filename is ~/.history.

**clear_history**( )
Clear the current history. (Note: this function is not available if the installed
version of GNU readline doesn't support it.) New in version 2.4.

**get_history_length()**
Return the desired length of the history file. Negative values imply unlimited history file size.

**set_history_length**( length)
Set the number of lines to save in the history file. 
**write_history_file()** uses this value to truncate the history file when saving. Negative values imply unlimited history file size.

**get_current_history_length()**
Return the number of lines currently in the history. (This is different from **get_history_length()**, which returns the maximum number of lines that will be written to a history file.) New in version 2.3.

**get_history_item**( index)
Return the current contents of history item at index. New in version 2.3.

**remove_history_item**( pos)
Remove history item specified by its position from the history. New in version 2.4.

**replace_history_item**( pos, line)
Replace history item specified by its position with the given line. New in version 2.4.

**redisplay()**
Change what's displayed on the screen to reflect the current contents of the line buffer. New in version 2.3.

**set_startup_hook**( [function])
Set or remove the startup_hook function. If function is specified, it will be used as the new startup_hook function; if omitted or **None**, any hook function already installed is removed. The startup_hook function is called with no arguments just before readline prints the first prompt.
**set_pre_input_hook**([*function*])

Set or remove the pre_input_hook function. If *function* is specified, it will be used as the new pre_input_hook function; if omitted or *None*, any hook function already installed is removed. The pre_input_hook function is called with no arguments after the first prompt has been printed and just before readline starts reading input characters.

**set_completer**([*function*])

Set or remove the completer function. If *function* is specified, it will be used as the new completer function; if omitted or *None*, any completer function already installed is removed. The completer function is called as *function*(text, state), for state in 0, 1, 2, ..., until it returns a non-string value. It should return the next possible completion starting with *text*.

**get_completer**()

Get the completer function, or *None* if no completer function has been set. New in version 2.3.

**get_begidx**()

Get the beginning index of the readline tab-completion scope.

**get_endidx**()

Get the ending index of the readline tab-completion scope.

**set_completer_delims**(*string*)

Set the readline word delimiters for tab-completion.

**get_completer_delims**()

Get the readline word delimiters for tab-completion.

**add_history**(*line*)

Append a line to the history buffer, as if it was the last line typed.

---

**See Also:**

Module [rlcompleter](https://docs.python.org/3/library/rlcompleter.html):
Completion of Python identifiers at the interactive prompt.

Subsections

- 15.7.1 Example

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
15.7.1 Example

The following example demonstrates how to use the `readline` module's history reading and writing functions to automatically load and save a history file named `.pyhist` from the user's home directory. The code below would normally be executed automatically during interactive sessions from the user's PYTHONSTARTUP file.

```python
import os
histfile = os.path.join(os.environ['HOME'], '.pyhist')
try:
    readline.read_history_file(histfile)
except IOError:
    pass
import atexit
atexit.register(readline.write_history_file, histfile)
del os, histfile
```

The following example extends the `code.InteractiveConsole` class to support history save/restore.

```python
import code
import readline
import atexit
import os

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>", histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except IOError:
                pass
        atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.write_history_file(histfile)
```

---

Python Library Reference
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
The `rlcompleter` module defines a completion function suitable for the `readline` module by completing valid Python identifiers and keywords.

When this module is imported on a UNIX platform with the `readline` module available, an instance of the `Completer` class is automatically created and its `complete` method is set as the `readline` completer.

Example:

```python
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__ readline.get_line_buffer readline.read_init_file
readline.__file__ readline.insert_text readline.set_completer
readline.__name__ readline.parse_and_bind

>>>
```

The `rlcompleter` module is designed for use with Python's interactive mode. A user can add the following lines to his or her initialization file (identified by the PYTHONSTARTUP environment variable) to get automatic Tab completion:

```
try:
    import readline
except ImportError:
    print "Module readline not available."
else:
    import rlcompleter
    readline.parse_and_bind("tab: complete")
```

On platforms without `readline`, the `Completer` class defined by this module can still be used for custom purposes.

---

**Subsections**

- **15.8.1 Completer Objects**
Completer Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
15.8.1 Completer Objects

Completer objects have the following method:

```
complete( text, state )
```

Return the stateth completion for text.

If called for text that doesn't include a period character (".") it will complete from names currently defined in __main__, __builtin__ and keywords (as defined by the keyword module).

If called for a dotted name, it will try to evaluate anything without obvious side-effects (functions will not be evaluated, but it can generate calls to __getattr__() up to the last part, and find matches for the rest via the dir() function.
16. Unix Specific Services

The modules described in this chapter provide interfaces to features that are unique to the Unix operating system, or in some cases to some or many variants of it. Here's an overview:

- **posix**: The most common POSIX system calls (normally used via module `os`).
- **pwd**: The password database (`getpwnam()` and friends).
- **spwd**: The shadow password database (`getspnam()` and friends).
- **grp**: The group database (`getgrnam()` and friends).
- **crypt**: The `crypt()` function used to check Unix passwords.
- **dl**: Call C functions in shared objects.
- **termios**: POSIX style tty control.
- **tty**: Utility functions that perform common terminal control operations.
- **pty**: Pseudo-Terminal Handling for SGI and Linux.
- **fcntl**: The `fcntl()` and `ioctl()` system calls.
- **pipes**: A Python interface to Unix shell pipelines.
- **posixfile**: A file-like object with support for locking.
- **resource**: An interface to provide resource usage information on the current process.
- **nis**: Interface to Sun's NIS (Yellow Pages) library.
- **syslog**: An interface to the Unix syslog library routines.
- **commands**: Utility functions for running external commands.
16.1 posix -- The most common POSIX system calls

Availability: Unix.

This module provides access to operating system functionality that is standardized by the C Standard and the POSIX standard (a thinly disguised UNIX interface).

Do not import this module directly. Instead, import the module os, which provides a portable version of this interface. On UNIX, the os module provides a superset of the posix interface. On non-UNIX operating systems the posix module is not available, but a subset is always available through the os interface. Once os is imported, there is no performance penalty in using it instead of posix. In addition, os provides some additional functionality, such as automatically calling putenv() when an entry in os.environ is changed.

The descriptions below are very terse; refer to the corresponding UNIX manual (or POSIX documentation) entry for more information. Arguments called path refer to a pathname given as a string.

Errors are reported as exceptions; the usual exceptions are given for type errors, while errors reported by the system calls raise error (a synonym for the standard exception OSError), described below.

Subsections

- 16.1.1 Large File Support
- 16.1.2 Module Contents

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
16.1.1 Large File Support

Several operating systems (including AIX, HPUX, Irix and Solaris) provide support for files that are larger than 2 Gb from a C programming model where `int` and `long` are 32-bit values. This is typically accomplished by defining the relevant size and offset types as 64-bit values. Such files are sometimes referred to as large files.

Large file support is enabled in Python when the size of an `off_t` is larger than a `long` and the `long long` type is available and is at least as large as an `off_t`. Python longs are then used to represent file sizes, offsets and other values that can exceed the range of a Python int. It may be necessary to configure and compile Python with certain compiler flags to enable this mode. For example, it is enabled by default with recent versions of Irix, but with Solaris 2.6 and 2.7 you need to do something like:

```
CFLAGS=``getconf LFS_CFLAGS`` OPT=-g -O2 $CFLAGS \
./configure
```

On large-file-capable Linux systems, this might work:

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT=-g -O2 $C \
./configure
```
16.1.2 Module Contents

Module `posix` defines the following data item:

environ

A dictionary representing the string environment at the time the interpreter was started. For example, `environ['HOME']` is the pathname of your home directory, equivalent to `getenv("HOME")` in C.

Modifying this dictionary does not affect the string environment passed on by `execv()`, `popen()` or `system()`; if you need to change the environment, pass `environ` to `execve()` or add variable assignments and export statements to the command string for `system()` or `popen()`.

Note: The `os` module provides an alternate implementation of `environ` which updates the environment on modification. Note also that updating `os.environ` will render this dictionary obsolete. Use of the `os` module version of this is recommended over direct access to the `posix` module.

Additional contents of this module should only be accessed via the `os` module; refer to the documentation for that module for further information.
16.2 pwd -- The password database

Availability: **Unix**.

This module provides access to the *Unix* user account and password database. It is available on all *Unix* versions.

Password database entries are reported as a tuple-like object, whose attributes correspond to the members of the *passwd* structure (Attribute field below, see `<pwd.h>`):

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><em>pw_name</em></td>
<td>Login name</td>
</tr>
<tr>
<td>1</td>
<td><em>pw_passwd</em></td>
<td>Optional encrypted password</td>
</tr>
<tr>
<td>2</td>
<td><em>pw_uid</em></td>
<td>Numerical user ID</td>
</tr>
<tr>
<td>3</td>
<td><em>pw_gid</em></td>
<td>Numerical group ID</td>
</tr>
<tr>
<td>4</td>
<td><em>pw_gecos</em></td>
<td>User name or comment field</td>
</tr>
<tr>
<td>5</td>
<td><em>pw_dir</em></td>
<td>User home directory</td>
</tr>
<tr>
<td>6</td>
<td><em>pw_shell</em></td>
<td>User command interpreter</td>
</tr>
</tbody>
</table>

The uid and gid items are integers, all others are strings. **KeyError** is raised if the entry asked for cannot be found.

**Note:** In traditional *Unix* the field *pw_passwd* usually contains a password encrypted with a DES derived algorithm (see module *crypt*). However most modern unices use a so-called *shadow password* system. On those unices the *pw_passwd* field only contains an asterisk ('*') or the letter "x" where the encrypted password is stored in a file /etc/shadow which is not world readable. Whether the *pw_passwd* field contains anything useful is system-dependent. If available, the *spwd* module should be used where access to the encrypted password is required.

It defines the following items:

```
getpwuid(uid)
```
Return the password database entry for the given numeric user ID.

`getpwnam(name)`
Return the password database entry for the given user name.

`getpwall()`
Return a list of all available password database entries, in arbitrary order.

See Also:

Module `grp`:
An interface to the group database, similar to this.

Module `spwd`:
An interface to the shadow password database, similar to this.
16.3 spwd -- The shadow password database

Availability: **Unix**.

New in version 2.5.

This module provides access to the **UNIX** shadow password database. It is available on various **UNIX** versions.

You must have enough privileges to access the shadow password database (this usually means you have to be root).

Shadow password database entries are reported as a tuple-like object, whose attributes correspond to the members of the **spwd** structure (Attribute field below, see `<shadow.h>`):

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sp_nam</td>
<td>Login name</td>
</tr>
<tr>
<td>1</td>
<td>sp_pwd</td>
<td>Encrypted password</td>
</tr>
<tr>
<td>2</td>
<td>sp_lstchg</td>
<td>Date of last change</td>
</tr>
<tr>
<td>3</td>
<td>sp_min</td>
<td>Minimal number of days between changes</td>
</tr>
<tr>
<td>4</td>
<td>sp_max</td>
<td>Maximum number of days between changes</td>
</tr>
<tr>
<td>5</td>
<td>sp_warn</td>
<td>Number of days before password expires to warn user about it</td>
</tr>
<tr>
<td>6</td>
<td>sp_inact</td>
<td>Number of days after password expires until account is blocked</td>
</tr>
<tr>
<td>7</td>
<td>sp_expire</td>
<td>Number of days since 1970-01-01 until account is disabled</td>
</tr>
<tr>
<td>8</td>
<td>sp_flag</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The sp_nam and sp_pwd items are strings, all others are integers. **KeyError** is raised if the entry asked for cannot be found.

It defines the following items:
getspnam(name)
Return the shadow password database entry for the given user name.

getspall()
Return a list of all available shadow password database entries, in arbitrary order.

See Also:
Module grp:
An interface to the group database, similar to this.

Module pwd:
An interface to the normal password database, similar to this.
16.4 grp -- The group database

Availability: Unix.

This module provides access to the Unix group database. It is available on all Unix versions.

Group database entries are reported as a tuple-like object, whose attributes correspond to the members of the group structure (Attribute field below, see <pwd.h>):

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>gr_name</td>
<td>the name of the group</td>
</tr>
<tr>
<td>1</td>
<td>gr_passwd</td>
<td>the (encrypted) group password; often empty</td>
</tr>
<tr>
<td>2</td>
<td>gr_gid</td>
<td>the numerical group ID</td>
</tr>
<tr>
<td>3</td>
<td>gr_mem</td>
<td>all the group member's user names</td>
</tr>
</tbody>
</table>

The gid is an integer, name and password are strings, and the member list is a list of strings. (Note that most users are not explicitly listed as members of the group they are in according to the password database. Check both databases to get complete membership information.)

It defines the following items:

**getgrgid** (*gid*)

Return the group database entry for the given numeric group ID. *KeyError* is raised if the entry asked for cannot be found.

**getgrnam** (*name*)

Return the group database entry for the given group name. *KeyError* is raised if the entry asked for cannot be found.

**getgrall**()

Return a list of all available group entries, in arbitrary order.
See Also:

Module **pwd**: An interface to the user database, similar to this.

Module **spwd**: An interface to the shadow password database, similar to this.
16.5 crypt -- Function to check UNIX passwords

Availability: Unix.

This module implements an interface to the crypt(3) routine, which is a one-way hash function based upon a modified DES algorithm; see the UNIX man page for further details. Possible uses include allowing Python scripts to accept typed passwords from the user, or attempting to crack UNIX passwords with a dictionary.

Notice that the behavior of this module depends on the actual implementation of the crypt(3) routine in the running system. Therefore, any extensions available on the current implementation will also be available on this module.

\texttt{crypt( word, salt)}

\textit{word} will usually be a user's password as typed at a prompt or in a graphical interface. \textit{salt} is usually a random two-character string which will be used to perturb the DES algorithm in one of 4096 ways. The characters in \textit{salt} must be in the set \texttt{[./a-zA-Z0-9]}]. Returns the hashed password as a string, which will be composed of characters from the same alphabet as the salt (the first two characters represent the salt itself).

Since a few crypt(3) extensions allow different values, with different sizes in the \textit{salt}, it is recommended to use the full crypted password as salt when checking for a password.

A simple example illustrating typical use:

\begin{verbatim}
import crypt, getpass, pwd

def login():
    username = raw_input('Python login: ')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise "Sorry, currently no support for shadow passwords"
        cleartext = getpass.getpass()
        return crypt.crypt(cleartext, cryptedpasswd) == cryptedpasswd

    return False
\end{verbatim}
else:
    return 1
16.6 dl -- Call C functions in shared objects

Availability: Unix.

The dl module defines an interface to the dlopen() function, which is the most common interface on UNIX platforms for handling dynamically linked libraries. It allows the program to call arbitrary functions in such a library.

Warning: The dl module bypasses the Python type system and error handling. If used incorrectly it may cause segmentation faults, crashes or other incorrect behaviour.

Note: This module will not work unless sizeof(int) == sizeof(long) == sizeof(char *). If this is not the case, SystemError will be raised on import.

The dl module defines the following function:

```python
open( name[, mode = RTLD_LAZY])
```

Open a shared object file, and return a handle. Mode signifies late binding (RTLD_LAZY) or immediate binding (RTLD_NOW). Default is RTLD_LAZY. Note that some systems do not support RTLD_NOW.

Return value is a dlobject.

The dl module defines the following constants:

```
RTLD_LAZY
Useful as an argument to open().
```

```
RTLD_NOW
Useful as an argument to open(). Note that on systems which do not support immediate binding, this constant will not appear in the module. For maximum portability, use hasattr() to determine if the system supports immediate binding.
```
The `dl` module defines the following exception:

**exception error**

Exception raised when an error has occurred inside the dynamic loading and linking routines.

Example:

```python
>>> import dl, time
>>> a=dl.open('/lib/libc.so.6')
>>> a.call('time'), time.time()
(929723914, 929723914.498)
```

This example was tried on a Debian GNU/Linux system, and is a good example of the fact that using this module is usually a bad alternative.
16.6.1 DI Objects

DI objects, as returned by `open()` above, have the following methods:

**close()**
Free all resources, except the memory.

**sym(name)**
Return the pointer for the function named `name`, as a number, if it exists in the referenced shared object, otherwise `None`. This is useful in code like:

```python
>>> if a.sym('time'):
    ...    a.call('time')
    ... else:
    ...    time.time()
```

(Note that this function will return a non-zero number, as zero is the NULL pointer)

**call(name[, arg1[, arg2...]])**
Call the function named `name` in the referenced shared object. The arguments must be either Python integers, which will be passed as is, Python strings, to which a pointer will be passed, or `None`, which will be passed as NULL. Note that strings should only be passed to functions as `const char*`, as Python will not like its string mutated.

There must be at most 10 arguments, and arguments not given will be treated as `None`. The function's return value must be a C `long`, which is a Python integer.
16.7 termios -- POSIX style tty control

Availability: Unix.

This module provides an interface to the POSIX calls for tty I/O control. For a complete description of these calls, see the POSIX or UNIX manual pages. It is only available for those UNIX versions that support POSIX termios style tty I/O control (and then only if configured at installation time).

All functions in this module take a file descriptor `fd` as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or a file object, such as `sys.stdin` itself.

This module also defines all the constants needed to work with the functions provided here; these have the same name as their counterparts in C. Please refer to your system documentation for more information on using these terminal control interfaces.

The module defines the following functions:

**tcgetattr(fd)**
Return a list containing the tty attributes for file descriptor `fd`, as follows: `[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]` where `cc` is a list of the tty special characters (each a string of length 1, except the items with indices VMIN and VTIME, which are integers when these fields are defined). The interpretation of the flags and the speeds as well as the indexing in the `cc` array must be done using the symbolic constants defined in the `termios` module.

**tcsetattr(fd, when, attributes)**
Set the tty attributes for file descriptor `fd` from the `attributes`, which is a list like the one returned by `tcgetattr()`. The `when` argument determines when the attributes are changed: TCSANOW to change immediately, TCSADRAIN to change after transmitting all queued output, or
TCSAFLUSH to change after transmitting all queued output and discarding all queued input.

**tcsendbreak**( *fd*, *duration*)
Send a break on file descriptor *fd*. A zero *duration* sends a break for 0.25-0.5 seconds; a nonzero *duration* has a system dependent meaning.

**tcdrain**( *fd*)
Wait until all output written to file descriptor *fd* has been transmitted.

**tcflush**( *fd*, *queue*)
Discard queued data on file descriptor *fd*. The *queue* selector specifies which queue: TCIFFLUSH for the input queue, TCOFLUSH for the output queue, or TCI0FLUSH for both queues.

**tcflow**( *fd*, *action*)
Suspend or resume input or output on file descriptor *fd*. The *action* argument can be TCOFF to suspend output, TCOON to restart output, TCI0FF to suspend input, or TCION to restart input.

---

**See Also:**

Module **tty**:
Convenience functions for common terminal control operations.

---

**Subsections**

- [16.7.1 Example](#)

---

**Python Library Reference**

Previous: **16.6.1 Di Objects**  Up: **16. Unix Specific Services**  Next: **16.7.1 Example**

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
16.7.1 Example

Here's a function that prompts for a password with echoing turned off. Note the technique using a separate `tcgetattr()` call and a `try ... finally` statement to ensure that the old tty attributes are restored exactly no matter what happens:

```python
def getpass(prompt = "Password: "):  
    import termios, sys  
    fd = sys.stdin.fileno()  
    old = termios.tcgetattr(fd)  
    new = termios.tcgetattr(fd)  
    try:  
        termios.tcsetattr(fd, termios.TCSADRAIN, new)  
        passwd = raw_input(prompt)  
    finally:  
        termios.tcsetattr(fd, termios.TCSADRAIN, old)  
    return passwd
```

Python Library Reference

Previous: 16.7 termios  Up: 16.7 termios  Next: 16.8 tty

*Release 2.5, documentation updated on 19th September, 2006.*

*See [About this document...](#) for information on suggesting changes.*
16.8 tty -- Terminal control functions

Availability: Unix.

The tty module defines functions for putting the tty into cbreak and raw modes.

Because it requires the termios module, it will work only on UNIX.

The tty module defines the following functions:

```
setraw(fd[, when])
   Change the mode of the file descriptor fd to raw. If when is omitted, it
   defaults to termios.TCSAFLUSH, and is passed to
   termios.tcsetattr().

setcbreak(fd[, when])
   Change the mode of file descriptor fd to cbreak. If when is omitted, it
   defaults to termios.TCSAFLUSH, and is passed to
   termios.tcsetattr().
```

See Also:

Module termios:
   Low-level terminal control interface.
16.9 pty -- Pseudo-terminal utilities

Availability: IRIX, Linux.

The pty module defines operations for handling the pseudo-terminal concept: starting another process and being able to write to and read from its controlling terminal programmatically.

Because pseudo-terminal handling is highly platform dependant, there is code to do it only for SGI and Linux. (The Linux code is supposed to work on other platforms, but hasn't been tested yet.)

The pty module defines the following functions:

**fork()**
Fork. Connect the child's controlling terminal to a pseudo-terminal. Return value is \((\text{pid}, \, \text{fd})\). Note that the child gets \(\text{pid} \, 0\), and the \(\text{fd}\) is invalid. The parent's return value is the \(\text{pid}\) of the child, and \(\text{fd}\) is a file descriptor connected to the child's controlling terminal (and also to the child's standard input and output).

**openpty()**
Open a new pseudo-terminal pair, using \texttt{os.openpty()} if possible, or emulation code for SGI and generic Unix systems. Return a pair of file descriptors \((\text{master}, \, \text{slave})\), for the master and the slave end, respectively.

**spawn( argv[, master_read[, stdin_read]])**
Spawn a process, and connect its controlling terminal with the current process's standard io. This is often used to baffle programs which insist on reading from the controlling terminal.

The functions \textit{master_read} and \textit{stdin_read} should be functions which read from a file-descriptor. The defaults try to read 1024 bytes each time they are called.
Release 2.5, documentation updated on 19th September, 2006.  
See About this document... for information on suggesting changes.
16.10 fcntl -- The fcntl() and ioctl() system calls

Availability: Unix.

This module performs file control and I/O control on file descriptors. It is an interface to the fcntl() and ioctl() UNIX routines.

All functions in this module take a file descriptor \textit{fd} as their first argument. This can be an integer file descriptor, such as returned by \texttt{sys.stdin.fileno()}, or a file object, such as \texttt{sys.stdin} itself, which provides a \texttt{fileno()} which returns a genuine file descriptor.

The module defines the following functions:

\textbf{fcntl}(\textit{fd}, \textit{op}[\textit{arg}])

Perform the requested operation on file descriptor \textit{fd} (file objects providing a \texttt{fileno()} method are accepted as well). The operation is defined by \textit{op} and is operating system dependent. These codes are also found in the fcntl module. The argument \textit{arg} is optional, and defaults to the integer value \texttt{0}. When present, it can either be an integer value, or a string. With the argument missing or an integer value, the return value of this function is the integer return value of the C fcntl() call. When the argument is a string it represents a binary structure, e.g. created by \texttt{struct.pack()}. The binary data is copied to a buffer whose address is passed to the C fcntl() call. The return value after a successful call is the contents of the buffer, converted to a string object. The length of the returned string will be the same as the length of the \textit{arg} argument. This is limited to 1024 bytes. If the information returned in the buffer by the operating system is larger than 1024 bytes, this is most likely to result in a segmentation violation or a more subtle data corruption.

If the fcntl() fails, an IOError is raised.

\textbf{ioctl}(\textit{fd}, \textit{op}[\textit{arg}[\textit{mutate\_flag}]]))
This function is identical to the `fcntl()` function, except that the operations are typically defined in the library module `termios` and the argument handling is even more complicated.

The parameter `arg` can be one of an integer, absent (treated identically to the integer 0), an object supporting the read-only buffer interface (most likely a plain Python string) or an object supporting the read-write buffer interface.

In all but the last case, behaviour is as for the `fcntl()` function.

If a mutable buffer is passed, then the behaviour is determined by the value of the `mutate_flag` parameter.

If it is false, the buffer's mutability is ignored and behaviour is as for a read-only buffer, except that the 1024 byte limit mentioned above is avoided - so long as the buffer you pass is as least as long as what the operating system wants to put there, things should work.

If `mutate_flag` is true, then the buffer is (in effect) passed to the underlying `ioctl()` system call, the latter's return code is passed back to the calling Python, and the buffer's new contents reflect the action of the `ioctl()`.

This is a slight simplification, because if the supplied buffer is less than 1024 bytes long it is first copied into a static buffer 1024 bytes long which is then passed to `ioctl()` and copied back into the supplied buffer.

If `mutate_flag` is not supplied, then from Python 2.5 it defaults to true, which is a change from versions 2.3 and 2.4. Supply the argument explicitly if version portability is a priority.

An example:

```python
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, " "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```
**flock**($fd$, $op$)
Perform the lock operation $op$ on file descriptor $fd$ (file objects providing a `fileno()` method are accepted as well). See the UNIX manual `flock(3)` for details. (On some systems, this function is emulated using `fcntl()`.)

**lockf**($fd$, $operation$, [$length$, [$start$, [whence]]])
This is essentially a wrapper around the `fcntl()` locking calls. $fd$ is the file descriptor of the file to lock or unlock, and $operation$ is one of the following values:

- LOCK_UN - unlock
- LOCK_SH - acquire a shared lock
- LOCK_EX - acquire an exclusive lock

When $operation$ is LOCK_SH or LOCK_EX, it can also be bit-wise OR'd with LOCK_NB to avoid blocking on lock acquisition. If LOCK_NB is used and the lock cannot be acquired, an IOError will be raised and the exception will have an `errno` attribute set to EACCES or EAGAIN (depending on the operating system; for portability, check for both values). On at least some systems, LOCK_EX can only be used if the file descriptor refers to a file opened for writing.

$length$ is the number of bytes to lock, $start$ is the byte offset at which the lock starts, relative to whence, and whence is as with `fileobj.seek()`, specifically:

- 0 - relative to the start of the file (SEEK_SET)
- 1 - relative to the current buffer position (SEEK_CUR)
- 2 - relative to the end of the file (SEEK_END)

The default for $start$ is 0, which means to start at the beginning of the file. The default for $length$ is 0 which means to lock to the end of the file. The default for whence is also 0.

Examples (all on a SVR4 compliant system):

```python
import struct, fcntl, os
f = open(...)```
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)

Note that in the first example the return value variable \( rv \) will hold an integer value; in the second example it will hold a string value. The structure lay-out for the \( lockdata \) variable is system dependent -- therefore using the \texttt{flock()} \) call may be better.

See Also:

Module \texttt{os}:

If the locking flags \texttt{O_SHLOCK} and \texttt{O_EXLOCK} are present in the \texttt{os} module, the \texttt{os.open()} function provides a more platform-independent alternative to the \texttt{lockf()} and \texttt{flock()} functions.
16.11 pipes -- Interface to shell pipelines

Availability: Unix.

The pipes module defines a class to abstract the concept of a pipeline -- a sequence of converters from one file to another.

Because the module uses /bin/sh command lines, a POSIX or compatible shell for os.system() and os.popen() is required.

The pipes module defines the following class:

```python
class Template()
    An abstraction of a pipeline.
```

Example:

```python
>>> import pipes
>>> t=pipes.Template()
>>> t.append('tr a-z A-Z', '--')
>>> f=t.open('/tmp/1', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('/tmp/1').read()
'HELLO WORLD'
```
16.11.1 Template Objects

Template objects following methods:

**reset()**
Restore a pipeline template to its initial state.

**clone()**
Return a new, equivalent, pipeline template.

**debug( flag)**
If *flag* is true, turn debugging on. Otherwise, turn debugging off. When debugging is on, commands to be executed are printed, and the shell is given `set -x` command to be more verbose.

**append( cmd, kind)**
Append a new action at the end. The *cmd* variable must be a valid bourne shell command. The *kind* variable consists of two letters.

The first letter can be either of ' - ' (which means the command reads its standard input), ' f ' (which means the command reads a given file on the command line) or ' . ' (which means the command reads no input, and hence must be first.)

Similarly, the second letter can be either of ' - ' (which means the command writes to standard output), ' f ' (which means the command writes a file on the command line) or ' . ' (which means the command does not write anything, and hence must be last.)

**prepend( cmd, kind)**
Add a new action at the beginning. See append( ) for explanations of the arguments.

**open( file, mode)**
Return a file-like object, open to *file*, but read from or written to by the pipeline. Note that only one of ' r ', ' w ' may be given.
`copy(infile, outfile)`
Copy `infile` to `outfile` through the pipe.
16.12 posixfile -- File-like objects with locking support

Availability: Unix.

**Deprecated since release 1.5.** The locking operation that this module provides is done better and more portably by the `fcntl.lockf()` call.

This module implements some additional functionality over the built-in file objects. In particular, it implements file locking, control over the file flags, and an easy interface to duplicate the file object. The module defines a new file object, the posixfile object. It has all the standard file object methods and adds the methods described below. This module only works for certain flavors of UNIX, since it uses `fcntl.fcntl()` for file locking.

To instantiate a posixfile object, use the `open()` function in the `posixfile` module. The resulting object looks and feels roughly the same as a standard file object.

The `posixfile` module defines the following constants:

**SEEK_SET**
Offset is calculated from the start of the file.

**SEEK_CUR**
Offset is calculated from the current position in the file.

**SEEK_END**
Offset is calculated from the end of the file.

The `posixfile` module defines the following functions:

`open(filename[, mode[, bufsize]])`
Create a new posixfile object with the given filename and mode. The `filename`, `mode` and `bufsize` arguments are interpreted the same way as by the built-in `open()` function.
**fileopen***( fileobject)***  
Create a new posixfile object with the given standard file object. The resulting object has the same filename and mode as the original file object.

The posixfile object defines the following additional methods:

**lock***( fmt, [len[, start[, whence]]])***  
Lock the specified section of the file that the file object is referring to. The format is explained below in a table. The *len* argument specifies the length of the section that should be locked. The default is 0. *start* specifies the starting offset of the section, where the default is 0. The *whence* argument specifies where the offset is relative to. It accepts one of the constants SEEK_SET, SEEK_CUR or SEEK_END. The default is SEEK_SET. For more information about the arguments refer to the *fcntl(2)* manual page on your system.

**flags***( [flags])***  
Set the specified flags for the file that the file object is referring to. The new flags are ORed with the old flags, unless specified otherwise. The format is explained below in a table. Without the *flags* argument a string indicating the current flags is returned (this is the same as the "?" modifier). For more information about the flags refer to the *fcntl(2)* manual page on your system.

**dup**()  
Duplicate the file object and the underlying file pointer and file descriptor. The resulting object behaves as if it were newly opened.

**dup2***( fd)***  
Duplicate the file object and the underlying file pointer and file descriptor. The new object will have the given file descriptor. Otherwise the resulting object behaves as if it were newly opened.

**file**()  
Return the standard file object that the posixfile object is based on. This is sometimes necessary for functions that insist on a standard file object.
All methods raise `IOError` when the request fails.

Format characters for the `lock()` method have the following meaning:

<table>
<thead>
<tr>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>unlock the specified region</td>
</tr>
<tr>
<td>r</td>
<td>request a read lock for the specified section</td>
</tr>
<tr>
<td>w</td>
<td>request a write lock for the specified section</td>
</tr>
</tbody>
</table>

In addition the following modifiers can be added to the format:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>wait until the lock has been granted</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>return the first lock conflicting with the requested lock, or None if there is no conflict.</td>
<td>(1)</td>
</tr>
</tbody>
</table>

Note:

(1)

The lock returned is in the format `(mode, len, start, whence, pid)` where `mode` is a character representing the type of lock ('r' or 'w'). This modifier prevents a request from being granted; it is for query purposes only.

Format characters for the `flags()` method have the following meanings:

<table>
<thead>
<tr>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>append only flag</td>
</tr>
<tr>
<td>c</td>
<td>close on exec flag</td>
</tr>
<tr>
<td>n</td>
<td>no delay flag (also called non-blocking flag)</td>
</tr>
<tr>
<td>s</td>
<td>synchronization flag</td>
</tr>
</tbody>
</table>

In addition the following modifiers can be added to the format:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>turn the specified flags 'off', instead of the default 'on'</td>
<td>(1)</td>
</tr>
<tr>
<td>=</td>
<td>replace the flags, instead of the default 'OR' operation</td>
<td>(1)</td>
</tr>
</tbody>
</table>
? return a string in which the characters represent the flags that are set.

Notes:

(1) The "!" and "=" modifiers are mutually exclusive.

(2) This string represents the flags after they may have been altered by the same call.

Examples:

```python
import posixfile

file = posixfile.open('/tmp/test', 'w')
file.lock('w|')

file.lock('u')
file.close()
```

Release 2.5, documentation updated on 19th September, 2006.
See `About this document...` for information on suggesting changes.
16.13 resource -- Resource usage information

Availability: Unix.

This module provides basic mechanisms for measuring and controlling system resources utilized by a program.

Symbolic constants are used to specify particular system resources and to request usage information about either the current process or its children.

A single exception is defined for errors:

**exception error**

The functions described below may raise this error if the underlying system call failures unexpectedly.

Subsections

- 16.13.1 Resource Limits
- 16.13.2 Resource Usage
16.13.1 Resource Limits

Resources usage can be limited using the `setrlimit()` function described below. Each resource is controlled by a pair of limits: a soft limit and a hard limit. The soft limit is the current limit, and may be lowered or raised by a process over time. The soft limit can never exceed the hard limit. The hard limit can be lowered to any value greater than the soft limit, but not raised. (Only processes with the effective UID of the super-user can raise a hard limit.)

The specific resources that can be limited are system dependent. They are described in the `getrlimit(2)` man page. The resources listed below are supported when the underlying operating system supports them; resources which cannot be checked or controlled by the operating system are not defined in this module for those platforms.

`getrlimit(resource)`

Returns a tuple `(soft, hard)` with the current soft and hard limits of `resource`. Raises `ValueError` if an invalid resource is specified, or `error` if the underlying system call fails unexpectedly.

`setrlimit(resource, limits)`

Sets new limits of consumption of `resource`. The `limits` argument must be a tuple `(soft, hard)` of two integers describing the new limits. A value of `-1` can be used to specify the maximum possible upper limit.

Raises `ValueError` if an invalid resource is specified, if the new soft limit exceeds the hard limit, or if a process tries to raise its hard limit (unless the process has an effective UID of super-user). Can also raise `error` if the underlying system call fails.

These symbols define resources whose consumption can be controlled using the `setrlimit()` and `getrlimit()` functions described below. The values of these symbols are exactly the constants used by C programs.

The UNIX man page for `getrlimit(2)` lists the available resources. Note that not all systems use the same symbol or same value to denote the same resource. This module does not attempt to mask platform differences -- symbols not defined for
a platform will not be available from this module on that platform.

**RLIMIT_CORE**
The maximum size (in bytes) of a core file that the current process can create. This may result in the creation of a partial core file if a larger core would be required to contain the entire process image.

**RLIMIT_CPU**
The maximum amount of processor time (in seconds) that a process can use. If this limit is exceeded, a **SIGXCPU** signal is sent to the process. (See the [signal](#) module documentation for information about how to catch this signal and do something useful, e.g. flush open files to disk.)

**RLIMITFSIZE**
The maximum size of a file which the process may create. This only affects the stack of the main thread in a multi-threaded process.

**RLIMIT_DATA**
The maximum size (in bytes) of the process's heap.

**RLIMIT_STACK**
The maximum size (in bytes) of the call stack for the current process.

**RLIMIT_RSS**
The maximum resident set size that should be made available to the process.

**RLIMIT_NPROC**
The maximum number of processes the current process may create.

**RLIMIT_NOFILE**
The maximum number of open file descriptors for the current process.

**RLIMIT_OFILE**
The BSD name for **RLIMIT_NOFILE**.

**RLIMIT_MEMLOCK**
The maximum address space which may be locked in memory.

**RLIMIT_VMEM**
The largest area of mapped memory which the process may occupy.

**RLIMIT_AS**
The maximum area (in bytes) of address space which may be taken by the process.
16.13.2 Resource Usage

These functions are used to retrieve resource usage information:

getrusage( who)

This function returns an object that describes the resources consumed by either the current process or its children, as specified by the who parameter. The who parameter should be specified using one of the RUSAGE_ * constants described below.

The fields of the return value each describe how a particular system resource has been used, e.g. amount of time spent running is user mode or number of times the process was swapped out of main memory. Some values are dependent on the clock tick internal, e.g. the amount of memory the process is using.

For backward compatibility, the return value is also accessible as a tuple of 16 elements.

The fields ru_utime and ru_stime of the return value are floating point values representing the amount of time spent executing in user mode and the amount of time spent executing in system mode, respectively. The remaining values are integers. Consult the getrusage(2) man page for detailed information about these values. A brief summary is presented here:

<table>
<thead>
<tr>
<th>Index</th>
<th>Field</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ru_utime</td>
<td>time in user mode (float)</td>
</tr>
<tr>
<td>1</td>
<td>ru_stime</td>
<td>time in system mode (float)</td>
</tr>
<tr>
<td>2</td>
<td>ru_maxrss</td>
<td>maximum resident set size</td>
</tr>
<tr>
<td>3</td>
<td>ru_ixrss</td>
<td>shared memory size</td>
</tr>
<tr>
<td>4</td>
<td>ru_idrss</td>
<td>unshared memory size</td>
</tr>
<tr>
<td>5</td>
<td>ru_isrss</td>
<td>unshared stack size</td>
</tr>
<tr>
<td>6</td>
<td>ru_minflt</td>
<td>page faults not requiring I/O</td>
</tr>
<tr>
<td>7</td>
<td>ru_majflt</td>
<td>page faults requiring I/O</td>
</tr>
<tr>
<td>8</td>
<td>ru_nswap</td>
<td>number of swap outs</td>
</tr>
<tr>
<td>9</td>
<td>ru_inblock</td>
<td>block input operations</td>
</tr>
<tr>
<td></td>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>---</td>
<td>-------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>10</td>
<td>ru_oublock</td>
<td>block output operations</td>
</tr>
<tr>
<td>11</td>
<td>ru_msgsnd</td>
<td>messages sent</td>
</tr>
<tr>
<td>12</td>
<td>ru_msgrcv</td>
<td>messages received</td>
</tr>
<tr>
<td>13</td>
<td>ru_nsignals</td>
<td>signals received</td>
</tr>
<tr>
<td>14</td>
<td>ru_nvcsw</td>
<td>voluntary context switches</td>
</tr>
<tr>
<td>15</td>
<td>ru_nivcsw</td>
<td>involuntary context switches</td>
</tr>
</tbody>
</table>

This function will raise a `ValueError` if an invalid `who` parameter is specified. It may also raise `error` exception in unusual circumstances.

Changed in version 2.3: Added access to values as attributes of the returned object.

`getpagesize()`

Returns the number of bytes in a system page. (This need not be the same as the hardware page size.) This function is useful for determining the number of bytes of memory a process is using. The third element of the tuple returned by `getrusage()` describes memory usage in pages; multiplying by page size produces number of bytes.

The following `RUSAGE_`* symbols are passed to the `getrusage()` function to specify which processes information should be provided for.

**RUSAGE_SELF**

`RUSAGE_SELF` should be used to request information pertaining only to the process itself.

**RUSAGE_CHILDREN**

Pass to `getrusage()` to request resource information for child processes of the calling process.

**RUSAGE_BOTH**

Pass to `getrusage()` to request resources consumed by both the current process and child processes. May not be available on all systems.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
16.14 nis -- Interface to Sun's NIS (Yellow Pages)

Availability: UNIX.

The nis module gives a thin wrapper around the NIS library, useful for central administration of several hosts.

Because NIS exists only on UNIX systems, this module is only available for UNIX.

The nis module defines the following functions:

**match**(key, mapname[, domain=default_domain])

Return the match for key in map mapname, or raise an error (nis.error) if there is none. Both should be strings, key is 8-bit clean. Return value is an arbitrary array of bytes (may contain NULL and other joys).

Note that mapname is first checked if it is an alias to another name.

Changed in version 2.5: The domain argument allows to override the NIS domain used for the lookup. If unspecified, lookup is in the default NIS domain.

**cat**(mapname[, domain=default_domain])

Return a dictionary mapping key to value such that match(key, mapname)==value. Note that both keys and values of the dictionary are arbitrary arrays of bytes.

Note that mapname is first checked if it is an alias to another name.

Changed in version 2.5: The domain argument allows to override the NIS domain used for the lookup. If unspecified, lookup is in the default NIS domain.

**maps**( [domain=default_domain])

Return a list of all valid maps.
Changed in version 2.5: The domain argument allows to override the NIS domain used for the lookup. If unspecified, lookup is in the default NIS domain.

get_default_domain()

Return the system default NIS domain. New in version 2.5.

The nis module defines the following exception:

exception error

An error raised when a NIS function returns an error code.
16.15 syslog -- UNIX syslog library routines

Availability: Unix.

This module provides an interface to the UNIX syslog library routines. Refer to the UNIX manual pages for a detailed description of the syslog facility.

The module defines the following functions:

**syslog**([priority,] message)
Send the string message to the system logger. A trailing newline is added if necessary. Each message is tagged with a priority composed of a facility and a level. The optional priority argument, which defaults to LOG_INFO, determines the message priority. If the facility is not encoded in priority using logical-or (LOG_INFO | LOG_USER), the value given in the openlog() call is used.

**openlog**(ident[, logopt[, facility]])
Logging options other than the defaults can be set by explicitly opening the log file with openlog() prior to calling syslog(). The defaults are (usually) ident = 'syslog', logopt = 0, facility = LOG_USER. The ident argument is a string which is prepended to every message. The optional logopt argument is a string - see below for possible values to combine. The optional facility argument sets the default facility for messages which do not have a facility explicitly encoded.

**closelog**()
Close the log file.

**setlogmask**(maskpri)
Set the priority mask to maskpri and return the previous mask value. Calls to syslog() with a priority level not set in maskpri are ignored. The default is to log all priorities. The function LOG_MASK(pri) calculates the
mask for the individual priority \textit{pri}. The function \texttt{LOG\_UPTO(pri)} calculates the mask for all priorities up to and including \textit{pri}. 

The module defines the following constants:

**Priority levels (high to low):**

\begin{itemize}
  \item LOG\_EMERG
  \item LOG\_ALERT
  \item LOG\_CRIT
  \item LOG\_ERR
  \item LOG\_WARNING
  \item LOG\_NOTICE
  \item LOG\_INFO
  \item LOG\_DEBUG
\end{itemize}

**Facilities:**

\begin{itemize}
  \item LOG\_KERN
  \item LOG\_USER
  \item LOG\_MAIL
  \item LOG\_DAEMON
  \item LOG\_AUTH
  \item LOG\_LPR
  \item LOG\_NEWS
  \item LOG\_UUCP
  \item LOG\_CRON
  \item LOG\_LOCAL0 to LOG\_LOCAL7
\end{itemize}

**Log options:**

\begin{itemize}
  \item LOG\_PID
  \item LOG\_CONS
  \item LOG\_NDELAY
  \item LOG\_NOWAIT
  \item LOG\_PERROR
\end{itemize}

\textit{if} defined \textit{in} \texttt{<syslog.h>}. 

---

**Python Library Reference**


Release 2.5, documentation updated on 19th September, 2006.

See About this document... for information on suggesting changes.
16.16 commands -- Utilities for running commands

Availability: Unix.

The commands module contains wrapper functions for os.popen() which take a system command as a string and return any output generated by the command and, optionally, the exit status.

The commands module defines the following functions:

- **getstatusoutput**(cmd)
  Execute the string cmd in a shell with os.popen() and return a 2-tuple (status, output). cmd is actually run as { cmd ; } 2>&1, so that the returned output will contain output or error messages. A trailing newline is stripped from the output. The exit status for the command can be interpreted according to the rules for the C function wait().

- **getoutput**(cmd)
  Like getstatusoutput(), except the exit status is ignored and the return value is a string containing the command's output.

- **getstatus**(file)
  Return the output of "ls -l ld file" as a string. This function uses the getoutput() function, and properly escapes backslashes and dollar signs in the argument.

Example:

```python
>>> import commands
>>> commands.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> commands.getstatusoutput('cat /bin/junk')
(256, 'cat: /bin/junk: No such file or directory')
>>> commands.getstatusoutput('/bin/junk')
(256, 'sh: /bin/junk: not found')
>>> commands.getoutput('ls /bin/ls')
'/bin/ls'
```
>>> commands.getstatus('/bin/ls')
'-rwxr-xr-x  1 root  13352 Oct 14 1994 /bin/ls'

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
17. Interprocess Communication and Networking

The modules described in this chapter provide mechanisms for different processes to communicate.

Some modules only work for two processes that are on the same machine, e.g. signal and subprocess. Other modules support networking protocols that two or more processes can used to communicate across machines.

The list of modules described in this chapter is:

- subprocess: Subprocess management.
- socket: Low-level networking interface.
- signal: Set handlers for asynchronous events.
- popen2: Subprocesses with accessible standard I/O streams.
- asyncore: A base class for developing asynchronous socket handling services.
- asynchat: Support for asynchronous command/response protocols.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
17.1 subprocess -- Subprocess management

New in version 2.4.

The `subprocess` module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several other, older modules and functions, such as:

```python
os.system
os.spawn*
os.popen*
popen2.*
commands.*
```

Information about how the `subprocess` module can be used to replace these modules and functions can be found in the following sections.

---

**Subsections**

- 17.1.1 Using the subprocess Module
  - 17.1.1.1 Convenience Functions
  - 17.1.1.2 Exceptions
  - 17.1.1.3 Security
- 17.1.2 Popen Objects
- 17.1.3 Replacing Older Functions with the subprocess Module
  - 17.1.3.1 Replacing `/bin/sh` shell backquote
  - 17.1.3.2 Replacing shell pipe line
  - 17.1.3.3 Replacing `os.system()`
  - 17.1.3.4 Replacing `os.spawn*`
  - 17.1.3.5 Replacing `os.popen*`
  - 17.1.3.6 Replacing `popen2.*`

---

**Python Library Reference**

Previous: 17. Interprocess Communication and  
Up: 17. Interprocess Communication and  
Next: 17.1.1 Using the subprocess
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
17.1.1 Using the subprocess Module

This module defines one class called Popen:

```python
class Popen(
    args, bufsize=0, executable=None, stdin=None, stdout=None,
    stderr=None, preexec_fn=None, close_fds=False, shell=False,
    cwd=None, env=None, universal_newlines=False,
    startupinfo=None, creationflags=0
)
```

Arguments are:

- `args` should be a string, or a sequence of program arguments. The program to execute is normally the first item in the `args` sequence or string, but can be explicitly set by using the `executable` argument.

On **Unix**, with `shell=False` (default): In this case, the Popen class uses `os.execvp()` to execute the child program. `args` should normally be a sequence. A string will be treated as a sequence with the string as the only item (the program to execute).

On **Unix**, with `shell=True`: If `args` is a string, it specifies the command string to execute through the shell. If `args` is a sequence, the first item specifies the command string, and any additional items will be treated as additional shell arguments.

On Windows: the Popen class uses `CreateProcess()` to execute the child program, which operates on strings. If `args` is a sequence, it will be converted to a string using the `list2cmdline` method. Please note that not all MS Windows applications interpret the command line the same way: `list2cmdline` is designed for applications using the same rules as the MS C runtime.

`bufsize`, if given, has the same meaning as the corresponding argument to the built-in `open()` function: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size. A negative `bufsize` means to use the system default, which usually means fully buffered. The default value for `bufsize` is 0 (unbuffered).
The *executable* argument specifies the program to execute. It is very seldom needed: Usually, the program to execute is defined by the *args* argument. If *shell=True*, the *executable* argument specifies which shell to use. On UNIX, the default shell is /bin/sh. On Windows, the default shell is specified by the COMSPEC environment variable.

*stdin*, *stdout* and *stderr* specify the executed programs' standard input, standard output and standard error file handles, respectively. Valid values are PIPE, an existing file descriptor (a positive integer), an existing file object, and None. PIPE indicates that a new pipe to the child should be created. With None, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be STDOUT, which indicates that the stderr data from the applications should be captured into the same file handle as for stdout.

If preexec_fn is set to a callable object, this object will be called in the child process just before the child is executed. (UNIX only)

If close_fds is true, all file descriptors except 0, 1 and 2 will be closed before the child process is executed. (UNIX only)

If *shell* is True, the specified command will be executed through the shell.

If *cwd* is not None, the child's current directory will be changed to *cwd* before it is executed. Note that this directory is not considered when searching the executable, so you can't specify the program's path relative to *cwd*.

If *env* is not None, it defines the environment variables for the new process.

If universal_newlines is True, the file objects stdout and stderr are opened as text files, but lines may be terminated by any of '
', the UNIX end-of-line convention, '', the Macintosh convention or '
', the Windows convention. All of these external representations are seen as '
' by the Python program. **Note:** This feature is only available if Python is built with universal newline support (the default). Also, the newlines attribute of the file objects stdout, stdin and stderr are not updated by the communicate() method.
The `startupinfo` and `creationflags`, if given, will be passed to the underlying `CreateProcess()` function. They can specify things such as appearance of the main window and priority for the new process. (Windows only)

Subsections

- [17.1.1.1 Convenience Functions](#)
- [17.1.1.2 Exceptions](#)
- [17.1.1.3 Security](#)
17.1.2 Popen Objects

Instances of the Popen class have the following methods:

**poll()**
Check if child process has terminated. Returns returncode attribute.

**wait()**
Wait for child process to terminate. Returns returncode attribute.

**communicate( input=None)**
Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional input argument should be a string to be sent to the child process, or None, if no data should be sent to the child.

communicate() returns a tuple (stdout, stderr).

*Note:* The data read is buffered in memory, so do not use this method if the data size is large or unlimited.

The following attributes are also available:

**stdin**
If the stdin argument is PIPE, this attribute is a file object that provides input to the child process. Otherwise, it is None.

**stdout**
If the stdout argument is PIPE, this attribute is a file object that provides output from the child process. Otherwise, it is None.

**stderr**
If the stderr argument is PIPE, this attribute is file object that provides error output from the child process. Otherwise, it is None.

**pid**
The process ID of the child process.
**returncode**

The child return code. A `None` value indicates that the process hasn't terminated yet. A negative value -N indicates that the child was terminated by signal N (UNIX only).
Replacing /bin/sh shell
17.1.3 Replacing Older Functions with the subprocess Module

In this section, "a ==> b" means that b can be used as a replacement for a.

Note: All functions in this section fail (more or less) silently if the executed program cannot be found; this module raises an OSError exception.

In the following examples, we assume that the subprocess module is imported with "from subprocess import *".

Subsections

- 17.1.3.1 Replacing /bin/sh shell backquote
- 17.1.3.2 Replacing shell pipe line
- 17.1.3.3 Replacing os.system()
- 17.1.3.4 Replacing os.spawn*
- 17.1.3.5 Replacing os.popen*
- 17.1.3.6 Replacing popen2.*
17.2 socket -- Low-level networking interface

This module provides access to the BSD socket interface. It is available on all modern UNIX systems, Windows, MacOS, BeOS, OS/2, and probably additional platforms. Note: Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

For an introduction to socket programming (in C), see the following papers: *An Introductory 4.3BSD Interprocess Communication Tutorial*, by Stuart Sechrest and *An Advanced 4.3BSD Interprocess Communication Tutorial*, by Samuel J. Leffler et al, both in the *UNIX Programmer's Manual, Supplementary Documents 1* (sections PS1:7 and PS1:8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For UNIX, refer to the manual pages; for Windows, see the WinSock (or Winsock 2) specification. For IPv6-ready APIs, readers may want to refer to RFC 2553 titled *Basic Socket Interface Extensions for IPv6*.

The Python interface is a straightforward transliteration of the UNIX system call and library interface for sockets to Python's object-oriented style: the `socket()` function returns a `socket object` whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with `read()` and `write()` operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

Socket addresses are represented as follows: A single string is used for the AF_UNIX address family. A pair (host, port) is used for the AF_INET address family, where host is a string representing either a hostname in Internet domain notation like 'daring.cwi.nl' or an IPv4 address like '100.50.200.5', and port is an integral port number. For AF_INET6 address family, a four-tuple (host, port, flowinfo, scopeid) is used, where flowinfo and scopeid represents `sin6_flowinfo` and `sin6_scope_id` member in struct `sockaddr_in6` in C. For socket module methods, `flowinfo` and `scopeid` can be omitted just for backward compatibility. Note,
however, omission of scopeid can cause problems in manipulating scoped IPv6 addresses. Other address families are currently not supported. The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created.

For IPv4 addresses, two special forms are accepted instead of a host address: the empty string represents INADDR_ANY, and the string '<broadcast>' represents INADDR_BROADCAST. The behavior is not available for IPv6 for backward compatibility, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.

If you use a hostname in the host portion of IPv4/v6 socket address, the program may show a nondeterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in host portion.

New in version 2.5: AF_NETLINK sockets are represented as pairs pid, groups.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; errors related to socket or address semantics raise the error socket.error.

Non-blocking mode is supported through setblocking(). A generalization of this based on timeouts is supported through settimeout().

The module socket exports the following constants and functions:

exception error
This exception is raised for socket-related errors. The accompanying value is either a string telling what went wrong or a pair (errno, string) representing an error returned by a system call, similar to the value accompanying os.error. See the module errno, which contains names for the error codes defined by the underlying operating system.

exception herror
This exception is raised for address-related errors, i.e. for functions that use
h_errno in the C API, including gethostbyname_ex() and gethostbyaddr().

The accompanying value is a pair \((h\_errno, \ string)\) representing an error returned by a library call. \string represents the description of \(h\_errno\), as returned by the \texttt{hstrerror()} C function.

**exception gaierror**

This exception is raised for address-related errors, for getaddrinfo() and getnameinfo(). The accompanying value is a pair \((\text{error}, \ string)\) representing an error returned by a library call. \string represents the description of \text{error}, as returned by the \texttt{gai_strerror()} C function. The \text{error} value will match one of the \texttt{EAI_*} constants defined in this module.

**exception timeout**

This exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to settimeout(). The accompanying value is a string whose value is currently always `\texttt{\textasciitilde timed out}`. New in version 2.3.

**AF_UNIX**

**AF_INET**

**AF_INET6**

These constants represent the address (and protocol) families, used for the first argument to \texttt{socket()}. If the \texttt{AF_UNIX} constant is not defined then this protocol is unsupported.

**SOCK_STREAM**

**SOCK_DGRAM**

**SOCK_RAW**

**SOCK_RDM**

**SOCK_SEQPACKET**

These constants represent the socket types, used for the second argument to \texttt{socket()}. (Only \texttt{SOCK_STREAM} and \texttt{SOCK_DGRAM} appear to be generally useful.)

**SO_***

**SOMAXCONN**
Many constants of these forms, documented in the UNIX documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the setsockopt() and getsockopt() methods of socket objects. In most cases, only those symbols that are defined in the UNIX header files are defined; for a few symbols, default values are provided.

**has_ipv6**

This constant contains a boolean value which indicates if IPv6 is supported on this platform. New in version 2.3.

**getaddrinfo** (*host*, *port*[,...], *family*[,...], *socktype*[,...], *proto*[,...], *flags*[,...])

Resolves the *host/port* argument, into a sequence of 5-tuples that contain all the necessary argument for the sockets manipulation. *host* is a domain name, a string representation of IPv4/v6 address or None. *port* is a string service name (like 'http'), a numeric port number or None.

The rest of the arguments are optional and must be numeric if specified. For *host* and *port*, by passing either an empty string or None, you can pass NULL to the C API. The getaddrinfo() function returns a list of 5-tuples with the following structure:

```
(family, socktype, proto, canonnname, sockaddr)
```

*family*, *socktype*, *proto* are all integer and are meant to be passed to the socket() function. *canonnname* is a string representing the canonical name of the *host*. It can be a numeric IPv4/v6 address when
AI_CANONNAME is specified for a numeric host. sockaddr is a tuple describing a socket address, as described above. See the source for the httpplib and other library modules for a typical usage of the function. New in version 2.2.

getfqdn([name])
Return a fully qualified domain name for name. If name is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by gethostbyaddr() is checked, then aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available, the hostname as returned by gethostname() is returned. New in version 2.0.

gethostbyname( hostname)
Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as '100.50.200.5'. If the host name is an IPv4 address itself it is returned unchanged. See gethostbyname_ex() for a more complete interface. gethostbyname() does not support IPv6 name resolution, and getaddrinfo() should be used instead for IPv4/v6 dual stack support.

gethostbyname_ex( hostname)
Translate a host name to IPv4 address format, extended interface. Return a triple (hostname, aliaslist, ipaddrlist) where hostname is the primary host name responding to the given ip_address, aliaslist is a (possibly empty) list of alternative host names for the same address, and ipaddrlist is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). gethostbyname_ex() does not support IPv6 name resolution, and getaddrinfo() should be used instead for IPv4/v6 dual stack support.

gethostname()
Return a string containing the hostname of the machine where the Python interpreter is currently executing. If you want to know the current machine's IP address, you may want to use gethostbyname(gethostname()). This operation assumes that there is a valid address-to-host mapping for the host, and the assumption does not always hold. Note: gethostname()
doesn't always return the fully qualified domain name; use 
\texttt{gethostbyaddr(gethostname())} (see below).

\textbf{gethostbyaddr}( ip\_address) 
Return a triple \texttt{(hostname, aliaslist, ipaddrlist)} where \texttt{hostname} is the primary host name responding to the given \texttt{ip\_address}, \texttt{aliaslist} is a (possibly empty) list of alternative host names for the same address, and \texttt{ipaddrlist} is a list of IPv4/v6 addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function \texttt{getfqdn()}. \texttt{gethostbyaddr} supports both IPv4 and IPv6.

\textbf{getnameinfo}( sockaddr, flags) 
Translate a socket address \texttt{sockaddr} into a 2-tuple \texttt{(host, port)}. Depending on the settings of \texttt{flags}, the result can contain a fully-qualified domain name or numeric address representation in \texttt{host}. Similarly, \texttt{port} can contain a string port name or a numeric port number. New in version 2.2.

\textbf{getprotobynamel}( protocolname) 
Translate an Internet protocol name (for example, 'icmp') to a constant suitable for passing as the (optional) third argument to the \texttt{socket()} function. This is usually only needed for sockets opened in \texttt{``raw''} mode (SOCK\_RAW); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

\textbf{getservbyname}( servicename[, protocolname])
Translate an Internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be 'tcp' or 'udp', otherwise any protocol will match.

\textbf{getservbyport}( port[, protocolname])
Translate an Internet port number and protocol name to a service name for that service. The optional protocol name, if given, should be 'tcp' or 'udp', otherwise any protocol will match.

\textbf{socket}( [family[, type[, proto]]])
Create a new socket using the given address family, socket type and
protocol number. The address family should be AF_INET (the default), AF_INET6 or AF_UNIX. The socket type should be SOCK_STREAM (the default), SOCK_DGRAM or perhaps one of the other "SOCK_" constants. The protocol number is usually zero and may be omitted in that case.

**ssl**(sock[, keyfile, certfile])

Initiate a SSL connection over the socket **sock**. **keyfile** is the name of a PEM formatted file that contains your private key. **certfile** is a PEM formatted certificate chain file. On success, a new **SSLObject** is returned.

**Warning:** This does not do any certificate verification!

**socketpair**([[family[, type[, proto]]]])

Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the **socket**() function above. The default family is AF_UNIX if defined on the platform; otherwise, the default is AF_INET. Availability: **UNIX**. New in version 2.4.

**fromfd**(fd, family, type[, proto])

Duplicate the file descriptor **fd** (an integer as returned by a file object's **fileno**() method) and build a socket object from the result. Address family, socket type and protocol number are as for the **socket**() function above. The file descriptor should refer to a socket, but this is not checked -- subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the **UNIX** inet daemon). The socket is assumed to be in blocking mode. Availability: **UNIX**.

**ntohl**(x)

Convert 32-bit integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

**ntohs**(x)

Convert 16-bit integers from network to host byte order. On machines
where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

htonl\((x)\)

Convert 32-bit integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

htons\((x)\)

Convert 16-bit integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

inet_aton\((ip_string)\)

Convert an IPv4 address from dotted-quad string format (for example, '123.45.67.89') to 32-bit packed binary format, as a string four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type struct in_addr, which is the C type for the 32-bit packed binary this function returns.

If the IPv4 address string passed to this function is invalid, socket.error will be raised. Note that exactly what is valid depends on the underlying C implementation of inet_aton().

inet_aton() does not support IPv6, and getnameinfo() should be used instead for IPv4/v6 dual stack support.

inet_ntoa\((packed_ip)\)

Convert a 32-bit packed IPv4 address (a string four characters in length) to its standard dotted-quad string representation (for example, '123.45.67.89'). This is useful when conversing with a program that uses the standard C library and needs objects of type struct in_addr, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the string passed to this function is not exactly 4 bytes in length, socket.error will be raised. inet_ntoa() does not support IPv6, and getnameinfo() should be used instead for IPv4/v6 dual stack support.
**inet_pton**( address_family, ip_string)

Convert an IP address from its family-specific string format to a packed, binary format. `inet_pton()` is useful when a library or network protocol calls for an object of type `struct in_addr` (similar to `inet_aton()`) or `struct in6_addr`.

Supported values for `address_family` are currently AF_INET and AF_INET6. If the IP address string `ip_string` is invalid, `socket.error` will be raised. Note that exactly what is valid depends on both the value of `address_family` and the underlying implementation of `inet_pton()`.

Availability: UNIX (maybe not all platforms). New in version 2.3.

**inet_ntop**( address_family, packed_ip)

Convert a packed IP address (a string of some number of characters) to its standard, family-specific string representation (for example, '7.10.0.5' or '5aef:2b::8') `inet_ntop()` is useful when a library or network protocol returns an object of type `struct in_addr` (similar to `inet_ntoa()`) or `struct in6_addr`.

Supported values for `address_family` are currently AF_INET and AF_INET6. If the string `packed_ip` is not the correct length for the specified address family, `ValueError` will be raised. A `socket.error` is raised for errors from the call to `inet_ntop()`.

Availability: UNIX (maybe not all platforms). New in version 2.3.

**ge/defaulttimeout()**

Return the default timeout in floating seconds for new socket objects. A value of `None` indicates that new socket objects have no timeout. When the socket module is first imported, the default is `None`. New in version 2.3.

**setdefaulttimeout**( timeout)

Set the default timeout in floating seconds for new socket objects. A value of `None` indicates that new socket objects have no timeout. When the socket module is first imported, the default is `None`. New in version 2.3.

SocketType
This is a Python type object that represents the socket object type. It is the same as `type(socket(...))`.

**See Also:**

Module **SocketServer**: Classes that simplify writing network servers.

**Subsections**

- [17.2.1 Socket Objects](#)
- [17.2.2 SSL Objects](#)
- [17.2.3 Example](#)

*Release 2.5, documentation updated on 19th September, 2006.*

See [About this document...](#) for information on suggesting changes.
17.2.1 Socket Objects

Socket objects have the following methods. Except for `makefile()` these correspond to UNIX system calls applicable to sockets.

**accept()**
Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair `(conn, address)` where `conn` is a new socket object usable to send and receive data on the connection, and `address` is the address bound to the socket on the other end of the connection.

**bind( address)**
Bind the socket to `address`. The socket must not already be bound. (The format of `address` depends on the address family -- see above.) **Note:** This method has historically accepted a pair of parameters for AF_INET addresses instead of only a tuple. This was never intentional and is no longer available in Python 2.0 and later.

**close()**
Close the socket. All future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

**connect( address)**
Connect to a remote socket at `address`. (The format of `address` depends on the address family -- see above.) **Note:** This method has historically accepted a pair of parameters for AF_INET addresses instead of only a tuple. This was never intentional and is no longer available in Python 2.0 and later.

**connect_ex( address)**
Like `connect(address)`, but return an error indicator instead of raising an exception for errors returned by the C-level `connect()` call (other problems, such as `host not found`, can still raise exceptions). The error indicator is 0 if the operation succeeded, otherwise the value of the `errno`
variable. This is useful to support, for example, asynchronous connects.  

**Note:** This method has historically accepted a pair of parameters for 
`AF_INET` addresses instead of only a tuple. This was never intentional and 
is no longer available in Python 2.0 and later.

**fileno()**

Return the socket's file descriptor (a small integer). This is useful with 
`select.select()`.

Under Windows the small integer returned by this method cannot be used 
where a file descriptor can be used (such as `os.fdopen()`). UNIX does 
not have this limitation.

**getpeername()**

Return the remote address to which the socket is connected. This is useful 
to find out the port number of a remote IPv4/v6 socket, for instance. (The 
format of the address returned depends on the address family -- see above.) 
On some systems this function is not supported.

**getsockname()**

Return the socket's own address. This is useful to find out the port number 
of an IPv4/v6 socket, for instance. (The format of the address returned 
depends on the address family -- see above.)

**getsockopt( level, optname[, buflen])**

Return the value of the given socket option (see the UNIX man page 
`getsockopt(2)`). The needed symbolic constants (`SO_*` etc.) are defined in 
this module. If `buflen` is absent, an integer option is assumed and its integer 
value is returned by the function. If `buflen` is present, it specifies the 
maximum length of the buffer used to receive the option in, and this buffer 
is returned as a string. It is up to the caller to decode the contents of the 
buffer (see the optional built-in module `struct` for a way to decode C 
structures encoded as strings).

**listen( backlog)**

Listen for connections made to the socket. The `backlog` argument specifies 
the maximum number of queued connections and should be at least 1; the 
maximum value is system-dependent (usually 5).
**makefile**([`mode`, `bufsize`])

Return a file object associated with the socket. (File objects are described in 3.9, `File Objects.`) The file object references a dup() ped version of the socket file descriptor, so the file object and socket object may be closed or garbage-collected independently. The socket must be in blocking mode. The optional mode and bufsize arguments are interpreted the same way as by the built-in file() function; see `Built-in Functions` (section 2.1) for more information.

**`recv`**(`bufsize`, `flags`)

Receive data from the socket. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by bufsize. See the UNIX manual page recv(2) for the meaning of the optional argument flags; it defaults to zero. **Note:** For best match with hardware and network realities, the value of bufsize should be a relatively small power of 2, for example, 4096.

**`recvfrom`**(`bufsize`, `flags`)

Receive data from the socket. The return value is a pair (string, address) where string is a string representing the data received and address is the address of the socket sending the data. The optional flags argument has the same meaning as for recv() above. (The format of address depends on the address family -- see above.)

**`send`**(`string`, `flags`)

Send data to the socket. The socket must be connected to a remote socket. The optional flags argument has the same meaning as for recv() above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data.

**`sendall`**(`string`, `flags`)

Send data to the socket. The socket must be connected to a remote socket. The optional flags argument has the same meaning as for recv() above. Unlike send(), this method continues to send data from string until either all data has been sent or an error occurs. **None** is returned on success. On error, an exception is raised, and there is no way to determine how much
data, if any, was successfully sent.

sendto( string[, flags], address)
Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by address. The optional flags argument has the same meaning as for recv() above. Return the number of bytes sent. (The format of address depends on the address family -- see above.)

setblocking( flag)
Set blocking or non-blocking mode of the socket: if flag is 0, the socket is set to non-blocking, else to blocking mode. Initially all sockets are in blocking mode. In non-blocking mode, if a recv() call doesn't find any data, or if a send() call can't immediately dispose of the data, a error exception is raised; in blocking mode, the calls block until they can proceed. s.setblocking(0) is equivalent to s.settimeout(0); s.setblocking(1) is equivalent to s.settimeout(None).

settimeout( value)
Set a timeout on blocking socket operations. The value argument can be a nonnegative float expressing seconds, or None. If a float is given, subsequent socket operations will raise an timeout exception if the timeout period value has elapsed before the operation has completed. Setting a timeout of None disables timeouts on socket operations. s.settimeout(0.0) is equivalent to s.setblocking(0); s.settimeout(None) is equivalent to s.setblocking(1). New in version 2.3.

gettimeout()
Return the timeout in floating seconds associated with socket operations, or None if no timeout is set. This reflects the last call to setblocking() or settimeout(). New in version 2.3.

Some notes on socket blocking and timeouts: A socket object can be in one of three modes: blocking, non-blocking, or timeout. Sockets are always created in blocking mode. In blocking mode, operations block until complete. In non-blocking mode, operations fail (with an error that is unfortunately system-
dependent) if they cannot be completed immediately. In timeout mode, operations fail if they cannot be completed within the timeout specified for the socket. The setblocking() method is simply a shorthand for certain settimeout() calls.

Timeout mode internally sets the socket in non-blocking mode. The blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. A consequence of this is that file objects returned by the makefile() method must only be used when the socket is in blocking mode; in timeout or non-blocking mode file operations that cannot be completed immediately will fail.

Note that the connect() operation is subject to the timeout setting, and in general it is recommended to call settimeout() before calling connect().

`setsockopt(level, optname, value)`
Set the value of the given socket option (see the UNIX manual page `setsockopt(2)`). The needed symbolic constants are defined in the socket module (SO_* etc.). The value can be an integer or a string representing a buffer. In the latter case it is up to the caller to ensure that the string contains the proper bits (see the optional built-in module `struct` for a way to encode C structures as strings).

`shutdown(how)`
Shut down one or both halves of the connection. If `how` is SHUT_RD, further receives are disallowed. If `how` is SHUT_WR, further sends are disallowed. If `how` is SHUT_RDWR, further sends and receives are disallowed.

Note that there are no methods read() or write(); use recv() and send() without flags argument instead.

Socket objects also have these (read-only) attributes that correspond to the values given to the socket constructor.

`family`
The socket family. New in version 2.5.
type
The socket type. New in version 2.5.

proto
The socket protocol. New in version 2.5.
17.2.2 SSL Objects

SSL objects have the following methods.

**write(s)**
Writes the string $s$ to the on the object's SSL connection. The return value is the number of bytes written.

**read([n])**
If $n$ is provided, read $n$ bytes from the SSL connection, otherwise read until EOF. The return value is a string of the bytes read.

**server()**
Returns a string containing the ASN.1 distinguished name identifying the server's certificate. (See below for an example showing what distinguished names look like.)

**issuer()**
Returns a string containing the ASN.1 distinguished name identifying the issuer of the server's certificate.
17.2.3 Example

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence `socket()`, `bind()`, `listen()`, `accept()` (possibly repeating the `accept()` to service more than one client), while a client only needs the sequence `socket()`, `connect()`. Also note that the server does not `send()`/`recv()` on the socket it is listening on but on the new socket returned by `accept()`.

The first two examples support IPv4 only.

```python
# Echo server program
import socket

HOST = ''    # Symbolic name meaning the local host
PORT = 50007    # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()

# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007    # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)
```

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to
connect to the all addresses returned as a result of the name resolution, and sends
traffic to the first one connected successfully.

```python
# Echo server program
import socket
import sys

HOST = ''  # Symbolic name meaning the local host
PORT = 50007  # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error, msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except socket.error, msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print 'could not open socket'
s.close()
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()

# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'  # The remote host
PORT = 50007  # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error, msg:
        s = None
        continue
    try:
        s.connect(sa)
```
s.connect(sa)
except socket.error, msg:
    s.close()
    s = None
    continue
    break
if s is None:
    print 'could not open socket'
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)

This example connects to an SSL server, prints the server and issuer's distinguished names, sends some bytes, and reads part of the response:

import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('www.verisign.com', 443))
ssl_sock = socket.ssl(s)
print repr(ssl_sock.server())
print repr(ssl_sock.issuer())

# Set a simple HTTP request -- use httpplib in actual code.
ssl_sock.write('''GET / HTTP/1.0
Host: www.verisign.com
''
)

# Read a chunk of data. Will not necessarily
# read all the data returned by the server.
data = ssl_sock.read()

# Note that you need to close the underlying socket, not the SSL obj
del ssl_sock
s.close()

At this writing, this SSL example prints the following output (line breaks inserted for readability):

'/C=US/ST=California/L=Mountain View/
O=VeriSign, Inc./OU=Production Services/
OU=Terms of use at www.verisign.com/rpa (c)00/
CN=www.verisign.com'
'/O=VeriSign Trust Network/OU=VeriSign, Inc./
OU=VeriSign International Server CA - Class 3/
OU=www.verisign.com/CPS Incorp.by Ref. LABILITY LTD.(c)97 VeriSign
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
17.3 signal -- Set handlers for asynchronous events

This module provides mechanisms to use signal handlers in Python. Some general rules for working with signals and their handlers:

- A handler for a particular signal, once set, remains installed until it is explicitly reset (Python emulates the BSD style interface regardless of the underlying implementation), with the exception of the handler for SIGCHLD, which follows the underlying implementation.

- There is no way to "block" signals temporarily from critical sections (since this is not supported by all UNIX flavors).

- Although Python signal handlers are called asynchronously as far as the Python user is concerned, they can only occur between the "atomic" instructions of the Python interpreter. This means that signals arriving during long calculations implemented purely in C (such as regular expression matches on large bodies of text) may be delayed for an arbitrary amount of time.

- When a signal arrives during an I/O operation, it is possible that the I/O operation raises an exception after the signal handler returns. This is dependent on the underlying UNIX system's semantics regarding interrupted system calls.

- Because the C signal handler always returns, it makes little sense to catch synchronous errors like SIGFPE or SIGSEGV.

- Python installs a small number of signal handlers by default: SIGPIPE is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and SIGINT is translated into a KeyboardInterrupt exception. All of these can be overridden.

- Some care must be taken if both signals and threads are used in the same program. The fundamental thing to remember in using signals and threads
simultaneously is: always perform signal() operations in the main thread of execution. Any thread can perform an alarm(), getsignal(), or pause(); only the main thread can set a new signal handler, and the main thread will be the only one to receive signals (this is enforced by the Python signal module, even if the underlying thread implementation supports sending signals to individual threads). This means that signals can't be used as a means of inter-thread communication. Use locks instead.

The variables defined in the signal module are:

**SIG_DFL**
This is one of two standard signal handling options; it will simply perform the default function for the signal. For example, on most systems the default action for SIGQUIT is to dump core and exit, while the default action for SIGCLD is to simply ignore it.

**SIG_IGN**
This is another standard signal handler, which will simply ignore the given signal.

**SIG***
All the signal numbers are defined symbolically. For example, the hangup signal is defined as signal.SIGHUP; the variable names are identical to the names used in C programs, as found in <signal.h>. The UNIX man page for `signal()` lists the existing signals (on some systems this is signal(2), on others the list is in signal(7)). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

**NSIG**
One more than the number of the highest signal number.

The signal module defines the following functions:

**alarm(time)**
If time is non-zero, this function requests that a SIGALRM signal be sent to the process in time seconds. Any previously scheduled alarm is canceled (only one alarm can be scheduled at any time). The returned value is then
the number of seconds before any previously set alarm was to have been delivered. If time is zero, no alarm is scheduled, and any scheduled alarm is canceled. The return value is the number of seconds remaining before a previously scheduled alarm. If the return value is zero, no alarm is currently scheduled. (See the UNIX man page alarm(2).) Availability: UNIX.

getsignal(signalnum)
Return the current signal handler for the signal signalnum. The returned value may be a callable Python object, or one of the special values signal.SIG_IGN, signal.SIG_DFL or None. Here, signal.SIG_IGN means that the signal was previously ignored, signal.SIG_DFL means that the default way of handling the signal was previously in use, and None means that the previous signal handler was not installed from Python.

pause()
Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing. Not on Windows. (See the UNIX man page signal(2).)

signal(signalnum, handler)
Set the handler for signal signalnum to the function handler. handler can be a callable Python object taking two arguments (see below), or one of the special values signal.SIG_IGN or signal.SIG_DFL. The previous signal handler will be returned (see the description of getsignal() above). (See the UNIX man page signal(2).)

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a ValueError exception to be raised.

The handler is called with two arguments: the signal number and the current stack frame (None or a frame object; for a description of frame objects, see the reference manual section on the standard type hierarchy or see the attribute descriptions in the inspect module).

Subsections
• 17.3.1 Example

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
17.3.1 Example

Here is a minimal example program. It uses the `alarm()` function to limit the time spent waiting to open a file; this is useful if the file is for a serial device that may not be turned on, which would normally cause the `os.open()` to hang indefinitely. The solution is to set a 5-second alarm before opening the file; if the operation takes too long, the alarm signal will be sent, and the handler raises an exception.

```python
import signal, os

def handler(signum, frame):
    print 'Signal handler called with signal', signum
    raise IOError, "Couldn't open device!"

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)  # Disable the alarm
```

*Release 2.5, documentation updated on 19th September, 2006.*
*See [About this document...](http://example.com/about) for information on suggesting changes.*
17.3.1 Example Up: 17. Interprocess Communication and Next: 17.4.1 Popen3 and Popen4
17.4 popen2 -- Subprocesses with accessible I/O streams

Availability: Unix, Windows.

This module allows you to spawn processes and connect to their input/output/error pipes and obtain their return codes under UNIX and Windows.

Note that starting with Python 2.0, this functionality is available using functions from the os module which have the same names as the factory functions here, but the order of the return values is more intuitive in the os module variants.

The primary interface offered by this module is a trio of factory functions. For each of these, if bufsize is specified, it specifies the buffer size for the I/O pipes. mode, if provided, should be the string 'b' or 't'; on Windows this is needed to determine whether the file objects should be opened in binary or text mode. The default value for mode is 't'.

On UNIX, cmd may be a sequence, in which case arguments will be passed directly to the program without shell intervention (as with os.spawnv()). If cmd is a string it will be passed to the shell (as with os.system()).

The only way to retrieve the return codes for the child processes is by using the poll() or wait() methods on the Popen3 and Popen4 classes; these are only available on UNIX. This information is not available when using the popen2(), popen3(), and popen4() functions, or the equivalent functions in the os module. (Note that the tuples returned by the os module's functions are in a different order from the ones returned by the popen2 module.)

```python
popen2(cmd[, bufsize[, mode]])
```

Executes cmd as a sub-process. Returns the file objects (child_stdout, child_stdin).

```python
popen3(cmd[, bufsize[, mode]])
```

Executes cmd as a sub-process. Returns the file objects (child_stdout,
popen4( cmd[, bufsize[, mode]])

Executes cmd as a sub-process. Returns the file objects (child_stdout_and_stderr, child_stdin). New in version 2.0.

On UNIX, a class defining the objects returned by the factory functions is also available. These are not used for the Windows implementation, and are not available on that platform.

class Popen3( cmd[, capturestderr[, bufsize]])

This class represents a child process. Normally, Popen3 instances are created using the popen2() and popen3() factory functions described above.

If not using one of the helper functions to create Popen3 objects, the parameter cmd is the shell command to execute in a sub-process. The capturestderr flag, if true, specifies that the object should capture standard error output of the child process. The default is false. If the bufsize parameter is specified, it specifies the size of the I/O buffers to/from the child process.

class Popen4( cmd[, bufsize])

Similar to Popen3, but always captures standard error into the same file object as standard output. These are typically created using popen4(). New in version 2.0.
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
17.4.1 Popen3 and Popen4 Objects

Instances of the Popen3 and Popen4 classes have the following methods:

**poll()**
Returns -1 if child process hasn't completed yet, or its return code otherwise.

**wait()**
Waits for and returns the status code of the child process. The status code encodes both the return code of the process and information about whether it exited using the `exit()` system call or died due to a signal. Functions to help interpret the status code are defined in the `os` module; see section 14.1.5 for the W*( ) family of functions.

The following attributes are also available:

**fromchild**
A file object that provides output from the child process. For Popen4 instances, this will provide both the standard output and standard error streams.

**tochild**
A file object that provides input to the child process.

**childerr**
A file object that provides error output from the child process, if `capturestderr` was true for the constructor, otherwise None. This will always be None for Popen4 instances.

**pid**
The process ID of the child process.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
17.4.2 Flow Control Issues

Any time you are working with any form of inter-process communication, control flow needs to be carefully thought out. This remains the case with the file objects provided by this module (or the os module equivalents).

When reading output from a child process that writes a lot of data to standard error while the parent is reading from the child's standard output, a deadlock can occur. A similar situation can occur with other combinations of reads and writes. The essential factors are that more than _PC_PIPE_BUF bytes are being written by one process in a blocking fashion, while the other process is reading from the other process, also in a blocking fashion.

There are several ways to deal with this situation.

The simplest application change, in many cases, will be to follow this model in the parent process:

```python
import popen2
r, w, e = popen2.popen3('python slave.py')
e.readlines()
r.readlines()
r.close()
e.close()
w.close()
```

with code like this in the child:

```python
import os
import sys

# note that each of these print statements
# writes a single long string

print >>sys.stderr, 400 * 'this is a test
' os.close(sys.stderr.fileno())
print >>sys.stdout, 400 * 'this is another test
'
```

In particular, note that sys.stderr must be closed after writing all data, or readlines() won't return. Also note that os.close() must be used, as sys.stderr.close() won't close stderr (otherwise assigning to
sys.stderr will silently close it, so no further errors can be printed).

Applications which need to support a more general approach should integrate I/O over pipes with their select() loops, or use separate threads to read each of the individual files provided by whichever popen*() function or Popen* class was used.
17.5 asyncore -- Asynchronous socket handler

This module provides the basic infrastructure for writing asynchronous socket service clients and servers.

There are only two ways to have a program on a single processor do "more than one thing at a time." Multi-threaded programming is the simplest and most popular way to do it, but there is another very different technique, that lets you have nearly all the advantages of multi-threading, without actually using multiple threads. It's really only practical if your program is largely I/O bound. If your program is processor bound, then pre-emptive scheduled threads are probably what you really need. Network servers are rarely processor bound, however.

If your operating system supports the select() system call in its I/O library (and nearly all do), then you can use it to juggle multiple communication channels at once; doing other work while your I/O is taking place in the "background." Although this strategy can seem strange and complex, especially at first, it is in many ways easier to understand and control than multi-threaded programming. The asyncore module solves many of the difficult problems for you, making the task of building sophisticated high-performance network servers and clients a snap. For "conversational" applications and protocols the companion asyncchat module is invaluable.

The basic idea behind both modules is to create one or more network channels, instances of class asyncore.dispatcher and asyncchat.async_chat. Creating the channels adds them to a global map, used by the loop() function if you do not provide it with your own map.

Once the initial channel(s) is(are) created, calling the loop() function activates channel service, which continues until the last channel (including any that have been added to the map during asynchronous service) is closed.

```
loop([timeout[, use_poll[, map[,count]]]])
```
Enter a polling loop that terminates after count passes or all open channels have been closed. All arguments are optional. The \texttt{(count)} parameter defaults to None, resulting in the loop terminating only when all channels have been closed. The \texttt{timeout} argument sets the timeout parameter for the appropriate \texttt{select()} or \texttt{poll()} call, measured in seconds; the default is 30 seconds. The \texttt{use\_poll} parameter, if true, indicates that \texttt{poll()} should be used in preference to \texttt{select()} (the default is \texttt{False}).

The \texttt{map} parameter is a dictionary whose items are the channels to watch. As channels are closed they are deleted from their map. If \texttt{map} is omitted, a global map is used. Channels (instances of \texttt{asyncore.dispatcher}, \texttt{asynchat.async\_chat} and subclasses thereof) can freely be mixed in the map.

\textbf{class \texttt{dispatcher}()}  

The \texttt{dispatcher} class is a thin wrapper around a low-level socket object. To make it more useful, it has a few methods for event-handling which are called from the asynchronous loop. Otherwise, it can be treated as a normal non-blocking socket object.

Two class attributes can be modified, to improve performance, or possibly even to conserve memory.

\textbf{ac\_in\_buffer\_size}  
The asynchronous input buffer size (default 4096).

\textbf{ac\_out\_buffer\_size}  
The asynchronous output buffer size (default 4096).

The firing of low-level events at certain times or in certain connection states tells the asynchronous loop that certain higher-level events have taken place. For example, if we have asked for a socket to connect to another host, we know that the connection has been made when the socket becomes writable for the first time (at this point you know that you may write to it with the expectation of success). The implied higher-level events are:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{handle_connect()}</td>
<td>Implied by the first write event</td>
</tr>
<tr>
<td>\texttt{handle_close()}</td>
<td>Implied by a read event with no data available</td>
</tr>
</tbody>
</table>
handle_accept() | Implied by a read event on a listening socket

During asynchronous processing, each mapped channel's `readable()` and `writable()` methods are used to determine whether the channel's socket should be added to the list of channels `select()`ed or `poll()`ed for read and write events.

Thus, the set of channel events is larger than the basic socket events. The full set of methods that can be overridden in your subclass follows:

**handle_read()**
Called when the asynchronous loop detects that a `read()` call on the channel's socket will succeed.

**handle_write()**
Called when the asynchronous loop detects that a writable socket can be written. Often this method will implement the necessary buffering for performance. For example:

```python
    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]
```

**handle_expt()**
Called when there is out of band (OOB) data for a socket connection. This will almost never happen, as OOB is tenuously supported and rarely used.

**handle_connect()**
Called when the active opener's socket actually makes a connection. Might send a ``welcome'' banner, or initiate a protocol negotiation with the remote endpoint, for example.

**handle_close()**
Called when the socket is closed.

**handle_error()**
Called when an exception is raised and not otherwise handled. The default version prints a condensed traceback.
**handle_accept()**
Called on listening channels (passive openers) when a connection can be established with a new remote endpoint that has issued a `connect()` call for the local endpoint.

**readable()**
Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which read events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in read events.

**writable()**
Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which write events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in write events.

In addition, each channel delegates or extends many of the socket methods. Most of these are nearly identical to their socket partners.

**create_socket**(family, type)
This is identical to the creation of a normal socket, and will use the same options for creation. Refer to the `socket` documentation for information on creating sockets.

**connect**(address)
As with the normal socket object, `address` is a tuple with the first element the host to connect to, and the second the port number.

**send**(data)
Send `data` to the remote end-point of the socket.

**recv**(buffer_size)
Read at most `buffer_size` bytes from the socket's remote end-point. An empty string implies that the channel has been closed from the other end.

**listen**(backlog)
Listen for connections made to the socket. The *backlog* argument specifies the maximum number of queued connections and should be at least 1; the maximum value is system-dependent (usually 5).

**bind( address)**

Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family -- see above.)

**accept()**

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair (*conn, address*) where *conn* is a new socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection.

**close()**

Close the socket. All future operations on the socket object will fail. The remote end-point will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

---

**Subsections**

- [17.5.1 asyncore Example basic HTTP client](#)

---

RELEASE 2.5, DOCUMENTATION UPDATED ON 19TH SEPTEMBER, 2006.

SEE [ABOUT THIS DOCUMENT...](#) FOR INFORMATION ON SUGGESTING CHANGES.
17.5.1 asyncore Example basic HTTP client

Here is a very basic HTTP client that uses the dispatcher class to implement its socket handling:

```python
import asyncore, socket

class http_client(asyncore.dispatcher):
    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connect((host, 80))
        self.buffer = 'GET %s HTTP/1.0\r\n\r\n' % path

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print self.recv(8192)

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]

c = http_client('www.python.org', '/')
asyncore.loop()
```

Python Library Reference

Previous: 17.5 asyncore Up: 17.5 asyncore Next: 17.6 asynchat

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
17.6 asynchat -- Asynchronous socket command/response handler

This module builds on the asyncore infrastructure, simplifying asynchronous clients and servers and making it easier to handle protocols whose elements are terminated by arbitrary strings, or are of variable length. asynchat defines the abstract class async_chat that you subclass, providing implementations of the collect_incoming_data() and found_terminator() methods. It uses the same asynchronous loop as asyncore, and the two types of channel, asyncore.dispatcher and asyncchat.async_chat, can freely be mixed in the channel map. Typically an asyncore.dispatcher server channel generates new asyncchat.async_chat channel objects as it receives incoming connection requests.

class async_chat()
    This class is an abstract subclass of asyncore.dispatcher. To make practical use of the code you must subclass async_chat, providing meaningful collect_incoming_data() and found_terminator() methods. The asyncore.dispatcher methods can be used, although not all make sense in a message/response context.

    Like asyncore.dispatcher, async_chat defines a set of events that are generated by an analysis of socket conditions after a select() call. Once the polling loop has been started the async_chat object's methods are called by the event-processing framework with no action on the part of the programmer.

    Unlike asyncore.dispatcher, async_chat allows you to define a first-in-first-out queue (fifo) of producers. A producer need have only one method, more(), which should return data to be transmitted on the channel. The producer indicates exhaustion (i.e. that it contains no more data) by having its more() method return the empty string. At this point the async_chat object removes the producer from the fifo and starts using the next producer, if any. When the producer fifo is empty the
handle_write() method does nothing. You use the channel object's set_terminator() method to describe how to recognize the end of, or an important breakpoint in, an incoming transmission from the remote endpoint.

To build a functioning async_chat subclass your input methods collect_incoming_data() and found_terminator() must handle the data that the channel receives asynchronously. The methods are described below.

close_when_done()
Pushes a None on to the producer fifo. When this producer is popped off the fifo it causes the channel to be closed.

collect_incoming_data(data)
Called with data holding an arbitrary amount of received data. The default method, which must be overridden, raises a NotImplementedError exception.

discard_buffers()
In emergencies this method will discard any data held in the input and/or output buffers and the producer fifo.

found_terminator()
Called when the incoming data stream matches the termination condition set by set_terminator. The default method, which must be overridden, raises a NotImplementedError exception. The buffered input data should be available via an instance attribute.

get_terminator()
Returns the current terminator for the channel.

handle_close()
Called when the channel is closed. The default method silently closes the channel's socket.

handle_read()
Called when a read event fires on the channel's socket in the asynchronous loop. The default method checks for the termination condition established by set_terminator(), which can be either the appearance of a particular string in the input stream or the receipt of a particular number of characters. When the terminator is found, handle_read calls the found_terminator() method after calling collect_incoming_data() with any data preceding the terminating condition.

**handle_write()**

Called when the application may write data to the channel. The default method calls the initiate_send() method, which in turn will call refill_buffer() to collect data from the producer fifo associated with the channel.

**push(data)**

Creates a simple_producer object (see below) containing the data and pushes it on to the channel's producer_fifo to ensure its transmission. This is all you need to do to have the channel write the data out to the network, although it is possible to use your own producers in more complex schemes to implement encryption and chunking, for example.

**push_with_producer(producer)**

Takes a producer object and adds it to the producer fifo associated with the channel. When all currently-pushed producers have been exhausted the channel will consume this producer's data by calling its more() method and send the data to the remote endpoint.

**readable()**

Should return True for the channel to be included in the set of channels tested by the select() loop for readability.

**refill_buffer()**

Refills the output buffer by calling the more() method of the producer at the head of the fifo. If it is exhausted then the producer is popped off the fifo and the next producer is activated. If the current producer is, or becomes, None then the channel is closed.
**set_terminator**(term)
Sets the terminating condition to be recognised on the channel. term may be any of three types of value, corresponding to three different ways to handle incoming protocol data.

<table>
<thead>
<tr>
<th>term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Will call <code>found_terminator()</code> when the string is found in the input stream</td>
</tr>
<tr>
<td>integer</td>
<td>Will call <code>found_terminator()</code> when the indicated number of characters have been received</td>
</tr>
<tr>
<td>None</td>
<td>The channel continues to collect data forever</td>
</tr>
</tbody>
</table>

Note that any data following the terminator will be available for reading by the channel after `found_terminator()` is called.

**writable()**
Should return `True` as long as items remain on the producer fifo, or the channel is connected and the channel's output buffer is non-empty.

---

**Subsections**

- [17.6.1 asynchat - Auxiliary Classes and Functions](#)
- [17.6.2 asynchat Example](#)

---

**Python Library Reference**

Previous: [17.5.1 asyncore Example basic](#)  Up: [17. Interprocess Communication and](#)  Next: [17.6.1 asynchat - Auxiliary](#)

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
17.6.1 asynchat - Auxiliary Classes and Functions

**class simple_producer**( \(data[, buffer\_size=512]\))

A simple_producer takes a chunk of data and an optional buffer size. Repeated calls to its more() method yield successive chunks of the data no larger than buffer_size.

**more()**

Produces the next chunk of information from the producer, or returns the empty string.

**class fifo**( \([list=None]\))

Each channel maintains a fifo holding data which has been pushed by the application but not yet popped for writing to the channel. A fifo is a list used to hold data and/or producers until they are required. If the list argument is provided then it should contain producers or data items to be written to the channel.

**is_empty()**

Returns True iff the fifo is empty.

**first()**

Returns the least-recently push()ed item from the fifo.

**push**( \(data\))

Adds the given data (which may be a string or a producer object) to the producer fifo.

**pop()**

If the fifo is not empty, returns True, first(), deleting the popped item. Returns False, None for an empty fifo.

The asynchat module also defines one utility function, which may be of use in network and textual analysis operations.
find_prefix_at_end( haystack, needle)

Returns True if string haystack ends with any non-empty prefix of string needle.
17.6.2 asynchat Example

The following partial example shows how HTTP requests can be read with async_chat. A web server might create an `http_request_handler` object for each incoming client connection. Notice that initially the channel terminator is set to match the blank line at the end of the HTTP headers, and a flag indicates that the headers are being read.

Once the headers have been read, if the request is of type POST (indicating that further data are present in the input stream) then the `Content-Length:` header is used to set a numeric terminator to read the right amount of data from the channel.

The `handle_request()` method is called once all relevant input has been marshalled, after setting the channel terminator to `None` to ensure that any extraneous data sent by the web client are ignored.

```python
class http_request_handler(asynchat.async_chat):
    def __init__(self, conn, addr, sessions, log):
        asynchat.async_chat.__init__(self, conn=conn)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = ""
        self.set_terminator("\r\n\r\n")
        self.reading_headers = True
        self.handling = False
        self.cgi_data = None
        self.log = log

    def collect_incoming_data(self, self, data):
        """Buffer the data""
        self.ibuffer.append(data)

    def found_terminator(self):
        if self.reading_headers:
            self.reading_headers = False
            self.parse_headers("".join(self.ibuffer))
            self.ibuffer = []
            if self.op.upper() == "POST":
                clen = self.headers.getheader("content-length")
                self.set_terminator(int(clen))
            else:
                self.handling = True
```
self.set_terminator(None)
self.handle_request()

elif not self.handling:
    self.set_terminator(None)  # browsers sometimes over-send
    self.cgi_data = parse(self.headers, ".join(self.ibuffer
    self.handling = True
    self.ibuffer = []
    self.handle_request()
The modules described in this chapter implement Internet protocols and support for related technology. They are all implemented in Python. Most of these modules require the presence of the system-dependent module `socket`, which is currently supported on most popular platforms. Here is an overview:

**webbrowser**
Easy-to-use controller for Web browsers.

**cgi**
Common Gateway Interface support, used to interpret forms in server-side scripts.

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

**wsgiref**
WSGI Utilities and Reference Implementation

**wsgiref.util**

**wsgiref.headers**

**wsgiref.simple_server**

**wsgiref.validate**

**wsgiref.handlers**

**urllib**
Open an arbitrary network resource by URL (requires sockets).

**urllib2**
An extensible library for opening URLs using a variety of protocols

**httplib**
HTTP and HTTPS protocol client (requires sockets).

**ftplib**
FTP protocol client (requires sockets).

**gopherlib**
Gopher protocol client (requires sockets).

**poplib**
POP3 protocol client (requires sockets).

**imaplib**
IMAP4 protocol client (requires sockets).

**nntplib**
NNTP protocol client (requires sockets).

**smtplib**
SMTP protocol client (requires sockets).

**smtpd**
Implement a flexible SMTP server

**telnetlib**
Telnet client class.

UUID objects (universally unique identifiers)
**uuid**

according to RFC 4122

**urlparse**

Parse URLs into components.

**SocketServer**

A framework for network servers.

**BaseHTTPServer**

Basic HTTP server (base class for SimpleHTTPServer and CGIHTTPServer).

**SimpleHTTPServer**

This module provides a basic request handler for HTTP servers.

**CGIHTTPServer**

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

**cookielib**

Cookie handling for HTTP clients

**Cookie**

Support for HTTP state management (cookies).

**xmlrpclib**

XML-RPC client access.

**SimpleXMLRPCServer**

Basic XML-RPC server implementation.

**DocXMLRPCServer**

Self-documenting XML-RPC server implementation.
18. Internet Protocols and

18.1.1 Browser Controller Objects
18.1 webbrowser -- Convenient Web-browser controller

The **webbrowser** module provides a high-level interface to allow displaying Web-based documents to users. Under most circumstances, simply calling the `open()` function from this module will do the right thing.

Under **UNIX**, graphical browsers are preferred under X11, but text-mode browsers will be used if graphical browsers are not available or an X11 display isn't available. If text-mode browsers are used, the calling process will block until the user exits the browser.

If the environment variable `BROWSER` exists, it is interpreted to override the platform default list of browsers, as a `os.pathsep-separated` list of browsers to try in order. When the value of a list part contains the string `%s`, then it is interpreted as a literal browser command line to be used with the argument URL substituted for `%s`; if the part does not contain `%s`, it is simply interpreted as the name of the browser to launch.

For non-**UNIX** platforms, or when a remote browser is available on **UNIX**, the controlling process will not wait for the user to finish with the browser, but allow the remote browser to maintain its own windows on the display. If remote browsers are not available on **UNIX**, the controlling process will launch a new browser and wait.

The script **webbrowser** can be used as a command-line interface for the module. It accepts an URL as the argument. It accepts the following optional parameters: `-n` opens the URL in a new browser window, if possible; `-t` opens the URL in a new browser page ("tab"). The options are, naturally, mutually exclusive.

The following exception is defined:

**exception Error**

   Exception raised when a browser control error occurs.

The following functions are defined:
**open**(*url[, new=0[, autoraise=1]]*)

Display *url* using the default browser. If *new* is 0, the *url* is opened in the same browser window. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page ("tab") is opened if possible. If *autoraise* is true, the window is raised if possible (note that under many window managers this will occur regardless of the setting of this variable). Changed in version 2.5: *new* can now be 2.

**open_new**(*url*)

Open *url* in a new window of the default browser, if possible, otherwise, open *url* in the only browser window.

**open_new_tab**(*url*)

Open *url* in a new page ("tab") of the default browser, if possible, otherwise equivalent to **open_new**. New in version 2.5.

**get**([*name]*)

Return a controller object for the browser type *name*. If *name* is empty, return a controller for a default browser appropriate to the caller's environment.

**register**(*name, constructor[, instance]*)

Register the browser type *name*. Once a browser type is registered, the **get()** function can return a controller for that browser type. If *instance* is not provided, or is *None*, *constructor* will be called without parameters to create an instance when needed. If *instance* is provided, *constructor* will never be called, and may be *None*.

This entry point is only useful if you plan to either set the BROWSER variable or call **get** with a nonempty argument matching the name of a handler you declare.

A number of browser types are predefined. This table gives the type names that may be passed to the **get()** function and the corresponding instantiations for the controller classes, all defined in this module.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Class Name</th>
<th>Notes</th>
</tr>
</thead>
</table>


| 'mozilla'           | Mozilla('mozilla')               |
| 'firefox'           | Mozilla('mozilla')               |
| 'netscape'          | Mozilla('netscape')              |
| 'galeon'            | Galeon('galeon')                 |
| 'epiphany'          | Galeon('epiphany')               |
| 'skipstone'         | BackgroundBrowser('skipstone')   |
| 'kfmclient'         | Konqueror()                      |
| 'konqueror'         | Konqueror()                      |
| 'kfm'               | Konqueror()                      |
| 'mosaic'            | BackgroundBrowser('mosaic')      |
| 'opera'             | Opera()                          |
| 'grail'             | Grail()                          |
| 'links'             | GenericBrowser('links')          |
| 'elinks'            | Elinks('elinks')                 |
| 'lynx'              | GenericBrowser('lynx')           |
| 'w3m'               | GenericBrowser('w3m')            |
| 'windows-default'   | WindowsDefault                   |
| 'internet-config'   | InternetConfig                   |
| 'macosx'            | MacOSX('default')                |

Notes:

(1) "Konqueror" is the file manager for the KDE desktop environment for UNIX, and only makes sense to use if KDE is running. Some way of reliably detecting KDE would be nice; the KDEDIR variable is not sufficient. Note also that the name "kfm" is used even when using the konqueror command with KDE 2 -- the implementation selects the best strategy for running Konqueror.

(2) Only on Windows platforms.

(3)
Only on MacOS platforms; requires the standard MacPython ic module, described in the *Macintosh Library Modules* manual.

(4)

Only on MacOS X platform.

Here are some simple examples:

```python
url = 'http://www.python.org'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url + '/doc')

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

---

**Subsections**

- [18.1.1 Browser Controller Objects](#)

---

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
18.1.1 Browser Controller Objects

Browser controllers provide two methods which parallel two of the module-level convenience functions:

**open** *(url[, new[, autoraise=1]])*
Display *url* using the browser handled by this controller. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page ("tab") is opened if possible.

**open_new** *(url)*
Open *url* in a new window of the browser handled by this controller, if possible, otherwise, open *url* in the only browser window. Alias open_new.

**open_new_tab** *(url)*
Open *url* in a new page ("tab") of the browser handled by this controller, if possible, otherwise equivalent to open_new. New in version 2.5.
18.2 cgi -- Common Gateway Interface support.

Support module for Common Gateway Interface (CGI) scripts.

This module defines a number of utilities for use by CGI scripts written in Python.

Subsections

- 18.2.1 Introduction
- 18.2.2 Using the cgi module
- 18.2.3 Higher Level Interface
- 18.2.4 Old classes
- 18.2.5 Functions
- 18.2.6 Caring about security
- 18.2.7 Installing your CGI script on a UNIX system
- 18.2.8 Testing your CGI script
- 18.2.9 Debugging CGI scripts
- 18.2.10 Common problems and solutions
18.2.1 Introduction

A CGI script is invoked by an HTTP server, usually to process user input submitted through an HTML <FORM> or <ISINDEX> element.

Most often, CGI scripts live in the server's special cgi-bin directory. The HTTP server places all sorts of information about the request (such as the client's hostname, the requested URL, the query string, and lots of other goodies) in the script's shell environment, executes the script, and sends the script's output back to the client.

The script's input is connected to the client too, and sometimes the form data is read this way; at other times the form data is passed via the ``query string'' part of the URL. This module is intended to take care of the different cases and provide a simpler interface to the Python script. It also provides a number of utilities that help in debugging scripts, and the latest addition is support for file uploads from a form (if your browser supports it).

The output of a CGI script should consist of two sections, separated by a blank line. The first section contains a number of headers, telling the client what kind of data is following. Python code to generate a minimal header section looks like this:

```python
print "Content-Type: text/html"  # HTML is following
print                        # blank line, end of headers
```

The second section is usually HTML, which allows the client software to display nicely formatted text with header, in-line images, etc. Here's Python code that prints a simple piece of HTML:

```python
print "<TITLE>CGI script output</TITLE>"
print "<H1>This is my first CGI script</H1>"
print "Hello, world!"
```
18.2.2 Using the cgi module

Begin by writing "import cgi". Do not use "fromcgiimport *" -- the module defines all sorts of names for its own use or for backward compatibility that you don't want in your namespace.

When you write a new script, consider adding the line:

```python
import cgitb; cgitb.enable()
```

This activates a special exception handler that will display detailed reports in the Web browser if any errors occur. If you'd rather not show the guts of your program to users of your script, you can have the reports saved to files instead, with a line like this:

```python
import cgitb; cgitb.enable(display=0, logdir="/tmp")
```

It's very helpful to use this feature during script development. The reports produced by cgitb provide information that can save you a lot of time in tracking down bugs. You can always remove the cgitb line later when you have tested your script and are confident that it works correctly.

To get at submitted form data, it's best to use the FieldStorage class. The other classes defined in this module are provided mostly for backward compatibility. Instantiate it exactly once, without arguments. This reads the form contents from standard input or the environment (depending on the value of various environment variables set according to the CGI standard). Since it may consume standard input, it should be instantiated only once.

The FieldStorage instance can be indexed like a Python dictionary, and also supports the standard dictionary methods has_key() and keys(). The built-in len() is also supported. Form fields containing empty strings are ignored and do not appear in the dictionary; to keep such values, provide a true value for the optional keep_blank_values keyword parameter when creating the FieldStorage instance.

For instance, the following code (which assumes that the Content-Type: header and blank line have already been printed) checks that the fields name and addr
are both set to a non-empty string:

```python
form = cgi.FieldStorage()
if not (form.has_key("name") and form.has_key("addr")):
    print "<H1>Error</H1>"
    print "Please fill in the name and addr fields."
    return
print "<p>name: ", form["name"].value
print "<p>addr: ", form["addr"].value
...further form processing here...
```

Here the fields, accessed through "form[key]", are themselves instances of `FieldStorage` (or `MiniFieldStorage`, depending on the form encoding). The `value` attribute of the instance yields the string value of the field. The `getvalue()` method returns this string value directly; it also accepts an optional second argument as a default to return if the requested key is not present.

If the submitted form data contains more than one field with the same name, the object retrieved by "form[key]" is not a `FieldStorage` or `MiniFieldStorage` instance but a list of such instances. Similarly, in this situation, "form.getvalue(key)" would return a list of strings. If you expect this possibility (when your HTML form contains multiple fields with the same name), use the `getlist()` function, which always returns a list of values (so that you do not need to special-case the single item case). For example, this code concatenates any number of username fields, separated by commas:

```python
value = form.getlist("username")
usernames = ",".join(value)
```

If a field represents an uploaded file, accessing the value via the `value` attribute or the `getvalue()` method reads the entire file in memory as a string. This may not be what you want. You can test for an uploaded file by testing either the `filename` attribute or the `file` attribute. You can then read the data at leisure from the `file` attribute:

```python
fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
    linecount = 0
    while 1:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```
The file upload draft standard entertains the possibility of uploading multiple files from one field (using a recursive multipart/* encoding). When this occurs, the item will be a dictionary-like FieldStorage item. This can be determined by testing its type attribute, which should be multipart/form-data (or perhaps another MIME type matching multipart/*). In this case, it can be iterated over recursively just like the top-level form object.

When a form is submitted in the `"old"` format (as the query string or as a single data part of type application/x-www-form-urlencoded), the items will actually be instances of the class MiniFieldStorage. In this case, the list, file, and filename attributes are always None.
18.2.3 Higher Level Interface

New in version 2.2.

The previous section explains how to read CGI form data using the FieldStorage class. This section describes a higher level interface which was added to this class to allow one to do it in a more readable and intuitive way. The interface doesn't make the techniques described in previous sections obsolete -- they are still useful to process file uploads efficiently, for example.

The interface consists of two simple methods. Using the methods you can process form data in a generic way, without the need to worry whether only one or more values were posted under one name.

In the previous section, you learned to write following code anytime you expected a user to post more than one value under one name:

```python
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

This situation is common for example when a form contains a group of multiple checkboxes with the same name:

```html
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

In most situations, however, there's only one form control with a particular name in a form and then you expect and need only one value associated with this name. So you write a script containing for example this code:

```python
user = form.getvalue("user").upper()
```

The problem with the code is that you should never expect that a client will provide valid input to your scripts. For example, if a curious user appends another "user=foo" pair to the query string, then the script would crash, because in this situation the getvalue("user") method call returns a list instead of a string. Calling the toupper() method on a list is not valid (since
lists do not have a method of this name) and results in an AttributeError exception.

Therefore, the appropriate way to read form data values was to always use the code which checks whether the obtained value is a single value or a list of values. That's annoying and leads to less readable scripts.

A more convenient approach is to use the methods getfirst() and getlist() provided by this higher level interface.

**getfirst**(name[, default])

This method always returns only one value associated with form field *name*. The method returns only the first value in case that more values were posted under such name. Please note that the order in which the values are received may vary from browser to browser and should not be counted on.\footnote{18.1} If no such form field or value exists then the method returns the value specified by the optional parameter default. This parameter defaults to None if not specified.

**getlist**(name)

This method always returns a list of values associated with form field *name*. The method returns an empty list if no such form field or value exists for name. It returns a list consisting of one item if only one such value exists.

Using these methods you can write nice compact code:

```python
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", ").upper()  # This way it's safe.
for item in form.getlist("item"):  
do_something(item)
```

---

**Footnotes**

... on.\footnote{18.1}

Note that some recent versions of the HTML specification do state what order the field values should be supplied in, but knowing whether a request was received from a conforming browser, or even from a browser at all, is
tedious and error-prone.
18.2.4 Old classes

These classes, present in earlier versions of the cgi module, are still supported for backward compatibility. New applications should use the FieldStorage class.

SvFormContentDict stores single value form content as dictionary; it assumes each field name occurs in the form only once.

FormContentDict stores multiple value form content as a dictionary (the form items are lists of values). Useful if your form contains multiple fields with the same name.

Other classes (FormContent, InterpFormContentDict) are present for backwards compatibility with really old applications only. If you still use these and would be inconvenienced when they disappeared from a next version of this module, drop me a note.
18.2.5 Functions

These are useful if you want more control, or if you want to employ some of the algorithms implemented in this module in other circumstances.

**parse** *(fp[, keep_blank_values[, strict_parsing]])*

Parse a query in the environment or from a file (the file defaults to `sys.stdin`). The `keep_blank_values` and `strict_parsing` parameters are passed to `parse_qs()` unchanged.

**parse_qs** *(qs[, keep_blank_values[, strict_parsing]])*

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument `keep_blank_values` is a flag indicating whether blank values in URL encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument `strict_parsing` is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

Use the `urllib.urlencode()` function to convert such dictionaries into query strings.

**parse_qsl** *(qs[, keep_blank_values[, strict_parsing]])*

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a list of name, value pairs.

The optional argument `keep_blank_values` is a flag indicating whether blank values in URL encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated
as if they were not included.

The optional argument strict_parsing is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a ValueError exception.

Use the urllib.urlencode() function to convert such lists of pairs into query strings.

parse_multipart(fp, pdict)
Parse input of type multipart/form-data (for file uploads). Arguments are fp for the input file and pdict for a dictionary containing other parameters in the Content-Type: header.

Returns a dictionary just like parse_qs() keys are the field names, each value is a list of values for that field. This is easy to use but not much good if you are expecting megabytes to be uploaded -- in that case, use the FieldStorage class instead which is much more flexible.

Note that this does not parse nested multipart parts -- use FieldStorage for that.

parse_header(string)
Parse a MIME header (such as Content-Type:) into a main value and a dictionary of parameters.

test()
Robust test CGI script, usable as main program. Writes minimal HTTP headers and formats all information provided to the script in HTML form.

print_environ()
Format the shell environment in HTML.

print_form(form)
Format a form in HTML.

print_directory()
Format the current directory in HTML.
print_environ_usage()
Print a list of useful (used by CGI) environment variables in HTML.

escape(s[, quote])
Convert the characters "&", "<" and ">" in string s to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag quote is true, the quotation mark character ("""") is also translated; this helps for inclusion in an HTML attribute value, as in <A HREF="...">. If the value to be quoted might include single- or double-quote characters, or both, consider using the quoteattr() function in the xml.sax.saxutils module instead.
18.2.6 Caring about security

There's one important rule: if you invoke an external program (via the `os.system()` or `os.popen()` functions, or others with similar functionality), make very sure you don't pass arbitrary strings received from the client to the shell. This is a well-known security hole whereby clever hackers anywhere on the Web can exploit a gullible CGI script to invoke arbitrary shell commands. Even parts of the URL or field names cannot be trusted, since the request doesn't have to come from your form!

To be on the safe side, if you must pass a string gotten from a form to a shell command, you should make sure the string contains only alphanumeric characters, dashes, underscores, and periods.
18.2.7 Installing your CGI script on a UNIX system

Read the documentation for your HTTP server and check with your local system administrator to find the directory where CGI scripts should be installed; usually this is in a directory cgi-bin in the server tree.

Make sure that your script is readable and executable by `"others"`; the UNIX file mode should be `0755` octal (use `"chmod 0755 filename"`). Make sure that the first line of the script contains `#!` starting in column 1 followed by the pathname of the Python interpreter, for instance:

```
#!/usr/local/bin/python
```

Make sure the Python interpreter exists and is executable by `"others"`.

Make sure that any files your script needs to read or write are readable or writable, respectively, by `"others"` -- their mode should be `0644` for readable and `0666` for writable. This is because, for security reasons, the HTTP server executes your script as user `"nobody"`, without any special privileges. It can only read (write, execute) files that everybody can read (write, execute). The current directory at execution time is also different (it is usually the server's cgi-bin directory) and the set of environment variables is also different from what you get when you log in. In particular, don't count on the shell's search path for executables (PATH) or the Python module search path (PYTHONPATH) to be set to anything interesting.

If you need to load modules from a directory which is not on Python's default module search path, you can change the path in your script, before importing other modules. For example:

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(This way, the directory inserted last will be searched first!)

Instructions for non-UNIX systems will vary; check your HTTP server's documentation (it will usually have a section on CGI scripts).
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.2.8 Testing your CGI script

Unfortunately, a CGI script will generally not run when you try it from the command line, and a script that works perfectly from the command line may fail mysteriously when run from the server. There's one reason why you should still test your script from the command line: if it contains a syntax error, the Python interpreter won't execute it at all, and the HTTP server will most likely send a cryptic error to the client.

Assuming your script has no syntax errors, yet it does not work, you have no choice but to read the next section.
18.2.9 Debugging CGI scripts

First of all, check for trivial installation errors -- reading the section above on installing your CGI script carefully can save you a lot of time. If you wonder whether you have understood the installation procedure correctly, try installing a copy of this module file (cgi.py) as a CGI script. When invoked as a script, the file will dump its environment and the contents of the form in HTML form. Give it the right mode etc, and send it a request. If it's installed in the standard cgi-bin directory, it should be possible to send it a request by entering a URL into your browser of the form:

   http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home

If this gives an error of type 404, the server cannot find the script - perhaps you need to install it in a different directory. If it gives another error, there's an installation problem that you should fix before trying to go any further. If you get a nicely formatted listing of the environment and form content (in this example, the fields should be listed as ``addr'' with value ``At Home'' and ``name'' with value ``Joe Blow''), the cgi.py script has been installed correctly. If you follow the same procedure for your own script, you should now be able to debug it.

The next step could be to call the cgi module's test() function from your script: replace its main code with the single statement

  cgi.test()

This should produce the same results as those gotten from installing the cgi.py file itself.

When an ordinary Python script raises an unhandled exception (for whatever reason: of a typo in a module name, a file that can't be opened, etc.), the Python interpreter prints a nice traceback and exits. While the Python interpreter will still do this when your CGI script raises an exception, most likely the traceback will end up in one of the HTTP server's log files, or be discarded altogether.

Fortunately, once you have managed to get your script to execute some code, you can easily send tracebacks to the Web browser using the cgitb module. If you
haven't done so already, just add the line:

```python
import cgitb; cgitb.enable()
```

to the top of your script. Then try running it again; when a problem occurs, you should see a detailed report that will likely make apparent the cause of the crash.

If you suspect that there may be a problem in importing the `cgitb` module, you can use an even more robust approach (which only uses built-in modules):

```python
import sys
sys.stderr = sys.stdout
print "Content-Type: text/plain"
print
...your code here...
```

This relies on the Python interpreter to print the traceback. The content type of the output is set to plain text, which disables all HTML processing. If your script works, the raw HTML will be displayed by your client. If it raises an exception, most likely after the first two lines have been printed, a traceback will be displayed. Because no HTML interpretation is going on, the traceback will be readable.
18.2.10 Common problems and solutions

- Most HTTP servers buffer the output from CGI scripts until the script is completed. This means that it is not possible to display a progress report on the client's display while the script is running.

- Check the installation instructions above.

- Check the HTTP server's log files. ("tail -f logfile" in a separate window may be useful!)

- Always check a script for syntax errors first, by doing something like "python script.py".

- If your script does not have any syntax errors, try adding "import cgitb; cgitb.enable()" to the top of the script.

- When invoking external programs, make sure they can be found. Usually, this means using absolute path names -- PATH is usually not set to a very useful value in a CGI script.

- When reading or writing external files, make sure they can be read or written by the userid under which your CGI script will be running: this is typically the userid under which the web server is running, or some explicitly specified userid for a web server's "suexec" feature.

- Don't try to give a CGI script a set-uid mode. This doesn't work on most systems, and is a security liability as well.
18.3 cgitb -- Traceback manager for CGI scripts

New in version 2.2.

The `cgitb` module provides a special exception handler for Python scripts. (Its name is a bit misleading. It was originally designed to display extensive traceback information in HTML for CGI scripts. It was later generalized to also display this information in plain text.) After this module is activated, if an uncaught exception occurs, a detailed, formatted report will be displayed. The report includes a traceback showing excerpts of the source code for each level, as well as the values of the arguments and local variables to currently running functions, to help you debug the problem. Optionally, you can save this information to a file instead of sending it to the browser.

To enable this feature, simply add one line to the top of your CGI script:

```
import cgitb; cgitb.enable()
```

The options to the `enable()` function control whether the report is displayed in the browser and whether the report is logged to a file for later analysis.

```
enable([display[, logdir[, context[, format]]]])
```

This function causes the `cgitb` module to take over the interpreter's default handling for exceptions by setting the value of `sys.excepthook`.

The optional argument `display` defaults to 1 and can be set to 0 to suppress sending the traceback to the browser. If the argument `logdir` is present, the traceback reports are written to files. The value of `logdir` should be a directory where these files will be placed. The optional argument `context` is the number of lines of context to display around the current line of source code in the traceback; this defaults to 5. If the optional argument `format` is "html", the output is formatted as HTML. Any other value forces plain text output. The default value is "html".

```
handler([info])
```
This function handles an exception using the default settings (that is, show a report in the browser, but don't log to a file). This can be used when you've caught an exception and want to report it using `cgitb`. The optional `info` argument should be a 3-tuple containing an exception type, exception value, and traceback object, exactly like the tuple returned by `sys.exc_info()`. If the `info` argument is not supplied, the current exception is obtained from `sys.exc_info()`.
18.4 wsgiref -- WSGI Utilities and Reference Implementation

New in version 2.5.

The Web Server Gateway Interface (WSGI) is a standard interface between web server software and web applications written in Python. Having a standard interface makes it easy to use an application that supports WSGI with a number of different web servers.

Only authors of web servers and programming frameworks need to know every detail and corner case of the WSGI design. You don't need to understand every detail of WSGI just to install a WSGI application or to write a web application using an existing framework.

wsgiref is a reference implementation of the WSGI specification that can be used to add WSGI support to a web server or framework. It provides utilities for manipulating WSGI environment variables and response headers, base classes for implementing WSGI servers, a demo HTTP server that serves WSGI applications, and a validation tool that checks WSGI servers and applications for conformance to the WSGI specification (PEP 333).

Subsections

- 18.4.1 wsgiref.util - WSGI environment utilities
- 18.4.2 wsgiref.headers - WSGI response header tools
- 18.4.3 wsgiref.simple_server - a simple WSGI HTTP server
- 18.4.4 wsgiref.validate - WSGI conformance checker
- 18.4.5 wsgiref.handlers - server/gateway base classes
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.4.1 wsgiref.util - WSGI environment utilities

This module provides a variety of utility functions for working with WSGI environments. A WSGI environment is a dictionary containing HTTP request variables as described in PEP 333. All of the functions taking an `environ` parameter expect a WSGI-compliant dictionary to be supplied; please see PEP 333 for a detailed specification.

**guess_scheme**(*environ*)

Return a guess for whether `wsgi.url_scheme` should be ``http`` or ``https``, by checking for a HTTPS environment variable in the `environ` dictionary. The return value is a string.

This function is useful when creating a gateway that wraps CGI or a CGI-like protocol such as FastCGI. Typically, servers providing such protocols will include a HTTPS variable with a value of `1` ``yes``, or `on` when a request is received via SSL. So, this function returns ``https`` if such a value is found, and ``http`` otherwise.

**request_uri**(*environ [, include_query=1]*)

Return the full request URI, optionally including the query string, using the algorithm found in the `URL Reconstruction` section of PEP 333. If `include_query` is false, the query string is not included in the resulting URI.

**application_uri**(*environ*)

Similar to `request_uri`, except that the PATH_INFO and QUERY_STRING variables are ignored. The result is the base URI of the application object addressed by the request.

**shift_path_info**(*environ*)

Shift a single name from PATH_INFO to SCRIPT_NAME and return the name. The `environ` dictionary is modified in-place; use a copy if you need to keep the original PATH_INFO or SCRIPT_NAME intact.

If there are no remaining path segments in PATH_INFO, None is returned.
Typically, this routine is used to process each portion of a request URI path, for example to treat the path as a series of dictionary keys. This routine modifies the passed-in environment to make it suitable for invoking another WSGI application that is located at the target URI. For example, if there is a WSGI application at /foo, and the request URI path is /foo/bar/baz, and the WSGI application at /foo calls shift_path_info, it will receive the string ``bar'', and the environment will be updated to be suitable for passing to a WSGI application at /foo/bar. That is, SCRIPT_NAME will change from /foo to /foo/bar, and PATH_INFO will change from /bar/baz to /baz.

When PATH_INFO is just a ``/''', this routine returns an empty string and appends a trailing slash to SCRIPT_NAME, even though empty path segments are normally ignored, and SCRIPT_NAME doesn't normally end in a slash. This is intentional behavior, to ensure that an application can tell the difference between URIs ending in /x from ones ending in /x/ when using this routine to do object traversal.

**setup_testing_defaults**( environ)

Update environ with trivial defaults for testing purposes.

This routine adds various parameters required for WSGI, including HTTP_HOST, SERVER_NAME, SERVER_PORT, REQUEST_METHOD, SCRIPT_NAME, PATH_INFO, and all of the PEP 333-defined wsgi.* variables. It only supplies default values, and does not replace any existing settings for these variables.

This routine is intended to make it easier for unit tests of WSGI servers and applications to set up dummy environments. It should NOT be used by actual WSGI servers or applications, since the data is fake!

In addition to the environment functions above, the wsgiref.util module also provides these miscellaneous utilities:

**is_hop_by_hop**( header_name)

Return true if 'header_name' is an HTTP/1.1 ``Hop-by-Hop'' header, as defined by RFC 2616.
class FileWrapper(filelike [, blksize=8192])

A wrapper to convert a file-like object to an iterator. The resulting objects support both __getitem__ and __iter__ iteration styles, for compatibility with Python 2.1 and Jython. As the object is iterated over, the optional blksize parameter will be repeatedly passed to the filelike object's read() method to obtain strings to yield. When read() returns an empty string, iteration is ended and is not resumable.

If filelike has a close() method, the returned object will also have a close() method, and it will invoke the filelike object's close() method when called.
18.4.2 wsgiref.headers - WSGI response header tools

This module provides a single class, Headers, for convenient manipulation of WSGI response headers using a mapping-like interface.

class Headers( headers)

Create a mapping-like object wrapping headers, which must be a list of header name/value tuples as described in PEP 333. Any changes made to the new Headers object will directly update the headers list it was created with.

Headers objects support typical mapping operations including __getitem__, get, __setitem__, setdefault, __delitem__, __contains__ and has_key. For each of these methods, the key is the header name (treated case-insensitively), and the value is the first value associated with that header name. Setting a header deletes any existing values for that header, then adds a new value at the end of the wrapped header list. Headers' existing order is generally maintained, with new headers added to the end of the wrapped list.

Unlike a dictionary, Headers objects do not raise an error when you try to get or delete a key that isn't in the wrapped header list. Getting a nonexistent header just returns None, and deleting a nonexistent header does nothing.

Headers objects also support keys(), values(), and items() methods. The lists returned by keys() and items() can include the same key more than once if there is a multi-valued header. The len() of a Headers object is the same as the length of its items(), which is the same as the length of the wrapped header list. In fact, the items() method just returns a copy of the wrapped header list.

Calling str() on a Headers object returns a formatted string suitable for transmission as HTTP response headers. Each header is placed on a line with its value, separated by a colon and a space. Each line is terminated by
a carriage return and line feed, and the string is terminated with a blank line.

In addition to their mapping interface and formatting features, Headers objects also have the following methods for querying and adding multi-valued headers, and for adding headers with MIME parameters:

**get_all**( *name*)

Return a list of all the values for the named header.

The returned list will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

**add_header**( *name*, *value*, **_params*)

Add a (possibly multi-valued) header, with optional MIME parameters specified via keyword arguments.

*name* is the header field to add. Keyword arguments can be used to set MIME parameters for the header field. Each parameter must be a string or None. Underscores in parameter names are converted to dashes, since dashes are illegal in Python identifiers, but many MIME parameter names include dashes. If the parameter value is a string, it is added to the header value parameters in the form name="value". If it is None, only the parameter name is added. (This is used for MIME parameters without a value.) Example usage:

```python
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

The above will add a header that looks like this:

```
Content-Disposition: attachment; filename="bud.gif"
```
18.4.3 wsgiref.simple_server - a simple WSGI HTTP server

This module implements a simple HTTP server (based on BaseHTTPServer) that serves WSGI applications. Each server instance serves a single WSGI application on a given host and port. If you want to serve multiple applications on a single host and port, you should create a WSGI application that parses PATH_INFO to select which application to invoke for each request. (E.g., using the shift_path_info() function from wsgiref.util.)

```
make_server(  host, port, app [, server_class=WSGIServer [,  
  handler_class=WSGIRequestHandler]])
```

Create a new WSGI server listening on host and port, accepting connections for app. The return value is an instance of the supplied server_class, and will process requests using the specified handler_class. app must be a WSGI application object, as defined by PEP 333.

Example usage:

```
from wsgiref.simple_server import make_server, demo_app

httpd = make_server('', 8000, demo_app)
print "Serving HTTP on port 8000..."

# Respond to requests until process is killed
httpd.serve_forever()

# Alternative: serve one request, then exit
##httpd.handle_request()
```

```
demo_app( environ, start_response)
```

This function is a small but complete WSGI application that returns a text page containing the message ``Hello world!'' and a list of the key/value pairs provided in the environ parameter. It's useful for verifying that a WSGI server (such as wsgiref.simple_server) is able to run a simple WSGI application correctly.

```
class WSGIServer( server_address, RequestHandlerClass)
```
Create a \texttt{WSGIServer} instance. \texttt{server_address} should be a (host, port) tuple, and \texttt{RequestHandlerClass} should be the subclass of \texttt{BaseHTTPServer.BaseHTTPRequestHandler} that will be used to process requests.

You do not normally need to call this constructor, as the \texttt{make_server()} function can handle all the details for you.

\texttt{WSGIServer} is a subclass of \texttt{BaseHTTPServer.HTTPServer}, so all of its methods (such as \texttt{serve_forever()} and \texttt{handle_request()}) are available. \texttt{WSGIServer} also provides these WSGI-specific methods:

\begin{verbatim}
set_app(application)
\end{verbatim}

Sets the callable \texttt{application} as the WSGI application that will receive requests.

\begin{verbatim}
get_app()
\end{verbatim}

Returns the currently-set application callable.

Normally, however, you do not need to use these additional methods, as \texttt{set_app()} is normally called by \texttt{make_server()}, and the \texttt{get_app()} exists mainly for the benefit of request handler instances.

\textbf{class WSGIRequestHandler(request, client_address, server)}

Create an HTTP handler for the given \texttt{request} (i.e. a socket), \texttt{client_address} (a (host, port) tuple), and \texttt{server} (\texttt{WSGIServer} instance).

You do not need to create instances of this class directly; they are automatically created as needed by \texttt{WSGIServer} objects. You can, however, subclass this class and supply it as a \texttt{handler_class} to the \texttt{make_server()} function. Some possibly relevant methods for overriding in subclasses:

\begin{verbatim}
get_environ()
\end{verbatim}

Returns a dictionary containing the WSGI environment for a request. The default implementation copies the contents of the \texttt{WSGIServer} object's \texttt{base_environ} dictionary attribute and then adds various headers derived from the HTTP request. Each call to this method
should return a new dictionary containing all of the relevant CGI environment variables as specified in PEP 333.

**get_stderr()**

Return the object that should be used as the `wsgi.errors` stream. The default implementation just returns `sys.stderr`.

**handle()**

Process the HTTP request. The default implementation creates a handler instance using a `wsgiref.handlers` class to implement the actual WSGI application interface.
18.4.4 `wsgiref.validate` - WSGI conformance checker

When creating new WSGI application objects, frameworks, servers, or middleware, it can be useful to validate the new code's conformance using `wsgiref.validate`. This module provides a function that creates WSGI application objects that validate communications between a WSGI server or gateway and a WSGI application object, to check both sides for protocol conformance.

Note that this utility does not guarantee complete PEP 333 compliance; an absence of errors from this module does not necessarily mean that errors do not exist. However, if this module does produce an error, then it is virtually certain that either the server or application is not 100% compliant.

This module is based on the `paste.lint` module from Ian Bicking's ``Python Paste`` library.

`validator(application)`

Wrap application and return a new WSGI application object. The returned application will forward all requests to the original application, and will check that both the application and the server invoking it are conforming to the WSGI specification and to RFC 2616.

Any detected nonconformance results in an `AssertionError` being raised; note, however, that how these errors are handled is server-dependent. For example, `wsgiref.simple_server` and other servers based on `wsgiref.handlers` (that don't override the error handling methods to do something else) will simply output a message that an error has occurred, and dump the traceback to `sys.stderr` or some other error stream.

This wrapper may also generate output using the `warnings` module to indicate behaviors that are questionable but which may not actually be prohibited by PEP 333. Unless they are suppressed using Python command-line options or the `warnings` API, any such warnings will be written to `sys.stderr` (not `wsgi.errors`), unless they happen to be the same
object).
18.4.5 `wsgiref.handlers - server/gateway`

base classes

This module provides base handler classes for implementing WSGI servers and gateways. These base classes handle most of the work of communicating with a WSGI application, as long as they are given a CGI-like environment, along with input, output, and error streams.

**class CGIHandler()**

CGI-based invocation via `sys.stdin`, `sys.stdout`, `sys.stderr` and `os.environ`. This is useful when you have a WSGI application and want to run it as a CGI script. Simply invoke `CGIHandler().run(app)`, where `app` is the WSGI application object you wish to invoke.

This class is a subclass of `BaseCGIHandler` that sets `wsgi.run_once` to `true`, `wsgi.multithread` to `false`, and `wsgi.multiprocess` to `true`, and always uses `sys` and `os` to obtain the necessary CGI streams and environment.

**class BaseCGIHandler(stdin, stdout, stderr, environ [, multithread=True [, multiprocessing=False]])**

Similar to `CGIHandler`, but instead of using the `sys` and `os` modules, the CGI environment and I/O streams are specified explicitly. The `multithread` and `multiprocess` values are used to set the `wsgi.multithread` and `wsgi.multiprocess` flags for any applications run by the handler instance.

This class is a subclass of `SimpleHandler` intended for use with software other than HTTP ``origin servers``. If you are writing a gateway protocol implementation (such as CGI, FastCGI, SCGI, etc.) that uses a **Status** header to send an HTTP status, you probably want to subclass this instead of `SimpleHandler`.

\[ stdin, stdout, stderr, environ [,multithread=True [,] \]
class SimpleHandler
(multiprocess=False))

Similar to BaseCGIHandler, but designed for use with HTTP origin servers. If you are writing an HTTP server implementation, you will probably want to subclass this instead of BaseCGIHandler.

This class is a subclass of BaseHandler. It overrides the __init__(), get_stdin(), get_stderr(), add_cgi_vars(), _write(), and _flush() methods to support explicitly setting the environment and streams via the constructor. The supplied environment and streams are stored in the stdin, stdout, stderr, and environ attributes.

class BaseHandler()

This is an abstract base class for running WSGI applications. Each instance will handle a single HTTP request, although in principle you could create a subclass that was reusable for multiple requests.

BaseHandler instances have only one method intended for external use:

    run(app)

    Run the specified WSGI application, app.

All of the other BaseHandler methods are invoked by this method in the process of running the application, and thus exist primarily to allow customizing the process.

The following methods MUST be overridden in a subclass:

    _write(data)

    Buffer the string data for transmission to the client. It's okay if this method actually transmits the data; BaseHandler just separates write and flush operations for greater efficiency when the underlying system actually has such a distinction.

    _flush()

    Force buffered data to be transmitted to the client. It's okay if this method is a no-op (i.e., if _write() actually sends the data).
get_stdin()

Return an input stream object suitable for use as the wsgi.input of the request currently being processed.

get_stderr()

Return an output stream object suitable for use as the wsgi.errors of the request currently being processed.

add_cgi_vars()

Insert CGI variables for the current request into the environ attribute.

Here are some other methods and attributes you may wish to override. This list is only a summary, however, and does not include every method that can be overridden. You should consult the docstrings and source code for additional information before attempting to create a customized BaseHandler subclass.

Attributes and methods for customizing the WSGI environment:

wsgi_multithread

The value to be used for the wsgi.multithread environment variable. It defaults to true in BaseHandler, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_multiprocess

The value to be used for the wsgi.multiprocess environment variable. It defaults to true in BaseHandler, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_run_once

The value to be used for the wsgi.run_once environment variable. It defaults to false in BaseHandler, but CGIHandler sets it to true by default.

os_environ

The default environment variables to be included in every request's WSGI environment. By default, this is a copy of os.environ at the
time that `wsgiref.handlers` was imported, but subclasses can
either create their own at the class or instance level. Note that the
dictionary should be considered read-only, since the default value is
shared between multiple classes and instances.

**server_software**

If the `origin_server` attribute is set, this attribute's value is used
to set the default `SERVER_SOFTWARE` WSGI environment variable,
and also to set a default `Server:` header in HTTP responses. It is
ignored for handlers (such as `BaseCGIHandler` and
`CGIHandler`) that are not HTTP origin servers.

**get_scheme()**

Return the URL scheme being used for the current request. The default
implementation uses the `guess_scheme()` function from
`wsgiref.util` to guess whether the scheme should be ``http`` or
``https``, based on the current request's `environ` variables.

**setup_environ()**

Set the `environ` attribute to a fully-populated WSGI environment.
The default implementation uses all of the above methods and
attributes, plus the `get_stdin()`, `get_stderr()`, and
`add_cgi_vars()` methods and the `wsgi_file_wrapper` attribute. It also inserts a `SERVER_SOFTWARE` key if not present, as
long as the `origin_server` attribute is a true value and the
`server_software` attribute is set.

Methods and attributes for customizing exception handling:

**log_exception( exc_info)**

Log the `exc_info` tuple in the server log. `exc_info` is a `(type, value, traceback)` tuple. The default implementation simply writes the
traceback to the request's `wsgi.errors` stream and flushes it.
Subclasses can override this method to change the format or retarget
the output, mail the traceback to an administrator, or whatever other
action may be deemed suitable.

**traceback_limit**
The maximum number of frames to include in tracebacks output by the default `log_exception()` method. If `None`, all frames are included.

**error_output** *(environ, start_response)*

This method is a WSGI application to generate an error page for the user. It is only invoked if an error occurs before headers are sent to the client.

This method can access the current error information using `sys.exc_info()`, and should pass that information to `start_response` when calling it (as described in the ``Error Handling`` section of PEP 333).

The default implementation just uses the `error_status`, `error_headers`, and `error_body` attributes to generate an output page. Subclasses can override this to produce more dynamic error output.

Note, however, that it's not recommended from a security perspective to spit out diagnostics to any old user; ideally, you should have to do something special to enable diagnostic output, which is why the default implementation doesn't include any.

**error_status**

The HTTP status used for error responses. This should be a status string as defined in PEP 333; it defaults to a 500 code and message.

**error_headers**

The HTTP headers used for error responses. This should be a list of WSGI response headers (`(name, value)` tuples), as described in PEP 333. The default list just sets the content type to `text/plain`.

**error_body**

The error response body. This should be an HTTP response body string. It defaults to the plain text, ``A server error occurred. Please contact the administrator.``

Methods and attributes for PEP 333's ``Optional Platform-Specific File
Handling" feature:

**wsgi_file_wrapper**

A `wsgi.file_wrapper` factory, or None. The default value of this attribute is the `FileWrapper` class from `wsgiref.util`.

**sendfile()**

Override to implement platform-specific file transmission. This method is called only if the application's return value is an instance of the class specified by the `wsgi_file_wrapper` attribute. It should return a true value if it was able to successfully transmit the file, so that the default transmission code will not be executed. The default implementation of this method just returns a false value.

Miscellaneous methods and attributes:

**origin_server**

This attribute should be set to a true value if the handler's `_write()` and `_flush()` are being used to communicate directly to the client, rather than via a CGI-like gateway protocol that wants the HTTP status in a special `Status:` header.

This attribute's default value is true in `BaseHandler`, but false in `BaseCGIHandler` and `CGIhandler`.

**http_version**

If `origin_server` is true, this string attribute is used to set the HTTP version of the response set to the client. It defaults to "1.0".
18.5 urllib -- Open arbitrary resources by URL

This module provides a high-level interface for fetching data across the World Wide Web. In particular, the `urlopen()` function is similar to the built-in function `open()`, but accepts Universal Resource Locators (URLs) instead of filenames. Some restrictions apply -- it can only open URLs for reading, and no seek operations are available.

It defines the following public functions:

```python
urlopen([url[, data[, proxies]]])
```

Open a network object denoted by a URL for reading. If the URL does not have a scheme identifier, or if it has file: as its scheme identifier, this opens a local file (without universal newlines); otherwise it opens a socket to a server somewhere on the network. If the connection cannot be made the IOError exception is raised. If all went well, a file-like object is returned. This supports the following methods: `read()`, `readline()`, `readlines()`, `fileno()`, `close()`, `info()` and `geturl()`. It also has proper support for the iterator protocol. One caveat: the `read()` method, if the size argument is omitted or negative, may not read until the end of the data stream; there is no good way to determine that the entire stream from a socket has been read in the general case.

Except for the `info()` and `geturl()` methods, these methods have the same interface as for file objects -- see section 3.9 in this manual. (It is not a built-in file object, however, so it can't be used at those few places where a true built-in file object is required.)

The `info()` method returns an instance of the class `mimetools.Message` containing meta-information associated with the URL. When the method is HTTP, these headers are those returned by the server at the head of the retrieved HTML page (including Content-Length and Content-Type). When the method is FTP, a Content-Length header will be present if (as is now usual) the server passed back a file length in
response to the FTP retrieval request. A Content-Type header will be present if the MIME type can be guessed. When the method is local-file, returned headers will include a Date representing the file's last-modified time, a Content-Length giving file size, and a Content-Type containing a guess at the file's type. See also the description of the `mimetypes` module.

The `geturl()` method returns the real URL of the page. In some cases, the HTTP server redirects a client to another URL. The `urlopen()` function handles this transparently, but in some cases the caller needs to know which URL the client was redirected to. The `geturl()` method can be used to get at this redirected URL.

If the `url` uses the `http:` scheme identifier, the optional `data` argument may be given to specify a POST request (normally the request type is `GET`). The `data` argument must be in standard `application/x-www-form-urlencoded` format; see the `urlencode()` function below.

The `urlopen()` function works transparently with proxies which do not require authentication. In a UNIX or Windows environment, set the `http_proxy`, `ftp_proxy` or `gopher_proxy` environment variables to a URL that identifies the proxy server before starting the Python interpreter. For example (the "%" is the command prompt):

```bash
% http_proxy="http://www.someproxy.com:3128"
% export http_proxy
% python
...
```

In a Windows environment, if no proxy environment variables are set, proxy settings are obtained from the registry's Internet Settings section.

In a Macintosh environment, `urlopen()` will retrieve proxy information from Internet Config.

Alternatively, the optional `proxies` argument may be used to explicitly specify proxies. It must be a dictionary mapping scheme names to proxy URLs, where an empty dictionary causes no proxies to be used, and `None` (the default value) causes environmental proxy settings to be used as discussed above. For example:

```python
# Use http://www.someproxy.com:3128 for http proxying
```
proxies = {'http': 'http://www.someproxy.com:3128'}
filehandle = urllib.urlopen(some_url, proxies=proxies)
# Don't use any proxies
filehandle = urllib.urlopen(some_url, proxies={})
# Use proxies from environment - both versions are equivalent
filehandle = urllib.urlopen(some_url, proxies=None)
filehandle = urllib.urlopen(some_url)

The **urlopen()** function does not support explicit proxy specification. If you need to override environmental proxy settings, use **URLopener**, or a subclass such as **FancyURLopener**.

Proxies which require authentication for use are not currently supported; this is considered an implementation limitation.

Changed in version 2.3: Added the **proxies** support.

**urlretrieve(url[, filename[, reporthook[, data]]])**

Copy a network object denoted by a URL to a local file, if necessary. If the URL points to a local file, or a valid cached copy of the object exists, the object is not copied. Return a tuple (**filename, headers**) where **filename** is the local file name under which the object can be found, and **headers** is whatever the **info()** method of the object returned by **urlopen()** returned (for a remote object, possibly cached). Exceptions are the same as for **urlopen()**.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a hook function that will be called once on establishment of the network connection and once after each block read thereafter. The hook will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be -1 on older FTP servers which do not return a file size in response to a retrieval request.

If the **url** uses the **http:** scheme identifier, the optional **data** argument may be given to specify a POST request (normally the request type is **GET**). The **data** argument must in standard application/x-www-form-urlencoded format; see the **urlencode()** function below.

Changed in version 2.5: **urlretrieve()** will raise
ContentTooShortError when it detects that the amount of data available was less than the expected amount (which is the size reported by a **Content-Length** header). This can occur, for example, when the download is interrupted.

The **Content-Length** is treated as a lower bound: if there's more data to read, urllretrieve reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the **content** attribute of the exception instance.

If no **Content-Length** header was supplied, urllretrieve can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

**_urlopener**

The public functions urlopen() and urllretrieve() create an instance of the FancyURLopener class and use it to perform their requested actions. To override this functionality, programmers can create a subclass of URLopener or FancyURLopener, then assign an instance of that class to the urllib._urlopener variable before calling the desired function. For example, applications may want to specify a different User-Agent: header than URLopener defines. This can be accomplished with the following code:

```python
import urllib

class AppURLopener(urllib.FancyURLopener):
    version = "App/1.7"

    urllib._urlopener = AppURLopener()
```

**urlcleanup()**

Clear the cache that may have been built up by previous calls to urllretrieve().

**quote(string[, safe])**

Replace special characters in string using the "%xx" escape. Letters, digits, and the characters "_,-." are never quoted. The optional safe parameter
specifies additional characters that should not be quoted -- its default value is '/'.

Example: `quote('/~connolly/')` yields '/%7econnolly/'.

**quote_plus** *(string[, safe]*)

Like `quote()`, but also replaces spaces by plus signs, as required for quoting HTML form values. Plus signs in the original string are escaped unless they are included in `safe`. It also does not have `safe` default to '/'.

**unquote** *(string)*

Replace "%xx" escapes by their single-character equivalent.

Example: `unquote('/%7Econnolly/')` yields '/~connolly/'.

**unquote_plus** *(string)*

Like `unquote()`, but also replaces plus signs by spaces, as required for unquoting HTML form values.

**urlencode** *(query[, doseq]*)

Convert a mapping object or a sequence of two-element tuples to a "url-encoded" string, suitable to pass to `urlopen()` above as the optional `data` argument. This is useful to pass a dictionary of form fields to a POST request. The resulting string is a series of `key=value` pairs separated by "&" characters, where both `key` and `value` are quoted using `quote_plus()` above. If the optional parameter `doseq` is present and evaluates to true, individual `key=value` pairs are generated for each element of the sequence. When a sequence of two-element tuples is used as the `query` argument, the first element of each tuple is a key and the second is a value. The order of parameters in the encoded string will match the order of parameter tuples in the sequence. The **cgi** module provides the functions `parse_qs()` and `parse_qsl()` which are used to parse query strings into Python data structures.

**pathname2url** *(path)*

Convert the pathname `path` from the local syntax for a path to the form used in the path component of a URL. This does not produce a complete URL.
The return value will already be quoted using the quote() function.

url2pathname(path)
Convert the path component path from an encoded URL to the local syntax for a path. This does not accept a complete URL. This function uses unquote() to decode path.

class URLopener([proxies[, **x509]])
Base class for opening and reading URLs. Unless you need to support opening objects using schemes other than http:, ftp:, gopher: or file:, you probably want to use FancyURLopener.

By default, the URLopener class sends a User-Agent: header of "urllib/VVV", where VVV is the urllib version number. Applications can define their own User-Agent: header by subclassing URLopener or FancyURLopener and setting the class attribute version to an appropriate string value in the subclass definition.

The optional proxies parameter should be a dictionary mapping scheme names to proxy URLs, where an empty dictionary turns proxies off completely. Its default value is None, in which case environmental proxy settings will be used if present, as discussed in the definition of urlopen(), above.

Additional keyword parameters, collected in x509, may be used for authentication of the client when using the https: scheme. The keywords key_file and cert_file are supported to provide an SSL key and certificate; both are needed to support client authentication.

URLopener objects will raise an IOError exception if the server returns an error code.

class FancyURLopener(...)
FancyURLopener subclasses URLopener providing default handling for the following HTTP response codes: 301, 302, 303, 307 and 401. For the 30x response codes listed above, the Location: header is used to fetch the actual URL. For 401 response codes (authentication required), basic HTTP authentication is performed. For the 30x response codes, recursion is
bounded by the value of the `maxtries` attribute, which defaults to 10.

For all other response codes, the method `http_error_default()` is called which you can override in subclasses to handle the error appropriately.

**Note:** According to the letter of [RFC 2616](https://www.rfc-editor.org/rfc/rfc2616), 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and `urllib` reproduces this behaviour.

The parameters to the constructor are the same as those for `URLopener`.

**Note:** When performing basic authentication, a `FancyURLopener` instance calls its `prompt_user_passwd()` method. The default implementation asks the users for the required information on the controlling terminal. A subclass may override this method to support more appropriate behavior if needed.

**exception ContentTooShortError**( `msg[, content]` )

This exception is raised when the `urlretrieve()` function detects that the amount of the downloaded data is less than the expected amount (given by the `Content-Length` header). The `content` attribute stores the downloaded (and supposedly truncated) data. New in version 2.5.

Restrictions:

- Currently, only the following protocols are supported: HTTP, (versions 0.9 and 1.0), Gopher (but not Gopher-+), FTP, and local files.
- The caching feature of `urlretrieve()` has been disabled until I find the time to hack proper processing of Expiration time headers.
- There should be a function to query whether a particular URL is in the cache.
- For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.
• The `urlopen()` and `urlretrieve()` functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive Web client using these functions without using threads.

• The data returned by `urlopen()` or `urlretrieve()` is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the `Content-Type:` header. For the Gopher protocol, type information is encoded in the URL; there is currently no easy way to extract it. If the returned data is HTML, you can use the module `htmllib` to parse it.

• The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a `/`, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing `/` has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the `ftplib` module, subclassing `FancyURLopener`, or changing `_urlopener` to meet your needs.

• This module does not support the use of proxies which require authentication. This may be implemented in the future.

• Although the `urllib` module contains (undocumented) routines to parse and unpars URL strings, the recommended interface for URL manipulation is in module `urlparse`.

---

**Subsections**

• [18.5.1 URLopener Objects](#)
• [18.5.2 Examples](#)
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.5.1 URLopener Objects

URLopener and FancyURLopener objects have the following attributes.

**open**( `fullurl`, `data` )

Open `fullurl` using the appropriate protocol. This method sets up cache and proxy information, then calls the appropriate open method with its input arguments. If the scheme is not recognized, `open_unknown()` is called. The `data` argument has the same meaning as the `data` argument of `urlopen()`.

**open_unknown**( `fullurl`, `data` )

Overridable interface to open unknown URL types.

**retrieve**( `url`, `filename`, `reporthook`, `data` )

Retrieves the contents of `url` and places it in `filename`. The return value is a tuple consisting of a local filename and either a `mimetypes.Message` object containing the response headers (for remote URLs) or `None` (for local URLs). The caller must then open and read the contents of `filename`. If `filename` is not given and the URL refers to a local file, the input filename is returned. If the URL is non-local and `filename` is not given, the filename is the output of `tempfile.mktemp()` with a suffix that matches the suffix of the last path component of the input URL. If `reporthook` is given, it must be a function accepting three numeric parameters. It will be called after each chunk of data is read from the network. `reporthook` is ignored for local URLs.

If the `url` uses the `http:` scheme identifier, the optional `data` argument may be given to specify a POST request (normally the request type is GET). The `data` argument must in standard application/x-www-form-urlencoded format; see the `urlencode()` function below.

**version**

Variable that specifies the user agent of the opener object. To get `urllib` to tell servers that it is a particular user agent, set this in a subclass as a class variable or in the constructor before calling the base constructor.
The FancyURLopener class offers one additional method that should be overloaded to provide the appropriate behavior:

\[ \text{prompt}_\text{user}_\text{passwd}(\text{host, realm}) \]

Return information needed to authenticate the user at the given host in the specified security realm. The return value should be a tuple, \((\text{user, password})\), which can be used for basic authentication.

The implementation prompts for this information on the terminal; an application should override this method to use an appropriate interaction model in the local environment.
18.5.2 Examples

Here is an example session that uses the "GET" method to retrieve a URL containing parameters:

```python
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.music-al.com/cgi-bin/query?%s" % params)
>>> print f.read()
```

The following example uses the "POST" method instead:

```python
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.music-al.com/cgi-bin/query", params)
>>> print f.read()
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings:

```python
>>> import urllib
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.FancyURLopener(proxies)
>>> f = opener.open("http://www.python.org")
>>> f.read()
```

The following example uses no proxies at all, overriding environment settings:

```python
>>> import urllib
>>> opener = urllib.FancyURLopener({})
>>> f = opener.open("http://www.python.org/")
>>> f.read()
```
18.6 urllib2 -- extensible library for opening URLs

The urllib2 module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world -- basic and digest authentication, redirections, cookies and more.

The urllib2 module defines the following functions:

**urlopen(url[, data])**

Open the URL `url`, which can be either a string or a `Request` object.

`data` may be a string specifying additional data to send to the server, or `None` if no such data is needed. Currently HTTP requests are the only ones that use `data`; the HTTP request will be a POST instead of a GET when the `data` parameter is provided. `data` should be a buffer in the standard `application/x-www-form-urlencoded` format. The `urllib.urlencode()` function takes a mapping or sequence of 2-tuples and returns a string in this format.

This function returns a file-like object with two additional methods:

- `geturl()` -- return the URL of the resource retrieved
- `info()` -- return the meta-information of the page, as a dictionary-like object

Raises `URLError` on errors.

Note that `None` may be returned if no handler handles the request (though the default installed global `OpenerDirector` uses `UnknownHandler` to ensure this never happens).

**install_opener(opener)**

Install an `OpenerDirector` instance as the default global opener. Installing an opener is only necessary if you want urlopen to use that
opener; otherwise, simply call `OpenerDirector.open()` instead of `urlopen()`. The code does not check for a real `OpenerDirector`, and any class with the appropriate interface will work.

**build_opener**([`handler`, ...])

Return an `OpenerDirector` instance, which chains the handlers in the order given. `handlers` can be either instances of `BaseHandler`, or subclasses of `BaseHandler` (in which case it must be possible to call the constructor without any parameters). Instances of the following classes will be in front of the `handlers`, unless the `handlers` contain them, instances of them or subclasses of them: `ProxyHandler`, `UnknownHandler`, `HTTPHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`, `HTTPErrorProcessor`.

If the Python installation has SSL support (`socket.ssl()` exists), `HTTPSHandler` will also be added.

Beginning in Python 2.3, a `BaseHandler` subclass may also change its `handler_order` member variable to modify its position in the handlers list.

The following exceptions are raised as appropriate:

**exception URLError**

The handlers raise this exception (or derived exceptions) when they run into a problem. It is a subclass of `IOError`.

**exception HTTPError**

A subclass of `URLError`, it can also function as a non-exceptional file-like return value (the same thing that `urlopen()` returns). This is useful when handling exotic HTTP errors, such as requests for authentication.

**exception GopherError**

A subclass of `URLError`, this is the error raised by the Gopher handler.

The following classes are provided:
class Request( url[, data][, headers][, origin_req_host][, unverifiable])

This class is an abstraction of a URL request.

url should be a string containing a valid URL.

data may be a string specifying additional data to send to the server, or None if no such data is needed. Currently HTTP requests are the only ones that use data; the HTTP request will be a POST instead of a GET when the data parameter is provided. data should be a buffer in the standard application/x-www-form-urlencoded format. The urllib.urlencode() function takes a mapping or sequence of 2-tuples and returns a string in this format.

headers should be a dictionary, and will be treated as if add_header() was called with each key and value as arguments.

The final two arguments are only of interest for correct handling of third-party HTTP cookies:

origin_req_host should be the request-host of the origin transaction, as defined by RFC 2965. It defaults to cookielib.request_host(self). This is the host name or IP address of the original request that was initiated by the user. For example, if the request is for an image in an HTML document, this should be the request-host of the request for the page containing the image.

unverifiable should indicate whether the request is unverifiable, as defined by RFC 2965. It defaults to False. An unverifiable request is one whose URL the user did not have the option to approve. For example, if the request is for an image in an HTML document, and the user had no option to approve the automatic fetching of the image, this should be true.

class OpenerDirector()

The OpenerDirector class opens URLs via BaseHandlers chained together. It manages the chaining of handlers, and recovery from errors.

class BaseHandler()

This is the base class for all registered handlers -- and handles only the simple mechanics of registration.
class HTTPDefaultErrorHandler()
   A class which defines a default handler for HTTP error responses; all
   responses are turned into HTTPError exceptions.

class HTTPRedirectHandler()
   A class to handle redirections.

class HTTPCookieProcessor([cookiejar])
   A class to handle HTTP Cookies.

class ProxyHandler([proxies])
   Cause requests to go through a proxy. If proxies is given, it must be a
dictionary mapping protocol names to URLs of proxies. The default is to
read the list of proxies from the environment variables <protocol>_proxy.

class HTTPPasswordMgr()
   Keep a database of (realm, uri) -> (user, password) mappings.

class HTTPPasswordMgrWithDefaultRealm()
   Keep a database of (realm, uri) -> (user, password) mappings. A
realm of None is considered a catch-all realm, which is searched if no other
realm fits.

class AbstractBasicAuthHandler([password_mgr])
   This is a mixin class that helps with HTTP authentication, both to the
remote host and to a proxy. password_mgr, if given, should be something
that is compatible with HTTPPasswordMgr; refer to section 18.6.7 for
information on the interface that must be supported.

class HTTPBasicAuthHandler([password_mgr])
   Handle authentication with the remote host. password_mgr, if given, should
be something that is compatible with HTTPPasswordMgr; refer to
section 18.6.7 for information on the interface that must be supported.

class ProxyBasicAuthHandler([password_mgr])
   Handle authentication with the proxy. password_mgr, if given, should be
something that is compatible with HTTPPasswordMgr; refer to section 18.6.7 for information on the interface that must be supported.

class AbstractDigestAuthHandler([password_mgr])
   This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section 18.6.7 for information on the interface that must be supported.

class HTTPDigestAuthHandler([password_mgr])
   Handle authentication with the remote host. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section 18.6.7 for information on the interface that must be supported.

class ProxyDigestAuthHandler([password_mgr])
   Handle authentication with the proxy. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section 18.6.7 for information on the interface that must be supported.

class HTTPHandler()
   A class to handle opening of HTTP URLs.

class HTTPSHandler()
   A class to handle opening of HTTPS URLs.

class FileHandler()
   Open local files.

class FTPHandler()
   Open FTP URLs.

class CacheFTPHandler()
   Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class GopherHandler()
   Open gopher URLs.
class UnknownHandler()

A catch-all class to handle unknown URLs.

Subsections

- 18.6.1 Request Objects
- 18.6.2 OpenerDirector Objects
- 18.6.3 BaseHandler Objects
- 18.6.4 HTTPRedirectHandler Objects
- 18.6.5 HTTPCookieProcessor Objects
- 18.6.6 ProxyHandler Objects
- 18.6.7 HTTPPasswordMgr Objects
- 18.6.8 AbstractBasicAuthHandler Objects
- 18.6.9 HTTPBasicAuthHandler Objects
- 18.6.10 ProxyBasicAuthHandler Objects
- 18.6.11 AbstractDigestAuthHandler Objects
- 18.6.12 HTTPDigestAuthHandler Objects
- 18.6.13 ProxyDigestAuthHandler Objects
- 18.6.14 HTTPHandler Objects
- 18.6.15 HTTPSHandler Objects
- 18.6.16 FileHandler Objects
- 18.6.17 FTPHandler Objects
- 18.6.18 CacheFTPHandler Objects
- 18.6.19 GopherHandler Objects
- 18.6.20 UnknownHandler Objects
- 18.6.21 HTTPErrorProcessor Objects
- 18.6.22 Examples
18.6.1 Request Objects

The following methods describe all of Request's public interface, and so all must be overridden in subclasses.

**add_data** *(data)*
Set the Request data to *data*. This is ignored by all handlers except HTTP handlers -- and there it should be a byte string, and will change the request to be POST rather than GET.

**get_method** ()
Return a string indicating the HTTP request method. This is only meaningful for HTTP requests, and currently always returns 'GET' or 'POST'.

**has_data** ()
Return whether the instance has a non-None data.

**get_data** ()
Return the instance's data.

**add_header** *(key, val)*
Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the *key* collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header.

**add_unredirected_header** *(key, header)*
Add a header that will not be added to a redirected request. New in version 2.4.

**has_header** *(header)*
Return whether the instance has the named header (checks both regular and
unredirected. New in version 2.4.

get_full_url()
Return the URL given in the constructor.

get_type()
Return the type of the URL -- also known as the scheme.

get_host()
Return the host to which a connection will be made.

get_selector()
Return the selector -- the part of the URL that is sent to the server.

set_proxy( host, type)
Prepare the request by connecting to a proxy server. The host and type will replace those of the instance, and the instance's selector will be the original URL given in the constructor.

get_origin_req_host()
Return the request-host of the origin transaction, as defined by RFC 2965. See the documentation for the Request constructor.

is_unverifiable()
Return whether the request is unverifiable, as defined by RFC 2965. See the documentation for the Request constructor.
18.6.2 OpenerDirector Objects

OpenerDirector instances have the following methods:

**add_handler(handler)**

*handler* should be an instance of `BaseHandler`. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case).

- `protocol_open()` -- signal that the handler knows how to open protocol URLs.
- `http_error_type()` -- signal that the handler knows how to handle HTTP errors with HTTP error code `type`.
- `protocol_error()` -- signal that the handler knows how to handle errors from (non-http) `protocol`.
- `protocol_request()` -- signal that the handler knows how to pre-process `protocol` requests.
- `protocol_response()` -- signal that the handler knows how to post-process `protocol` responses.

**open(url[, data])**

Open the given `url` (which can be a request object or a string), optionally passing the given `data`. Arguments, return values and exceptions raised are the same as those of `urlopen()` (which simply calls the `open()` method on the currently installed global OpenerDirector).

**error(proto[, arg[, ...]])**

Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler; refer to the `http_error_*()` methods of the handler classes.

Return values and exceptions raised are the same as those of `urlopen()`.

OpenerDirector objects open URLs in three stages:
The order in which these methods are called within each stage is determined by sorting the handler instances.

1. Every handler with a method named like `protocol_request()` has that method called to pre-process the request.

2. Handlers with a method named like `protocol_open()` are called to handle the request. This stage ends when a handler either returns a non-`None` value (ie. a response), or raises an exception (usually `URLError`). Exceptions are allowed to propagate.

   In fact, the above algorithm is first tried for methods named `default_open`. If all such methods return `None`, the algorithm is repeated for methods named like `protocol_open()`. If all such methods return `None`, the algorithm is repeated for methods named `unknown_open()`.

   Note that the implementation of these methods may involve calls of the parent `OpenerDirector` instance's `.open()` and `.error()` methods.

3. Every handler with a method named like `protocol_response()` has that method called to post-process the response.
18.6.3 BaseHandler Objects

BaseHandler objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use:

**add_parent**( director)

Add a director as parent.

**close**( )

Remove any parents.

The following members and methods should only be used by classes derived from BaseHandler. **Note:** The convention has been adopted that subclasses defining protocol_request() or protocol_response() methods are named *Processor; all others are named *Handler.

**parent**

A valid OpenerDirector, which can be used to open using a different protocol, or handle errors.

**default_open**( req)

This method is *not* defined in BaseHandler, but subclasses should define it if they want to catch all URLs.

This method, if implemented, will be called by the parent OpenerDirector. It should return a file-like object as described in the return value of the open() of OpenerDirector, or None. It should raise URLError, unless a truly exceptional thing happens (for example, MemoryError should not be mapped to URLError).

This method will be called before any protocol-specific open method.

**protocol_open**( req)

This method is *not* defined in BaseHandler, but subclasses should define it if they want to handle URLs with the given protocol.
This method, if defined, will be called by the parent OpenerDirector. Return values should be the same as for default_open().

unknown_open( req)
This method is not defined in BaseHandler, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

This method, if implemented, will be called by the parent OpenerDirector. Return values should be the same as for default_open().

http_error_default( req, fp, code, msg, hdrs)
This method is not defined in BaseHandler, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the OpenerDirector getting the error, and should not normally be called in other circumstances.

req will be a Request object, fp will be a file-like object with the HTTP error body, code will be the three-digit code of the error, msg will be the user-visible explanation of the code and hdrs will be a mapping object with the headers of the error.

Return values and exceptions raised should be the same as those of urlopen().

http_error_nnn( req, fp, code, msg, hdrs)
nnn should be a three-digit HTTP error code. This method is also not defined in BaseHandler, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code nnn occurs.

Subclasses should override this method to handle specific HTTP errors.

Arguments, return values and exceptions raised should be the same as for http_error_default().

protocol_request( req)
This method is not defined in BaseHandler, but subclasses should define
it if they want to pre-process requests of the given protocol.

This method, if defined, will be called by the parent **OpenerDirector**. 
*req* will be a **Request** object. The return value should be a **Request**
object.

**protocol_response**( *req*, *response*)

This method is *not* defined in **BaseHandler**, but subclasses should define
it if they want to post-process responses of the given protocol.

This method, if defined, will be called by the parent **OpenerDirector**. 
*req* will be a **Request** object. *response* will be an object implementing the
same interface as the return value of *urlopen()* The return value should
implement the same interface as the return value of *urlopen()*.
18.6.4 HTTPRedirectHandler Objects

Note: Some HTTP redirections require action from this module's client code. If this is the case, HTTPError is raised. See RFC 2616 for details of the precise meanings of the various redirection codes.

redirect_request( req, fp, code, msg, hdrs)

Return a Request or None in response to a redirect. This is called by the default implementations of the http_error_30*() methods when a redirection is received from the server. If a redirection should take place, return a new Request to allow http_error_30*() to perform the redirect. Otherwise, raise HTTPError if no other handler should try to handle this URL, or return None if you can't but another handler might.

Note: The default implementation of this method does not strictly follow RFC 2616, which says that 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and the default implementation reproduces this behavior.

http_error_301( req, fp, code, msg, hdrs)

Redirect to the Location: URL. This method is called by the parent OpenerDirector when getting an HTTP `moved permanently' response.

http_error_302( req, fp, code, msg, hdrs)

The same as http_error_301( ), but called for the `found' response.

http_error_303( req, fp, code, msg, hdrs)

The same as http_error_301( ), but called for the `see other' response.

http_error_307( req, fp, code, msg, hdrs)
The same as `http_error_301()`, but called for the `temporary redirect' response.
18.6.5 HTTPCookieProcessor Objects

New in version 2.4.

HTTPCookieProcessor instances have one attribute:

**cookiejar**

The `cookielib.CookieJar` in which cookies are stored.
18.6.6 ProxyHandler Objects

`protocol_open(request)`

The ProxyHandler will have a method `protocol_open()` for every protocol which has a proxy in the proxies dictionary given in the constructor. The method will modify requests to go through the proxy, by calling `request.set_proxy()`, and call the next handler in the chain to actually execute the protocol.
18.6.7 HTTPPasswordMgr Objects

These methods are available on HTTPPasswordMgr and HTTPPasswordMgrWithDefaultRealm objects.

**add_password** *(realm, uri, user, passwd)*

*uri* can be either a single URI, or a sequence of URIs. *realm, user* and *passwd* must be strings. This causes *(user, passwd)* to be used as authentication tokens when authentication for *realm* and a super-URI of any of the given URIs is given.

**find_user_password** *(realm, authuri)*

Get user/password for given realm and URI, if any. This method will return *(None, None)* if there is no matching user/password.

For HTTPPasswordMgrWithDefaultRealm objects, the realm None will be searched if the given *realm* has no matching user/password.

---

**Python Library Reference**

Previous: 18.6.6 ProxyHandler Objects  Up: 18.6 urllib2  Next: 18.6.8 AbstractBasicAuthHandler Objects

*Release 2.5, documentation updated on 19th September, 2006.*

See *About this document...* for information on suggesting changes.
18.6.8 AbstractBasicAuthHandler Objects

http_error_auth_reqed( authreq, host, req, headers)
Handle an authentication request by getting a user/password pair, and re-
trying the request. authreq should be the name of the header where the
information about the realm is included in the request, host specifies the
URL and path to authenticate for, req should be the (failed) Request
object, and headers should be the error headers.

host is either an authority (e.g. "python.org") or a URL containing an
authority component (e.g. "http://python.org/"). In either case, the
authority must not contain a userinfo component (so, "python.org" and
"python.org:80" are fine, "joe:password@python.org" is not).
18.6.9 `HTTPBasicAuthHandler` Objects

`http_error_401(req, fp, code, msg, hdrs)`
- Retry the request with authentication information, if available.
18.6.10 ProxyBasicAuthHandler Objects

```
http_error_407( req, fp, code, msg, hdrs)

Retry the request with authentication information, if available.
```

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.6.11 AbstractDigestAuthHandler Objects

`http_error_auth_reqed(authreq, host, req, headers)`

`authreq` should be the name of the header where the information about the realm is included in the request, `host` should be the host to authenticate to, `req` should be the (failed) `Request` object, and `headers` should be the error headers.
18.6.12 HTTPDigestAuthHandler Objects

`http_error_401( req, fp, code, msg, hdrs)`

Retry the request with authentication information, if available.
18.6.13 ProxyDigestAuthHandler Objects

`http_error_407(req, fp, code, msg, hdrs)`

Retry the request with authentication information, if available.
18.6.14 HTTPHandler Objects

http_open(req)

Send an HTTP request, which can be either GET or POST, depending on req.has_data().
18.6.15 HTTPSHandler Objects

`https_open(req)`

Send an HTTPS request, which can be either GET or POST, depending on `req.has_data()`.

---

*Previous: 18.6.14 HTTPHandler Objects Up: 18.6 urllib2 Next: 18.6.16 FileHandler Objects*

*Release 2.5, documentation updated on 19th September, 2006.*

*See [About this document...](#) for information on suggesting changes.*
18.6.16 FileHandler Objects

`file_open(req)`

Open the file locally, if there is no host name, or the host name is 'localhost'. Change the protocol to `ftp` otherwise, and retry opening it using `parent`.
18.6.17 FTPHandler Objects

`ftp_open(req)`

Open the FTP file indicated by `req`. The login is always done with empty username and password.
18.6.18 CacheFTPHandler Objects

CacheFTPHandler objects are FTPHandler objects with the following additional methods:

**setTimeOut**(\(t\))
- Set timeout of connections to \(t\) seconds.

**setMaxConns**(\(m\))
- Set maximum number of cached connections to \(m\).
18.6.19 GopherHandler Objects

`gopher_open(req)`
Open the gopher resource indicated by `req`.

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.6.20 UnknownHandler Objects

unknown_open()

Raise a URLError exception.
18.6.21 HTTPErrorProcessor Objects

New in version 2.4.

unknown_open()

Process HTTP error responses.

For 200 error codes, the response object is returned immediately.

For non-200 error codes, this simply passes the job on to the protocol_error_code() handler methods, via OpenerDirector.error(). Eventually, urllib2.HTTPDefaultErrorHandler will raise an HTTPError if no other handler handles the error.
18.6.22 Examples

This example gets the python.org main page and displays the first 100 bytes of it:

```python
>>> import urllib2
>>> f = urllib2.urlopen('http://www.python.org/')
>>> print f.read(100)
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<!?xml-stylesheet href="./css/ht2html

Here we are sending a data-stream to the stdin of a CGI and reading the data it returns to us. Note that this example will only work when the Python installation supports SSL.

```python
>>> import urllib2
>>> req = urllib2.Request(url='https://localhost/cgi-bin/test.cgi',
    data='This data is passed to stdin of the CGI')
>>> f = urllib2.urlopen(req)
>>> print f.read()
Got Data: "This data is passed to stdin of the CGI"
```

The code for the sample CGI used in the above example is:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print 'Content-type: text-plain

Got Data: "%s"' % data
```

Use of Basic HTTP Authentication:

```
import urllib2
# Create an OpenerDirector with support for Basic HTTP Authentication
auth_handler = urllib2.HTTPBasicAuthHandler()
auth_handler.add_password('realm', 'host', 'username', 'password')
opener = urllib2.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib2.install_opener(opener)
urllib2.urlopen('http://www.example.com/login.html')
```

`build_opener()` provides many handlers by default, including a `ProxyHandler`. By default, `ProxyHandler` uses the environment variables named `<scheme>_proxy`, where `<scheme>` is the URL scheme involved. For example, the http_proxy environment variable is read to obtain the HTTP
proxy's URL.

This example replaces the default `ProxyHandler` with one that uses programatically-supplied proxy URLs, and adds proxy authorization support with `ProxyBasicAuthHandler`.

```python
proxy_handler = urllib2.ProxyHandler({'http': 'http://www.example.com:3128/'}
proxy_auth_handler = urllib2.HTTPBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = build_opener(proxy_handler, proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it direc
t opener.open('http://www.example.com/login.html')
```

Adding HTTP headers:

Use the `headers` argument to the `Request` constructor, or:

```python
import urllib2
req = urllib2.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
r = urllib2.urlopen(req)
```

`OpenerDirector` automatically adds a `User-Agent:` header to every `Request`. To change this:

```python
import urllib2
opener = urllib2.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
 opener.open('http://www.example.com/')
```

Also, remember that a few standard headers (Content-Length:, Content-Type: and Host:) are added when the Request is passed to `urlopen()` (or `OpenerDirector.open()`).
This module defines classes which implement the client side of the HTTP and HTTPS protocols. It is normally not used directly -- the module urllib uses it to handle URLs that use HTTP and HTTPS.

**Note:** HTTPS support is only available if the socket module was compiled with SSL support.

**Note:** The public interface for this module changed substantially in Python 2.0. The HTTP class is retained only for backward compatibility with 1.5.2. It should not be used in new code. Refer to the online docstrings for usage.

The module provides the following classes:

**class HTTPConnection**(`host[, port]`)  
An HTTPConnection instance represents one transaction with an HTTP server. It should be instantiated passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. For example, the following calls all create instances that connect to the server at the same host and port:

```python
>>> h1 = httplib.HTTPConnection('www.cwi.nl')
>>> h2 = httplib.HTTPConnection('www.cwi.nl:80')
>>> h3 = httplib.HTTPConnection('www.cwi.nl', 80)
```

New in version 2.0.

**class HTTPSConnection**(`host[, port, key_file, cert_file]`)  
A subclass of HTTPConnection that uses SSL for communication with secure servers. Default port is 443. *key_file* is the name of a PEM formatted file that contains your private key. *cert_file* is a PEM formatted
certificate chain file.

**Warning:** This does not do any certificate verification!

New in version 2.0.

```python
class HTTPResponse( sock[, debuglevel=0][, strict=0])
```
Class whose instances are returned upon successful connection. Not instantiated directly by user. New in version 2.0.

The following exceptions are raised as appropriate:

**exception HTTPException**
The base class of the other exceptions in this module. It is a subclass of `Exception`. New in version 2.0.

**exception NotConnected**
A subclass of `HTTPException`. New in version 2.0.

**exception InvalidURL**
A subclass of `HTTPException`, raised if a port is given and is either non-numeric or empty. New in version 2.3.

**exception UnknownProtocol**
A subclass of `HTTPException`. New in version 2.0.

**exception UnknownTransferEncoding**
A subclass of `HTTPException`. New in version 2.0.

**exception UnimplementedFileMode**
A subclass of `HTTPException`. New in version 2.0.

**exception IncompleteRead**
A subclass of `HTTPException`. New in version 2.0.

**exception ImproperConnectionState**
A subclass of `HTTPException`. New in version 2.0.

**exception CannotSendRequest**
A subclass of `ImproperConnectionState`. New in version 2.0.

**exception CannotSendHeader**

A subclass of `ImproperConnectionState`. New in version 2.0.

**exception ResponseNotReady**

A subclass of `ImproperConnectionState`. New in version 2.0.

**exception BadStatusLine**

A subclass of `HTTPException`. Raised if a server responds with a HTTP status code that we don't understand. New in version 2.0.

The constants defined in this module are:

**HTTP_PORT**

The default port for the HTTP protocol (always 80).

**HTTPS_PORT**

The default port for the HTTPS protocol (always 443).

and also the following constants for integer status codes:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTINUE</td>
<td>100</td>
<td>HTTP/1.1, RFC 2616, Section 10.1.1</td>
</tr>
<tr>
<td>SWITCHING_PROTOCOLS</td>
<td>101</td>
<td>HTTP/1.1, RFC 2616, Section 10.1.2</td>
</tr>
<tr>
<td>PROCESSING</td>
<td>102</td>
<td>WEBDAV, RFC 2518, Section 10.1</td>
</tr>
<tr>
<td>OK</td>
<td>200</td>
<td>HTTP/1.1, RFC 2616, Section 10.2.1</td>
</tr>
<tr>
<td>CREATED</td>
<td>201</td>
<td>HTTP/1.1, RFC 2616, Section 10.2.2</td>
</tr>
<tr>
<td>ACCEPTED</td>
<td>202</td>
<td>HTTP/1.1, RFC 2616, Section 10.2.3</td>
</tr>
<tr>
<td>NON_AUTHORITATIVE_INFORMATION</td>
<td>203</td>
<td>HTTP/1.1, RFC 2616, Section 10.2.4</td>
</tr>
<tr>
<td>Status Code</td>
<td>HTTP/1.1, RFC</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>204</td>
<td>2616, Section 10.2.5</td>
<td>NO_CONTENT</td>
</tr>
<tr>
<td>205</td>
<td>2616, Section 10.2.6</td>
<td>RESET_CONTENT</td>
</tr>
<tr>
<td>206</td>
<td>2616, Section 10.2.7</td>
<td>PARTIAL_CONTENT</td>
</tr>
<tr>
<td>207</td>
<td>WEBDAV RFC 2518, Section 10.2</td>
<td>MULTI_STATUS</td>
</tr>
<tr>
<td>226</td>
<td>Delta encoding in HTTP, RFC 3229, Section 10.4.1</td>
<td>IM_USED</td>
</tr>
<tr>
<td>300</td>
<td>2616, Section 10.3.1</td>
<td>MULTIPLE_CHOICES</td>
</tr>
<tr>
<td>301</td>
<td>2616, Section 10.3.2</td>
<td>MOVED_PERMANENTLY</td>
</tr>
<tr>
<td>302</td>
<td>2616, Section 10.3.3</td>
<td>FOUND</td>
</tr>
<tr>
<td>303</td>
<td>2616, Section 10.3.4</td>
<td>SEE_OTHER</td>
</tr>
<tr>
<td>304</td>
<td>2616, Section 10.3.5</td>
<td>NOT_MODIFIED</td>
</tr>
<tr>
<td>305</td>
<td>2616, Section 10.3.6</td>
<td>USE_PROXY</td>
</tr>
<tr>
<td>307</td>
<td>2616, Section 10.3.8</td>
<td>TEMPORARY_REDIRECT</td>
</tr>
<tr>
<td>400</td>
<td>2616, Section 10.4.1</td>
<td>BAD_REQUEST</td>
</tr>
<tr>
<td>401</td>
<td>2616, Section 10.4.2</td>
<td>UNAUTHORIZED</td>
</tr>
<tr>
<td>402</td>
<td>2616, Section 10.4.3</td>
<td>PAYMENT_REQUIRED</td>
</tr>
<tr>
<td>403</td>
<td>2616, Section 10.4.4</td>
<td>FORBIDDEN</td>
</tr>
<tr>
<td>404</td>
<td>2616, Section 10.4.5</td>
<td>NOT_FOUND</td>
</tr>
<tr>
<td>405</td>
<td>HTTP/1.1, RFC</td>
<td>METHOD_NOT_ALLOWED</td>
</tr>
<tr>
<td>Error Code</td>
<td>Status Code</td>
<td>Definition</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>NOT_ACCEPTABLE</td>
<td>406</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.6</td>
</tr>
<tr>
<td>PROXY_AUTHENTICATION_REQUIRED</td>
<td>407</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.7</td>
</tr>
<tr>
<td>REQUEST_TIMEOUT</td>
<td>408</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.8</td>
</tr>
<tr>
<td>CONFLICT</td>
<td>409</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.9</td>
</tr>
<tr>
<td>GONE</td>
<td>410</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.11</td>
</tr>
<tr>
<td>LENGTH_REQUIRED</td>
<td>411</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.12</td>
</tr>
<tr>
<td>PRECONDITION_FAILED</td>
<td>412</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.13</td>
</tr>
<tr>
<td>REQUEST_ENTITY_TOO_LARGE</td>
<td>413</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.14</td>
</tr>
<tr>
<td>REQUEST_URI_TOO_LONG</td>
<td>414</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.15</td>
</tr>
<tr>
<td>UNSUPPORTED_MEDIA_TYPE</td>
<td>415</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.16</td>
</tr>
<tr>
<td>REQUESTED_RANGE_NOT_SATISFIABLE</td>
<td>416</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.17</td>
</tr>
<tr>
<td>EXPECTATION_FAILED</td>
<td>417</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.18</td>
</tr>
<tr>
<td>UNPROCESSABLE_ENTITY</td>
<td>422</td>
<td>WEBDAV, RFC 2518, Section 10.3</td>
</tr>
<tr>
<td>LOCKED</td>
<td>423</td>
<td>WEBDAV RFC 2518, Section 10.4</td>
</tr>
<tr>
<td>FAILED_DEPENDENCY</td>
<td>424</td>
<td>WEBDAV, RFC 2518, Section 10.5</td>
</tr>
<tr>
<td>UPGRADE_REQUIRED</td>
<td>426</td>
<td>HTTP Upgrade to TLS, RFC 2817, Section 6</td>
</tr>
<tr>
<td>INTERNAL_SERVER_ERROR</td>
<td>500</td>
<td>HTTP/1.1, RFC 2616, Section 10.5.1</td>
</tr>
<tr>
<td>Status Code</td>
<td>HTTP Version</td>
<td>RFC Reference</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>NOT_IMPLEMENTED</td>
<td>501</td>
<td>HTTP/1.1, RFC 2616, Section 10.5.2</td>
</tr>
<tr>
<td>BAD_GATEWAY</td>
<td>502</td>
<td>HTTP/1.1 RFC 2616, Section 10.5.3</td>
</tr>
<tr>
<td>SERVICE_UNAVAILABLE</td>
<td>503</td>
<td>HTTP/1.1, RFC 2616, Section 10.5.4</td>
</tr>
<tr>
<td>GATEWAY_TIMEOUT</td>
<td>504</td>
<td>HTTP/1.1 RFC 2616, Section 10.5.5</td>
</tr>
<tr>
<td>HTTP_VERSION_NOT_SUPPORTED</td>
<td>505</td>
<td>HTTP/1.1, RFC 2616, Section 10.5.6</td>
</tr>
<tr>
<td>INSUFFICIENT_STORAGE</td>
<td>507</td>
<td>WEBDAV, RFC 2518, Section 10.6</td>
</tr>
<tr>
<td>NOT_EXTENDED</td>
<td>510</td>
<td>An HTTP Extension Framework, RFC 2774, Section 7</td>
</tr>
</tbody>
</table>

**responses**

This dictionary maps the HTTP 1.1 status codes to the W3C names.

Example: `httplib.responses[httplib.NOT_FOUND]` is 'Not Found'. New in version 2.5.
18.7.1 HTTPConnection Objects

HTTPConnection instances have the following methods:

**request**(method, url[, body[, headers]])
This will send a request to the server using the HTTP request method method and the selector url. If the body argument is present, it should be a string of data to send after the headers are finished. The header Content-Length is automatically set to the correct value. The headers argument should be a mapping of extra HTTP headers to send with the request.

**getresponse()**
Should be called after a request is sent to get the response from the server. Returns an HTTPResponse instance. **Note:** Note that you must have read the whole response before you can send a new request to the server.

**set_debuglevel**(level)
Set the debugging level (the amount of debugging output printed). The default debug level is 0, meaning no debugging output is printed.

**connect()**
Connect to the server specified when the object was created.

**close()**
Close the connection to the server.

As an alternative to using the request() method described above, you can also send your request step by step, by using the four functions below.

**putrequest**(request, selector[, skip_host[, skip_accept_encoding]])
This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the request string, the selector string, and the HTTP version (HTTP/1.1). To disable automatic sending of Host: or Accept-Encoding: headers (for example to accept additional content encodings), specify skip_host or skip_accept_encoding with non-False values. Changed in version 2.4:
skip_accept_encoding argument added.

**putheader( header, argument[, ...])**

Send an RFC 822-style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

**endheaders()**

Send a blank line to the server, signalling the end of the headers.

**send( data)**

Send data to the server. This should be used directly only after the **endheaders()** method has been called and before **getresponse()** is called.
18.7.2 HTTPResponse Objects

HTTPResponse instances have the following methods and attributes:

**read**([*amt*])
Reads and returns the response body, or up to the next *amt* bytes.

**getheader**(*name*, *default*)
Get the contents of the header *name*, or *default* if there is no matching header.

**getheaders**()
Return a list of (header, value) tuples. New in version 2.4.

**msg**
A `mimetypes.Message` instance containing the response headers.

**version**
HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

**status**
Status code returned by server.

**reason**
Reason phrase returned by server.
18.7.3 Examples

Here is an example session that uses the "GET" method:

```python
>>> import httplib
>>> conn = httplib.HTTPConnection("www.python.org")
>>> conn.request("GET", "/index.html")
>>> r1 = conn.getresponse()
>>> print r1.status, r1.reason
200 OK
>>> data1 = r1.read()
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print r2.status, r2.reason
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

Here is an example session that shows how to "POST" requests:

```python
>>> import httplib, urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
... "Accept": "text/plain"}
>>> conn = httplib.HTTPConnection("musi-cal.mojam.com:80")
>>> conn.request("POST", "/cgi-bin/query", params, headers)
>>> response = conn.getresponse()
>>> print response.status, response.reason
200 OK
>>> data = response.read()
>>> conn.close()
```
18.8 ftplib -- FTP protocol client

This module defines the class FTP and a few related items. The FTP class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other ftp servers. It is also used by the module urllib to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet RFC 959.

Here's a sample session using the ftplib module:

```python
>>> from ftplib import FTP
>>> ftp = FTP('ftp.cwi.nl')  # connect to host, default port
>>> ftp.login()  # user anonymous, passwd anonymous@
>>> ftp.retrlines('LIST')  # list directory contents
total 24418
  drwxrwsr-x  5 ftp-usr pdmaint  1536 Mar 20 09:48 .
  dr-xr-srwt 105 ftp-usr pdmaint  1536 Mar 21 14:32 ..
  -rw-r--r--  1 ftp-usr pdmaint  5305 Mar 20 09:48 INDEX
  .
  .
>>> ftp.retrbinary('RETR README', open('README', 'wb').write)
'226 Transfer complete.'
>>> ftp.quit()
```

The module defines the following items:

**class FTP([host[ , user[ , passwd[ , acct ]]]]]**

Return a new instance of the FTP class. When *host* is given, the method call `connect(host)` is made. When *user* is given, additionally the method call `login(user, passwd, acct)` is made (where *passwd* and *acct* default to the empty string when not given).

**all_errors**

The set of all exceptions (as a tuple) that methods of FTP instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed below as well as `socket.error` and IOError.

**exception error_reply**
Exception raised when an unexpected reply is received from the server.

**exception error_temp**
Exception raised when an error code in the range 400-499 is received.

**exception error_perm**
Exception raised when an error code in the range 500-599 is received.

**exception error_proto**
Exception raised when a reply is received from the server that does not begin with a digit in the range 1-5.

### See Also:

Module **netrc**:  
Parser for the .netrc file format. The file .netrc is typically used by FTP clients to load user authentication information before prompting the user.

The file Tools/scripts/ftpmirror.py in the Python source distribution is a script that can mirror FTP sites, or portions thereof, using the ftplib module. It can be used as an extended example that applies this module.

### Subsections

- [18.8.1 FTP Objects](#)
18.8.1 FTP Objects

Several methods are available in two flavors: one for handling text files and another for binary files. These are named for the command which is used followed by "lines" for the text version or "binary" for the binary version.

FTP instances have the following methods:

**set_debuglevel( level)**
Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

**connect( host[, port])**
Connect to the given host and port. The default port number is 21, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made.

**getwelcome()**
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

**login([user[, passwd[, acct]]])**
Log in as the given user. The passwd and acct parameters are optional and default to the empty string. If no user is specified, it defaults to 'anonymous'. If user is 'anonymous', the default passwd is 'anonymous@'. This function should be called only once for each instance, after a connection has been established; it should not be called at all if a host and user were given when the instance was created. Most FTP
commands are only allowed after the client has logged in.

abort()
Abort a file transfer that is in progress. Using this does not always work, but it's worth a try.

sendcmd(command)
Send a simple command string to the server and return the response string.

voidcmd(command)
Send a simple command string to the server and handle the response. Return nothing if a response code in the range 200-299 is received. Raise an exception otherwise.

retrbinary(command, callback[, maxblocksize[, rest]])
Retrieve a file in binary transfer mode. command should be an appropriate "RETR" command: 'RETR filename'. The callback function is called for each block of data received, with a single string argument giving the data block. The optional maxblocksize argument specifies the maximum chunk size to read on the low-level socket object created to do the actual transfer (which will also be the largest size of the data blocks passed to callback). A reasonable default is chosen. rest means the same thing as in the transfercmd() method.

retrlines(command[, callback])
Retrieve a file or directory listing in ASCII transfer mode. command should be an appropriate "RETR" command (see retrbinary()) or a "LIST" command (usually just the string 'LIST'). The callback function is called for each line, with the trailing CRLF stripped. The default callback prints the line to sys.stdout.

set_pasv(boolean)
Enable ``passive'' mode if boolean is true, other disable passive mode. (In Python 2.0 and before, passive mode was off by default; in Python 2.1 and later, it is on by default.)

storbinary(command, file[, blocksize])
Store a file in binary transfer mode. *command* should be an appropriate "STOR" command: "STOR filename". *file* is an open file object which is read until EOF using its `read()` method in blocks of size `blocksize` to provide the data to be stored. The `blocksize` argument defaults to 8192. Changed in version 2.1: default for `blocksize` added.

**storlines**( *command*, *file*)

Store a file in ASCII transfer mode. *command* should be an appropriate "STOR" command (see `storbinary()`). Lines are read until EOF from the open file object *file* using its `readline()` method to provide the data to be stored.

**transfercmd**( *cmd*, *rest*)

Initiate a transfer over the data connection. If the transfer is active, send a "EPRT" or "PORT" command and the transfer command specified by *cmd*, and accept the connection. If the server is passive, send a "EPSV" or "PASV" command, connect to it, and start the transfer command. Either way, return the socket for the connection.

If optional *rest* is given, a "REST" command is sent to the server, passing *rest* as an argument. *rest* is usually a byte offset into the requested file, telling the server to restart sending the file's bytes at the requested offset, skipping over the initial bytes. Note however that RFC 959 requires only that *rest* be a string containing characters in the printable range from ASCII code 33 to ASCII code 126. The `transfercmd()` method, therefore, converts *rest* to a string, but no check is performed on the string's contents. If the server does not recognize the "REST" command, an `error_reply` exception will be raised. If this happens, simply call `transfercmd()` without a *rest* argument.

**ntransfercmd**( *cmd*, *rest*)

Like `transfercmd()`, but returns a tuple of the data connection and the expected size of the data. If the expected size could not be computed, `None` will be returned as the expected size. *cmd* and *rest* means the same thing as in `transfercmd()`.

**nlst**( *argument*, ...)

Return a list of files as returned by the "NLST" command. The optional argument is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the "NLST" command.

**dir**(*argument[, ...]*)
Produce a directory listing as returned by the "LIST" command, printing it to standard output. The optional argument is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the "LIST" command. If the last argument is a function, it is used as a callback function as for retrlines(); the default prints to sys.stdout. This method returns None.

**rename**(*fromname, toname*)
Rename file fromname on the server to toname.

**delete**(*filename*)
Remove the file named filename from the server. If successful, returns the text of the response, otherwise raises error_perm on permission errors or error_reply on other errors.

**cwd**(*pathname*)
Set the current directory on the server.

**mkd**(*pathname*)
Create a new directory on the server.

**pwd**()
Return the pathname of the current directory on the server.

**rmd**(*dirname*)
Remove the directory named dirname on the server.

**size**(*filename*)
Request the size of the file named filename on the server. On success, the size of the file is returned as an integer, otherwise None is returned. Note that the "SIZE" command is not standardized, but is supported by many
common server implementations.

\textbf{quit()}

Send a "QUIT" command to the server and close the connection. This is the `polite" way to close a connection, but it may raise an exception of the server responds with an error to the "QUIT" command. This implies a call to the close() method which renders the FTP instance useless for subsequent calls (see below).

\textbf{close()}

Close the connection unilaterally. This should not be applied to an already closed connection such as after a successful call to quit(). After this call the FTP instance should not be used any more (after a call to close() or quit() you cannot reopen the connection by issuing another login() method).
18.9 gopherlib -- Gopher protocol client

Deprecated since release 2.5. The gopher protocol is not in active use anymore.

This module provides a minimal implementation of client side of the Gopher protocol. It is used by the module urllib to handle URLs that use the Gopher protocol.

The module defines the following functions:

send_selector(selector, host[, port])
Send a selector string to the gopher server at host and port (default 70). Returns an open file object from which the returned document can be read.

send_query(selector, query, host[, port])
Send a selector string and a query string to a gopher server at host and port (default 70). Returns an open file object from which the returned document can be read.

Note that the data returned by the Gopher server can be of any type, depending on the first character of the selector string. If the data is text (first character of the selector is "0"), lines are terminated by CRLF, and the data is terminated by a line consisting of a single ".", and a leading "." should be stripped from lines that begin with ". .". Directory listings (first character of the selector is "1") are transferred using the same protocol.
18.10 poplib -- POP3 protocol client

This module defines a class, POP3, which encapsulates a connection to a POP3 server and implements the protocol as defined in RFC 1725. The POP3 class supports both the minimal and optional command sets. Additionally, this module provides a class POP3_SSL, which provides support for connecting to POP3 servers that use SSL as an underlying protocol layer.

Note that POP3, though widely supported, is obsolescent. The implementation quality of POP3 servers varies widely, and too many are quite poor. If your mailserver supports IMAP, you would be better off using the imaplib. IMAP4 class, as IMAP servers tend to be better implemented.

A single class is provided by the poplib module:

```
class POP3( host[, port])
   This class implements the actual POP3 protocol. The connection is created when the instance is initialized. If port is omitted, the standard POP3 port (110) is used.
```

```
class POP3_SSL( host[, port[, keyfile[, certfile]]])
   This is a subclass of POP3 that connects to the server over an SSL encrypted socket. If port is not specified, 995, the standard POP3-over-SSL port is used. keyfile and certfile are also optional - they can contain a PEM formatted private key and certificate chain file for the SSL connection.
```

New in version 2.4.

One exception is defined as an attribute of the poplib module:

```
exception error_proto
   Exception raised on any errors. The reason for the exception is passed to the constructor as a string.
```

See Also:
Module **imaplib**: The standard Python IMAP module.

**Frequently Asked Questions About Fetchmail**
The FAQ for the fetchmail POP/IMAP client collects information on POP3 server variations and RFC noncompliance that may be useful if you need to write an application based on the POP protocol.

Subsections

- 18.10.1 POP3 Objects
- 18.10.2 POP3 Example

Release 2.5, documentation updated on 19th September, 2006. See [About this document...](#) for information on suggesting changes.
18.10.1 POP3 Objects

All POP3 commands are represented by methods of the same name, in lowercase; most return the response text sent by the server.

An POP3 instance has the following methods:

set_debuglevel( level)
Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

getwelcome()
Returns the greeting string sent by the POP3 server.

user( username)
Send user command, response should indicate that a password is required.

pass_( password)
Send password, response includes message count and mailbox size. Note: the mailbox on the server is locked until quit() is called.

apop( user, secret)
Use the more secure APOP authentication to log into the POP3 server.

rpop( user)
Use RPOP authentication (similar to UNIX r-commands) to log into POP3 server.

stat()
Get mailbox status. The result is a tuple of 2 integers: (message count, mailbox size).
list([which])
Request message list, result is in the form (response, ['mesg_num octets', ...], octets). If which is set, it is the message to list.

retr(which)
Retrieve whole message number which, and set its seen flag. Result is in form (response, ['line', ...], octets).

dele(which)
Flag message number which for deletion. On most servers deletions are not actually performed until QUIT (the major exception is Eudora QPOP, which deliberately violates the RFCs by doing pending deletes on any disconnect).

rset()
Remove any deletion marks for the mailbox.

noop()
Do nothing. Might be used as a keep-alive.

quit()
Signoff: commit changes, unlock mailbox, drop connection.

top(which, howmuch)
Retrieves the message header plus howmuch lines of the message after the header of message number which. Result is in form (response, ['line', ...], octets).

The POP3 TOP command this method uses, unlike the RETR command, doesn't set the message's seen flag; unfortunately, TOP is poorly specified in the RFCs and is frequently broken in off-brand servers. Test this method by hand against the POP3 servers you will use before trusting it.

uidl([which])
Return message digest (unique id) list. If which is specified, result contains the unique id for that message in the form 'response mesgnum uid', otherwise result is list (response, ['mesgnum uid', ...], octets).
Instances of POP3_SSL have no additional methods. The interface of this subclass is identical to its parent.
18.10.2 POP3 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```python
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print j
```

At the end of the module, there is a test section that contains a more extensive example of usage.
18.11 imaplib -- IMAP4 protocol client

This module defines three classes, IMAP4, IMAP4_SSL and IMAP4_stream, which encapsulate a connection to an IMAP4 server and implement a large subset of the IMAP4rev1 client protocol as defined in RFC 2060. It is backward compatible with IMAP4 (RFC 1730) servers, but note that the "STATUS" command is not supported in IMAP4.

Three classes are provided by the imaplib module, IMAP4 is the base class:

```python
class IMAP4([host, port])
```
This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If host is not specified, '' (the local host) is used. If port is omitted, the standard IMAP4 port (143) is used.

Three exceptions are defined as attributes of the IMAP4 class:

```python
exception IMAP4.error
```
Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

```python
exception IMAP4.abort
```
IMAP4 server errors cause this exception to be raised. This is a sub-class of IMAP4.error. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.

```python
exception IMAP4.readonly
```
This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of IMAP4.error. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

There's also a subclass for secure connections:
class IMAP4_SSL([host[, port[, keyfile[, certfile]]]])

This is a subclass derived from IMAP4 that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If host is not specified, '' (the local host) is used. If port is omitted, the standard IMAP4-over-SSL port (993) is used. keyfile and certfile are also optional - they can contain a PEM formatted private key and certificate chain file for the SSL connection.

The second subclass allows for connections created by a child process:

class IMAP4_stream(command)

This is a subclass derived from IMAP4 that connects to the stdin/stdout file descriptors created by passing command to os.popen2(). New in version 2.3.

The following utility functions are defined:

**Internaldate2tuple** (datestr)

Converts an IMAP4 INTERNALDATE string to Coordinated Universal Time. Returns a time module tuple.

**Int2AP** (num)

Converts an integer into a string representation using characters from the set [A .. P].

**ParseFlags** (flagstr)

Converts an IMAP4 "FLAGS" response to a tuple of individual flags.

**Time2Internaldate** (date_time)

Converts a time module tuple to an IMAP4 "INTERNALDATE" representation. Returns a string in the form: "DD-Mmm-YYYY HH:MM:SS +HHMM" (including double-quotes).

Note that IMAP4 message numbers change as the mailbox changes; in particular, after an "EXPUNGE" command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.
At the end of the module, there is a test section that contains a more extensive example of usage.

**See Also:**

Documents describing the protocol, and sources and binaries for servers implementing it, can all be found at the University of Washington's *IMAP Information Center* ([http://www.cac.washington.edu/imap/](http://www.cac.washington.edu/imap/)).

---

**Subsections**

- 18.11.1 IMAP4 Objects
- 18.11.2 IMAP4 Example

---

Release 2.5, documentation updated on 19th September, 2006. See [About this document...](#) for information on suggesting changes.
18.11.1 IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either upper-case or lower-case.

All arguments to commands are converted to strings, except for "AUTHENTICATE", and the last argument to "APPEND" which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn't enclosed with either parentheses or double quotes) each string is quoted. However, the password argument to the "LOGIN" command is always quoted. If you want to avoid having an argument string quoted (eg: the flags argument to "STORE") then enclose the string in parentheses (eg: r'\Deleted').

Each command returns a tuple: (type, [data, ...]) where type is usually 'OK' or 'NO', and data is either the text from the command response, or mandated results from the command. Each data is either a string, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie: 'literal' value).

The message_set options to commands below is a string specifying one or more messages to be acted upon. It may be a simple message number ('1'), a range of message numbers ('2:4'), or a group of non-contiguous ranges separated by commas ('1:3, 6:9'). A range can contain an asterisk to indicate an infinite upper bound ('3:*').

An IMAP4 instance has the following methods:

**append**( mailbox, flags, date_time, message)

Append message to named mailbox.

**authenticate**( mechanism, authobject)

Authenticate command -- requires response processing.

mechanism specifies which authentication mechanism is to be used - it should appear in the instance variable capabilities in the form AUTH=mechanism.
authobject must be a callable object:

\[
data = \text{authobject}(\text{response})
\]

It will be called to process server continuation responses. It should return data that will be encoded and sent to server. It should return None if the client abort response "*" should be sent instead.

check()
Checkpoint mailbox on server.

close()
Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before "LOGOUT".

copy( message_set, new_mailbox)
Copy message_set messages onto end of new_mailbox.

create( mailbox)
Create new mailbox named mailbox.

delete( mailbox)
Delete old mailbox named mailbox.

deleteacl( mailbox, who)
Delete the ACLs (remove any rights) set for who on mailbox. New in version 2.4.

expunge()
Permanently remove deleted items from selected mailbox. Generates an "EXPUNGE" response for each deleted message. Returned data contains a list of "EXPUNGE" message numbers in order received.

fetch( message_set, message_parts)
Fetch (parts of) messages. message_parts should be a string of message part names enclosed within parentheses, eg: ""(UID BODY[TEXT])"". Returned data are tuples of message part envelope and data.
**getacl**(*mailbox*)

Get the "ACL"s for *mailbox*. The method is non-standard, but is supported by the "Cyrus" server.

**getannotation**(*mailbox*, *entry*, *attribute*)

Retrieve the specified "ANNOTATION"s for *mailbox*. The method is non-standard, but is supported by the "Cyrus" server. New in version 2.5.

**getquota**(*root*)

Get the "quota" *root"s resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087. New in version 2.3.

**getquotaroot**(*mailbox*)

Get the list of "quota" "roots" for the named *mailbox*. This method is part of the IMAP4 QUOTA extension defined in rfc2087. New in version 2.3.

**list**(^[directory[, pattern]]^)

List mailbox names in *directory* matching *pattern*. *directory* defaults to the top-level mail folder, and *pattern* defaults to match anything. Returned data contains a list of "LIST" responses.

**login**(*user*, *password*)

Identify the client using a plaintext password. The *password* will be quoted.

**login_cram_md5**(*user*, *password*)

Force use of "CRAM-MD5" authentication when identifying the client to protect the password. Will only work if the server "CAPABILITY" response includes the phrase "AUTH=CRAM-MD5". New in version 2.3.

**logout**()

Shutdown connection to server. Returns server "BYE" response.

**lsub**(^[directory[, pattern]]^)

List subscribed mailbox names in directory matching *pattern*. *directory* defaults to the top level directory and *pattern* defaults to match any
mailbox. Returned data are tuples of message part envelope and data.

myrights(mailbox)
Show my ACLs for a mailbox (i.e. the rights that I have on mailbox). New in version 2.4.

namespace()
Returns IMAP namespaces as defined in RFC2342. New in version 2.3.

noop()
Send "NOOP" to server.

open(host, port)
Opens socket to port at host. The connection objects established by this method will be used in the read, readline, send, and shutdown methods. You may override this method.

partial(message_num, message_part, start, length)
Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

proxyauth(user)
Assume authentication as user. Allows an authorised administrator to proxy into any user's mailbox. New in version 2.3.

read(size)
Reads size bytes from the remote server. You may override this method.

readline()
Reads one line from the remote server. You may override this method.

recent()
Prompt server for an update. Returned data is None if no new messages, else value of "RECENT" response.

rename(oldmailbox, newmailbox)
Rename mailbox named oldmailbox to newmailbox.
**response**(code)

Return data for response code if received, or None. Returns the given code, instead of the usual type.

**search**(charset, criterion[, ...])

Search mailbox for matching messages. charset may be None, in which case no "CHARSET" will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error.

Example:

```python
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

**select**(mailbox[, readonly])

Select a mailbox. Returned data is the count of messages in mailbox ("EXISTS" response). The default mailbox is 'INBOX'. If the readonly flag is set, modifications to the mailbox are not allowed.

**send**(data)

Sends data to the remote server. You may override this method.

**setacl**(mailbox, who, what)

Set an "ACL" for mailbox. The method is non-standard, but is supported by the "Cyrus" server.

**setannotation**(mailbox, entry, attribute[, ...])

Set "ANNOTATION"s for mailbox. The method is non-standard, but is supported by the "Cyrus" server. New in version 2.5.

**setquota**(root, limits)

Set the "quota" root's resource limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087. New in version 2.3.
**shutdown()**

Close connection established in **open**. You may override this method.

**socket()**

Returns socket instance used to connect to server.

**sort**(sort_criteria, charset, search_criterion[, ...])

The **sort** command is a variant of **search** with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the **search_criterion** argument(s); a parenthesized list of **sort_criteria**, and the searching **charset**. Note that unlike **search**, the searching **charset** argument is mandatory. There is also a **uid** **sort** command which corresponds to **sort** the way that **uid** **search** corresponds to **search**. The **sort** command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

This is an "**IMAP4rev1**" extension command.

**status**(mailbox, names)

Request named status conditions for **mailbox**.

**store**(message_set, command, flag_list)

Alters flag dispositions for messages in mailbox. **command** is specified by section 6.4.6 of **RFC 2060** as being one of "**FLAGS**", "+**FLAGS**", or "-**FLAGS**", optionally with a suffix of "**.SILENT**".

For example, to set the delete flag on all messages:

```python
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    M.store(num, '+FLAGS', '\Deleted')
M.expunge()
```

**subscribe**(mailbox)

Subscribe to new mailbox.
thread(threading_algorithm, charset, search_criterion[, ...])

The `thread` command is a variant of `search` with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the `search_criterion` argument(s); a `threading_algorithm`, and the searching `charset`. Note that unlike `search`, the searching `charset` argument is mandatory. There is also a `uid thread` command which corresponds to `thread` the way that `uid search` corresponds to `search`. The `thread` command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.

This is an "IMAP4rev1" extension command. New in version 2.4.

uid(command, arg[, ...])

Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied; if none are provided, the server will return an error and an exception will be raised.

unsubscribe(mailbox)

Unsubscribe from old mailbox.

xatom(name[, arg[, ...]])

Allow simple extension commands notified by server in "CAPABILITY" response.

Instances of IMAP4_SSL have just one additional method:

ssl()

Returns SSLObjec tinstance used for the secure connection with the server.
The following attributes are defined on instances of IMAP4:

**PROTOCOL_VERSION**

The most recent supported protocol in the "CAPABILITY" response from the server.

**debug**

Integer value to control debugging output. The initialize value is taken from the module variable Debug. Values greater than three trace each command.
18.11.2 IMAP4 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```python
import getpass, imaplib

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print 'Message %s\n%s\n' % (num, data[0][1])
M.close()
M.logout()
```

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.12 nntplib -- NNTP protocol client

This module defines the class `NNTP` which implements the client side of the NNTP protocol. It can be used to implement a news reader or poster, or automated news processors. For more information on NNTP (Network News Transfer Protocol), see Internet RFC 977.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles:

```python
>>> s = NNTP('news.cwi.nl')
>>> resp, count, first, last, name = s.group('comp.lang.python')
>>> print 'Group', name, 'has', count, 'articles, range', first, 'to', last
Group comp.lang.python has 59 articles, range 3742 to 3803
>>> resp, subs = s.xhdr('subject', first + '-' + last)
>>> for id, sub in subs[-10:]: print id, sub
... 3792 Re: Removing elements from a list while iterating...
  3793 Re: Who likes Info files?
  3794 Emacs and doc strings
  3795 a few questions about the Mac implementation
  3796 Re: executable python scripts
  3797 Re: executable python scripts
  3798 Re: a few questions about the Mac implementation
  3799 Re: PROPOSAL: A Generic Python Object Interface for Python C Mo
  3802 Re: executable python scripts
  3803 Re: \POSIX{} wait and SIGCHLD
>>> s.quit()
'205 news.cwi.nl closing connection. Goodbye.'
```

To post an article from a file (this assumes that the article has valid headers):

```python
>>> s = NNTP('news.cwi.nl')
>>> f = open('/tmp/article')
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 news.cwi.nl closing connection. Goodbye.'
```

The module itself defines the following items:

```python
class NNTP(
```
Return a new instance of the NNTP class, representing a connection to the NNTP server running on host `host`, listening at port `port`. The default `port` is 119. If the optional `user` and `password` are provided, or if suitable credentials are present in `./netrc` and the optional flag `usenetrc` is true (the default), the "AUTHINFO USER" and "AUTHINFO PASS" commands are used to identify and authenticate the user to the server. If the optional flag `readermode` is true, then a "mode reader" command is sent before authentication is performed. Reader mode is sometimes necessary if you are connecting to an NNTP server on the local machine and intend to call reader-specific commands, such as "group". If you get unexpected `NNTPPermanentErrors`, you might need to set `readermode`. `readermode` defaults to `None`. `usenetrc` defaults to `True`.

Changed in version 2.4: `usenetrc` argument added.

**exception NNTPError**
Derived from the standard exception `Exception`, this is the base class for all exceptions raised by the `nntplib` module.

**exception NNTPReplyError**
Exception raised when an unexpected reply is received from the server. For backwards compatibility, the exception `error_reply` is equivalent to this class.

**exception NNTPTemporaryError**
Exception raised when an error code in the range 400-499 is received. For backwards compatibility, the exception `error_temp` is equivalent to this class.

**exception NNTPPermanentError**
Exception raised when an error code in the range 500-599 is received. For backwards compatibility, the exception `error_perm` is equivalent to this class.

**exception NNTPProtocolError**
Exception raised when a reply is received from the server that does not begin with a digit in the range 1-5. For backwards compatibility, the
exception error_proto is equivalent to this class.

**exception NNTPDataError**

Exception raised when there is some error in the response data. For backwards compatibility, the exception error_data is equivalent to this class.

Subsections

- [18.12.1 NNTP Objects](#)

---

Previous: [18.11.2 IMAP4 Example](#)  Up: [18. Internet Protocols and](#)  Next: [18.12.1 NNTP Objects](#)

*Release 2.5, documentation updated on 19th September, 2006.*

See [About this document...](#) for information on suggesting changes.
18.12.1 NNTP Objects

NNTP instances have the following methods. The response that is returned as the first item in the return tuple of almost all methods is the server's response: a string beginning with a three-digit code. If the server's response indicates an error, the method raises one of the above exceptions.

getwelcome()
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

set_debuglevel( level)
Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request or response. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the connection (including message text).

newgroups( date, time, [file])
Send a "NEWGROUPS" command. The date argument should be a string of the form 'yymmd' indicating the date, and time should be a string of the form 'hhmmss' indicating the time. Return a pair (response, groups) where groups is a list of group names that are new since the given date and time. If the file parameter is supplied, then the output of the "NEWGROUPS" command is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling write( ) on it to store the lines of the command output. If file is supplied, then the returned list is an empty list.

newnews( group, date, time, [file])
Send a "NEWNEWS" command. Here, group is a group name or ' * ', and date and time have the same meaning as for newgroups( ). Return a pair (response, articles) where articles is a list of message ids. If the file parameter is supplied, then the output of the "NEWNEWS" command is
stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling write() on it to store the lines of the command output. If file is supplied, then the returned list is an empty list.

**list([file])**

Send a "LIST" command. Return a pair (response, list) where list is a list of tuples. Each tuple has the form (group, last, first, flag), where group is a group name, last and first are the last and first article numbers (as strings), and flag is 'y' if posting is allowed, 'n' if not, and 'm' if the newsgroup is moderated. (Note the ordering: last, first.) If the file parameter is supplied, then the output of the "LIST" command is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling write() on it to store the lines of the command output. If file is supplied, then the returned list is an empty list.

**descriptions(grouppattern)**

Send a "LIST NEWSGROUPS" command, where grouppattern is a wildmat string as specified in RFC2980 (it's essentially the same as DOS or UNIX shell wildcard strings). Return a pair (response, list), where list is a list of tuples containing (name, title).

New in version 2.4.

**description(group)**

Get a description for a single group group. If more than one group matches (if 'group' is a real wildmat string), return the first match. If no group matches, return an empty string.

This elides the response code from the server. If the response code is needed, use descriptions().

New in version 2.4.

**group(name)**

Send a "GROUP" command, where name is the group name. Return a tuple (response, count, first, last, name) where count is the (estimated)
number of articles in the group, \textit{first} is the first article number in the group, \textit{last} is the last article number in the group, and \textit{name} is the group name. The numbers are returned as strings.

\textbf{help([file])}

Send a "HELP" command. Return a pair \((\text{response}, \text{list})\) where \text{list} is a list of help strings. If the \text{file} parameter is supplied, then the output of the "HELP" command is stored in a file. If \text{file} is a string, then the method will open a file object with that name, write to it then close it. If \text{file} is a file object, then it will start calling \text{write()} on it to store the lines of the command output. If \text{file} is supplied, then the returned \text{list} is an empty list.

\textbf{stat(id)}

Send a "STAT" command, where \text{id} is the message id (enclosed in "<" and ")") or an article number (as a string). Return a triple \((\text{response}, \text{number}, \text{id})\) where \text{number} is the article number (as a string) and \text{id} is the message id (enclosed in "<" and ">").

\textbf{next()}

Send a "NEXT" command. Return as for \text{stat()}. 

\textbf{last()}

Send a "LAST" command. Return as for \text{stat()}. 

\textbf{head(id)}

Send a "HEAD" command, where \text{id} has the same meaning as for \text{stat()}. Return a tuple \((\text{response}, \text{number}, \text{id}, \text{list})\) where the first three are the same as for \text{stat()}, and \text{list} is a list of the article's headers (an uninterpreted list of lines, without trailing newlines). 

\textbf{body(id,[file])}

Send a "BODY" command, where \text{id} has the same meaning as for \text{stat()}. If the \text{file} parameter is supplied, then the body is stored in a file. If \text{file} is a string, then the method will open a file object with that name, write to it then close it. If \text{file} is a file object, then it will start calling \text{write()} on it to store the lines of the body. Return as for \text{head()}. If \text{file} is supplied, then the returned \text{list} is an empty list.
**article**(*id*)
Send an "ARTICLE" command, where *id* has the same meaning as for `stat()`. Return as for `head()`.

**slave()**
Send a "SLAVE" command. Return the server's *response*.

**xhdr**(*header*, *string*, [*file]*)
Send an "XHDR" command. This command is not defined in the RFC but is a common extension. The *header* argument is a header keyword, e.g. 'subject'. The *string* argument should have the form '*first-last*' where *first* and *last* are the first and last article numbers to search. Return a pair (*response*, *list*), where *list* is a list of pairs (*id*, *text*), where *id* is an article number (as a string) and *text* is the text of the requested header for that article. If the *file* parameter is supplied, then the output of the "XHDR" command is stored in a file. If *file* is a string, then the method will open a file object with that name, write to it then close it. If *file* is a file object, then it will start calling `write()` on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list.

**post**(*file*)
Post an article using the "POST" command. The *file* argument is an open file object which is read until EOF using its `readline()` method. It should be a well-formed news article, including the required headers. The `post()` method automatically escapes lines beginning with ".".

**ihave**(*id*, *file*)
Send an "IHAVE" command. *id* is a message id (enclosed in "<" and ">"). If the response is not an error, treat *file* exactly as for the `post()` method.

**date()**
Return a triple (*response*, *date*, *time*), containing the current date and time in a form suitable for the `newnews()` and `newgroups()` methods. This is an optional NNTP extension, and may not be supported by all servers.

**xgtitle**(*name*, [*file]*)
Process an "XGTITLE" command, returning a pair \( \text{response, list} \), where \text{list} is a list of tuples containing \( \text{name, title} \). If the \text{file} parameter is supplied, then the output of the "XGTITLE" command is stored in a file. If \text{file} is a string, then the method will open a file object with that name, write to it then close it. If \text{file} is a file object, then it will start calling \text{write()}\ on it to store the lines of the command output. If \text{file} is supplied, then the returned \text{list} is an empty list. This is an optional NNTP extension, and may not be supported by all servers.

RFC2980 says ``It is suggested that this extension be deprecated''. Use \text{descriptions()} or \text{description()} instead.

\text{xover}( \text{start, end, [file]} )
Return a pair \( \text{resp, list} \). \text{list} is a list of tuples, one for each article in the range delimited by the \text{start} and \text{end} article numbers. Each tuple is of the form \( \text{article number, subject, poster, date, id, references, size, lines} \). If the \text{file} parameter is supplied, then the output of the "XOVER" command is stored in a file. If \text{file} is a string, then the method will open a file object with that name, write to it then close it. If \text{file} is a file object, then it will start calling \text{write()}\ on it to store the lines of the command output. If \text{file} is supplied, then the returned \text{list} is an empty list. This is an optional NNTP extension, and may not be supported by all servers.

\text{xpath}( \text{id} )
Return a pair \( \text{resp, path} \), where \text{path} is the directory path to the article with message ID \text{id}. This is an optional NNTP extension, and may not be supported by all servers.

\text{quit()}
Send a "QUIT" command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.
18.13 smtplib -- SMTP protocol client

The smtplib module defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult RFC 821 (Simple Mail Transfer Protocol) and RFC 1869 (SMTP Service Extensions).

class SMTP([host[, port[, local_hostname]]])

A SMTP instance encapsulates an SMTP connection. It has methods that support a full repertoire of SMTP and ESMTP operations. If the optional host and port parameters are given, the SMTP `connect()` method is called with those parameters during initialization. An `SMTPConnectError` is raised if the specified host doesn't respond correctly.

For normal use, you should only require the initialization/connect, `sendmail()`, and `quit()` methods. An example is included below.

A nice selection of exceptions is defined as well:

exception SMTPException
    Base exception class for all exceptions raised by this module.

exception SMTPServerDisconnected
    This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the SMTP instance before connecting it to a server.

exception SMTPResponseException
    Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code. The error code is stored in the `smtp_code` attribute of the error, and the `smtp_error` attribute is set to the error message.
exception **SMTPSenderRefused**
Sender address refused. In addition to the attributes set by on all SMTPResponseException exceptions, this sets `sender' to the string that the SMTP server refused.

exception **SMTPRecipientsRefused**
All recipient addresses refused. The errors for each recipient are accessible through the attribute recipients, which is a dictionary of exactly the same sort as SMTP.sendmail() returns.

exception **SMTPDataError**
The SMTP server refused to accept the message data.

exception **SMTPConnectError**
Error occurred during establishment of a connection with the server.

exception **SMTPHeloError**
The server refused our "HELO" message.

---

**See Also:**

[RFC 821, Simple Mail Transfer Protocol](http://example.com)
Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

[RFC 1869, SMTP Service Extensions](http://example.com)
Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.

---

**Subsections**

- [18.13.1 SMTP Objects](http://example.com)
- [18.13.2 SMTP Example](http://example.com)
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.13.1 SMTP Objects

An SMTP instance has the following methods:

**set_debuglevel**( *level*)
Set the debug output level. A true value for *level* results in debug messages for connection and for all messages sent to and received from the server.

**connect**( *host[, port]*)
Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25). If the hostname ends with a colon ("":"") followed by a number, that suffix will be stripped off and the number interpreted as the port number to use. This method is automatically invoked by the constructor if a host is specified during instantiation.

**docmd**( *cmd[, argstring]*)
Send a command *cmd* to the server. The optional argument *argstring* is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, `SMTPServerDisconnected` will be raised.

**helo**( *hostname*)
Identify yourself to the SMTP server using "HELO". The hostname argument defaults to the fully qualified domain name of the local host.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the `sendmail()` when necessary.
**ehlo** ([hostname])
Identify yourself to an ESMTP server using "EHLO". The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by has_extn().

Unless you wish to use has_extn() before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by sendmail() when necessary.

**has_extn** (name)
Return True if name is in the set of SMTP service extensions returned by the server, False otherwise. Case is ignored.

**verify** (address)
Check the validity of an address on this server using SMTP "VRFY". Returns a tuple consisting of code 250 and a full RFC 822 address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

*Note*: Many sites disable SMTP "VRFY" in order to foil spammers.

**login** (user, password)
Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous "EHLO" or "HELO" command this session, this method tries ESMTP "EHLO" first. This method will return normally if the authentication was successful, or may raise the following exceptions:

**SMTPHeloError**
The server didn't reply properly to the "HELO" greeting.

**SMTPAuthenticationError**
The server didn't accept the username/password combination.

**SMTPError**
No suitable authentication method was found.

**starttls**([keyfile[, certfile]])
Put the SMTP connection in TLS (Transport Layer Security) mode. All
SMTP commands that follow will be encrypted. You should then call
ehlo() again.

If keyfile and certfile are provided, these are passed to the socket
module's ssl() function.

sendmail( from_addr, to_addrs, msg[, mail_options, rcpt_options])

Send mail. The required arguments are an RFC 822 from-address string, a
list of RFC 822 to-address strings (a bare string will be treated as a list with
1 address), and a message string. The caller may pass a list of ESMTP
options (such as "8bitmime") to be used in "MAIL FROM" commands as
mail_options. ESMTP options (such as "DSN" commands) that should be
used with all "RCPT" commands can be passed as rcpt_options. (If you need
to use different ESMTP options to different recipients you have to use the
low-level methods such as mail, rcpt and data to send the message.)

Note: The from_addr and to_addrs parameters are used to construct the
message envelope used by the transport agents. The SMTP does not modify
the message headers in any way.

If there has been no previous "EHLO" or "HELO" command this session,
this method tries ESMTP "EHLO" first. If the server does ESMTP, message
size and each of the specified options will be passed to it (if the option is in
the feature set the server advertises). If "EHLO" fails, "HELO" will be tried
and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one
recipient. Otherwise it will throw an exception. That is, if this method does
not throw an exception, then someone should get your mail. If this method
does not throw an exception, it returns a dictionary, with one entry for each
recipient that was refused. Each entry contains a tuple of the SMTP error
code and the accompanying error message sent by the server.

This method may raise the following exceptions:

SMTPRecipientsRefused

All recipients were refused. Nobody got the mail. The recipients
attribute of the exception object is a dictionary with information about
the refused recipients (like the one returned when at least one recipient was accepted).

**SMTPHeloError**

The server didn't reply properly to the "HELO" greeting.

**SMTPSenderRefused**

The server didn't accept the *from_addr*.

**SMTPDataError**

The server replied with an unexpected error code (other than a refusal of a recipient).

Unless otherwise noted, the connection will be open even after an exception is raised.

**quit()**

Terminate the SMTP session and close the connection.

Low-level methods corresponding to the standard SMTP/ESMTP commands "HELP", "RSET", "NOOP", "MAIL", "RCPT", and "DATA" are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.
18.13.2 SMTP Example

This example prompts the user for addresses needed in the message envelope (`To` and `From` addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn't do any processing of the RFC 822 headers. In particular, the `To` and `From` addresses must be included in the message headers explicitly.

```python
import smtplib

def prompt(prompt):
    return raw_input(prompt).strip()

fromaddr = prompt("From:")
toaddrs = prompt("To:")
print "Enter message, end with ^D (Unix) or ^Z (Windows):"

# Add the From: and To: headers at the start!
msg = ("From: %s\nTo: %s\n\n" % (fromaddr,",".join(toaddrs)))
while 1:
    try:
        line = raw_input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print "Message length is " + repr(len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```
18.14 smtpd -- SMTP Server

This module offers several classes to implement SMTP servers. One is a generic do-nothing implementation, which can be overridden, while the other two offer specific mail-sending strategies.

Subsections

- 18.14.1 SMTPServer Objects
- 18.14.2 DebuggingServer Objects
- 18.14.3 PureProxy Objects
- 18.14.4 MailmanProxy Objects
18.14.1 SMTPServer Objects

class SMTPServer(localaddr, remoteaddr)

Create a new SMTPServer object, which binds to local address localaddr. It will treat remoteaddr as an upstream SMTP relayer. It inherits from asyncore.dispatcher, and so will insert itself into asyncore's event loop on instantiation.

process_message(peer, mailfrom, rcpttos, data)

Raise NotImplementedError exception. Override this in subclasses to do something useful with this message. Whatever was passed in the constructor as remoteaddr will be available as the _remoteaddr attribute. peer is the remote host's address, mailfrom is the envelope originator, rcpttos are the envelope recipients and data is a string containing the contents of the e-mail (which should be in RFC 2822 format).
18.14.2 DebuggingServer Objects

class DebuggingServer( localaddr, remoteaddr)
    Create a new debugging server. Arguments are as per SMTPServer.
    Messages will be discarded, and printed on stdout.

Python Library Reference


Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.14.3 PureProxy Objects

class PureProxy( localaddr, remoteaddr)

Create a new pure proxy server. Arguments are as per SMTPServer. Everything will be relayed to remoteaddr. Note that running this has a good chance to make you into an open relay, so please be careful.
18.14.4 MailmanProxy Objects

class MailmanProxy( localaddr, remoteaddr)
    Create a new pure proxy server. Arguments are as per SMTPServer. Everything will be relayed to remoteaddr, unless local mailman configurations knows about an address, in which case it will be handled via mailman. Note that running this has a good chance to make you into an open relay, so please be careful.
18.15 telnetlib -- Telnet client

The telnetlib module provides a Telnet class that implements the Telnet protocol. See RFC 854 for details about the protocol. In addition, it provides symbolic constants for the protocol characters (see below), and for the telnet options. The symbolic names of the telnet options follow the definitions in arpa/telnet.h, with the leading TELOPT_ removed. For symbolic names of options which are traditionally not included in arpa/telnet.h, see the module source itself.

The symbolic constants for the telnet commands are: IAC, DONT, DO, WONT, WILL, SE (Subnegotiation End), NOP (No Operation), DM (Data Mark), BRK (Break), IP (Interrupt process), AO (Abort output), AYT (Are You There), EC (Erase Character), EL (Erase Line), GA (Go Ahead), SB (Subnegotiation Begin).

class Telnet([host[, port]])

Telnet represents a connection to a Telnet server. The instance is initially not connected by default; the open() method must be used to establish a connection. Alternatively, the host name and optional port number can be passed to the constructor, to, in which case the connection to the server will be established before the constructor returns.

Do not reopen an already connected instance.

This class has many read_*() methods. Note that some of them raise EOFError when the end of the connection is read, because they can return an empty string for other reasons. See the individual descriptions below.

See Also:

RFC 854, Telnet Protocol Specification
Definition of the Telnet protocol.
18.15.1 Telnet Objects

18.15.2 Telnet Example
18.15.1 Telnet Objects

Telnet instances have the following methods:

- **read_until(expected[, timeout])**
  Read until a given string, `expected`, is encountered or until `timeout` seconds have passed.

  When no match is found, return whatever is available instead, possibly the empty string. Raise **EOFError** if the connection is closed and no cooked data is available.

- **read_all()**
  Read all data until EOF; block until connection closed.

- **read_some()**
  Read at least one byte of cooked data unless EOF is hit. Return `' '` if EOF is hit. Block if no data is immediately available.

- **read_very_eager()**
  Read everything that can be without blocking in I/O (eager).

  Raise **EOFError** if connection closed and no cooked data available. Return `' '` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

- **read_eager()**
  Read readily available data.

  Raise **EOFError** if connection closed and no cooked data available. Return `' '` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

- **read_lazy()**
  Process and return data already in the queues (lazy).
Raise EOFError if connection closed and no data available. Return '' if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

**read very lazy()**
Return any data available in the cooked queue (very lazy).

Raise EOFError if connection closed and no data available. Return '' if no cooked data available otherwise. This method never blocks.

**read sb data()**
Return the data collected between a SB/SE pair (suboption begin/end). The callback should access these data when it was invoked with a SE command. This method never blocks.

New in version 2.3.

**open( host[, port])**
Connect to a host. The optional second argument is the port number, which defaults to the standard Telnet port (23).

Do not try to reopen an already connected instance.

**msg( msg[, *args])**
Print a debug message when the debug level is > 0. If extra arguments are present, they are substituted in the message using the standard string formatting operator.

**set debuglevel( debuglevel)**
Set the debug level. The higher the value of debuglevel, the more debug output you get (on sys.stdout).

**close()**
Close the connection.

**get socket()**
Return the socket object used internally.
fileno()
Return the file descriptor of the socket object used internally.

write(buffer)
Write a string to the socket, doubling any IAC characters. This can block if the connection is blocked. May raise socket.error if the connection is closed.

interact()
Interaction function, emulates a very dumb Telnet client.

mt_interact()
Multithreaded version of interact().

expect(list[, timeout])
Read until one from a list of a regular expressions matches.

The first argument is a list of regular expressions, either compiled (re.RegexObject instances) or uncompiled (strings). The optional second argument is a timeout, in seconds; the default is to block indefinitely.

Return a tuple of three items: the index in the list of the first regular expression that matches; the match object returned; and the text read up till and including the match.

If end of file is found and no text was read, raise EOFError. Otherwise, when nothing matches, return (-1, None, text) where text is the text received so far (may be the empty string if a timeout happened).

If a regular expression ends with a greedy match (such as .* ) or if more than one expression can match the same input, the results are indeterministic, and may depend on the I/O timing.

set_option_negotiation_callback(callback)
Each time a telnet option is read on the input flow, this callback (if set) is called with the following parameters: callback(telnet socket, command (DO/DONT/WILL/WONT), option). No other action is done afterwards by
telnetlib.
18.15.2 Telnet Example

A simple example illustrating typical use:

```python
import getpass
import sys
import telnetlib

HOST = "localhost"
user = raw_input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)
	n.read_until("login: ")
tn.write(user + "\n")
if password:
    tn.read_until("Password: ")
    tn.write(password + "\n")

tn.write("ls\n")
tn.write("exit\n")

print tn.read_all()
```

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.16 uuid -- UUID objects according to RFC 4122

New in version 2.5.

This module provides immutable UUID objects (the UUID class) and the functions uuid1(), uuid3(), uuid4(), uuid5() for generating version 1, 3, 4, and 5 UUIDs as specified in RFC 4122.

If all you want is a unique ID, you should probably call uuid1() or uuid4(). Note that uuid1() may compromise privacy since it creates a UUID containing the computer's network address. uuid4() creates a random UUID.

class UUID([hex[, bytes[, bytes_le[, fields[, int[, version]]]]]])

Create a UUID from either a string of 32 hexadecimal digits, a string of 16 bytes as the bytes argument, a string of 16 bytes in little-endian order as the bytes_le argument, a tuple of six integers (32-bit time_low, 16-bit time_mid, 16-bit time_hi_version, 8-bit clock_seq_hi_variant, 8-bit clock_seq_low, 48-bit node) as the fields argument, or a single 128-bit integer as the int argument. When a string of hex digits is given, curly braces, hyphens, and a URN prefix are all optional. For example, these expressions all yield the same UUID:

```python
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes='\x12\x34\x56\x78'*4)
UUID(bytes_le='\x78\x56\x34\x12\x12\x34\x12\x78\x56' + '\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x56781234))
UUID(int=0x12345678123456781234567812345678)
```

Exactly one of hex, bytes, bytes_le, fields, or int must be given. The version argument is optional; if given, the resulting UUID will have its variant and version number set according to RFC 4122, overriding bits in the given hex, bytes, bytes_le, fields, or int.
UUID instances have these read-only attributes:

**bytes**
The UUID as a 16-byte string (containing the six integer fields in big-endian byte order).

**bytes_le**
The UUID as a 16-byte string (with time_low, time_mid, and time_hi_version in little-endian byte order).

**fields**
A tuple of the six integer fields of the UUID, which are also available as six individual attributes and two derived attributes:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>time_low</td>
<td>the first 32 bits of the UUID</td>
</tr>
<tr>
<td>time_mid</td>
<td>the next 16 bits of the UUID</td>
</tr>
<tr>
<td>time_hi_version</td>
<td>the next 16 bits of the UUID</td>
</tr>
<tr>
<td>clock_seq_hi_variant</td>
<td>the next 8 bits of the UUID</td>
</tr>
<tr>
<td>clock_seq_low</td>
<td>the next 8 bits of the UUID</td>
</tr>
<tr>
<td>node</td>
<td>the last 48 bits of the UUID</td>
</tr>
<tr>
<td>time</td>
<td>the 60-bit timestamp</td>
</tr>
<tr>
<td>clock_seq</td>
<td>the 14-bit sequence number</td>
</tr>
</tbody>
</table>

**hex**
The UUID as a 32-character hexadecimal string.

**int**
The UUID as a 128-bit integer.

**urn**
The UUID as a URN as specified in RFC 4122.

**variant**
The UUID variant, which determines the internal layout of the UUID. This will be one of the integer constants RESERVED_NCS, RFC_4122, RESERVED_MICROSOFT, or RESERVED_FUTURE.
version
The UUID version number (1 through 5, meaningful only when the variant is RFC_4122).

The uuid module defines the following functions:

getnode()
Get the hardware address as a 48-bit positive integer. The first time this runs, it may launch a separate program, which could be quite slow. If all attempts to obtain the hardware address fail, we choose a random 48-bit number with its eighth bit set to 1 as recommended in RFC 4122. "Hardware address" means the MAC address of a network interface, and on a machine with multiple network interfaces the MAC address of any one of them may be returned.

uuid1([node[, clock_seq]])
Generate a UUID from a host ID, sequence number, and the current time. If node is not given, getnode() is used to obtain the hardware address. If clock_seq is given, it is used as the sequence number; otherwise a random 14-bit sequence number is chosen.

uuid3(namespace, name)
Generate a UUID based on the MD5 hash of a namespace identifier (which is a UUID) and a name (which is a string).

uuid4()
Generate a random UUID.

uuid5(namespace, name)
Generate a UUID based on the SHA-1 hash of a namespace identifier (which is a UUID) and a name (which is a string).

The uuid module defines the following namespace identifiers for use with uuid3() or uuid5().

NAMESPACE_DNS
When this namespace is specified, the name string is a fully-qualified domain name.
NAMESPACE_URL
When this namespace is specified, the name string is a URL.

NAMESPACE_OID
When this namespace is specified, the name string is an ISO OID.

NAMESPACE_X500
When this namespace is specified, the name string is an X.500 DN in DER or a text output format.

The uuid module defines the following constants for the possible values of the variant attribute:

RESERVED_NCS
Reserved for NCS compatibility.

RFC_4122
Specifies the UUID layout given in RFC 4122.

RESERVED_MICROSOFT
Reserved for Microsoft compatibility.

RESERVED_FUTURE
Reserved for future definition.

See Also:

RFC 4122, A Universally Unique IDentifier (UUID) URN Namespace
This specification defines a Uniform Resource Name namespace for UUIDs, the internal format of UUIDs, and methods of generating UUIDs.

Subsections

• 18.16.1 Example
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.16.1 Example

Here are some examples of typical usage of the uuid module:

```python
>>> import uuid

# make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

# make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

# make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

# make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9ae199e5d')

# make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

# convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

# get the raw 16 bytes of the UUID
>>> x.bytes
'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

# make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```
18.17 urlparse -- Parse URLs into components

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a ```relative URL``` to an absolute URL given a ```base URL```.

The module has been designed to match the Internet RFC on Relative Uniform Resource Locators (and discovered a bug in an earlier draft!). It supports the following URL schemes: file, ftp, gopher, hd1, http, https, imap, mailto, mms, news, nntp, prospero, rsync, rtsp, rtspu, sftp, shttp, sip, sips, snews, svn, svn+ssh, telnet, wais.

New in version 2.5: Support for the sftp and sips schemes.

The `urlparse` module defines the following functions:

`urlparse(urlstring[, default_scheme[, allow_fragments]])`

Parse a URL into six components, returning a 6-tuple. This corresponds to the general structure of a URL: `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the `path` component, which is retained if present. For example:

```python
>>> from urlparse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o
('http', 'www.cwi.nl:80', '/%7Eguido/Python.html', '', '', '')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```
If the `default_scheme` argument is specified, it gives the default addressing scheme, to be used only if the URL does not specify one. The default value for this argument is the empty string.

If the `allow_fragments` argument is false, fragment identifiers are not allowed, even if the URL's addressing scheme normally does support them. The default value for this argument is `True`.

The return value is actually an instance of a subclass of tuple. This class has the following additional read-only convenience attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Index</th>
<th>Value</th>
<th>Value if not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme</td>
<td>0</td>
<td>URL scheme specifier</td>
<td>empty string</td>
</tr>
<tr>
<td>netloc</td>
<td>1</td>
<td>Network location part</td>
<td>empty string</td>
</tr>
<tr>
<td>path</td>
<td>2</td>
<td>Hierarchical path</td>
<td>empty string</td>
</tr>
<tr>
<td>params</td>
<td>3</td>
<td>Parameters for last path element</td>
<td>empty string</td>
</tr>
<tr>
<td>query</td>
<td>4</td>
<td>Query component</td>
<td>empty string</td>
</tr>
<tr>
<td>fragment</td>
<td>5</td>
<td>Fragment identifier</td>
<td>empty string</td>
</tr>
<tr>
<td>username</td>
<td></td>
<td>User name</td>
<td>None</td>
</tr>
<tr>
<td>password</td>
<td></td>
<td>Password</td>
<td>None</td>
</tr>
<tr>
<td>hostname</td>
<td></td>
<td>Host name (lower case)</td>
<td>None</td>
</tr>
<tr>
<td>port</td>
<td></td>
<td>Port number as integer, if present</td>
<td>None</td>
</tr>
</tbody>
</table>

See section 18.17.1,``Results of `urlparse()` and `urlsplit()`,'' for more information on the result object.

Changed in version 2.5: Added attributes to return value.

**urlunparse**(parts)

Construct a URL from a tuple as returned by `urlparse()`. The `parts` argument be any six-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).
urlsplit( urlstring[, default_scheme[, allow_segments]])

This is similar to urlparse(), but does not split the params from the URL. This should generally be used instead of urlparse() if the more recent URL syntax allowing parameters to be applied to each segment of the path portion of the URL (see RFC 2396) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-tuple: (addressing scheme, network location, path, query, fragment identifier).

The return value is actually an instance of a subclass of tuple. This class has the following additional read-only convenience attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Index</th>
<th>Value</th>
<th>Value if not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme</td>
<td>0</td>
<td>URL scheme specifier</td>
<td>empty string</td>
</tr>
<tr>
<td>netloc</td>
<td>1</td>
<td>Network location part</td>
<td>empty string</td>
</tr>
<tr>
<td>path</td>
<td>2</td>
<td>Hierarchical path</td>
<td>empty string</td>
</tr>
<tr>
<td>query</td>
<td>3</td>
<td>Query component</td>
<td>empty string</td>
</tr>
<tr>
<td>fragment</td>
<td>4</td>
<td>Fragment identifier</td>
<td>empty string</td>
</tr>
<tr>
<td>username</td>
<td></td>
<td>User name</td>
<td>None</td>
</tr>
<tr>
<td>password</td>
<td></td>
<td>Password</td>
<td>None</td>
</tr>
<tr>
<td>hostname</td>
<td></td>
<td>Host name (lower case)</td>
<td>None</td>
</tr>
<tr>
<td>port</td>
<td></td>
<td>Port number as integer, if present</td>
<td>None</td>
</tr>
</tbody>
</table>

See section 18.17.1, `Results of urlparse() and urlsplit()`, for more information on the result object.

New in version 2.2. Changed in version 2.5: Added attributes to return value.

urlunsplit(parts)

Combine the elements of a tuple as returned by urlsplit() into a complete URL as a string. The parts argument be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent). New in version 2.2.
urljoin(\(base, url[, allow\_fragments]\))

Construct a full (``absolute'') URL by combining a ``base URL'' (\(base\)) with a ``relative URL'' (\(url\)). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example:

```python
>>> from urlparse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The \(allow\_fragments\) argument has the same meaning and default as for urlparse().

urldefrag(\(url\))

If \(url\) contains a fragment identifier, returns a modified version of \(url\) with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in \(url\), returns \(url\) unmodified and an empty string.

---

**See Also:**

- **RFC 1738, Uniform Resource Locators (URL)**
  This specifies the formal syntax and semantics of absolute URLs.

- **RFC 1808, Relative Uniform Resource Locators**
  This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of ``Abnormal Examples'' which govern the treatment of border cases.

- **RFC 2396, Uniform Resource Identifiers (URI): Generic Syntax**
  Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

---

**Subsections**

- **18.17.1 Results of urlparse() and urlsplit()**
Results of urlparse()
18.17.1 Results of urlparse() and urlsplit()

The result objects from the urlparse() and urlsplit() functions are subclasses of the tuple type. These subclasses add the attributes described in those functions, as well as provide an additional method:

geturl()

Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme will always be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

The result of this method is a fixpoint if passed back through the original parsing function:

```python
>>> import urlparse
>>> url = 'HTTP://www.Python.org/doc/#'

>>> r1 = urlparse.urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/

>>> r2 = urlparse.urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

New in version 2.5.

The following classes provide the implementations of the parse results:

class BaseResult

Base class for the concrete result classes. This provides most of the attribute definitions. It does not provide a geturl() method. It is derived from tuple, but does not override the __init__() or __new__() methods.

class ParseResult(scheme, netloc, path, params, query, fragment)

Concrete class for urlparse() results. The __new__() method is overridden to support checking that the right number of arguments are passed.
class SplitResult(scheme, netloc, path, query, fragment)

Concrete class for urlsplit() results. The __new__() method is overridden to support checking that the right number of arguments are passed.
18.18 SocketServer -- A framework for network servers

The SocketServer module simplifies the task of writing network servers.

There are four basic server classes: TCPServer uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. UDPServer uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The more infrequently used UnixStreamServer and UnixDatagramServer classes are similar, but use UNIX domain sockets; they're not available on non-UNIX platforms. For more details on network programming, consult a book such as W. Richard Steven's UNIX Network Programming or Ralph Davis's Win32 Network Programming.

These four classes process requests synchronously; each request must be completed before the next request can be started. This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the ForkingMixIn and ThreadingMixIn mix-in classes can be used to support asynchronous behaviour.

Creating a server requires several steps. First, you must create a request handler class by subclassing the BaseRequestHandler class and overriding its handle() method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server's address and the request handler class. Finally, call the handle_request() or serve_forever() method of the server object to process one or many requests.

When inheriting from ThreadingMixIn for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The ThreadingMixIn class defines an attribute daemon_threads, which indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is False, meaning that Python will not exit until all
threads created by ThreadingMixIn have exited.

Server classes have the same external methods and attributes, no matter what network protocol they use:

Subsections

- 18.18.1 Server Creation Notes
- 18.18.2 Server Objects
- 18.18.3 RequestHandler Objects
### 18.18.1 Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:

```
+------------+
| BaseServer |
+------------+
    |        |
    v
+-----------+  +------------------+
| TCPServer | -->| UnixStreamServer |
+-----------+  +------------------+
    |        |
    v
+-----------+  +--------------------+
| UDPServer | -->| UnixDatagramServer |
+-----------+  +--------------------+
```

Note that `UnixDatagramServer` derives from `UDPServer`, not from `UnixStreamServer` -- the only difference between an IP and a UNIX stream server is the address family, which is simply repeated in both UNIX server classes.

Forking and threading versions of each type of server can be created using the `ForkingMixIn` and `ThreadingMixIn` mix-in classes. For instance, a threading UDP server class is created as follows:

```python
class ThreadingUDPServer(ThreadingMixIn, UDPServer): pass
```

The mix-in class must come first, since it overrides a method defined in `UDPServer`. Setting the various member variables also changes the behavior of the underlying server mechanism.

To implement a service, you must derive a class from `BaseRequestHandler` and redefine its `handle()` method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses `StreamRequestHandler` or `DatagramRequestHandler`.
Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service "deaf" while one request is being handled - which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the request handler class handle() method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor fork() (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use select() to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used).
18.18.2 Server Objects

fileno()

Return an integer file descriptor for the socket on which the server is
listening. This function is most commonly passed to select.select(),
to allow monitoring multiple servers in the same process.

handle_request()

Process a single request. This function calls the following methods in order:
get_request(), verify_request(), and
process_request(). If the user-provided handle() method of the
handler class raises an exception, the server's handle_error() method
will be called.

serve_forever()

Handle an infinite number of requests. This simply calls
handle_request() inside an infinite loop.

address_family

The family of protocols to which the server's socket belongs.
socket.AF_INET and socket.AF_UNIX are two possible values.

RequestHandlerClass

The user-provided request handler class; an instance of this class is created
for each request.

server_address

The address on which the server is listening. The format of addresses varies
depending on the protocol family; see the documentation for the socket
module for details. For Internet protocols, this is a tuple containing a string
giving the address, and an integer port number: ('127.0.0.1', 80),
for example.

socket

The socket object on which the server will listen for incoming requests.

The server classes support the following class variables:
**allow_reuse_address**
Whether the server will allow the reuse of an address. This defaults to `False`, and can be set in subclasses to change the policy.

**request_queue_size**
The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to `request_queue_size` requests. Once the queue is full, further requests from clients will get a ``Connection denied'' error. The default value is usually 5, but this can be overridden by subclasses.

**socket_type**
The type of socket used by the server; `socket.SOCK_STREAM` and `socket.SOCK_DGRAM` are two possible values.

There are various server methods that can be overridden by subclasses of base server classes like `TCPServer`; these methods aren't useful to external users of the server object.

**finish_request()**
Actually processes the request by instantiating `RequestHandlerClass` and calling its `handle()` method.

**get_request()**
Must accept a request from the socket, and return a 2-tuple containing the `new` socket object to be used to communicate with the client, and the client's address.

**handle_error**( `request, client_address`)  
This function is called if the `RequestHandlerClass`'s `handle()` method raises an exception. The default action is to print the traceback to standard output and continue handling further requests.

**process_request**( `request, client_address`)  
Calls `finish_request()` to create an instance of the `RequestHandlerClass`. If desired, this function can create a new process or thread to handle the request; the `ForkingMixIn` and `ThreadingMixIn` classes do this.
**server_activate()**

Called by the server's constructor to activate the server. The default behavior just listens to the server's socket. May be overridden.

**server_bind()**

Called by the server's constructor to bind the socket to the desired address. May be overridden.

**verify_request(request, client_address)**

Must return a Boolean value; if the value is True, the request will be processed, and if it's False, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns True.
18.18.3 RequestHandler Objects

The request handler class must define a new handle() method, and can override any of the following methods. A new instance is created for each request.

**finish()**

Called after the handle() method to perform any clean-up actions required. The default implementation does nothing. If setup() or handle() raise an exception, this function will not be called.

**handle()**

This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as self.request; the client address as self.client_address; and the server instance as self.server, in case it needs access to per-server information.

The type of self.request is different for datagram or stream services. For stream services, self.request is a socket object; for datagram services, self.request is a string. However, this can be hidden by using the request handler subclasses StreamRequestHandler or DatagramRequestHandler, which override the setup() and finish() methods, and provide self.rfile and self.wfile attributes. self.rfile and self.wfile can be read or written, respectively, to get the request data or return data to the client.

**setup()**

Called before the handle() method to perform any initialization actions required. The default implementation does nothing.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.19 BaseHTTPServer -- Basic HTTP server

This module defines two classes for implementing HTTP servers (Web servers). Usually, this module isn't used directly, but is used as a basis for building functioning Web servers. See the SimpleHTTPServer and CGIHTTPServer modules.

The first class, HTTPServer, is a SocketServer.TCPServer subclass. It creates and listens at the HTTP socket, dispatching the requests to a handler. Code to create and run the server looks like this:

```python
def run(server_class=BaseHTTPServer.HTTPServer,
       handler_class=BaseHTTPServer.BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

**class HTTPServer**( `server_address`, `RequestHandlerClass` )

This class builds on the TCPServer class by storing the server address as instance variables named server_name and server_port. The server is accessible by the handler, typically through the handler's server instance variable.

**class BaseHTTPRequestHandler**( `request`, `client_address`, `server` )

This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). BaseHTTPRequestHandler provides a number of class and instance variables, and methods for use by subclasses.

The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method "SPAM", the `do_SPAM()` method will be called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to
override or extend the __init__() method.

BaseHTTPRequestHandler has the following instance variables:

**client_address**
Contains a tuple of the form (host, port) referring to the client's address.

**command**
Contains the command (request type). For example, 'GET'.

**path**
Contains the request path.

**request_version**
Contains the version string from the request. For example, 'HTTP/1.0'.

**headers**
Holds an instance of the class specified by the MessageClass class variable. This instance parses and manages the headers in the HTTP request.

**rfile**
Contains an input stream, positioned at the start of the optional input data.

**wfile**
Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream.

BaseHTTPRequestHandler has the following class variables:

**server_version**
Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form name[/version]. For example, 'BaseHTTP/0.2'.

**sys_version**
Contains the Python system version, in a form usable by the version_string method and the server_version class variable. For example, 'Python/1.4'.

**error_message_format**
Specifications a format string for building an error response to the client. It uses parenthesized, keyed format specifiers, so the format operand must be a dictionary. The *code* key should be an integer, specifying the numeric HTTP error code value. *message* should be a string containing a (detailed) error message of what occurred, and *explain* should be an explanation of the error code number. Default *message* and *explain* values can found in the *responses* class variable.

**protocol_version**
This specifies the HTTP protocol version used in responses. If set to 'HTTP/1.1', the server will permit HTTP persistent connections; however, your server *must* then include an accurate *Content-Length* header (using *send_header()*) in all of its responses to clients. For backwards compatibility, the setting defaults to 'HTTP/1.0'.

**MessageClass**
Specifies a *rfc822.Message*-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to *mimetypes.Message*.

**responses**
This variable contains a mapping of error code integers to two-element tuples containing a short and long message. For example, `{*code*: (*shortmessage*, *longmessage*)}`. The *shortmessage* is usually used as the *message* key in an error response, and *longmessage* as the *explain* key (see the *error_message_format* class variable).

A *BaseHTTPRequestHandler* instance has the following methods:

**handle()**
Calls *handle_one_request()* once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it; instead, implement appropriate *do_*(**) methods.

**handle_one_request()**
This method will parse and dispatch the request to the appropriate *do_*(**) method. You should never need to override it.
**send_error**(*code*, *message*)
Sends and logs a complete error reply to the client. The numeric *code* specifies the HTTP error code, with *message* as optional, more specific text. A complete set of headers is sent, followed by text composed using the error_message_format class variable.

**send_response**(*code*, *message*)
Sends a response header and logs the accepted request. The HTTP response line is sent, followed by *Server* and *Date* headers. The values for these two headers are picked up from the version_string() and date_time_string() methods, respectively.

**send_header**(*keyword*, *value*)
Writes a specific HTTP header to the output stream. *keyword* should specify the header keyword, with *value* specifying its value.

**end_headers()**
Sends a blank line, indicating the end of the HTTP headers in the response.

**log_request**([*code*, *size]*)
Logs an accepted (successful) request. *code* should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the *size* parameter.

**log_error**(*...*)
Logs an error when a request cannot be fulfilled. By default, it passes the message to log_message(), so it takes the same arguments (*format* and additional values).

**log_message**(*format*, *...*)
Logs an arbitrary message to sys.stderr. This is typically overridden to create custom error logging mechanisms. The *format* argument is a standard printf-style format string, where the additional arguments to log_message() are applied as inputs to the formatting. The client address and current date and time are prefixed to every message logged.

**version_string()**
Returns the server software's version string. This is a combination of the server_version and sys_version class variables.

```
Returns the server software's version string. This is a combination of the server_version and sys_version class variables.

date_time_string([timestamp])
Returns the date and time given by timestamp (which must be in the format returned by time.time()), formatted for a message header. If timestamp is omitted, it uses the current date and time.

The result looks like 'Sun, 06 Nov 1994 08:49:37 GMT'. New in version 2.5: The timestamp parameter.

log_date_time_string()
Returns the current date and time, formatted for logging.

address_string()
Returns the client address, formatted for logging. A name lookup is performed on the client's IP address.
```

See Also:

Module **CGIHTTPServer**: Extended request handler that supports CGI scripts.

Module **SimpleHTTPServer**: Basic request handler that limits response to files actually under the document root.
18.20 SimpleHTTPServer -- Simple HTTP request handler

The SimpleHTTPServer module defines a request-handler class, interface-compatible with BaseHTTPServer.BaseHTTPRequestHandler, that serves files only from a base directory.

The SimpleHTTPServer module defines the following class:

```python
class SimpleHTTPRequestHandler(request, client_address, server)
```

This class is used to serve files from the current directory and below, directly mapping the directory structure to HTTP requests.

A lot of the work, such as parsing the request, is done by the base class BaseHTTPServer.BaseHTTPRequestHandler. This class implements the do_GET() and do_HEAD() functions.

The SimpleHTTPRequestHandler defines the following member variables:

- **server_version**
  - This will be "SimpleHTTP/" + __version__, where __version__ is defined in the module.

- **extensions_map**
  - A dictionary mapping suffixes into MIME types. The default is signified by an empty string, and is considered to be application/octet-stream. The mapping is used case-insensitively, and so should contain only lower-cased keys.

The SimpleHTTPRequestHandler defines the following methods:

- **do_HEAD()**
  - This method serves the 'HEAD' request type: it sends the headers it would send for the equivalent GET request. See the do_GET() method for a more
complete explanation of the possible headers.

**do_GET()**

The request is mapped to a local file by interpreting the request as a path relative to the current working directory.

If the request was mapped to a directory, the directory is checked for a file named `index.html` or `index.htm` (in that order). If found, the file's contents are returned; otherwise a directory listing is generated by calling the `list_directory()` method. This method uses `os.listdir()` to scan the directory, and returns a 404 error response if the `listdir()` fails.

If the request was mapped to a file, it is opened and the contents are returned. Any `IOError` exception in opening the requested file is mapped to a 404, 'File not found' error. Otherwise, the content type is guessed by calling the `guess_type()` method, which in turn uses the `extensions_map` variable.

A 'Content-type:' header with the guessed content type is output, followed by a 'Content-Length:' header with the file's size and a 'Last-Modified:' header with the file's modification time.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output. If the file's MIME type starts with `text/` the file is opened in text mode; otherwise binary mode is used.

For example usage, see the implementation of the `test()` function. New in version 2.5: The 'Last-Modified' header.

---

**See Also:**

Module [BaseHTTPServer](https://docs.python.org/2/library/basehttpserver.html): Base class implementation for Web server and request handler.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.21 CGIHTTPServer -- CGI-capable HTTP request handler

The CGIHTTPServer module defines a request-handler class, interface compatible with BaseHTTPServer.BaseHTTPRequestHandler and inherits behavior from SimpleHTTPServer.SimpleHTTPRequestHandler but can also run CGI scripts.

**Note:** This module can run CGI scripts on UNIX and Windows systems; on Mac OS it will only be able to run Python scripts within the same process as itself.

**Note:** CGI scripts run by the CGIHTTPRequestHandler class cannot execute redirects (HTTP code 302), because code 200 (script output follows) is sent prior to execution of the CGI script. This pre-empts the status code.

The CGIHTTPServer module defines the following class:

```python
class CGIHTTPRequestHandler(request, client_address, server)
```

This class is used to serve either files or output of CGI scripts from the current directory and below. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in SimpleHTTPServer.SimpleHTTPRequestHandler.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only directory-based CGI are used -- the other common server configuration is to treat special extensions as denoting CGI scripts.

The `do_GET()` and `do_HEAD()` functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the `cgi_directories` path.

The CGIHTTPRequestHandler defines the following data member:

```python
cgi_directories```
This defaults to ["/cgi-bin", "/htbin"] and describes directories to treat as containing CGI scripts.

The CGIHTTPRequestHandler defines the following methods:

**do_POST()**

This method serves the 'POST' request type, only allowed for CGI scripts. Error 501, "Can only POST to CGI scripts", is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

For example usage, see the implementation of the test() function.

---

**See Also:**

Module **BaseHTTPServer**: Base class implementation for Web server and request handler.

---

*Python Library Reference*


Release 2.5, documentation updated on 19th September, 2006.

See About this document... for information on suggesting changes.
18.22 cookielib -- Cookie handling for HTTP clients

New in version 2.4.

The cookielib module defines classes for automatic handling of HTTP cookies. It is useful for accessing web sites that require small pieces of data - cookies - to be set on the client machine by an HTTP response from a web server, and then returned to the server in later HTTP requests.

Both the regular Netscape cookie protocol and the protocol defined by RFC 2965 are handled. RFC 2965 handling is switched off by default. RFC 2109 cookies are parsed as Netscape cookies and subsequently treated either as Netscape or RFC 2965 cookies according to the 'policy' in effect. Note that the great majority of cookies on the Internet are Netscape cookies. cookielib attempts to follow the de-facto Netscape cookie protocol (which differs substantially from that set out in the original Netscape specification), including taking note of the max-age and port cookie-attributes introduced with RFC 2965. Note: The various named parameters found in Set-Cookie: and Set-Cookie2: headers (eg. domain and expires) are conventionally referred to as attributes. To distinguish them from Python attributes, the documentation for this module uses the term cookie-attribute instead.

The module defines the following exception:

**exception LoadError**

Instances of FileCookieJar raise this exception on failure to load cookies from a file. Note: For backwards-compatibility with Python 2.4 (which raised an IOError), LoadError is a subclass of IOError.

The following classes are provided:

**class CookieJar**(policy=None)

policy is an object implementing the CookiePolicy interface.

The CookieJar class stores HTTP cookies. It extracts cookies from
HTTP requests, and returns them in HTTP responses. CookieJar instances automatically expire contained cookies when necessary. Subclasses are also responsible for storing and retrieving cookies from a file or database.

**class FileCookieJar**( `filename, delayload=None, policy=None` )

*policy* is an object implementing the CookiePolicy interface. For the other arguments, see the documentation for the corresponding attributes.

A CookieJar which can load cookies from, and perhaps save cookies to, a file on disk. Cookies are NOT loaded from the named file until either the `load()` or `revert()` method is called. Subclasses of this class are documented in section 18.22.2.

**class CookiePolicy()**

This class is responsible for deciding whether each cookie should be accepted from / returned to the server.

**class DefaultCookiePolicy**( `blocked_domains=None, allowed_domains=None, netscape=True, rfc2965=False, rfc2109=False, hide_cookie2=False, strict_domain=False, strict_rfc2965_unverifiable=True, strict_ns_domain=None, strict_ns_set_initial_dollar=False, strict...` )

Constructor arguments should be passed as keyword arguments only. `blocked_domains` is a sequence of domain names that we never accept cookies from, nor return cookies to. `allowed_domains` if not None, this is a sequence of the only domains for which we accept and return cookies. For all other arguments, see the documentation for CookiePolicy and DefaultCookiePolicy objects.

DefaultCookiePolicy implements the standard accept / reject rules for Netscape and RFC 2965 cookies. By default, RFC 2109 cookies (ie. cookies received in a Set-Cookie: header with a version cookie-attribute of 1) are treated according to the RFC 2965 rules. However, if RFC 2965 handling is turned off or `rfc2109_as_netscape` is True, RFC 2109 cookies are 'downgraded' by the CookieJar instance to Netscape cookies,
by setting the version attribute of the Cookie instance to 0. DefaultCookiePolicy also provides some parameters to allow some fine-tuning of policy.

**class Cookie()**

This class represents Netscape, RFC 2109 and RFC 2965 cookies. It is not expected that users of cookielib construct their own Cookie instances. Instead, if necessary, call make_cookies() on a CookieJar instance.

**See Also:**

Module **urllib2**: URL opening with automatic cookie handling.

Module **Cookie**: HTTP cookie classes, principally useful for server-side code. The cookielib and Cookie modules do not depend on each other.

http://wwwsearch.sf.net/ClientCookie/
Extensions to this module, including a class for reading Microsoft Internet Explorer cookies on Windows.

http://www.netscape.com/newsref/std/cookie_spec.html
The specification of the original Netscape cookie protocol. Though this is still the dominant protocol, the 'Netscape cookie protocol' implemented by all the major browsers (and cookielib) only bears a passing resemblance to the one sketched out in cookie_spec.html.

RFC 2109, **HTTP State Management Mechanism**

RFC 2965, **HTTP State Management Mechanism**
The Netscape protocol with the bugs fixed. Uses Set-Cookie2: in place of Set-Cookie:. Not widely used.

http://kristol.org/cookie/errata.html
Unfinished errata to RFC 2965.
Subsections

- [18.22.1](#) CookieJar and FileCookieJar Objects
- [18.22.2](#) FileCookieJar subclasses and co-operation with web browsers
- [18.22.3](#) CookiePolicy Objects
- [18.22.4](#) DefaultCookiePolicy Objects
- [18.22.5](#) Cookie Objects
- [18.22.6](#) Examples

Python Library Reference

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
18.22.1 CookieJar and FileCookieJar Objects

CookieJar objects support the iterator protocol for iterating over contained Cookie objects.

CookieJar has the following methods:

**add_cookie_header**( *request*)
Add correct Cookie: header to *request*.

If policy allows (i.e., the rfc2965 and hide_cookie2 attributes of the CookieJar's CookiePolicy instance are true and false respectively), the Cookie2: header is also added when appropriate.

The *request* object (usually a urllib2.Request instance) must support the methods get_full_url(), get_host(), get_type(), unverifiable(), get_origin_req_host(), has_header(), get_header(), header_items(), and add_unredirected_header(), as documented by urllib2.

**extract_cookies**( *response*, *request*)
Extract cookies from HTTP *response* and store them in the CookieJar, where allowed by policy.

The CookieJar will look for allowable Set-Cookie: and Set-Cookie2: headers in the *response* argument, and store cookies as appropriate (subject to the CookiePolicy.set_ok() method's approval).

The *response* object (usually the result of a call to urllib2.urlopen(), or similar) should support an info() method, which returns an object with a getallmatchingheaders() method (usually a mimetools.Message instance).

The *request* object (usually a urllib2.Request instance) must support the methods get_full_url(), get_host(), unverifiable(), and get_origin_req_host(), as documented by urllib2. The request is used to set default values for cookie-attributes as well as for
checking that the cookie is allowed to be set.

**set_policy**(policy)
Set the CookiePolicy instance to be used.

**make_cookies**(response, request)
Return sequence of Cookie objects extracted from response object.

See the documentation for *extract_cookies* for the interfaces required of the response and request arguments.

**set_cookie_if_ok**(cookie, request)
Set a Cookie if policy says it's OK to do so.

**set_cookie**(cookie)
Set a Cookie, without checking with policy to see whether or not it should be set.

**clear**(domain[, path[, name]])
Clear some cookies.

If invoked without arguments, clear all cookies. If given a single argument, only cookies belonging to that domain will be removed. If given two arguments, cookies belonging to the specified domain and URL path are removed. If given three arguments, then the cookie with the specified domain, path and name is removed.

 Raises KeyError if no matching cookie exists.

**clear_session_cookies**( )
Discard all session cookies.

Discards all contained cookies that have a true discard attribute (usually because they had either no max-age or expires cookie-attribute, or an explicit discard cookie-attribute). For interactive browsers, the end of a session usually corresponds to closing the browser window.

Note that the *save()* method won't save session cookies anyway, unless
you ask otherwise by passing a true `ignore_discard` argument.

FileCookieJar implements the following additional methods:

```python
save(filename=None, ignore_discard=False, ignore_expires=False)
```
Save cookies to a file.

This base class raises `NotImplementedError`. Subclasses may leave this method unimplemented.

`filename` is the name of file in which to save cookies. If `filename` is not specified, `self.filename` is used (whose default is the value passed to the constructor, if any); if `self.filename` is `None`, `ValueError` is raised.

`ignore_discard`: save even cookies set to be discarded. `ignore_expires`: save even cookies that have expired

The file is overwritten if it already exists, thus wiping all the cookies it contains. Saved cookies can be restored later using the `load()` or `revert()` methods.

```python
load(filename=None, ignore_discard=False, ignore_expires=False)
```
Load cookies from a file.

Old cookies are kept unless overwritten by newly loaded ones.

Arguments are as for `save()`.

The named file must be in the format understood by the class, or `LoadError` will be raised. Also, `IOError` may be raised, for example if the file does not exist. **Note:** For backwards-compatibility with Python 2.4 (which raised an `IOError`), `LoadError` is a subclass of `IOError`.

```python
revert(filename=None, ignore_discard=False, ignore_expires=False)
```
Clear all cookies and reload cookies from a saved file.

`revert()` can raise the same exceptions as `load()`. If there is a failure, the object's state will not be altered.
FileCookieJar instances have the following public attributes:

**filename**
Filename of default file in which to keep cookies. This attribute may be assigned to.

**delayload**
If true, load cookies lazily from disk. This attribute should not be assigned to. This is only a hint, since this only affects performance, not behaviour (unless the cookies on disk are changing). A CookieJar object may ignore it. None of the FileCookieJar classes included in the standard library lazily loads cookies.
18.22.2 FileCookieJar subclasses and cooperation with web browsers

The following CookieJar subclasses are provided for reading and writing. Further CookieJar subclasses, including one that reads Microsoft Internet Explorer cookies, are available at http://wwwsearch.sf.net/ClientCookie/.

**class MozillaCookieJar**(*filename*, *delayload=None*, *policy=None*)

A FileCookieJar that can load from and save cookies to disk in the Mozilla cookies.txt file format (which is also used by the Lynx and Netscape browsers). **Note:** This loses information about RFC 2965 cookies, and also about newer or non-standard cookie-attributes such as port.

**Warning:** Back up your cookies before saving if you have cookies whose loss / corruption would be inconvenient (there are some subtleties which may lead to slight changes in the file over a load / save round-trip).

Also note that cookies saved while Mozilla is running will get clobbered by Mozilla.

**class LWPCookieJar**(*filename*, *delayload=None*, *policy=None*)

A FileCookieJar that can load from and save cookies to disk in format compatible with the libwww-perl library's Set-Cookie3 file format. This is convenient if you want to store cookies in a human-readable file.
18.22.3 CookiePolicy Objects

Objects implementing the CookiePolicy interface have the following methods:

set_ok(cookie, request)
Return boolean value indicating whether cookie should be accepted from server.

cookie is a cookielib.Cookie instance. request is an object implementing the interface defined by the documentation for CookieJar.extract_cookies().

return_ok(cookie, request)
Return boolean value indicating whether cookie should be returned to server.

cookie is a cookielib.Cookie instance. request is an object implementing the interface defined by the documentation for CookieJar.add_cookie_header().

domain_return_ok(domain, request)
Return false if cookies should not be returned, given cookie domain.

This method is an optimization. It removes the need for checking every cookie with a particular domain (which might involve reading many files). Returning true from domain_return_ok() and path_return_ok() leaves all the work to return_ok().

If domain_return_ok() returns true for the cookie domain, path_return_ok() is called for the cookie path. Otherwise, path_return_ok() and return_ok() are never called for that cookie domain. If path_return_ok() returns true, return_ok() is called with the Cookie object itself for a full check. Otherwise, return_ok() is never called for that cookie path.

Note that domain_return_ok() is called for every cookie domain, not
just for the request domain. For example, the function might be called with both ".example.com" and "www.example.com" if the request domain is "www.example.com". The same goes for path_return_ok().

The request argument is as documented for return_ok().

\[ \text{path\_return\_ok( } \text{path, request) } \]

Return false if cookies should not be returned, given cookie path.

See the documentation for domain_return_ok().

In addition to implementing the methods above, implementations of the CookiePolicy interface must also supply the following attributes, indicating which protocols should be used, and how. All of these attributes may be assigned to.

\textit{netscape}

Implement Netscape protocol.

\textit{rfc2965}

Implement RFC 2965 protocol.

\textit{hide\_cookie2}

Don't add Cookie2: header to requests (the presence of this header indicates to the server that we understand RFC 2965 cookies).

The most useful way to define a CookiePolicy class is by subclassing from DefaultCookiePolicy and overriding some or all of the methods above. CookiePolicy itself may be used as a 'null policy' to allow setting and receiving any and all cookies (this is unlikely to be useful).
18.22.4 DefaultCookiePolicy Objects

Implements the standard rules for accepting and returning cookies.

Both RFC 2965 and Netscape cookies are covered. RFC 2965 handling is switched off by default.

The easiest way to provide your own policy is to override this class and call its methods in your overridden implementations before adding your own additional checks:

```python
import cookielib

class MyCookiePolicy(cookielib.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not cookielib.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

In addition to the features required to implement the CookiePolicy interface, this class allows you to block and allow domains from setting and receiving cookies. There are also some strictness switches that allow you to tighten up the rather loose Netscape protocol rules a little bit (at the cost of blocking some benign cookies).

A domain blacklist and whitelist is provided (both off by default). Only domains not in the blacklist and present in the whitelist (if the whitelist is active) participate in cookie setting and returning. Use the `blocked_domains` constructor argument, and `blocked_domains()` and `set_blocked_domains()` methods (and the corresponding argument and methods for `allowed_domains`). If you set a whitelist, you can turn it off again by setting it to `None`.

Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, "example.com" matches a blacklist entry of "example.com", but "www.example.com" does not. Domains that do start with a dot are matched by more specific domains too. For example, both "www.example.com" and "www.coyote.example.com" match ".example.com" (but "example.com" itself does not). IP addresses are an exception, and must match exactly. For example, if blocked_domains contains
"192.168.1.2" and ".168.1.2", 192.168.1.2 is blocked, but 193.168.1.2 is not.

DefaultCookiePolicy implements the following additional methods:

**blocked_domains()**
Return the sequence of blocked domains (as a tuple).

**set_blockedDomains(blocked_domains)**
Set the sequence of blocked domains.

**is_blocked(domain)**
Return whether *domain* is on the blacklist for setting or receiving cookies.

**allowed_domains()**
Return None, or the sequence of allowed domains (as a tuple).

**set_allowed_domains(allowed_domains)**
Set the sequence of allowed domains, or None.

**is_not_allowed(domain)**
Return whether *domain* is not on the whitelist for setting or receiving cookies.

DefaultCookiePolicy instances have the following attributes, which are all initialised from the constructor arguments of the same name, and which may all be assigned to.

**rfc2109_as_netscape**
If true, request that the CookieJar instance downgrade RFC 2109 cookies (i.e. cookies received in a Set-Cookie: header with a version cookie-attribute of 1) to Netscape cookies by setting the version attribute of the Cookie instance to 0. The default value is None, in which case RFC 2109 cookies are downgraded if and only if RFC 2965 handling is turned off. Therefore, RFC 2109 cookies are downgraded by default. New in version 2.5.

General strictness switches:
**strict_domain**
Don't allow sites to set two-component domains with country-code top-level domains like .co.uk, .gov.uk, .co.nz etc. This is far from perfect and isn't guaranteed to work!

RFC 2965 protocol strictness switches:

**strict_rfc2965_unverifiable**
Follow RFC 2965 rules on unverifiable transactions (usually, an unverifiable transaction is one resulting from a redirect or a request for an image hosted on another site). If this is false, cookies are *never* blocked on the basis of verifiability.

Netscape protocol strictness switches:

**strict_ns_unverifiable**
apply RFC 2965 rules on unverifiable transactions even to Netscape cookies

**strict_ns_domain**
Flags indicating how strict to be with domain-matching rules for Netscape cookies. See below for acceptable values.

**strict_ns_set_initial_dollar**
Ignore cookies in Set-Cookie: headers that have names starting with '$'.

**strict_ns_set_path**
Don't allow setting cookies whose path doesn't path-match request URI.

**strict_ns_domain** is a collection of flags. Its value is constructed by or-ing together (for example, DomainStrictNoDots|DomainStrictNonDomain means both flags are set).

**DomainStrictNoDots**
When setting cookies, the 'host prefix' must not contain a dot (eg. www.foo.bar.com can't set a cookie for .bar.com, because www.foo contains a dot).

**DomainStrictNonDomain**
Cookies that did not explicitly specify a domain cookie-attribute can only be returned to a domain equal to the domain that set the cookie (eg. spam.example.com won't be returned cookies from example.com that had no domain cookie-attribute).

**DomainRFC2965Match**
When setting cookies, require a full RFC 2965 domain-match.

The following attributes are provided for convenience, and are the most useful combinations of the above flags:

**DomainLiberal**
Equivalent to 0 (ie. all of the above Netscape domain strictness flags switched off).

**DomainStrict**
Equivalent to DomainStrictNoDots|DomainStrictNonDomain.
**18.22.5 Cookie Objects**

Cookie instances have Python attributes roughly corresponding to the standard cookie-attributes specified in the various cookie standards. The correspondence is not one-to-one, because there are complicated rules for assigning default values, because the max-age and expires cookie-attributes contain equivalent information, and because RFC 2109 cookies may be 'downgraded' by cookielib from version 1 to version 0 (Netscape) cookies.

Assignment to these attributes should not be necessary other than in rare circumstances in a CookiePolicy method. The class does not enforce internal consistency, so you should know what you're doing if you do that.

**version**

Integer or None. Netscape cookies have version 0. RFC 2965 and RFC 2109 cookies have a version cookie-attribute of 1. However, note that cookielib may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case version is 0.

**name**

Cookie name (a string).

**value**

Cookie value (a string), or None.

**port**

String representing a port or a set of ports (eg. '80', or '80,8080'), or None.

**path**

Cookie path (a string, eg. '/acme/rocket_launchers').

**secure**

True if cookie should only be returned over a secure connection.

**expires**

Integer expiry date in seconds since epoch, or None. See also the is_expired() method.
**discard**
True if this is a session cookie.

**comment**
String comment from the server explaining the function of this cookie, or None.

**comment_url**
URL linking to a comment from the server explaining the function of this cookie, or None.

**rfc2109**
True if this cookie was received as an RFC 2109 cookie (ie. the cookie arrived in a `Set-Cookie:` header, and the value of the Version cookie-attribute in that header was 1). This attribute is provided because `cookielib` may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case `version` is 0. New in version 2.5.

**port_specified**
True if a port or set of ports was explicitly specified by the server (in the `Set-Cookie: / Set-Cookie2:` header).

**domain_specified**
True if a domain was explicitly specified by the server.

**domain_initial_dot**
True if the domain explicitly specified by the server began with a dot (`'.').

Cookies may have additional non-standard cookie-attributes. These may be accessed using the following methods:

**has_nonstandard_attr**(name)
Return true if cookie has the named cookie-attribute.

**get_nonstandard_attr**(name, default= None)
If cookie has the named cookie-attribute, return its value. Otherwise, return `default`. 
**set_nonstandard_attr**( *name*, *value*)

Set the value of the named cookie-attribute.

The **Cookie** class also defines the following method:

**is_expired**( *[now=None]*)

True if cookie has passed the time at which the server requested it should expire. If *now* is given (in seconds since the epoch), return whether the cookie has expired at the specified time.
18.22.6 Examples

The first example shows the most common usage of cookielib:

```python
import cookielib, urllib2
cj = cookielib.CookieJar()
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

This example illustrates how to open a URL using your Netscape, Mozilla, or Lynx cookies (assumes UNIX/Netscape convention for location of the cookies file):

```python
import os, cookielib, urllib2
cj = cookielib.MozillaCookieJar()
cj.load(os.path.join(os.environ["HOME"], ".netscape/cookies.txt"))
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

The next example illustrates the use of DefaultCookiePolicy. Turn on RFC 2965 cookies, be more strict about domains when setting and returning Netscape cookies, and block some domains from setting cookies or having them returned:

```python
import urllib2
from cookielib import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```
18.23 Cookie -- HTTP state management

The Cookie module defines classes for abstracting the concept of cookies, an HTTP state management mechanism. It supports both simple string-only cookies, and provides an abstraction for having any serializable data-type as cookie value.

The module formerly strictly applied the parsing rules described in the RFC 2109 and RFC 2068 specifications. It has since been discovered that MSIE 3.0x doesn't follow the character rules outlined in those specs. As a result, the parsing rules used are a bit less strict.

**exception CookieError**
Exception failing because of RFC 2109 invalidity: incorrect attributes, incorrect Set-Cookie: header, etc.

**class BaseCookie([input])**
This class is a dictionary-like object whose keys are strings and whose values are Morsel instances. Note that upon setting a key to a value, the value is first converted to a Morsel containing the key and the value.

If input is given, it is passed to the load() method.

**class SimpleCookie([input])**
This class derives from BaseCookie and overrides value_decode() and value_encode() to be the identity and str() respectively.

**class SerialCookie([input])**
This class derives from BaseCookie and overrides value_decode() and value_encode() to be the pickle.loads() and pickle.dumps().

**Deprecated since release 2.3.** Reading pickled values from untrusted cookie data is a huge security hole, as pickle strings can be crafted to cause
arbitrary code to execute on your server. It is supported for backwards compatibility only, and may eventually go away.

**class SmartCookie([input])**

This class derives from `BaseCookie`. It overrides `value_decode()` to be `pickle.loads()` if it is a valid pickle, and otherwise the value itself. It overrides `value_encode()` to be `pickle.dumps()` unless it is a string, in which case it returns the value itself.

**Deprecated since release 2.3.** The same security warning from `SerialCookie` applies here.

A further security note is warranted. For backwards compatibility, the `Cookie` module exports a class named `Cookie` which is just an alias for `SmartCookie`. This is probably a mistake and will likely be removed in a future version. You should not use the `Cookie` class in your applications, for the same reason why you should not use the `SerialCookie` class.

See Also:

Module `cookielib`:

HTTP cookie handling for web `clients`. The `cookielib` and `Cookie` modules do not depend on each other.

**RFC 2109, HTTP State Management Mechanism**

This is the state management specification implemented by this module.

Subsections

- [18.23.1 Cookie Objects](#)
- [18.23.2 Morsel Objects](#)
- [18.23.3 Example](#)
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.23.1 Cookie Objects

value_decode( val)
Return a decoded value from a string representation. Return value can be any type. This method does nothing in BaseCookie -- it exists so it can be overridden.

value_encode( val)
Return an encoded value. val can be any type, but return value must be a string. This method does nothing in BaseCookie -- it exists so it can be overridden.

In general, it should be the case that value_encode() and value_decode() are inverses on the range of value_decode.

output([attrs, header[, sep]])
Return a string representation suitable to be sent as HTTP headers. attrs and header are sent to each Morsel's output() method. sep is used to join the headers together, and is by default the combination '\r\n' (CRLF). Changed in version 2.5: The default separator has been changed from '\n' to match the cookie specification.

js_output([attrs])
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP headers was sent.

The meaning for attrs is the same as in output().

load( rawdata)
If rawdata is a string, parse it as an HTTP_COOKIE and add the values found there as Morsels. If it is a dictionary, it is equivalent to:

    for k, v in rawdata.items():
        cookie[k] = v
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
18.23.2 Morsel Objects

class Morsel()

Abstract a key/value pair, which has some RFC 2109 attributes.

Morsels are dictionary-like objects, whose set of keys is constant -- the valid RFC 2109 attributes, which are

- expires
- path
- comment
- domain
- max-age
- secure
- version

The keys are case-insensitive.

value

The value of the cookie.

coded_value

The encoded value of the cookie -- this is what should be sent.

key

The name of the cookie.

set(key, value, coded_value)

Set the key, value and coded_value members.

isReservedKey(K)

Whether K is a member of the set of keys of a Morsel.

output([attrs[, header]])

Return a string representation of the Morsel, suitable to be sent as an HTTP header. By default, all the attributes are included, unless attrs is given, in which case it should be a list of attributes to use. header is by default
"Set-Cookie:"

js_output([attrs])
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP header was sent.

The meaning for attrs is the same as in output().

OutputString([attrs])
Return a string representing the Morsel, without any surrounding HTTP or JavaScript.

The meaning for attrs is the same as in output().
18.23.3 Example

The following example demonstrates how to use the Cookie module.

```python
>>> import Cookie
>>> C = Cookie.SimpleCookie()
>>> C = Cookie.SerialCookie()
>>> C = Cookie.SmartCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print C # generate HTTP headers
Set-Cookie: sugar=wafer
Set-Cookie: fig=newton
>>> print C.output() # same thing
Set-Cookie: sugar=wafer
Set-Cookie: fig=newton
>>> C = Cookie.SmartCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print C.output(header="Cookie:")
Cookie: rocky=road; Path=/cookie
>>> print C.output(attrs=[], header="Cookie:")
Cookie: rocky=road
>>> C = Cookie.SmartCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP h

```
Set-Cookie: number=7
Set-Cookie: string=seven
>>> C = Cookie.SerialCookie()
>>> C["number"] = 7
>>> C["string"] = "seven"
>>> C["number"].value
7
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number="I7\012."
Set-Cookie: string="S'seven'\012p1\012."
>>> C = Cookie.SmartCookie()
>>> C["number"] = 7
>>> C["string"] = "seven"
>>> C["number"].value
7
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number="I7\012."
Set-Cookie: string=seven

Python Library Reference

Previous: 18.23.2 Morsel Objects Up: 18.23 Cookie Next: 18.24 xmlrpclib

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
**18.24 xmlrpclib -- XML-RPC client access**

New in version 2.2.

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

```python
class ServerProxy(  
    uri[, transport[, encoding[, verbose[, allow_none[,  
    use_datetime]]]]]]
)
```

A `ServerProxy` instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal `SafeTransport` instance for https: URLs and an internal HTTP `Transport` instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag. If `allow_none` is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a `TypeError`. This is a commonly-used extension to the XML-RPC specification, but isn't supported by all clients and servers; see [http://ontosys.com/xml-rpc/extensions.php](http://ontosys.com/xml-rpc/extensions.php) for a description. The `use_datetime` flag can be used to cause date/time values to be presented as `datetime` `datetime` objects; this is false by default. `datetime.datetime`, `datetime.date` and `datetime.time` objects may be passed to calls. `datetime.datetime` date objects are converted with a time of `"00:00:00"`. `datetime.datetime` time objects are converted using today's date.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP `Authorization` header, and sent to the
remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

ServerProxy instance methods take Python basic types and objects as arguments and return Python basic types and classes. Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>The True and False constants</td>
</tr>
<tr>
<td>integers</td>
<td>Pass in directly</td>
</tr>
<tr>
<td>floating-point numbers</td>
<td>Pass in directly</td>
</tr>
<tr>
<td>strings</td>
<td>Pass in directly</td>
</tr>
<tr>
<td>arrays</td>
<td>Any Python sequence type containing conformable elements. Arrays are returned as lists</td>
</tr>
<tr>
<td>structures</td>
<td>A Python dictionary. Keys must be strings, values may be any conformable type.</td>
</tr>
<tr>
<td>dates</td>
<td>in seconds since the epoch (pass in an instance of the DateTime class) or a datetime.datetime, datetime.date or datetime.time instance</td>
</tr>
<tr>
<td>binary data</td>
<td>pass in an instance of the Binary wrapper class</td>
</tr>
</tbody>
</table>

This is the full set of data types supported by XML-RPC. Method calls may also raise a special Fault instance, used to signal XML-RPC server errors, or ProtocolError used to signal an error in the HTTP/HTTPS transport layer. Both Fault and ProtocolError derive from a base class called
**Error.** Note that even though starting with Python 2.2 you can subclass built-in types, the xmlrpc module currently does not marshal instances of such subclasses.

When passing strings, characters special to XML such as "<", ">", and "&" will be automatically escaped. However, it's the caller's responsibility to ensure that the string is free of characters that aren't allowed in XML, such as the control characters with ASCII values between 0 and 31; failing to do this will result in an XML-RPC request that isn't well-formed XML. If you have to pass arbitrary strings via XML-RPC, use the Binary wrapper class described below.

**Server** is retained as an alias for **ServerProxy** for backwards compatibility. New code should use **ServerProxy**.

Changed in version 2.5: The **use_datetime** flag was added.

---

**See Also:**

**XML-RPC HOWTO**
A good description of XML operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

**XML-RPC Hacks page**
Extensions for various open-source libraries to support introspection and multicall.

---

**Subsections**

- [18.24.1 ServerProxy Objects](#)
- [18.24.2 Boolean Objects](#)
- [18.24.3 DateTime Objects](#)
- [18.24.4 Binary Objects](#)
- [18.24.5 Fault Objects](#)
- [18.24.6 ProtocolError Objects](#)
18.24.7 MultiCall Objects
18.24.8 Convenience Functions
18.24.9 Example of Client Usage

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.24.1 ServerProxy Objects

A ServerProxy instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a Fault or ProtocolError object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved `system` member:

`system.listMethods()`
This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

`system.methodSignature(name)`
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply "string, array". If it expects three integers and returns a string, its signature is "string, int, int, int".

If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other that list.

`system.methodHelp(name)`
This method takes one parameter, the name of a method implemented by
the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

Introspection methods are currently supported by servers written in PHP, C and Microsoft .NET. Partial introspection support is included in recent updates to UserLand Frontier. Introspection support for Perl, Python and Java is available at the XML-RPC Hacks page.
18.24.2 Boolean Objects

This class may be initialized from any Python value; the instance returned depends only on its truth value. It supports various Python operators through \_\_cmp\_\_(), \_\_repr\_\_(), \_\_int\_\_(), and \_\_nonzero\_\_() methods, all implemented in the obvious ways.

It also has the following method, supported mainly for internal use by the unmarshalling code:

**encode**( out)

Write the XML-RPC encoding of this Boolean item to the out stream object.

---

**Python Library Reference**

Previous: 18.24.1 ServerProxy Objects  Up: 18.24 xmlrpclib  Next: 18.24.3 DateTime Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.24.3 DateTime Objects

This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a `datetime.datetime`, `datetime.date` or `datetime.time` instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

**decode**( string)

Accept a string as the instance's new time value.

**encode**( out)

Write the XML-RPC encoding of this `DateTime` item to the `out` stream object.

It also supports certain of Python's built-in operators through `__cmp__()` and `__repr__()` methods.

---

Python Library Reference

Previous: 18.24.2 Boolean Objects  Up: 18.24 xmlrpclib  Next: 18.24.4 Binary Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.24.4 Binary Objects

This class may be initialized from string data (which may include NULs). The primary access to the content of a Binary object is provided by an attribute:

data
   The binary data encapsulated by the Binary instance. The data is provided as an 8-bit string.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

dedecode( string)
   Accept a base64 string and decode it as the instance's new data.

dencode( out)
   Write the XML-RPC base 64 encoding of this binary item to the out stream object.

It also supports certain of Python's built-in operators through a ___cmp___( ) method.
18.24.5 Fault Objects

A Fault object encapsulates the content of an XML-RPC fault tag. Fault objects have the following members:

**faultCode**
A string indicating the fault type.

**faultString**
A string containing a diagnostic message associated with the fault.
18.24.6 ProtocolError Objects

A ProtocolError object describes a protocol error in the underlying transport layer (such as a 404 `not found' error if the server named by the URI does not exist). It has the following members:

**url**

The URI or URL that triggered the error.

**errcode**

The error code.

**errmsg**

The error message or diagnostic string.

**headers**

A string containing the headers of the HTTP/HTTPS request that triggered the error.
**18.24.7 MultiCall Objects**

New in version 2.4.

In [http://www.xmlrpc.com/discuss/msgReader%241208](http://www.xmlrpc.com/discuss/msgReader%241208), an approach is presented to encapsulate multiple calls to a remote server into a single request.

```python
class MultiCall(server):
    Create an object used to boxcar method calls. server is the eventual target of the call. Calls can be made to the result object, but they will immediately return None, and only store the call name and parameters in the MultiCall object. Calling the object itself causes all stored calls to be transmitted as a single system.multicall request. The result of this call is a generator; iterating over this generator yields the individual results.

A usage example of this class is

```python
multicall = MultiCall(server_proxy)
multicall.add(2, 3)
multicall.get_address("Guido")
add_result, address = multicall()
```
Example of Client
18.24.8 Convenience Functions

**boolean**( value )

Convert any Python value to one of the XML-RPC Boolean constants, True or False.

**dumps**( params[, methodname[, methodresponse[, encoding[, allow_none]]]])

Convert *params* into an XML-RPC request. or into a response if *methodresponse* is true. *params* can be either a tuple of arguments or an instance of the Fault exception class. If *methodresponse* is true, only a single value can be returned, meaning that *params* must be of length 1. *encoding*, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python's None value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for *allow_none*.

**loads**( data[, use_datetime])

Convert an XML-RPC request or response into Python objects, a *(params, methodname)*. *params* is a tuple of argument; *methodname* is a string, or None if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a Fault exception. The *use_datetime* flag can be used to cause date/time values to be presented as datetime.datetime objects; this is false by default. Note that even if you call an XML-RPC method with datetime.date or datetime.time objects, they are converted to DateTime objects internally, so only datetime.datetime objects will be returned.

Changed in version 2.5: The *use_datetime* flag was added.
18.24.9 Example of Client Usage

```python
# simple test program (from the XML-RPC specification)
from xmlrpc import ServerProxy, Error

# server = ServerProxy("http://localhost:8000")  # local server
server = ServerProxy("http://betty.userland.com")

print server

try:
    print server.examples.getStateName(41)
except Error, v:
    print "ERROR", v
```

To access an XML-RPC server through a proxy, you need to define a custom transport. The following example, written by NoboNobo, shows how:

```python
import xmlrpclib, httplib

class ProxiedTransport(xmlrpclib.Transport):
    def set_proxy(self, proxy):
        self.proxy = proxy
    def make_connection(self, host):
        self.realhost = host
        h = httplib.HTTP(self.proxy)
        return h
    def send_request(self, connection, handler, request_body):
        connection.putrequest("POST", "http://%s%s" % (self.realhost, handler))
    def send_host(self, connection, host):
        connection.putheader('Host', self.realhost)

p = ProxiedTransport()
p.set_proxy('proxy-server:8080')
print server.currentTime.getCurrentTime()
```

Python Library Reference

Previous: 18.24.8 Convenience Functions  Up: 18.24 xmlrpclib  Next: 18.25 SimpleXMLRPCServer

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
18.25 SimpleXMLRPCServer -- Basic XML-RPC server

New in version 2.2.

The SimpleXMLRPCServer module provides a basic server framework for XML-RPC servers written in Python. Servers can either be free standing, using SimpleXMLRPCServer, or embedded in a CGI environment, using CGIXMLRPCRequestHandler.

```python
class SimpleXMLRPCServer(addr[, requestHandler[, logRequests[allow_none[, encoding]]]]):
```

Create a new server instance. This class provides methods for registration of functions that can be called by the XML-RPC protocol. The `requestHandler` parameter should be a factory for request handler instances; it defaults to SimpleXMLRPCRequestHandler. The `addr` and `requestHandler` parameters are passed to the `SocketServer.TCPServer` constructor. If `logRequests` is true (the default), requests will be logged; setting this parameter to false will turn off logging. The `allow_none` and `encoding` parameters are passed on to `xmlrpclib` and control the XML-RPC responses that will be returned from the server. Changed in version 2.5: The `allow_none` and `encoding` parameters were added.

```python
class CGIXMLRPCRequestHandler([allow_none[, encoding]])
```

Create a new instance to handle XML-RPC requests in a CGI environment. The `allow_none` and `encoding` parameters are passed on to `xmlrpclib` and control the XML-RPC responses that will be returned from the server. New in version 2.3. Changed in version 2.5: The `allow_none` and `encoding` parameters were added.

```python
class SimpleXMLRPCRequestHandler()```

Create a new request handler instance. This request handler supports POST requests and modifies logging so that the `logRequests` parameter to the
SimpleXMLRPCServer constructor parameter is honored.

Subsections

- 18.25.1 SimpleXMLRPCServer Objects
- 18.25.2 CGIXMLRPCRequestHandler

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.25.1 SimpleXMLRPCServer Objects

The SimpleXMLRPCServer class is based on SocketServer.TCPServer and provides a means of creating simple, stand alone XML-RPC servers.

**register_function**(*function*, *name*)

Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with *function*, otherwise *function*.__name__ will be used. *name* can be either a normal or Unicode string, and may contain characters not legal in Python identifiers, including the period character.

**register_instance**(*instance*, *allow_dotted_names*)

Register an object which is used to expose method names which have not been registered using register_function(). If *instance* contains a _dispatch() method, it is called with the requested method name and the parameters from the request. Its API is def _dispatch(self, method, params) (note that params does not represent a variable argument list). If it calls an underlying function to perform its task, that function is called as func(*params), expanding the parameter list. The return value from _dispatch() is returned to the client as the result. If *instance* does not have a _dispatch() method, it is searched for an attribute matching the name of the requested method.

If the optional *allow_dotted_names* argument is true and the instance does not have a _dispatch() method, then if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

**Warning:** Enabling the *allow_dotted_names* option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your
Changed in version 2.3.5, 2.4.1: `allow_dotted_names` was added to plug a security hole; prior versions are insecure.

**register_introspection_functions()**

Registers the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`. New in version 2.3.

**register_multicall_functions()**

Registers the XML-RPC multicall function `system.multicall`.

**rpc_paths**

An attribute value that must be a tuple listing valid path portions of the URL for receiving XML-RPC requests. Requests posted to other paths will result in a 404 "no such page" HTTP error. If this tuple is empty, all paths will be considered valid. The default value is `('/', '/RPC2')`. New in version 2.5.

Example:

```python
from SimpleXMLRPCServer import SimpleXMLRPCServer

# Create server
server = SimpleXMLRPCServer(('localhost', 8000))
server.register_introspection_functions()

# Register pow() function; this will use the value of
# pow.__name__ as the name, which is just 'pow'.
server.register_function(pow)

# Register a function under a different name
def adder_function(x, y):
    return x + y
server.register_function(adder_function, 'add')

# Register an instance; all the methods of the instance are
# published as XML-RPC methods (in this case, just 'div').
class MyFuncs:
    def div(self, x, y):
        return x // y

server.register_instance(MyFuncs())
```
# Run the server's main loop
server.serve_forever()

The following client code will call the methods made available by the preceding server:

```
import xmlrpclib

s = xmlrpclib.Server('http://localhost:8000')
print s.pow(2,3)  # Returns 2**3 = 8
print s.add(2,3)  # Returns 5
print s.div(5,2)  # Returns 5//2 = 2

# Print list of available methods
print s.system.listMethods()
```
18.25.2 CGIXMLRPCRequestHandler

The CGIXMLRPCRequestHandler class can be used to handle XML-RPC requests sent to Python CGI scripts.

register_function(function[, name])

Register a function that can respond to XML-RPC requests. If name is given, it will be the method name associated with function, otherwise function.__name__ will be used. name can be either a normal or Unicode string, and may contain characters not legal in Python identifiers, including the period character.

register_instance(instance)

Register an object which is used to expose method names which have not been registered using register_function(). If instance contains a _dispatch() method, it is called with the requested method name and the parameters from the request; the return value is returned to the client as the result. If instance does not have a _dispatch() method, it is searched for an attribute matching the name of the requested method; if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

register_introspection_functions()

Register the XML-RPC introspection functions system.listMethods, system.methodHelp and system.methodSignature.

register_multicall_functions()

Register the XML-RPC multicall function system.multicall.

handle_request([request_text = None])

Handle a XML-RPC request. If request_text is given, it should be the POST data provided by the HTTP server, otherwise the contents of stdin will be used.
Example:

```python
class MyFuncs:
    def div(self, x, y): return x // y

handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

Python Library Reference

Previous: 18.25.1 SimpleXMLRPCServer Objects  Up: 18.25 SimpleXMLRPCServer  Next: 18.26 DocXMLRPCServer

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.26 DocXMLRPCServer -- Self-documenting XML-RPC server

New in version 2.3.

The DocXMLRPCServer module extends the classes found in SimpleXMLRPCServer to serve HTML documentation in response to HTTP GET requests. Servers can either be free standing, using DocXMLRPCServer, or embedded in a CGI environment, using DocCGIXMLRPCRequestHandler.

```python
class DocXMLRPCServer( addr[, requestHandler[, logRequests]])
```

Create a new server instance. All parameters have the same meaning as for SimpleXMLRPCServer; requestHandler defaults to DocXMLRPCRequestHandler.

```python
class DocCGIXMLRPCRequestHandler()
```

Create a new instance to handle XML-RPC requests in a CGI environment.

```python
class DocXMLRPCRequestHandler()
```

Create a new request handler instance. This request handler supports XML-RPC POST requests, documentation GET requests, and modifies logging so that the logRequests parameter to the DocXMLRPCServer constructor parameter is honored.

Subsections

- 18.26.1 DocXMLRPCServer Objects
- 18.26.2 DocCGIXMLRPCRequestHandler
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.26.1 DocXMLRPCServer Objects

The DocXMLRPCServer class is derived from SimpleXMLRPCServer.SimpleXMLRPCServer and provides a means of creating self-documenting, stand alone XML-RPC servers. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

```python
set_server_title(server_title)
```

Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

```python
set_server_name(server_name)
```

Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

```python
set_server_documentation(server_documentation)
```

Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

---

**Python Library Reference**


Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
18.26.2 DocCGIXMLRPCRequestHandler

The DocCGIXMLRPCRequestHandler class is derived from SimpleXMLRPCServer.CGIXMLRPCRequestHandler and provides a means of creating self-documenting, XML-RPC CGI scripts. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

**set_server_title**(server_title)

Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

**set_server_name**(server_name)

Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

**set_server_documentation**(server_documentation)

Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.
19. Multimedia Services

The modules described in this chapter implement various algorithms or interfaces that are mainly useful for multimedia applications. They are available at the discretion of the installation. Here's an overview:

- **audioop**: Manipulate raw audio data.
- **imageop**: Manipulate raw image data.
- **aifc**: Read and write audio files in AIFF or AIFC format.
- **sunau**: Provide an interface to the Sun AU sound format.
- **wave**: Provide an interface to the WAV sound format.
- **chunk**: Module to read IFF chunks.
- **colors**
  - **sys**: Conversion functions between RGB and other color systems.
- **rgbimg**: Read and write image files in ``SGI RGB'' format (the module is *not* SGI specific though!).
- **imghdr**: Determine the type of image contained in a file or byte stream.
- **sndhdr**: Determine type of a sound file.
- **ossaudiodev**: Access to OSS-compatible audio devices.

---

**Python Library Reference**

Previous: 18.26.2 DocCGIXMLRPCRequestHandler  
Up: Python Library Reference  
Next: 19.1 audioop

*Release 2.5, documentation updated on 19th September, 2006.*  
See [About this document...](#) for information on suggesting changes.
19.1 audioop -- Manipulate raw audio data

The audioop module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples 8, 16 or 32 bits wide, stored in Python strings. This is the same format as used by the al and sunaudiodev modules. All scalar items are integers, unless specified otherwise.

This module provides support for a-LAW, u-LAW and Intel/DVI ADPCM encodings.

A few of the more complicated operations only take 16-bit samples, otherwise the sample size (in bytes) is always a parameter of the operation.

The module defines the following variables and functions:

**exception error**
This exception is raised on all errors, such as unknown number of bytes per sample, etc.

**add(fragment1, fragment2, width)**
Return a fragment which is the addition of the two samples passed as parameters. width is the sample width in bytes, either 1, 2 or 4. Both fragments should have the same length.

**adpcm2lin(adpcmfragment, width, state)**
Decode an Intel/DVI ADPCM coded fragment to a linear fragment. See the description of lin2adpcm() for details on ADPCM coding. Return a tuple (sample, newstate) where the sample has the width specified in width.

**alaw2lin(fragment, width)**
Convert sound fragments in a-LAW encoding to linearly encoded sound fragments. a-LAW encoding always uses 8 bits samples, so width refers
only to the sample width of the output fragment here. New in version 2.5.

**avg**(*fragment*, *width*)
Return the average over all samples in the fragment.

**avgpp**(*fragment*, *width*)
Return the average peak-peak value over all samples in the fragment. No filtering is done, so the usefulness of this routine is questionable.

**bias**(*fragment*, *width*, *bias*)
Return a fragment that is the original fragment with a bias added to each sample.

**cross**(*fragment*, *width*)
Return the number of zero crossings in the fragment passed as an argument.

**findfactor**(*fragment*, *reference*)
Return a factor $F$ such that $\text{rms}(\text{add}(\text{fragment}, \text{mul}(\text{reference}, -F)))$ is minimal, i.e., return the factor with which you should multiply *reference* to make it match as well as possible to *fragment*. The fragments should both contain 2-byte samples.

The time taken by this routine is proportional to $\text{len}(\text{fragment})$.

**findfit**(*fragment*, *reference*)
Try to match *reference* as well as possible to a portion of *fragment* (which should be the longer fragment). This is (conceptually) done by taking slices out of *fragment*, using **findfactor**() to compute the best match, and minimizing the result. The fragments should both contain 2-byte samples. Return a tuple (*offset*, *factor*) where *offset* is the (integer) offset into *fragment* where the optimal match started and *factor* is the (floating-point) factor as per **findfactor**().

**findmax**(*fragment*, *length*)
Search *fragment* for a slice of length *length* samples (not bytes!) with maximum energy, i.e., return $i$ for which $\text{rms}(\text{fragment}[i*2: (i+\text{length})*2])$ is maximal. The fragments should both contain 2-byte samples.
samples.

The routine takes time proportional to \( \text{len}(\text{fragment}) \).

**getsample**\((\text{fragment}, \text{width}, \text{index})\)

Return the value of sample \( \text{index} \) from the fragment.

**lin2adpcm**\((\text{fragment}, \text{width}, \text{state})\)

Convert samples to 4 bit Intel/DVI ADPCM encoding. ADPCM coding is an adaptive coding scheme, whereby each 4 bit number is the difference between one sample and the next, divided by a (varying) step. The Intel/DVI ADPCM algorithm has been selected for use by the IMA, so it may well become a standard.

\( \text{state} \) is a tuple containing the state of the coder. The coder returns a tuple \( (\text{adpcmfrag}, \text{newstate}) \), and the \( \text{newstate} \) should be passed to the next call of \( \text{lin2adpcm}() \). In the initial call, \text{None} can be passed as the state. \( \text{adpcmfrag} \) is the ADPCM coded fragment packed 2 4-bit values per byte.

**lin2alaw**\((\text{fragment}, \text{width})\)

Convert samples in the audio fragment to a-LAW encoding and return this as a Python string. a-LAW is an audio encoding format whereby you get a dynamic range of about 13 bits using only 8 bit samples. It is used by the Sun audio hardware, among others. New in version 2.5.

**lin2lin**\((\text{fragment}, \text{width}, \text{newwidth})\)

Convert samples between 1-, 2- and 4-byte formats.

**lin2ulaw**\((\text{fragment}, \text{width})\)

Convert samples in the audio fragment to u-LAW encoding and return this as a Python string. u-LAW is an audio encoding format whereby you get a dynamic range of about 14 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

**minmax**\((\text{fragment}, \text{width})\)

Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.
\textbf{max}( \textit{fragment}, \textit{width} )

Return the maximum of the \textit{absolute value} of all samples in a fragment.

\textbf{maxpp}( \textit{fragment}, \textit{width} )

Return the maximum peak-peak value in the sound fragment.

\textbf{mul}( \textit{fragment}, \textit{width}, \textit{factor} )

Return a fragment that has all samples in the original fragment multiplied by the floating-point value \textit{factor}. Overflow is silently ignored.


Convert the frame rate of the input fragment.

\textit{state} is a tuple containing the state of the converter. The converter returns a tuple \textit{(newfragment, newstate)}, and \textit{newstate} should be passed to the next call of \textbf{ratecv}(). The initial call should pass None as the state.

The \textit{weightA} and \textit{weightB} arguments are parameters for a simple digital filter and default to 1 and 0 respectively.

\textbf{reverse}( \textit{fragment}, \textit{width} )

Reverse the samples in a fragment and returns the modified fragment.

\textbf{rms}( \textit{fragment}, \textit{width} )

Return the root-mean-square of the fragment, i.e.

$$\sqrt{\frac{\sum S_i^2}{n}}$$

This is a measure of the power in an audio signal.

\textbf{tomono}( \textit{fragment}, \textit{width}, \textit{lfactor}, \textit{rfactor} )

Convert a stereo fragment to a mono fragment. The left channel is multiplied by \textit{lfactor} and the right channel by \textit{rfactor} before adding the two channels to give a mono signal.
tostereo( fragment, width, lfactor, rfactor)
Generate a stereo fragment from a mono fragment. Each pair of samples in the stereo fragment are computed from the mono sample, whereby left channel samples are multiplied by lfactor and right channel samples by rfactor.

ulaw2lin( fragment, width)
Convert sound fragments in u-LAW encoding to linearly encoded sound fragments. u-LAW encoding always uses 8 bits samples, so width refers only to the sample width of the output fragment here.

Note that operations such as mul() or max() make no distinction between mono and stereo fragments, i.e. all samples are treated equal. If this is a problem the stereo fragment should be split into two mono fragments first and recombined later. Here is an example of how to do that:

```python
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(sample, width, lfactor)
    rsample = audioop.mul(sample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

If you use the ADPCM coder to build network packets and you want your protocol to be stateless (i.e. to be able to tolerate packet loss) you should not only transmit the data but also the state. Note that you should send the initial state (the one you passed to lin2adpcm()) along to the decoder, not the final state (as returned by the coder). If you want to use struct.struct() to store the state in binary you can code the first element (the predicted value) in 16 bits and the second (the delta index) in 8.

The ADPCM coders have never been tried against other ADPCM coders, only against themselves. It could well be that I misinterpreted the standards in which case they will not be interoperable with the respective standards.

The find*() routines might look a bit funny at first sight. They are primarily meant to do echo cancellation. A reasonably fast way to do this is to pick the most energetic piece of the output sample, locate that in the input sample and subtract the whole output sample from the input sample:
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)  # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)
    #     out_test])
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata, 2, -factor) + postf
    return audioop.add(inputdata, outputdata, 2)
19.2 *imageop* -- Manipulate raw image data

The *imageop* module contains some useful operations on images. It operates on images consisting of 8 or 32 bit pixels stored in Python strings. This is the same format as used by `gl.lrectwrite()` and the *imgfile* module.

The module defines the following variables and functions:

**exception error**
This exception is raised on all errors, such as unknown number of bits per pixel, etc.

**crop**(*image*, *psize*, *width*, *height*, *x0*, *y0*, *x1*, *y1*)
Return the selected part of *image*, which should by *width* by *height* in size and consist of pixels of *psize* bytes. *x0*, *y0*, *x1*, and *y1* are like the `gl.lrectread()` parameters, i.e. the boundary is included in the new image. The new boundaries need not be inside the picture. Pixels that fall outside the old image will have their value set to zero. If *x0* is bigger than *x1* the new image is mirrored. The same holds for the *y* coordinates.

**scale**(*image*, *psize*, *width*, *height*, *newwidth*, *newheight*)
Return *image* scaled to size *newwidth* by *newheight*. No interpolation is done, scaling is done by simple-minded pixel duplication or removal. Therefore, computer-generated images or dithered images will not look nice after scaling.

**tovideo**(*image*, *psize*, *width*, *height*)
Run a vertical low-pass filter over an image. It does so by computing each destination pixel as the average of two vertically-aligned source pixels. The main use of this routine is to forestall excessive flicker if the image is displayed on a video device that uses interlacing, hence the name.

**grey2mono**(*image*, *width*, *height*, *threshold*)
Convert a 8-bit deep greyscale image to a 1-bit deep image by thresholding
all the pixels. The resulting image is tightly packed and is probably only useful as an argument to \texttt{mono2grey()}.

**\texttt{dither2mono( image, width, height)\**

Convert an 8-bit greyscale image to a 1-bit monochrome image using a (simple-minded) dithering algorithm.

**\texttt{mono2grey( image, width, height, p0, p1)\**

Convert a 1-bit monochrome image to an 8 bit greyscale or color image. All pixels that are zero-valued on input get value \texttt{p0} on output and all one-value input pixels get value \texttt{p1} on output. To convert a monochrome black-and-white image to greyscale pass the values 0 and 255 respectively.

**\texttt{grey2grey4( image, width, height)\**

Convert an 8-bit greyscale image to a 4-bit greyscale image without dithering.

**\texttt{grey2grey2( image, width, height)\**

Convert an 8-bit greyscale image to a 2-bit greyscale image without dithering.

**\texttt{dither2grey2( image, width, height)\**

Convert an 8-bit greyscale image to a 2-bit greyscale image with dithering. As for \texttt{dither2mono()}, the dithering algorithm is currently very simple.

**\texttt{grey42grey( image, width, height)\**

Convert a 4-bit greyscale image to an 8-bit greyscale image.

**\texttt{grey22grey( image, width, height)\**

Convert a 2-bit greyscale image to an 8-bit greyscale image.

**\texttt{backward_compatible\**

If set to 0, the functions in this module use a non-backward compatible way of representing multi-byte pixels on little-endian systems. The SGI for which this module was originally written is a big-endian system, so setting this variable will have no effect. However, the code wasn't originally intended to run on anything else, so it made assumptions about byte order
which are not universal. Setting this variable to 0 will cause the byte order to be reversed on little-endian systems, so that it then is the same as on big-endian systems.
19.3 aifc -- Read and write AIFF and AIFC files

This module provides support for reading and writing AIFF and AIFF-C files. AIFF is Audio Interchange File Format, a format for storing digital audio samples in a file. AIFF-C is a newer version of the format that includes the ability to compress the audio data.

Caveat: Some operations may only work under IRIX; these will raise ImportError when attempting to import the cl module, which is only available on IRIX.

Audio files have a number of parameters that describe the audio data. The sampling rate or frame rate is the number of times per second the sound is sampled. The number of channels indicate if the audio is mono, stereo, or quadro. Each frame consists of one sample per channel. The sample size is the size in bytes of each sample. Thus a frame consists of nchannels*samplesize bytes, and a second's worth of audio consists of nchannels*samplesize*framerate bytes.

For example, CD quality audio has a sample size of two bytes (16 bits), uses two channels (stereo) and has a frame rate of 44,100 frames/second. This gives a frame size of 4 bytes (2*2), and a second's worth occupies 2*2*44100 bytes (176,400 bytes).

Module aifc defines the following function:

open( file[, mode])

Open an AIFF or AIFF-C file and return an object instance with methods that are described below. The argument file is either a string naming a file or a file object. mode must be 'r' or 'rb' when the file must be opened for reading, or 'w' or 'wb' when the file must be opened for writing. If omitted, file.mode is used if it exists, otherwise 'rb' is used. When used for writing, the file object should be seekable, unless you know ahead of time how many samples you are going to write in total and use
writeframesraw() and setnframes().

Objects returned by open() when a file is opened for reading have the following methods:

getnchannels()
    Return the number of audio channels (1 for mono, 2 for stereo).

getsampwidth()
    Return the size in bytes of individual samples.

getframerate()
    Return the sampling rate (number of audio frames per second).

getnframes()
    Return the number of audio frames in the file.

getcomptype()
    Return a four-character string describing the type of compression used in the audio file. For AIFF files, the returned value is 'NONE'.

getcompname()
    Return a human-readable description of the type of compression used in the audio file. For AIFF files, the returned value is 'not compressed'.

getparams()
    Return a tuple consisting of all of the above values in the above order.

getmarkers()
    Return a list of markers in the audio file. A marker consists of a tuple of three elements. The first is the mark ID (an integer), the second is the mark position in frames from the beginning of the data (an integer), the third is the name of the mark (a string).

getmark(id)
    Return the tuple as described in getmarkers() for the mark with the given id.
**readframes**(*nframes*)
Read and return the next *nframes* frames from the audio file. The returned data is a string containing for each frame the uncompressed samples of all channels.

**rewind()**
Rewind the read pointer. The next **readframes()** will start from the beginning.

**setpos**(*pos*)
Seek to the specified frame number.

**tell()**
Return the current frame number.

**close()**
Close the AIFF file. After calling this method, the object can no longer be used.

Objects returned by **open()** when a file is opened for writing have all the above methods, except for **readframes()** and **setpos()**. In addition the following methods exist. The **get**(* method can only be called after the corresponding **set**(* method have been called. Before the first **writeframes()** or **writeframesraw()**, all parameters except for the number of frames must be filled in.

**aiff()**
Create an AIFF file. The default is that an AIFF-C file is created, unless the name of the file ends in '.aiff' in which case the default is an AIFF file.

**aifc()**
Create an AIFF-C file. The default is that an AIFF-C file is created, unless the name of the file ends in '.aiff' in which case the default is an AIFF file.

**setnchannels**(*nchannels*)
Specify the number of channels in the audio file.
**setsampwidth**(*width*)
Specify the size in bytes of audio samples.

**setframerate**(*rate*)
Specify the sampling frequency in frames per second.

**setnframes**(*nframes*)
Specify the number of frames that are to be written to the audio file. If this parameter is not set, or not set correctly, the file needs to support seeking.

**setcomptype**(*type*, *name*)
Specify the compression type. If not specified, the audio data will not be compressed. In AIFF files, compression is not possible. The name parameter should be a human-readable description of the compression type, the type parameter should be a four-character string. Currently the following compression types are supported: NONE, ULAW, ALAW, G722.

**setparams**(*nchannels, sampwidth, framerate, comptype, compname*)
Set all the above parameters at once. The argument is a tuple consisting of the various parameters. This means that it is possible to use the result of a **getparams**() call as argument to **setparams**().

**setmark**(*id, pos, name*)
Add a mark with the given id (larger than 0), and the given name at the given position. This method can be called at any time before close().

**tell**()
Return the current write position in the output file. Useful in combination with setmark().

**writeframes**(*data*)
Write data to the output file. This method can only be called after the audio file parameters have been set.

**writeframesraw**(*data*)
Like writeframes(), except that the header of the audio file is not updated.
**close()**

Close the AIFF file. The header of the file is updated to reflect the actual size of the audio data. After calling this method, the object can no longer be used.
19.4 sunau -- Read and write Sun AU files

The sunau module provides a convenient interface to the Sun AU sound format. Note that this module is interface-compatible with the modules aifc and wave.

An audio file consists of a header followed by the data. The fields of the header are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>magic word</td>
<td>The four bytes &quot; . snd&quot;.</td>
</tr>
<tr>
<td>header size</td>
<td>Size of the header, including info, in bytes.</td>
</tr>
<tr>
<td>data size</td>
<td>Physical size of the data, in bytes.</td>
</tr>
<tr>
<td>encoding</td>
<td>Indicates how the audio samples are encoded.</td>
</tr>
<tr>
<td>sample rate</td>
<td>The sampling rate.</td>
</tr>
<tr>
<td># of channels</td>
<td>The number of channels in the samples.</td>
</tr>
<tr>
<td>info</td>
<td>ASCII string giving a description of the audio file (padded with null bytes).</td>
</tr>
</tbody>
</table>

Apart from the info field, all header fields are 4 bytes in size. They are all 32-bit unsigned integers encoded in big-endian byte order.

The sunau module defines the following functions:

**open**(*file*, *mode*)

If *file* is a string, open the file by that name, otherwise treat it as a seekable file-like object. *mode* can be any of

- `'r'`
  - Read only mode.
- `'w'`
  - Write only mode.
Note that it does not allow read/write files.

A mode of 'r' returns a AU_read object, while a mode of 'w' or 'wb' returns a AU_write object.

**openfp(file, mode)**

A synonym for open, maintained for backwards compatibility.

The sunau module defines the following exception:

**exception Error**

An error raised when something is impossible because of Sun AU specs or implementation deficiency.

The sunau module defines the following data items:

**AUDIO_FILE_MAGIC**

An integer every valid Sun AU file begins with, stored in big-endian form. This is the string ".snd" interpreted as an integer.

**AUDIO_FILE_ENCODING_MULAW_8**
**AUDIO_FILE_ENCODING_LINEAR_8**
**AUDIO_FILE_ENCODING_LINEAR_16**
**AUDIO_FILE_ENCODING_LINEAR_24**
**AUDIO_FILE_ENCODING_LINEAR_32**
**AUDIO_FILE_ENCODING_ALAW_8**

Values of the encoding field from the AU header which are supported by this module.

**AUDIO_FILE_ENCODING_FLOAT**
**AUDIO_FILE_ENCODING_DOUBLE**
**AUDIO_FILE_ENCODING_ADPCM_G721**
**AUDIO_FILE_ENCODING_ADPCM_G722**
**AUDIO_FILE_ENCODING_ADPCM_G723_3**
**AUDIO_FILE_ENCODING_ADPCM_G723_5**

Additional known values of the encoding field from the AU header, but which are not supported by this module.

---

**Subsections**
• **19.4.1 AU_read Objects**
• **19.4.2 AU_write Objects**

Release 2.5, documentation updated on 19th September, 2006.
See *About this document...* for information on suggesting changes.
19.4.1 AU_read Objects

AU_read objects, as returned by open() above, have the following methods:

**close()**
Close the stream, and make the instance unusable. (This is called automatically on deletion.)

**getnchannels()**
Returns number of audio channels (1 for mone, 2 for stereo).

**getsampwidth()**
Returns sample width in bytes.

**getframerate()**
Returns sampling frequency.

**getnframes()**
Returns number of audio frames.

**getcomptype()**
Returns compression type. Supported compression types are 'ULAW', 'ALAW' and 'NONE'.

**getcompname()**
Human-readable version of getcomptype(). The supported types have the respective names 'CCITT G.711 u-law', 'CCITT G.711 A-law' and 'not compressed'.

**getparams()**
Returns a tuple (nchannels, sampwidth, framerate, nframes, comptype, compname), equivalent to output of the get*() methods.

**readframes(n)**
Reads and returns at most n frames of audio, as a string of bytes. The data
will be returned in linear format. If the original data is in u-LAW format, it will be converted.

```python
rewind()
```

Rewind the file pointer to the beginning of the audio stream.

The following two methods define a term `"position"` which is compatible between them, and is otherwise implementation dependent.

```python
setpos(pos)
```

Set the file pointer to the specified position. Only values returned from `tell()` should be used for `pos`.

```python
tell()
```

Return current file pointer position. Note that the returned value has nothing to do with the actual position in the file.

The following two functions are defined for compatibility with the `aifc`, and don't do anything interesting.

```python
getmarkers()
```

Returns None.

```python
getmark(id)
```

Raise an error.
19.4.2 AU_write Objects

AU_write objects, as returned by open() above, have the following methods:

**setnchannels**(n)
Set the number of channels.

**setsampwidth**(n)
Set the sample width (in bytes.)

**setframerate**(n)
Set the frame rate.

**setnframes**(n)
Set the number of frames. This can be later changed, when and if more frames are written.

**setcomptype**(type, name)
Set the compression type and description. Only 'NONE' and 'ULAW' are supported on output.

**setparams**(tuple)
The tuple should be (nchannels, sampwidth, framerate, nframes, comptype, compname), with values valid for the set*() methods. Set all parameters.

**tell()**
Return current position in the file, with the same disclaimer for the AU_read.tell() and AU_read.setpos() methods.

**writeframesraw**(data)
Write audio frames, without correcting nframes.

**writeframes**(data)
Write audio frames and make sure nframes is correct.
close()

Make sure \textit{nframes} is correct, and close the file.

This method is called upon deletion.

Note that it is invalid to set any parameters after calling \texttt{writeframes()} or \texttt{writeframesraw()}. 

\textit{Python Library Reference}

Previous: 19.4.1 \texttt{AU\_read} Objects  Up: 19.4 \texttt{sunau}  Next: 19.5 \texttt{wave}

\textit{Release 2.5, documentation updated on 19th September, 2006.  
See About this document... for information on suggesting changes.}
19.5 wave -- Read and write WAV files

The `wave` module provides a convenient interface to the WAV sound format. It does not support compression/decompression, but it does support mono/stereo.

The `wave` module defines the following function and exception:

```python
open( file[, mode])
```

If `file` is a string, open the file by that name, other treat it as a seekable file-like object. `mode` can be any of

- `'r'`, `'rb'`
  Read only mode.
- `'w'`, `'wb'`
  Write only mode.

Note that it does not allow read/write WAV files.

A `mode` of 'r' or 'rb' returns a `Wave_read` object, while a `mode` of 'w' or 'wb' returns a `Wave_write` object. If `mode` is omitted and a file-like object is passed as `file`, `file.mode` is used as the default value for `mode` (the "b" flag is still added if necessary).

```python
openfp( file, mode)
```

A synonym for `open()`, maintained for backwards compatibility.

**exception Error**

An error raised when something is impossible because it violates the WAV specification or hits an implementation deficiency.

---

**Subsections**

- [19.5.1 Wave_read Objects](#)
- [19.5.2 Wave_write Objects](#)
Wave_read Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
19.5.1 Wave_read Objects

Wave_read objects, as returned by open(), have the following methods:

**close()**
Close the stream, and make the instance unusable. This is called automatically on object collection.

**getnchannels()**
Returns number of audio channels (1 for mono, 2 for stereo).

**getsampwidth()**
Returns sample width in bytes.

**getframerate()**
Returns sampling frequency.

**getnframes()**
Returns number of audio frames.

**getcomptype()**
Returns compression type ('NONE' is the only supported type).

**getcompname()**
Human-readable version of getcomptype(). Usually 'not compressed' parallels 'NONE'.

**getparams()**
Returns a tuple (nchannels, sampwidth, framerate, nframes, comptype, compname), equivalent to output of the get*() methods.

**readframes( n)**
Reads and returns at most n frames of audio, as a string of bytes.

**rewind()**
Rewind the file pointer to the beginning of the audio stream.

The following two methods are defined for compatibility with the `aifc` module, and don't do anything interesting.

**getmarkers()**
- Returns None.

**getmark(id)**
- Raise an error.

The following two methods define a term ``position'' which is compatible between them, and is otherwise implementation dependent.

**setpos(pos)**
- Set the file pointer to the specified position.

**tell()**
- Return current file pointer position.
19.5.2 Wave_write Objects

Wave_write objects, as returned by `open()`, have the following methods:

**close()**
Make sure `nframes` is correct, and close the file. This method is called upon deletion.

**setnchannels( n)**
Set the number of channels.

**setsampwidth( n)**
Set the sample width to `n` bytes.

**setframerate( n)**
Set the frame rate to `n`.

**setnframes( n)**
Set the number of frames to `n`. This will be changed later if more frames are written.

**setcomptype( type, name)**
Set the compression type and description.

**setparams( tuple)**
The `tuple` should be `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, with values valid for the `set*()` methods. Sets all parameters.

**tell()**
Return current position in the file, with the same disclaimer for the `Wave_read.tell()` and `Wave_read.setpos()` methods.

**writeframesraw( data)**
Write audio frames, without correcting `nframes`. 
writeframes(data)

Write audio frames and make sure nframes is correct.

Note that it is invalid to set any parameters after calling writeframes() or writeframesraw(), and any attempt to do so will raise wave.Error.
19.6 chunk -- Read IFF chunked data

This module provides an interface for reading files that use EA IFF 85 chunks. This format is used in at least the Audio Interchange File Format (AIFF/AIFF-C) and the Real Media File Format (RMFF). The WAVE audio file format is closely related and can also be read using this module.

A chunk has the following structure:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Chunk ID</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Size of chunk in big-endian byte order, not including the header</td>
</tr>
<tr>
<td>8</td>
<td>n</td>
<td>Data bytes, where n is the size given in the preceding field</td>
</tr>
<tr>
<td>8 + n</td>
<td>0 or 1</td>
<td>Pad byte needed if n is odd and chunk alignment is used</td>
</tr>
</tbody>
</table>

The ID is a 4-byte string which identifies the type of chunk.

The size field (a 32-bit value, encoded using big-endian byte order) gives the size of the chunk data, not including the 8-byte header.

Usually an IFF-type file consists of one or more chunks. The proposed usage of the Chunk class defined here is to instantiate an instance at the start of each chunk and read from the instance until it reaches the end, after which a new instance can be instantiated. At the end of the file, creating a new instance will fail with a EOFError exception.

class Chunk(file[, align, bigendian, inclheader])

Class which represents a chunk. The file argument is expected to be a file-like object. An instance of this class is specifically allowed. The only method that is needed is read(). If the methods seek() and tell() are present and don’t raise an exception, they are also used. If these methods are present and raise an exception, they are expected to not have altered the object. If the optional argument align is true, chunks are assumed to be aligned on 2-byte boundaries. If align is false, no alignment is assumed. The default value is true. If the optional argument bigendian is false, the
chunk size is assumed to be in little-endian order. This is needed for WAVE audio files. The default value is true. If the optional argument inclheader is true, the size given in the chunk header includes the size of the header. The default value is false.

A Chunk object supports the following methods:

**getname()**
- Returns the name (ID) of the chunk. This is the first 4 bytes of the chunk.

**getsize()**
- Returns the size of the chunk.

**close()**
- Close and skip to the end of the chunk. This does not close the underlying file.

The remaining methods will raise **IOError** if called after the **close()** method has been called.

**isatty()**
- Returns False.

**seek(pos[, whence])**
- Set the chunk's current position. The **whence** argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file's end). There is no return value. If the underlying file does not allow seek, only forward seeks are allowed.

**tell()**
- Return the current position into the chunk.

**read([size])**
- Read at most size bytes from the chunk (less if the read hits the end of the chunk before obtaining size bytes). If the size argument is negative or omitted, read all data until the end of the chunk. The bytes are returned as a
string object. An empty string is returned when the end of the chunk is encountered immediately.

**skip()**

Skip to the end of the chunk. All further calls to `read()` for the chunk will return '''. If you are not interested in the contents of the chunk, this method should be called so that the file points to the start of the next chunk.

---

**Footnotes**

... chunks.\(^{19.1}\)

19.7 colorsys -- Conversions between color systems

The colorsys module defines bidirectional conversions of color values between colors expressed in the RGB (Red Green Blue) color space used in computer monitors and three other coordinate systems: YIQ, HLS (Hue Lightness Saturation) and HSV (Hue Saturation Value). Coordinates in all of these color spaces are floating point values. In the YIQ space, the Y coordinate is between 0 and 1, but the I and Q coordinates can be positive or negative. In all other spaces, the coordinates are all between 0 and 1.

More information about color spaces can be found at http://www.poynton.com/ColorFAQ.html.

The colorsys module defines the following functions:

- `rgb_to_yiq(r, g, b)`: Convert the color from RGB coordinates to YIQ coordinates.
- `yiq_to_rgb(y, i, q)`: Convert the color from YIQ coordinates to RGB coordinates.
- `rgb_to_hls(r, g, b)`: Convert the color from RGB coordinates to HLS coordinates.
- `hls_to_rgb(h, l, s)`: Convert the color from HLS coordinates to RGB coordinates.
- `rgb_to_hsv(r, g, b)`: Convert the color from RGB coordinates to HSV coordinates.
- `hsv_to_rgb(h, s, v)`: Convert the color from HSV coordinates to RGB coordinates.

Example:
>>> import colorsys
>>> colorsys.rgb_to_hsv(.3, .4, .2)
(0.25, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.25, 0.5, 0.4)
(0.3, 0.4, 0.2)
19.8 rgbimg -- Read and write ``SGI RGB'' files

**Deprecated since release 2.5.** This module is not maintained anymore and seems to be unused.

The rgbimg module allows Python programs to access SGI imglib image files (also known as .rgb files). The module is far from complete, but is provided anyway since the functionality that there is enough in some cases. Currently, colormap files are not supported.

**Note:** This module is only built by default for 32-bit platforms; it is not expected to work properly on other systems.

The module defines the following variables and functions:

**exception error**
This exception is raised on all errors, such as unsupported file type, etc.

**sizeofimage**(*file*)
This function returns a tuple \((x, y)\) where \(x\) and \(y\) are the size of the image in pixels. Only 4 byte RGBA pixels, 3 byte RGB pixels, and 1 byte greyscale pixels are currently supported.

**longimagedata**(*file*)
This function reads and decodes the image on the specified file, and returns it as a Python string. The string has 4 byte RGBA pixels. The bottom left pixel is the first in the string. This format is suitable to pass to `gl.lrectwrite()`, for instance.

**longstoimage**(*data, x, y, z, file*)
This function writes the RGBA data in `data` to image file `file`. \(x\) and \(y\) give the size of the image. \(z\) is 1 if the saved image should be 1 byte greyscale, 3 if the saved image should be 3 byte RGB data, or 4 if the saved images should be 4 byte RGBA data. The input data always contains 4 bytes per
pixel. These are the formats returned by `gl.lrectread()`.

**ttob** *(flag)*

This function sets a global flag which defines whether the scan lines of the image are read or written from bottom to top (flag is zero, compatible with SGI GL) or from top to bottom (flag is one, compatible with X). The default is zero.
19.9 imghdr -- Determine the type of an image

The `imghdr` module determines the type of image contained in a file or byte stream.

The `imghdr` module defines the following function:

```python
what(filename[, h])
```
Tests the image data contained in the file named by `filename`, and returns a string describing the image type. If optional `h` is provided, the `filename` is ignored and `h` is assumed to contain the byte stream to test.

The following image types are recognized, as listed below with the return value from `what()`:

<table>
<thead>
<tr>
<th>Value</th>
<th>Image format</th>
</tr>
</thead>
<tbody>
<tr>
<td>'rgb'</td>
<td>SGI ImgLib Files</td>
</tr>
<tr>
<td>'gif'</td>
<td>GIF 87a and 89a Files</td>
</tr>
<tr>
<td>'pbm'</td>
<td>Portable Bitmap Files</td>
</tr>
<tr>
<td>'pgm'</td>
<td>Portable Graymap Files</td>
</tr>
<tr>
<td>'ppm'</td>
<td>Portable Pixmap Files</td>
</tr>
<tr>
<td>'tiff'</td>
<td>TIFF Files</td>
</tr>
<tr>
<td>'rast'</td>
<td>Sun Raster Files</td>
</tr>
<tr>
<td>'xbm'</td>
<td>X Bitmap Files</td>
</tr>
<tr>
<td>'jpeg'</td>
<td>JPEG data in JFIF or Exif formats</td>
</tr>
<tr>
<td>'bmp'</td>
<td>BMP files</td>
</tr>
<tr>
<td>'png'</td>
<td>Portable Network Graphics</td>
</tr>
</tbody>
</table>

New in version 2.5: Exif detection.

You can extend the list of file types `imghdr` can recognize by appending to this variable:
tests

A list of functions performing the individual tests. Each function takes two arguments: the byte-stream and an open file-like object. When `what()` is called with a byte-stream, the file-like object will be `None`.

The test function should return a string describing the image type if the test succeeded, or `None` if it failed.

Example:

```python
>>> import imghdr
>>> imghdr.what('/tmp/bass.gif')
'gif'
```
19.10 sndhdr -- Determine type of sound file

The sndhdr provides utility functions which attempt to determine the type of sound data which is in a file. When these functions are able to determine what type of sound data is stored in a file, they return a tuple (type, sampling_rate, channels, frames, bits_per_sample). The value for type indicates the data type and will be one of the strings 'aifc', 'aiff', 'au', 'hcom', 'sndr', 'sndt', 'voc', 'wav', '8svx', 'sb', 'ub', or 'ul'. The sampling_rate will be either the actual value or 0 if unknown or difficult to decode. Similarly, channels will be either the number of channels or 0 if it cannot be determined or if the value is difficult to decode. The value for frames will be either the number of frames or -1. The last item in the tuple, bits_per_sample, will either be the sample size in bits or 'A' for A-LAW or 'U' for u-LAW.

what(filename)

Determines the type of sound data stored in the file filename using whathdr(). If it succeeds, returns a tuple as described above, otherwise None is returned.

whathdr(filename)

Determines the type of sound data stored in a file based on the file header. The name of the file is given by filename. This function returns a tuple as described above on success, or None.
19.11 ossaudiodev -- Access to OSS-compatible audio devices

Availability: Linux, FreeBSD.

New in version 2.3.

This module allows you to access the OSS (Open Sound System) audio interface. OSS is available for a wide range of open-source and commercial Unices, and is the standard audio interface for Linux and recent versions of FreeBSD.

See Also:

Open Sound System Programmer's Guide
the official documentation for the OSS C API

The module defines a large number of constants supplied by the OSS device driver; see <sys/soundcard.h> on either Linux or FreeBSD for a listing.

ossaudiodev defines the following variables and functions:

exception OSSAudioError
This exception is raised on certain errors. The argument is a string describing what went wrong.

(If ossaudiodev receives an error from a system call such as open(), write(), or ioctl(), it raises IOError. Errors detected directly by ossaudiodev result in OSSAudioError.)

(For backwards compatibility, the exception class is also available as ossaudiodev.error.)
`open( [device, ]mode)`

Open an audio device and return an OSS audio device object. This object supports many file-like methods, such as `read()`, `write()`, and `fileno()` (although there are subtle differences between conventional UNIX read/write semantics and those of OSS audio devices). It also supports a number of audio-specific methods; see below for the complete list of methods.

`device` is the audio device filename to use. If it is not specified, this module first looks in the environment variable AUDIODEV for a device to use. If not found, it falls back to `/dev/dsp`.

`mode` is one of `'r'` for read-only (record) access, `'w'` for write-only (playback) access and `'rw'` for both. Since many sound cards only allow one process to have the recorder or player open at a time, it is a good idea to open the device only for the activity needed. Further, some sound cards are half-duplex: they can be opened for reading or writing, but not both at once.

Note the unusual calling syntax: the `first` argument is optional, and the second is required. This is a historical artifact for compatibility with the older `linuxaudiodev` module which `ossaudiodev` supersedes.

`openmixer([device])`

Open a mixer device and return an OSS mixer device object. `device` is the mixer device filename to use. If it is not specified, this module first looks in the environment variable MIXERDEV for a device to use. If not found, it falls back to `/dev/mixer`.

Subsections

- 19.11.1 Audio Device Objects
- 19.11.2 Mixer Device Objects
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
19.11.1 Audio Device Objects

Before you can write to or read from an audio device, you must call three methods in the correct order:

1. `setfmt()` to set the output format
2. `channels()` to set the number of channels
3. `speed()` to set the sample rate

Alternately, you can use the `setparameters()` method to set all three audio parameters at once. This is more convenient, but may not be as flexible in all cases.

The audio device objects returned by `open()` define the following methods and (read-only) attributes:

**close()**
Explicitly close the audio device. When you are done writing to or reading from an audio device, you should explicitly close it. A closed device cannot be used again.

**fileno()**
Return the file descriptor associated with the device.

**read(size)**
Read `size` bytes from the audio input and return them as a Python string. Unlike most UNIX device drivers, OSS audio devices in blocking mode (the default) will block `read()` until the entire requested amount of data is available.

**write(data)**
Write the Python string `data` to the audio device and return the number of bytes written. If the audio device is in blocking mode (the default), the entire string is always written (again, this is different from usual UNIX device semantics). If the device is in non-blocking mode, some data may not be written--see `writeall()`.
**writeall**(data)

Write the entire Python string *data* to the audio device: waits until the audio device is able to accept data, writes as much data as it will accept, and repeats until *data* has been completely written. If the device is in blocking mode (the default), this has the same effect as **write(); writeall()** is only useful in non-blocking mode. Has no return value, since the amount of data written is always equal to the amount of data supplied.

The following methods each map to exactly one **ioctl()** system call. The correspondence is obvious: for example, **setfmt()** corresponds to the **SNDCTL_DSP_SETFMT** ioctl, and **sync()** to **SNDCTL_DSP_SYNC** (this can be useful when consulting the OSS documentation). If the underlying **ioctl()** fails, they all raise **IOError**.

**nonblock()**

Put the device into non-blocking mode. Once in non-blocking mode, there is no way to return it to blocking mode.

**getfmts()**

Return a bitmask of the audio output formats supported by the soundcard. Some of the formats supported by OSS are:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFMT_MU_LAW</td>
<td>a logarithmic encoding (used by Sun .au files and /dev/audio)</td>
</tr>
<tr>
<td>AFMT_A_LAW</td>
<td>a logarithmic encoding</td>
</tr>
<tr>
<td>AFMTIMA_ADPCM</td>
<td>a 4:1 compressed format defined by the Interactive Multimedia Association</td>
</tr>
<tr>
<td>AFMT_U8</td>
<td>Unsigned, 8-bit audio</td>
</tr>
<tr>
<td>AFMT_S16_LE</td>
<td>Signed, 16-bit audio, little-endian byte order (as used by Intel processors)</td>
</tr>
<tr>
<td>AFMT_S16_BE</td>
<td>Signed, 16-bit audio, big-endian byte order (as used by 68k, PowerPC, Sparc)</td>
</tr>
<tr>
<td>AFMT_S8</td>
<td>Signed, 8 bit audio</td>
</tr>
<tr>
<td>AFMT_U16_LE</td>
<td>Unsigned, 16-bit little-endian audio</td>
</tr>
<tr>
<td>AFMT_U16_BE</td>
<td>Unsigned, 16-bit big-endian audio</td>
</tr>
</tbody>
</table>

Consult the OSS documentation for a full list of audio formats, and note
that most devices support only a subset of these formats. Some older
devices only support AFMT_U8; the most common format used today is
AFMT_S16_LE.

**setfmt** (*format*)

Try to set the current audio format to *format*--see **getfmts()** for a list.
Returns the audio format that the device was set to, which may not be the
requested format. May also be used to return the current audio format--do
this by passing an "`audio format" of AFMT_QUERY.

**channels** (*nchannels*)

Set the number of output channels to *nchannels*. A value of 1 indicates
monophonic sound, 2 stereophonic. Some devices may have more than 2
channels, and some high-end devices may not support mono. Returns the
number of channels the device was set to.

**speed** (*samplerate*)

Try to set the audio sampling rate to *samplerate* samples per second.
Returns the rate actually set. Most sound devices don't support arbitrary
sampling rates. Common rates are:

<table>
<thead>
<tr>
<th>Rate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8000</td>
<td>default rate for /dev/audio</td>
</tr>
<tr>
<td>11025</td>
<td>speech recording</td>
</tr>
<tr>
<td>22050</td>
<td></td>
</tr>
<tr>
<td>44100</td>
<td>CD quality audio (at 16 bits/sample and 2 channels)</td>
</tr>
<tr>
<td>96000</td>
<td>DVD quality audio (at 24 bits/sample)</td>
</tr>
</tbody>
</table>

**sync()**

Wait until the sound device has played every byte in its buffer. (This
happens implicitly when the device is closed.) The OSS documentation
recommends closing and re-opening the device rather than using **sync()**.

**reset()**

Immediately stop playing or recording and return the device to a state
where it can accept commands. The OSS documentation recommends
closing and re-opening the device after calling **reset()**.
**post()**
Tell the driver that there is likely to be a pause in the output, making it possible for the device to handle the pause more intelligently. You might use this after playing a spot sound effect, before waiting for user input, or before doing disk I/O.

The following convenience methods combine several ioctls, or one ioctl and some simple calculations.

**setparameters**( *format*, *nchannels*, *samplerate* [, *strict=\text{False}]*)
Set the key audio sampling parameters—sample format, number of channels, and sampling rate—in one method call. *format*, *nchannels*, and *samplerate* should be as specified in the *setfmt()*, *channels()*, and *speed()* methods. If *strict* is true, *setparameters()* checks to see if each parameter was actually set to the requested value, and raises *OSSAudioError* if not. Returns a tuple (*format*, *nchannels*, *samplerate*) indicating the parameter values that were actually set by the device driver (i.e., the same as the return values of *setfmt()*, *channels()*, and *speed()*).

For example,

```python
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

is equivalent to

```python
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(channels)
```

**bufsize()**
Returns the size of the hardware buffer, in samples.

**obufcount()**
Returns the number of samples that are in the hardware buffer yet to be played.

**obuffree()**
Returns the number of samples that could be queued into the hardware
buffer to be played without blocking.

Audio device objects also support several read-only attributes:

**closed**
Boolean indicating whether the device has been closed.

**name**
String containing the name of the device file.

**mode**
The I/O mode for the file, either "r", "rw", or "w".
19.11.2 Mixer Device Objects

The mixer object provides two file-like methods:

**close()**
This method closes the open mixer device file. Any further attempts to use the mixer after this file is closed will raise an IOError.

**fileno()**
Returns the file handle number of the open mixer device file.

The remaining methods are specific to audio mixing:

**controls()**
This method returns a bitmask specifying the available mixer controls ("Control" being a specific mixable "channel", such as SOUND_MIXER_PCM or SOUND_MIXER_SYNTH). This bitmask indicates a subset of all available mixer controls--the SOUND_MIXER_* constants defined at module level. To determine if, for example, the current mixer object supports a PCM mixer, use the following Python code:

```python
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

For most purposes, the SOUND_MIXER_VOLUME (master volume) and SOUND_MIXER_PCM controls should suffice--but code that uses the mixer should be flexible when it comes to choosing mixer controls. On the Gravis Ultrasound, for example, SOUND_MIXER_VOLUME does not exist.

**stereocontrols()**
Returns a bitmask indicating stereo mixer controls. If a bit is set, the corresponding control is stereo; if it is unset, the control is either monophonic or not supported by the mixer (use in combination with controls() to determine which).

See the code example for the controls() function for an example of
getting data from a bitmask.

**reccontrols()**

Returns a bitmask specifying the mixer controls that may be used to record. See the code example for `controls()` for an example of reading from a bitmask.

**get(control)**

Returns the volume of a given mixer control. The returned volume is a 2-tuple `(left_volume, right_volume)`. Volumes are specified as numbers from 0 (silent) to 100 (full volume). If the control is monophonic, a 2-tuple is still returned, but both volumes are the same.

Raises `OSSAudioError` if an invalid control was is specified, or `IOError` if an unsupported control is specified.

**set(control, (left, right))**

Sets the volume for a given mixer control to `(left, right)`. `left` and `right` must be ints and between 0 (silent) and 100 (full volume). On success, the new volume is returned as a 2-tuple. Note that this may not be exactly the same as the volume specified, because of the limited resolution of some soundcard's mixers.

Raises `OSSAudioError` if an invalid mixer control was specified, or if the specified volumes were out-of-range.

**get_recsrc()**

This method returns a bitmask indicating which control(s) are currently being used as a recording source.

**set_recsrc(bitmask)**

Call this function to specify a recording source. Returns a bitmask indicating the new recording source (or sources) if successful; raises `IOError` if an invalid source was specified. To set the current recording source to the microphone input:

```python
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```
Graphical User Interfaces

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20. Graphical User Interfaces with Tk

Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the Tkinter module, and its extension, the Tix module.

The Tkinter module is a thin object-oriented layer on top of Tcl/Tk. To use Tkinter, you don't need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. Tkinter is a set of wrappers that implement the Tk widgets as Python classes. In addition, the internal module _tkinter provides a threadsafe mechanism which allows Python and Tcl to interact.

Tk is not the only GUI for Python; see section 20.6, "Other User Interface Modules and Packages," for more information on other GUI toolkits for Python.

Tkinter  Interface to Tcl/Tk for graphical user interfaces
Tix  Tk Extension Widgets for Tkinter
ScrolledText  Text widget with a vertical scroll bar.
turtle  An environment for turtle graphics.
20.1 Tkinter -- Python interface to Tcl/Tk

The Tkinter module (``Tk interface'') is the standard Python interface to the Tk GUI toolkit. Both Tk and Tkinter are available on most UNIX platforms, as well as on Windows and Macintosh systems. (Tk itself is not part of Python; it is maintained at ActiveState.)

See Also:

* **Python Tkinter Resources**
  The Python Tkinter Topic Guide provides a great deal of information on using Tk from Python and links to other sources of information on Tk.

* **An Introduction to Tkinter**
  Fredrik Lundh's on-line reference material.

* **Tkinter reference: a GUI for Python**
  On-line reference material.

* **Tkinter for JPython**
  The Jython interface to Tkinter.

* **Python and Tkinter Programming**

Subsections

- [20.1.1 Tkinter Modules](#)
- [20.1.2 Tkinter Life Preserver](#)
  - [20.1.2.1 How To Use This Section](#)
  - [20.1.2.2 A Simple Hello World Program](#)
• 20.1.3 A (Very) Quick Look at Tcl/Tk
• 20.1.4 Mapping Basic Tk into Tkinter
• 20.1.5 How Tk and Tkinter are Related
• 20.1.6 Handy Reference
  ○ 20.1.6.1 Setting Options
  ○ 20.1.6.2 The Packer
  ○ 20.1.6.3 Packer Options
  ○ 20.1.6.4 Coupling Widget Variables
  ○ 20.1.6.5 The Window Manager
  ○ 20.1.6.6 Tk Option Data Types
  ○ 20.1.6.7 Bindings and Events
  ○ 20.1.6.8 The index Parameter
  ○ 20.1.6.9 Images

Previous: 20. Graphical User Interfaces Up: 20. Graphical User Interfaces
Next: 20.1.1 Tkinter Modules

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20.1.1 Tkinter Modules

Most of the time, the `Tkinter` module is all you really need, but a number of additional modules are available as well. The Tk interface is located in a binary module named `_tkinter`. This module contains the low-level interface to Tk, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter.

In addition to the Tk interface module, `Tkinter` includes a number of Python modules. The two most important modules are the `Tkinter` module itself, and a module called `Tkconstants`. The former automatically imports the latter, so to use Tkinter, all you need to do is to import one module:

```
import Tkinter
```

Or, more often:

```
from Tkinter import *
```

**class Tk**( `screenName=None, baseName=None, className='Tk', useTk=1` )

The `Tk` class is instantiated without arguments. This creates a toplevel widget of Tk which usually is the main window of an application. Each instance has its own associated Tcl interpreter. Changed in version 2.4: The `useTk` parameter was added.

**Tcl**( `screenName=None, baseName=None, className='Tk', useTk=0` )

The `Tcl` function is a factory function which creates an object much like that created by the `Tk` class, except that it does not initialize the Tk subsystem. This is most often useful when driving the Tcl interpreter in an environment where one doesn't want to create extraneous toplevel windows, or where one cannot (such as UNIX/Linux systems without an X server). An object created by the `Tcl` object can have a Toplevel window created (and the Tk subsystem initialized) by calling its `loadtk` method. New in version 2.4.

Other modules that provide Tk support include:
ScrolledText
Text widget with a vertical scroll bar built in.

tkColorChooser
Dialog to let the user choose a color.

tkCommonDialog
Base class for the dialogs defined in the other modules listed here.

tkFileDialog
Common dialogs to allow the user to specify a file to open or save.

tkFont
Utilities to help work with fonts.

tkMessageBox
Access to standard Tk dialog boxes.

tkSimpleDialog
Basic dialogs and convenience functions.

TkDnd
Drag-and-drop support for Tkinter. This is experimental and should become deprecated when it is replaced with the Tk DND.

turtle
Turtle graphics in a Tk window.
20.1.2 Tkinter Life Preserver

This section is not designed to be an exhaustive tutorial on either Tk or Tkinter. Rather, it is intended as a stop gap, providing some introductory orientation on the system.

Credits:

- Tkinter was written by Steen Lumholt and Guido van Rossum.
- Tk was written by John Ousterhout while at Berkeley.
- This Life Preserver was written by Matt Conway at the University of Virginia.
- The html rendering, and some liberal editing, was produced from a FrameMaker version by Ken Manheimer.
- Fredrik Lundh elaborated and revised the class interface descriptions, to get them current with Tk 4.2.
- Mike Clarkson converted the documentation to LaTeX, and compiled the User Interface chapter of the reference manual.

Subsections

- 20.1.2.1 How To Use This Section
- 20.1.2.2 A Simple Hello World Program

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20.1.3 A (Very) Quick Look at Tcl/Tk

The class hierarchy looks complicated, but in actual practice, application programmers almost always refer to the classes at the very bottom of the hierarchy.

Notes:

- These classes are provided for the purposes of organizing certain functions under one namespace. They aren't meant to be instantiated independently.

- The Tk class is meant to be instantiated only once in an application. Application programmers need not instantiate one explicitly, the system creates one whenever any of the other classes are instantiated.

- The Widget class is not meant to be instantiated, it is meant only for subclassing to make ```real``` widgets (in C++, this is called an `abstract class`).

To make use of this reference material, there will be times when you will need to know how to read short passages of Tk and how to identify the various parts of a Tk command. (See section 20.1.4 for the Tkinter equivalents of what's below.)

Tk scripts are Tcl programs. Like all Tcl programs, Tk scripts are just lists of tokens separated by spaces. A Tk widget is just its `class`, the `options` that help configure it, and the `actions` that make it do useful things.

To make a widget in Tk, the command is always of the form:

```
classCommand newPathname options
```

`classCommand`

denotes which kind of widget to make (a button, a label, a menu...)

`newPathname`

is the new name for this widget. All names in Tk must be unique. To help enforce this, widgets in Tk are named with `pathnames`, just like files in a file system. The top level widget, the `root`, is called `. (period)` and children
are delimited by more periods. For example, .myApp.controlPanel.okButton might be the name of a widget.

**options**

configure the widget's appearance and in some cases, its behavior. The options come in the form of a list of flags and values. Flags are proceeded by a `-`, like UNIX shell command flags, and values are put in quotes if they are more than one word.

For example:

```
button .fred -fg red -text "hi there"
```

Once created, the pathname to the widget becomes a new command. This new *widget command* is the programmer's handle for getting the new widget to perform some *action*. In C, you'd express this as someAction(fred, someOptions), in C++, you would express this as fred.someAction(someOptions), and in Tk, you say:

```
.fred someAction someOptions
```

Note that the object name, `.fred`, starts with a dot.

As you'd expect, the legal values for *someAction* will depend on the widget's class: `.fred disable` works if fred is a button (fred gets greyed out), but does not work if fred is a label (disabbling of labels is not supported in Tk).

The legal values of *someOptions* is action dependent. Some actions, like disable, require no arguments, others, like a text-entry box's delete command, would need arguments to specify what range of text to delete.
See About this document for information on suggesting changes.
20.1.4 Mapping Basic Tk into Tkinter

Class commands in Tk correspond to class constructors in Tkinter.

```python
button .fred

=====> fred = Button()
```

The master of an object is implicit in the new name given to it at creation time. In Tkinter, masters are specified explicitly.

```python
button .panel.fred

=====> fred = Button(panel)
```

The configuration options in Tk are given in lists of hyphenated tags followed by values. In Tkinter, options are specified as keyword-arguments in the instance constructor, and keyword-args for configure calls or as instance indices, in dictionary style, for established instances. See section 20.1.6 on setting options.

```python
button .fred -fg red

.fred configure -fg red

OR ==> fred.config(fg = "red")
```

In Tk, to perform an action on a widget, use the widget name as a command, and follow it with an action name, possibly with arguments (options). In Tkinter, you call methods on the class instance to invoke actions on the widget. The actions (methods) that a given widget can perform are listed in the Tkinter.py module.

```python
.fred invoke

=====> fred.invoke()
```

To give a widget to the packer (geometry manager), you call pack with optional arguments. In Tkinter, the Pack class holds all this functionality, and the various forms of the pack command are implemented as methods. All widgets in Tkinter are subclassed from the Packer, and so inherit all the packing methods. See the Tix module documentation for additional information on the Form geometry manager.

```python
pack .fred -side left

=====> fred.pack(side = "left")
```
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20.1.5 How Tk and Tkinter are Related

Note: This was derived from a graphical image; the image will be used more directly in a subsequent version of this document.

From the top down:

**Your App Here (Python)**
A Python application makes a Tkinter call.

**Tkinter (Python Module)**
This call (say, for example, creating a button widget), is implemented in the Tkinter module, which is written in Python. This Python function will parse the commands and the arguments and convert them into a form that makes them look as if they had come from a Tk script instead of a Python script.

**tkinter (C)**
These commands and their arguments will be passed to a C function in the tkinter - note the lowercase - extension module.

**Tk Widgets (C and Tcl)**
This C function is able to make calls into other C modules, including the C functions that make up the Tk library. Tk is implemented in C and some Tcl. The Tcl part of the Tk widgets is used to bind certain default behaviors to widgets, and is executed once at the point where the Python Tkinter module is imported. (The user never sees this stage).

**Tk (C)**
The Tk part of the Tk Widgets implement the final mapping to ...

**Xlib (C)**
The Xlib library to draw graphics on the screen.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20.2 Tix -- Extension widgets for Tk

The Tix (Tk Interface Extension) module provides an additional rich set of widgets. Although the standard Tk library has many useful widgets, they are far from complete. The Tix library provides most of the commonly needed widgets that are missing from standard Tk: HList, ComboBox, Control (a.k.a. SpinBox) and an assortment of scrollable widgets. Tix also includes many more widgets that are generally useful in a wide range of applications: Notebook, FileEntry, PanedWindow, etc; there are more than 40 of them.

With all these new widgets, you can introduce new interaction techniques into applications, creating more useful and more intuitive user interfaces. You can design your application by choosing the most appropriate widgets to match the special needs of your application and users.

See Also:

Tix Homepage
The home page for Tix. This includes links to additional documentation and downloads.

Tix Man Pages
On-line version of the man pages and reference material.

Tix Programming Guide
On-line version of the programmer's reference material.

Tix Development Applications
Tix applications for development of Tix and Tkinter programs. Tide applications work under Tk or Tkinter, and include TixInspect, an inspector to remotely modify and debug Tix/Tk/Tkinter applications.
20.2.1 Using Tix

20.2.2 Tix Widgets
  - 20.2.2.1 Basic Widgets
  - 20.2.2.2 File Selectors
  - 20.2.2.3 Hierarchical ListBox
  - 20.2.2.4 Tabular ListBox
  - 20.2.2.5 Manager Widgets
  - 20.2.2.6 Image Types
  - 20.2.2.7 Miscellaneous Widgets
  - 20.2.2.8 Form Geometry Manager

20.2.3 Tix Commands

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20.2.1 Using Tix

class Tix( screenName[, baseName[, className]])

Toplevel widget of Tix which represents mostly the main window of an application. It has an associated Tcl interpreter.

Classes in the Tix module subclasses the classes in the Tkinter module. The former imports the latter, so to use Tix with Tkinter, all you need to do is to import one module. In general, you can just import Tix, and replace the toplevel call to Tkinter.Tk with Tix.Tk:

```
import Tix
from Tkconstants import *
root = Tix.Tk()
```

To use Tix, you must have the Tix widgets installed, usually alongside your installation of the Tk widgets. To test your installation, try the following:

```
import Tix
root = Tix.Tk()
root.tk.eval('package require Tix')
```

If this fails, you have a Tk installation problem which must be resolved before proceeding. Use the environment variable TIX_LIBRARY to point to the installed Tix library directory, and make sure you have the dynamic object library (tix8183.dll or libtix8183.so) in the same directory that contains your Tk dynamic object library (tk8183.dll or libtk8183.so). The directory with the dynamic object library should also have a file called pkgIndex.tcl (case sensitive), which contains the line:

```
package ifneeded Tix 8.1 [list load "[file join $dir tix8183.dll"]"
```

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20.2.2 Tix Widgets

Tix introduces over 40 widget classes to the Tkinter repertoire. There is a demo of all the Tix widgets in the Demo/tix directory of the standard distribution.

Subsections

- 20.2.2.1 Basic Widgets
- 20.2.2.2 File Selectors
- 20.2.2.3 Hierachical ListBox
- 20.2.2.4 Tabular ListBox
- 20.2.2.5 Manager Widgets
- 20.2.2.6 Image Types
- 20.2.2.7 Miscellaneous Widgets
- 20.2.2.8 Form Geometry Manager

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20.2.3 Tix Commands

class tixCommand():
    The tix commands provide access to miscellaneous elements of Tix's internal state and the Tix application context. Most of the information manipulated by these methods pertains to the application as a whole, or to a screen or display, rather than to a particular window.

    To view the current settings, the common usage is:

        import Tix
        root = Tix.Tk()
        print root.tix_configure()

    tix_configure([cnf], **kw)
    Query or modify the configuration options of the Tix application context. If no option is specified, returns a dictionary all of the available options. If option is specified with no value, then the method returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. Option may be any of the configuration options.

    tix_cget(option)
    Returns the current value of the configuration option given by option. Option may be any of the configuration options.

    tix_getbitmap(name)
    Locates a bitmap file of the name name.xpm or name in one of the bitmap directories (see the tix_addbitmapdir() method). By using tix_getbitmap(), you can avoid hard coding the pathnames of the bitmap files in your application. When successful, it returns the complete pathname of the bitmap file, prefixed with the character "@". The returned value can be used to configure the bitmap option of the Tk and Tix widgets.
**tix_addbitmapdir(directory)**

Tix maintains a list of directories under which the `tix_getimage()` and `tix_getbitmap()` methods will search for image files. The standard bitmap directory is `$TIX_LIBRARY/bitmaps`. The `tix_addbitmapdir()` method adds `directory` into this list. By using this method, the image files of an application can also be located using the `tix_getimage()` or `tix_getbitmap()` method.

**tix_filedialog([dlgclass])**

Returns the file selection dialog that may be shared among different calls from this application. This method will create a file selection dialog widget when it is called for the first time. This dialog will be returned by all subsequent calls to `tix_filedialog()`. An optional dlgclass parameter can be passed as a string to specify what type of file selection dialog widget is desired. Possible options are `tix`, `FileSelectDialog` or `tixExFileSelectDialog`.

**tix_getimage(self, name)**

Locates an image file of the name `name.xpm`, `name.xbm` or `name.ppm` in one of the bitmap directories (see the `tix_addbitmapdir()` method above). If more than one file with the same name (but different extensions) exist, then the image type is chosen according to the depth of the X display: `xbm` images are chosen on monochrome displays and color images are chosen on color displays. By using `tix_getimage()`, you can avoid hard coding the pathnames of the image files in your application. When successful, this method returns the name of the newly created image, which can be used to configure the `image` option of the Tk and Tix widgets.

**tix_option_get(name)**

Gets the options maintained by the Tix scheme mechanism.

**tix_resetoptions(newScheme, newFontSet[, newScmPrio])**

Resets the scheme and fontset of the Tix application to `newScheme` and `newFontSet`, respectively. This affects only those widgets created after this call. Therefore, it is best to call the resetoptions method before the creation of any widgets in a Tix application.
The optional parameter `newScmPrio` can be given to reset the priority level of the Tk options set by the Tix schemes.

Because of the way Tk handles the X option database, after Tix has been imported and initied, it is not possible to reset the color schemes and font sets using the `tix_config()` method. Instead, the `tix_resetoptions()` method must be used.
20.3 ScrolledText -- Scrolled Text Widget

Availability: Tk.

The ScrolledText module provides a class of the same name which implements a basic text widget which has a vertical scroll bar configured to do the "right thing." Using the ScrolledText class is a lot easier than setting up a text widget and scroll bar directly. The constructor is the same as that of the Tkinter.Text class.

The text widget and scrollbar are packed together in a Frame, and the methods of the Grid and Pack geometry managers are acquired from the Frame object. This allows the ScrolledText widget to be used directly to achieve most normal geometry management behavior.

Should more specific control be necessary, the following attributes are available:

**frame**

The frame which surrounds the text and scroll bar widgets.

**vbar**

The scroll bar widget.
20.4 turtle -- Turtle graphics for Tk

Availability: Tk.

The turtle module provides turtle graphics primitives, in both an object-oriented and procedure-oriented ways. Because it uses Tkinter for the underlying graphics, it needs a version of python installed with Tk support.

The procedural interface uses a pen and a canvas which are automagically created when any of the functions are called.

The turtle module defines the following functions:

**degrees()**
Set angle measurement units to degrees.

**radians()**
Set angle measurement units to radians.

**setup(**kwargs**)**
Sets the size and position of the main window. Keywords are:
- **width**: either a size in pixels or a fraction of the screen. The default is 50% of the screen.
- **height**: either a size in pixels or a fraction of the screen. The default is 50% of the screen.
- **startx**: starting position in pixels from the left edge of the screen. None is the default value and centers the window horizontally on screen.
- **starty**: starting position in pixels from the top edge of the screen. None is the default value and centers the window vertically on screen.

Examples:

```
# Uses default geometry: 50% x 50% of screen, centered.
setup()

# Sets window to 200x200 pixels, in upper left of screen
```
setup (width=200, height=200, startx=0, starty=0)

# Sets window to 75% of screen by 50% of screen, and centers it.
setup(width=.75, height=0.5, startx=None, starty=None)

title(title_str)
    Set the window's title to title.

done()
    Enters the Tk main loop. The window will continue to be displayed until
    the user closes it or the process is killed.

reset()
    Clear the screen, re-center the pen, and set variables to the default values.

clear()
    Clear the screen.

tracer(flag)
    Set tracing on/off (according to whether flag is true or not). Tracing means
    line are drawn more slowly, with an animation of an arrow along the line.

speed(speed)
    Set the speed of the turtle. Valid values for the parameter speed are
    'fastest' (no delay), 'fast', (delay 5ms), 'normal' (delay 10ms),
    'slow' (delay 15ms), and 'slowest' (delay 20ms). New in version
    2.5.

delay(delay)
    Set the speed of the turtle to delay, which is given in ms. New in version
    2.5.

forward(distance)
    Go forward distance steps.

backward(distance)
    Go backward distance steps.
left(\textit{angle})
Turn left \textit{angle} units. Units are by default degrees, but can be set via the degrees() and radians() functions.

right(\textit{angle})
Turn right \textit{angle} units. Units are by default degrees, but can be set via the degrees() and radians() functions.

up()
Move the pen up -- stop drawing.

down()
Move the pen down -- draw when moving.

width(\textit{width})
Set the line width to \textit{width}.

color(\textit{s})
color((r, g, b))
color(r, g, b)
Set the pen color. In the first form, the color is specified as a Tk color specification as a string. The second form specifies the color as a tuple of the RGB values, each in the range \([0..1]\). For the third form, the color is specified giving the RGB values as three separate parameters (each in the range \([0..1]\)).

write(\textit{text[, move]})
Write \textit{text} at the current pen position. If \textit{move} is true, the pen is moved to the bottom-right corner of the text. By default, \textit{move} is false.

fill(\textit{flag})
The complete specifications are rather complex, but the recommended usage is: call \texttt{fill(1)} before drawing a path you want to fill, and call \texttt{fill(0)} when you finish to draw the path.

begin_fill()
Switch turtle into filling mode; Must eventually be followed by a corresponding end_fill() call. Otherwise it will be ignored. New in version 2.5.

**end_fill()**
End filling mode, and fill the shape; equivalent to fill(0). New in version 2.5.

**circle(radius[, extent])**
Draw a circle with radius radius whose center-point is radius units left of the turtle. extent determines which part of a circle is drawn: if not given it defaults to a full circle.

If extent is not a full circle, one endpoint of the arc is the current pen position. The arc is drawn in a counter clockwise direction if radius is positive, otherwise in a clockwise direction. In the process, the direction of the turtle is changed by the amount of the extent.

**goto(x, y)**
**goto((x, y))**
Go to co-ordinates x, y. The co-ordinates may be specified either as two separate arguments or as a 2-tuple.

**towards(x, y)**
Return the angle of the line from the turtle's position to the point x, y. The co-ordinates may be specified either as two separate arguments, as a 2-tuple, or as another pen object. New in version 2.5.

**heading()**
Return the current orientation of the turtle. New in version 2.3.

**setheading(angle)**
Set the orientation of the turtle to angle. New in version 2.3.

**position()**
Return the current location of the turtle as an (x, y) pair. New in version 2.3.
setx(x)
Set the x coordinate of the turtle to x. New in version 2.3.

sety(y)
Set the y coordinate of the turtle to y. New in version 2.3.

window_width()
Return the width of the canvas window. New in version 2.3.

window_height()
Return the height of the canvas window. New in version 2.3.

This module also does from math import *, so see the documentation for the math module for additional constants and functions useful for turtle graphics.

demo()
Exercise the module a bit.

exception Error
Exception raised on any error caught by this module.

For examples, see the code of the demo() function.

This module defines the following classes:

class Pen()
Define a pen. All above functions can be called as a methods on the given pen. The constructor automatically creates a canvas do be drawn on.

class Turtle()
Define a pen. This is essentially a synonym for Pen(); Turtle is an empty subclass of Pen.

class RawPen(canvas)
Define a pen which draws on a canvas canvas. This is useful if you want to use the module to create graphics in a ``real" program.
20.4.1 Turtle, Pen and RawPen Objects

Most of the global functions available in the module are also available as methods of the Turtle, Pen and RawPen classes, affecting only the state of the given pen.

The only method which is more powerful as a method is `degrees()`, which takes an optional argument letting you specify the number of units corresponding to a full circle:

```
degrees([fullcircle])
```

`fullcircle` is by default 360. This can cause the pen to have any angular units whatever: give `fullcircle 2*\pi` for radians, or 400 for gradians.
20.5 Idle

Idle is the Python IDE built with the Tkinter GUI toolkit.

IDLE has the following features:

- coded in 100% pure Python, using the Tkinter GUI toolkit
- cross-platform: works on Windows and UNIX (on Mac OS, there are currently problems with Tcl/Tk)
- multi-window text editor with multiple undo, Python colorizing and many other features, e.g. smart indent and call tips
- Python shell window (a.k.a. interactive interpreter)
- debugger (not complete, but you can set breakpoints, view and step)

---

Subsections

- **20.5.1 Menus**
  - 20.5.1.1 File menu
  - 20.5.1.2 Edit menu
  - 20.5.1.3 Windows menu
  - 20.5.1.4 Debug menu (in the Python Shell window only)
- **20.5.2 Basic editing and navigation**
  - 20.5.2.1 Automatic indentation
  - 20.5.2.2 Python Shell window
- **20.5.3 Syntax colors**
  - 20.5.3.1 Command line usage
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20.5.1 Menus

Subsections

- 20.5.1.1 File menu
- 20.5.1.2 Edit menu
- 20.5.1.3 Windows menu
- 20.5.1.4 Debug menu (in the Python Shell window only)

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20.5.2 Basic editing and navigation

- Backspace deletes to the left; Del deletes to the right
- Arrow keys and Page Up/Page Down to move around
- Home/End go to begin/end of line
- C-Home/C-End go to begin/end of file
- Some Emacs bindings may also work, including C-B, C-P, C-A, C-E, C-D, C-L
Command line usage
20.5.3 Syntax colors

The coloring is applied in a background `thread`, so you may occasionally see uncolorized text. To change the color scheme, edit the [Colors] section in config.txt.

Python syntax colors:

- **Keywords**: orange
- **Strings**: green
- **Comments**: red
- **Definitions**: blue

Shell colors:

- **Console output**: brown
- **stdout**: blue
- **stderr**: dark green
- **stdin**: black

Subsections

- [20.5.3.1 Command line usage](#)
See [About this document...] for information on suggesting changes.
20.6 Other Graphical User Interface Packages

There are an number of extension widget sets to Tkinter.

**Python megawidgets**

is a toolkit for building high-level compound widgets in Python using the Tkinter module. It consists of a set of base classes and a library of flexible and extensible megawidgets built on this foundation. These megawidgets include notebooks, comboboxes, selection widgets, paned widgets, scrolled widgets, dialog windows, etc. Also, with the Pmw.Blt interface to BLT, the busy, graph, stripchart, tabset and vector commands are be available.

The initial ideas for Pmw were taken from the Tk itcl extensions [incr Tk] by Michael McLennan and [incr Widgets] by Mark Ulferts. Several of the megawidgets are direct translations from the itcl to Python. It offers most of the range of widgets that [incr Widgets] does, and is almost as complete as Tix, lacking however Tix’s fast HList widget for drawing trees.

**Tkinter3000 Widget Construction Kit (WCK)**

is a library that allows you to write new Tkinter widgets in pure Python. The WCK framework gives you full control over widget creation, configuration, screen appearance, and event handling. WCK widgets can be very fast and light-weight, since they can operate directly on Python data structures, without having to transfer data through the Tk/Tcl layer.

Other GUI packages are also available for Python:

**wxPython**

wxPython is a cross-platform GUI toolkit for Python that is built around the popular wxWidgets C++ toolkit. It provides a native look and feel for applications on Windows, Mac OS X, and UNIX systems by using each platform's native widgets where ever possible, (GTK+ on UNIX-like systems). In addition to an extensive set of widgets, wxPython provides
classes for online documentation and context sensitive help, printing, HTML viewing, low-level device context drawing, drag and drop, system clipboard access, an XML-based resource format and more, including an ever growing library of user-contributed modules. Both the wxWidgets and wxPython projects are under active development and continuous improvement, and have active and helpful user and developer communities.

wxPython in Action
The wxPython book, by Noel Rappin and Robin Dunn.

PyQt
PyQt is a sip-wrapped binding to the Qt toolkit. Qt is an extensive C++ GUI toolkit that is available for UNIX, Windows and Mac OS X. sip is a tool for generating bindings for C++ libraries as Python classes, and is specifically designed for Python. An online manual is available at http://www.opendocspublishing.com/pyqt/ (errata are located at http://www.valdyas.org/python/book.html).

PyKDE
PyKDE is a sip-wrapped interface to the KDE desktop libraries. KDE is a desktop environment for UNIX computers; the graphical components are based on Qt.

EXPy
is a Python extension module which provides an interface to the FOX GUI. FOX is a C++ based Toolkit for developing Graphical User Interfaces easily and effectively. It offers a wide, and growing, collection of Controls, and provides state of the art facilities such as drag and drop, selection, as well as OpenGL widgets for 3D graphical manipulation. FOX also implements icons, images, and user-convenience features such as status line help, and tooltips.

Even though FOX offers a large collection of controls already, FOX leverages C++ to allow programmers to easily build additional Controls and GUI elements, simply by taking existing controls, and creating a derived class which simply adds or redefines the desired behavior.

PyGTK
is a set of bindings for the GTK widget set. It provides an object oriented
interface that is slightly higher level than the C one. It automatically does all the type casting and reference counting that you would have to do normally with the C API. There are also bindings to GNOME, and a tutorial is available.
21. Internationalization

The modules described in this chapter help you write software that is independent of language and locale by providing mechanisms for selecting a language to be used in program messages or by tailoring output to match local conventions.

The list of modules described in this chapter is:

- **gettext** Multilingual internationalization services.
- **locale** Internationalization services.

*Release 2.5, documentation updated on 19th September, 2006.*

See [About this document...](#) for information on suggesting changes.
21.1 gettext -- Multilingual internationalization services

The gettext module provides internationalization (I18N) and localization (L10N) services for your Python modules and applications. It supports both the GNU gettext message catalog API and a higher level, class-based API that may be more appropriate for Python files. The interface described below allows you to write your module and application messages in one natural language, and provide a catalog of translated messages for running under different natural languages.

Some hints on localizing your Python modules and applications are also given.

Subsections

- 21.1.1 GNU gettext API
- 21.1.2 Class-based API
  - 21.1.2.1 The NullTranslations class
  - 21.1.2.2 The GNUTranslations class
  - 21.1.2.3 Solaris message catalog support
  - 21.1.2.4 The Catalog constructor
- 21.1.3 Internationalizing your programs and modules
  - 21.1.3.1 Localizing your module
  - 21.1.3.2 Localizing your application
  - 21.1.3.3 Changing languages on the fly
  - 21.1.3.4 Deferred translations
  - 21.1.3.5 gettext() vs. lgettext()
- 21.1.4 Acknowledgements

Python Library Reference


Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
21.1.1 GNU gettext API

The `gettext` module defines the following API, which is very similar to the GNU `gettext` API. If you use this API you will affect the translation of your entire application globally. Often this is what you want if your application is monolingual, with the choice of language dependent on the locale of your user. If you are localizing a Python module, or if your application needs to switch languages on the fly, you probably want to use the class-based API instead.

`bindtextdomain`\( (\textit{domain}[, \textit{localedir}])\)

Bind the `domain` to the locale directory `localedir`. More concretely, `gettext` will look for binary `.mo` files for the given domain using the path (on UNiX): `localedir/language/LC_MESSAGES/domain.mo`, where `languages` is searched for in the environment variables `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG` respectively.

If `localedir` is omitted or `None`, then the current binding for `domain` is returned.\(^\text{21.1}\)

`bind_textdomain_codeset`\( (\textit{domain}[, \textit{codeset}])\)

Bind the `domain` to `codeset`, changing the encoding of strings returned by the `gettext()` family of functions. If `codeset` is omitted, then the current binding is returned.

New in version 2.4.

`textdomain`\( (\textit{[\textit{domain}]}))\)

Change or query the current global domain. If `domain` is `None`, then the current global domain is returned, otherwise the global domain is set to `domain`, which is returned.

`gettext`\( (\textit{message})\)

Return the localized translation of `message`, based on the current global domain, language, and locale directory. This function is usually aliased as `_` in the local namespace (see examples below).
lgettext( message)  
Equivalent to gettext(), but the translation is returned in the preferred 
system encoding, if no other encoding was explicitly set with
bind_textdomain_codeset().

New in version 2.4.

dgettext( domain, message)  
Like gettext(), but look the message up in the specified domain.

ldgettext( domain, message)  
Equivalent to dgettext(), but the translation is returned in the preferred 
system encoding, if no other encoding was explicitly set with
bind_textdomain_codeset().

New in version 2.4.

ngettext( singular, plural, n)  
Like gettext(), but consider plural forms. If a translation is found,
apply the plural formula to n, and return the resulting message (some
languages have more than two plural forms). If no translation is found,
return singular if n is 1; return plural otherwise.

The Plural formula is taken from the catalog header. It is a C or Python
expression that has a free variable n; the expression evaluates to the index
of the plural in the catalog. See the GNU gettext documentation for the
precise syntax to be used in .po files, and the formulas for a variety of
languages.

New in version 2.3.

lngettext( singular, plural, n)  
Equivalent to ngettext(), but the translation is returned in the preferred
system encoding, if no other encoding was explicitly set with
bind_textdomain_codeset().

New in version 2.4.
**dngettext** *(domain, singular, plural, n)*

Like **ngettext()**, but look the message up in the specified *domain*.

New in version 2.3.

**ldngettext** *(domain, singular, plural, n)*

Equivalent to **dngettext()**, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with **bind_textdomain_codeset()**.

New in version 2.4.

Note that GNU **gettext** also defines a **dcgettext()** method, but this was deemed not useful and so it is currently unimplemented.

Here's an example of typical usage for this API:

```python
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_
 = gettext.gettext
#
...
print _('This is a translatable string.')
```

---

**Footnotes**

... returned

The default locale directory is system dependent; for example, on RedHat Linux it is `/usr/share/locale`, but on Solaris it is `/usr/lib/locale`. The **gettext** module does not try to support these system dependent defaults; instead its default is `sys.prefix/share/locale`. For this reason, it is always best to call **bindtextdomain()** with an explicit absolute path at the start of your application.

---

**Python Library Reference**

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
21.1.2 Class-based API

The class-based API of the gettext module gives you more flexibility and greater convenience than the GNU gettext API. It is the recommended way of localizing your Python applications and modules. gettext defines a "translations" class which implements the parsing of GNU .mo format files, and has methods for returning either standard 8-bit strings or Unicode strings. Instances of this "translations" class can also install themselves in the built-in namespace as the function _().

find( domain[, localedir[, languages[, all]]])

This function implements the standard .mo file search algorithm. It takes a domain, identical to what textdomain() takes. Optional localedir is as in bindtextdomain() Optional languages is a list of strings, where each string is a language code.

If localedir is not given, then the default system locale directory is used. If languages is not given, then the following environment variables are searched: LANGUAGE, LC_ALL, LC_MESSAGES, and LANG. The first one returning a non-empty value is used for the languages variable. The environment variables should contain a colon separated list of languages, which will be split on the colon to produce the expected list of language code strings.

find() then expands and normalizes the languages, and then iterates through them, searching for an existing file built of these components:

localedir/language/LC_MESSAGES/domain.mo

The first such file name that exists is returned by find(). If no such file is found, then None is returned. If all is given, it returns a list of all file names, in the order in which they appear in the languages list or the environment variables.

translation( domain[, localedir[, languages[, class[, fallback[, codeset]]]]]])
Return a Translations instance based on the domain, localedir, and languages, which are first passed to find() to get a list of the associated .mo file paths. Instances with identical .mo file names are cached. The actual class instantiated is either class_ if provided, otherwise GNUTranslations. The class's constructor must take a single file object argument. If provided, codeset will change the charset used to encode translated strings.

If multiple files are found, later files are used as fallbacks for earlier ones. To allow setting the fallback, copy.copy is used to clone each translation object from the cache; the actual instance data is still shared with the cache.

If no .mo file is found, this function raises IOError if fallback is false (which is the default), and returns a NullTranslations instance if fallback is true.

Changed in version 2.4: Added the codeset parameter.

install( domain[, localedir[, unicode [, codeset[, names]]]])
This installs the function _ in Python's builtin namespace, based on domain, localedir, and codeset which are passed to the function translation(). The unicode flag is passed to the resulting translation object's install method.

For the names parameter, please see the description of the translation object's install method.

As seen below, you usually mark the strings in your application that are candidates for translation, by wrapping them in a call to the _() function, like this:

    print _('This string will be translated. ')

For convenience, you want the _() function to be installed in Python's builtin namespace, so it is easily accessible in all modules of your application.

Changed in version 2.4: Added the codeset parameter. Changed in version 2.5: Added the names parameter.
Footnotes

... used. 21.2

See the footnote for bindtextdomain() above.

Subsections

- 21.1.2.1 The NullTranslations class
- 21.1.2.2 The GNUTranslations class
- 21.1.2.3 Solaris message catalog support
- 21.1.2.4 The Catalog constructor

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
Localizing your module
21.1.3 Internationalizing your programs and modules

Internationalization (I18N) refers to the operation by which a program is made aware of multiple languages. Localization (L10N) refers to the adaptation of your program, once internationalized, to the local language and cultural habits. In order to provide multilingual messages for your Python programs, you need to take the following steps:

1. prepare your program or module by specially marking translatable strings
2. run a suite of tools over your marked files to generate raw messages catalogs
3. create language specific translations of the message catalogs
4. use the gettext module so that message strings are properly translated

In order to prepare your code for I18N, you need to look at all the strings in your files. Any string that needs to be translated should be marked by wrapping it in $\text{\_\_}('\ldots')$ -- that is, a call to the function $\text{\_\_}()$. For example:

```python
filename = 'mylog.txt'
message = $\text{\_\_}('writing a log message')$
fp = open(filename, 'w')
fp.write(message)
fp.close()
```

In this example, the string 'writing a log message' is marked as a candidate for translation, while the strings 'mylog.txt' and 'w' are not.

The Python distribution comes with two tools which help you generate the message catalogs once you've prepared your source code. These may or may not be available from a binary distribution, but they can be found in a source distribution, in the Tools/i18n directory.

The pygettext program scans all your Python source code looking for the strings you previously marked as translatable. It is similar to the GNU gettext program except that it understands all the intricacies of Python source code, but knows nothing about C or C++ source code. You don't need GNU gettext unless you're also going to be translating C code (such as C extension modules).
**pygettext** generates textual Uniforum-style human readable message catalog .pot files, essentially structured human readable files which contain every marked string in the source code, along with a placeholder for the translation strings. **pygettext** is a command line script that supports a similar command line interface as **xgettext**; for details on its use, run:

```
pygettext.py --help
```

Copies of these .pot files are then handed over to the individual human translators who write language-specific versions for every supported natural language. They send you back the filled in language-specific versions as a .po file. Using the **msgfmt.py** program (in the Tools/i18n directory), you take the .po files from your translators and generate the machine-readable .mo binary catalog files. The .mo files are what the gettext module uses for the actual translation processing during run-time.

How you use the gettext module in your code depends on whether you are internationalizing a single module or your entire application. The next two sections will discuss each case.

---

**Footnotes**

...**pygettext**

François Pinard has written a program called **xpot** which does a similar job. It is available as part of his **po-utils** package at [http://po-utils.progiciels-bpi.ca/](http://po-utils.progiciels-bpi.ca/).

...**msgfmt.py**

**msgfmt.py** is binary compatible with GNU **msgfmt** except that it provides a simpler, all-Python implementation. With this and **pygettext.py**, you generally won't need to install the GNU gettext package to internationalize your Python applications.

---

**Subsections**

- [21.1.3.1 Localizing your module](#)
21.1.3.2 Localizing your application
21.1.3.3 Changing languages on the fly
21.1.3.4 Deferred translations
21.1.3.5 gettext() vs. lgettext()
21.1.4 Acknowledgements

The following people contributed code, feedback, design suggestions, previous implementations, and valuable experience to the creation of this module:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer
21.2 locale -- Internationalization services

The `locale` module opens access to the POSIX locale database and functionality. The POSIX locale mechanism allows programmers to deal with certain cultural issues in an application, without requiring the programmer to know all the specifics of each country where the software is executed.

The `locale` module is implemented on top of the `_locale` module, which in turn uses an ANSI C locale implementation if available.

The `locale` module defines the following exception and functions:

**exception Error**

Exception raised when `setlocale()` fails.

**setlocale**( `category[, locale]` )

If `locale` is specified, it may be a string, a tuple of the form `(language code, encoding)`, or `None`. If it is a tuple, it is converted to a string using the locale aliasing engine. If `locale` is given and not `None`, `setlocale()` modifies the locale setting for the `category`. The available categories are listed in the data description below. The value is the name of a locale. An empty string specifies the user's default settings. If the modification of the locale fails, the exception `Error` is raised. If successful, the new locale setting is returned.

If `locale` is omitted or `None`, the current setting for `category` is returned.

`setlocale()` is not thread safe on most systems. Applications typically start with a call of

```python
import locale
locale.setlocale(locale.LC_ALL, '')
```

This sets the locale for all categories to the user's default setting (typically specified in the LANG environment variable). If the locale is not changed
thereafter, using multithreading should not cause problems.

Changed in version 2.0: Added support for tuple values of the `locale` parameter.

`localeconv()`

Returns the database of the local conventions as a dictionary. This dictionary has the following strings as keys:

<table>
<thead>
<tr>
<th>Category</th>
<th>Key</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_NUMERIC</td>
<td>'decimal_point'</td>
<td>Decimal point character.</td>
</tr>
<tr>
<td></td>
<td>'grouping'</td>
<td>Sequence specifying relative positions of thousands separator.</td>
</tr>
<tr>
<td></td>
<td>'thousands_sep'</td>
<td>Character used between groups.</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>'int_curr_symbol'</td>
<td>International currency symbol.</td>
</tr>
<tr>
<td></td>
<td>'currency_symbol'</td>
<td>Local currency symbol.</td>
</tr>
<tr>
<td></td>
<td>'p_cs_precedes/n_cs_precedes'</td>
<td>Whether symbol precedes value (for positive resp. negative values).</td>
</tr>
<tr>
<td></td>
<td>'p_sep_by_space/n_sep_by_space'</td>
<td>Whether symbol is separated by space (for positive resp. negative values).</td>
</tr>
</tbody>
</table>
from the value space (for positive resp. negative values).

| 'mon_decimal_point' | Decimal point used for monetary values. |
| 'frac_digits' | Number of fractional digits used in local formatting monetary values. |
| 'int_frac_digits' | Number of fractional digits used in international formatting of monetary values. |
| 'mon_thousands_sep' | Group separator used for monetary values. |
| 'mon_grouping' | Equivalent to 'grouping' for monetary values. |
| 'positive_sign' | Symbol used to annotate a positive monetary value. |
| 'negative_sign' | Symbol used to annotate a negative monetary value. |
| 'p_sign_posn/n_sign_posn' | The position of the sign (for positive resp. negative values), see below. |

All numeric values can be set to CHAR_MAX to indicate that there is no value specified in this locale.

The possible values for 'p_sign_posn' and 'n_sign_posn' are given below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Currency and value are surrounded by parentheses.</td>
</tr>
</tbody>
</table>
1 The sign should precede the value and currency symbol.
2 The sign should follow the value and currency symbol.
3 The sign should immediately precede the value.
4 The sign should immediately follow the value.

CHAR_MAX Nothing is specified in this locale.

**nl_langinfo** *(option)*

Return some locale-specific information as a string. This function is not available on all systems, and the set of possible options might also vary across platforms. The possible argument values are numbers, for which symbolic constants are available in the locale module.

**getdefaultlocale** *[envvars]*

Tries to determine the default locale settings and returns them as a tuple of the form *(language code, encoding)*.

According to POSIX, a program which has not called `setlocale(LC_ALL, '')` runs using the portable 'C' locale. Calling `setlocale(LC_ALL, '')` lets it use the default locale as defined by the LANG variable. Since we do not want to interfere with the current locale setting we thus emulate the behavior in the way described above.

To maintain compatibility with other platforms, not only the LANG variable is tested, but a list of variables given as envvars parameter. The first found to be defined will be used. envvars defaults to the search path used in GNU gettext; it must always contain the variable name "LANG". The GNU gettext search path contains 'LANGUAGE ', 'LC_ALL ', 'LC_CTYPE ', and 'LANG ', in that order.

Except for the code 'C ', the language code corresponds to RFC 1766. *language code* and *encoding* may be None if their values cannot be determined. New in version 2.0.

**getlocale** *[category]*

Returns the current setting for the given locale category as sequence containing *language code, encoding*. *category* may be one of the LC__*
values except LC_ALL. It defaults to LC_CTYPE.

Except for the code 'C', the language code corresponds to RFC 1766. language code and encoding may be None if their values cannot be determined. New in version 2.0.

getpreferredencoding([do_setlocale])
Return the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

On some systems, it is necessary to invoke setlocale to obtain the user preferences, so this function is not thread-safe. If invoking setlocale is not necessary or desired, do_setlocale should be set to False.

New in version 2.3.

normalize(localename)
Returns a normalized locale code for the given locale name. The returned locale code is formatted for use with setlocale(). If normalization fails, the original name is returned unchanged.

If the given encoding is not known, the function defaults to the default encoding for the locale code just like setlocale(). New in version 2.0.

resetlocale([category])
Sets the locale for category to the default setting.

The default setting is determined by calling getdefaultlocale(). category defaults to LC_ALL. New in version 2.0.

strcoll(string1, string2)
Compares two strings according to the current LC_COLLATE setting. As any other compare function, returns a negative, or a positive value, or 0, depending on whether string1 collates before or after string2 or is equal to it.
**strxfrm**(*string*)

Transforms a string to one that can be used for the built-in function `cmp()`, and still returns locale-aware results. This function can be used when the same string is compared repeatedly, e.g. when collating a sequence of strings.

**format**(*format*, *val*, *grouping*[*, monetary]*)

Formats a number *val* according to the current LC_NUMERIC setting. The format follows the conventions of the `%` operator. For floating point values, the decimal point is modified if appropriate. If *grouping* is true, also takes the grouping into account.

If *monetary* is true, the conversion uses monetary thousands separator and grouping strings.

Please note that this function will only work for exactly one `%` char specifier. For whole format strings, use `format_string()`.

Changed in version 2.5: Added the *monetary* parameter.

**format_string**(*format*, *val*, *grouping*)

Processes formatting specifiers as in `format % val`, but takes the current locale settings into account.

New in version 2.5.

**currency**(*val*, *symbol*, *grouping*[*, international]*)

Formats a number *val* according to the current LC_MONETARY settings.

The returned string includes the currency symbol if *symbol* is true, which is the default. If *grouping* is true (which is not the default), grouping is done with the value. If *international* is true (which is not the default), the international currency symbol is used.

Note that this function will not work with the `C` locale, so you have to set a locale via `setlocale()` first.

New in version 2.5.
**str** *(float)*
Formats a floating point number using the same format as the built-in function `str(float)`, but takes the decimal point into account.

**atof** *(string)*
Converts a string to a floating point number, following the **LC_NUMERIC** settings.

**atoi** *(string)*
Converts a string to an integer, following the **LC_NUMERIC** conventions.

**LC CTYPE**
Locale category for the character type functions. Depending on the settings of this category, the functions of module `string` dealing with case change their behaviour.

**LC_COLLATE**
Locale category for sorting strings. The functions `strcoll()` and `strxfrm()` of the `locale` module are affected.

**LC TIME**
Locale category for the formatting of time. The function `time.strftime()` follows these conventions.

**LC_MONETARY**
Locale category for formatting of monetary values. The available options are available from the `localeconv()` function.

**LC_MESSAGES**
Locale category for message display. Python currently does not support application specific locale-aware messages. Messages displayed by the operating system, like those returned by `os.strerror()` might be affected by this category.

**LC_NUMERIC**
Locale category for formatting numbers. The functions `format()`, `atoi()`, `atof()` and `str()` of the `locale` module are affected by that category. All other numeric formatting operations are not affected.
**LC_ALL**
Combination of all locale settings. If this flag is used when the locale is changed, setting the locale for all categories is attempted. If that fails for any category, no category is changed at all. When the locale is retrieved using this flag, a string indicating the setting for all categories is returned. This string can be later used to restore the settings.

**CHAR_MAX**
This is a symbolic constant used for different values returned by `localeconv()`.

The `nl_langinfo` function accepts one of the following keys. Most descriptions are taken from the corresponding description in the GNU C library.

**CODESET**
Return a string with the name of the character encoding used in the selected locale.

**D_T_FMT**
Return a string that can be used as a format string for `strftime(3)` to represent time and date in a locale-specific way.

**D_FMT**
Return a string that can be used as a format string for `strftime(3)` to represent a date in a locale-specific way.

**T_FMT**
Return a string that can be used as a format string for `strftime(3)` to represent a time in a locale-specific way.

**T_FMT_AMPM**
The return value can be used as a format string for `strftime` to represent time in the am/pm format.

**DAY_1 . . . DAY_7**
Return name of the n-th day of the week. **Warning:** This follows the US convention of `DAY_1` being Sunday, not the international convention (ISO 8601) that Monday is the first day of the week.

**ABDAY_1 . . . ABDAY_7**
Return abbreviated name of the n-th day of the week.

**MON_1 . . . MON_12**
Return name of the n-th month.

**ABMON_1 . . . ABMON_12**
Return abbreviated name of the n-th month.

**RADIXCHAR**
Return radix character (decimal dot, decimal comma, etc.)

**THOUSEP**
Return separator character for thousands (groups of three digits).

**YESEXPR**
Return a regular expression that can be used with the regex function to recognize a positive response to a yes/no question. **Warning:** The expression is in the syntax suitable for the `regex()` function from the C library, which might differ from the syntax used in `re`.

**NOEXPR**
Return a regular expression that can be used with the `regex(3)` function to recognize a negative response to a yes/no question.

**CRNCYSTR**
Return the currency symbol, preceded by "-" if the symbol should appear before the value, "+" if the symbol should appear after the value, or "." if the symbol should replace the radix character.

**ERA**
The return value represents the era used in the current locale.

Most locales do not define this value. An example of a locale which does define this value is the Japanese one. In Japan, the traditional representation of dates includes the name of the era corresponding to the then-emperor’s reign.

Normally it should not be necessary to use this value directly. Specifying the `E` modifier in their format strings causes the `strftime` function to use this information. The format of the returned string is not specified, and
therefore you should not assume knowledge of it on different systems.

**ERA_YEAR**
The return value gives the year in the relevant era of the locale.

**ERA_D_T_FMT**
This return value can be used as a format string for `strftime` to represent dates and times in a locale-specific era-based way.

**ERA_D_FMT**
This return value can be used as a format string for `strftime` to represent time in a locale-specific era-based way.

**ALT_DIGITS**
The return value is a representation of up to 100 values used to represent the values 0 to 99.

Example:

```python
>>> import locale
>>> loc = locale.getlocale(locale.LC_ALL) # get current locale
>>> locale.setlocale(locale.LC_ALL, 'de_DE') # use German locale; na
>>> locale.strcoll('f\xe4n', 'foo') # compare a string containing an
>>> locale.setlocale(locale.LC_ALL, '') # use user's preferred local
>>> locale.setlocale(locale.LC_ALL, 'C') # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc) # restore saved locale
```

---

Subsections

- [21.2.1 Background, details, hints, tips and caveats](#)
- [21.2.2 For extension writers and programs that embed Python](#)
- [21.2.3 Access to message catalogs](#)
21.2.1 Background, details, hints, tips and caveats

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementation are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

Initially, when a program is started, the locale is the "C" locale, no matter what the user's preferred locale is. The program must explicitly say that it wants the user's preferred locale settings by calling `setlocale(LC_ALL, '')`.

It is generally a bad idea to call `setlocale()` in some library routine, since as a side effect it affects the entire program. Saving and restoring it is almost as bad: it is expensive and affects other threads that happen to run before the settings have been restored.

If, when coding a module for general use, you need a locale independent version of an operation that is affected by the locale (such as `string.lower()`, or certain formats used with `time.strftime()`), you will have to find a way to do it without using the standard library routine. Even better is convincing yourself that using locale settings is okay. Only as a last resort should you document that your module is not compatible with non-"C" locale settings.

The case conversion functions in the `string` module are affected by the locale settings. When a call to the `setlocale()` function changes the LC_CTYPE settings, the variables `string.lowercase`, `string.uppercase` and `string.letters` are recalculated. Note that this code that uses these variable through `from ... import ...`, e.g. `from string import letters`, is not affected by subsequent `setlocale()` calls.

The only way to perform numeric operations according to the locale is to use the special functions defined by this module: `atof()`, `atoi()`, `format()`, `str()`.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
21.2.2 For extension writers and programs that embed Python

Extension modules should never call `setlocale()`, except to find out what the current locale is. But since the return value can only be used portably to restore it, that is not very useful (except perhaps to find out whether or not the locale is "C").

When Python code uses the `locale` module to change the locale, this also affects the embedding application. If the embedding application doesn't want this to happen, it should remove the `_locale` extension module (which does all the work) from the table of built-in modules in the `config.c` file, and make sure that the `_locale` module is not accessible as a shared library.
21.2.3 Access to message catalogs

The locale module exposes the C library's gettext interface on systems that provide this interface. It consists of the functions gettext(), dgettext(), dcgettext(), textdomain(), bindtextdomain(), and bind_textdomain_codeset(). These are similar to the same functions in the gettext module, but use the C library's binary format for message catalogs, and the C library's search algorithms for locating message catalogs.

Python applications should normally find no need to invoke these functions, and should use gettext instead. A known exception to this rule are applications that link use additional C libraries which internally invoke gettext() or dcgettext(). For these applications, it may be necessary to bind the text domain, so that the libraries can properly locate their message catalogs.
22. Program Frameworks

The modules described in this chapter are frameworks that will largely dictate the structure of your program. Currently the modules described here are all oriented toward writing command-line interfaces.

The full list of modules described in this chapter is:

- **cmd**: Build line-oriented command interpreters.
- **shlex**: Simple lexical analysis for UNIX shell-like languages.

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
22.1 cmd -- Support for line-oriented command interpreters

The Cmd class provides a simple framework for writing line-oriented command interpreters. These are often useful for test harnesses, administrative tools, and prototypes that will later be wrapped in a more sophisticated interface.

```python
class Cmd( [completekey[, stdin[, stdout]]])
```

A Cmd instance or subclass instance is a line-oriented interpreter framework. There is no good reason to instantiate Cmd itself; rather, it's useful as a superclass of an interpreter class you define yourself in order to inherit Cmd's methods and encapsulate action methods.

The optional argument `completekey` is the `readline` name of a completion key; it defaults to Tab. If `completekey` is not `None` and `readline` is available, command completion is done automatically.

The optional arguments `stdin` and `stdout` specify the input and output file objects that the Cmd instance or subclass instance will use for input and output. If not specified, they will default to `sys.stdin` and `sys.stdout`.

Changed in version 2.3: The `stdin` and `stdout` parameters were added.
22.1.1 Cmd Objects

A Cmd instance has the following methods:

**cmdloop([intro])**

Repeatedly issue a prompt, accept input, parse an initial prefix off the received input, and dispatch to action methods, passing them the remainder of the line as argument.

The optional argument is a banner or intro string to be issued before the first prompt (this overrides the intro class member).

If the `readline` module is loaded, input will automatically inherit bash-like history-list editing (e.g. Control-P scrolls back to the last command, Control-N forward to the next one, Control-F moves the cursor to the right non-destructively, Control-B moves the cursor to the left non-destructively, etc.).

An end-of-file on input is passed back as the string 'EOF'.

An interpreter instance will recognize a command name "foo" if and only if it has a method `do_foo()`. As a special case, a line beginning with the character "?" is dispatched to the method `do_help()`. As another special case, a line beginning with the character "!" is dispatched to the method `do_shell()` (if such a method is defined).

This method will return when the `postcmd()` method returns a true value. The stop argument to `postcmd()` is the return value from the command's corresponding `do_*()` method.

If completion is enabled, completing commands will be done automatically, and completing of commands args is done by calling `complete_foo()` with arguments `text`, `line`, `begidx`, and `endidx`. `text` is the string prefix we are attempting to match: all returned matches must begin with it. `line` is the current input line with leading whitespace removed, `begidx` and `endidx` are the beginning and ending indexes of the prefix text, which could be used to provide different completion depending upon which position the argument...
is in.

All subclasses of `Cmd` inherit a predefined `do_help()`. This method, called with an argument 'bar', invokes the corresponding method `help_bar()`. With no argument, `do_help()` lists all available help topics (that is, all commands with corresponding `help_*( )` methods), and also lists any undocumented commands.

**onecmd( str)**
Interpret the argument as though it had been typed in response to the prompt. This may be overridden, but should not normally need to be; see the `precmd()` and `postcmd()` methods for useful execution hooks. The return value is a flag indicating whether interpretation of commands by the interpreter should stop. If there is a `do_*( )` method for the command `str`, the return value of that method is returned, otherwise the return value from the `default( )` method is returned.

**emptyline( )**
Method called when an empty line is entered in response to the prompt. If this method is not overridden, it repeats the last nonempty command entered.

**default( line)**
Method called on an input line when the command prefix is not recognized. If this method is not overridden, it prints an error message and returns.

**completesdefault( text, line, begidx, endidx)**
Method called to complete an input line when no command-specific `complete_*( )` method is available. By default, it returns an empty list.

**precmd( line)**
Hook method executed just before the command line `line` is interpreted, but after the input prompt is generated and issued. This method is a stub in `Cmd`; it exists to be overridden by subclasses. The return value is used as the command which will be executed by the `onecmd( )` method; the `precmd( )` implementation may re-write the command or simply return `line` unchanged.
postcmd( stop, line)
Hook method executed just after a command dispatch is finished. This
method is a stub in Cmd; it exists to be overridden by subclasses. line is the
command line which was executed, and stop is a flag which indicates
whether execution will be terminated after the call to postcmd(); this
will be the return value of the onecmd() method. The return value of this
method will be used as the new value for the internal flag which
corresponds to stop; returning false will cause interpretation to continue.

preloop()
Hook method executed once when cmdloop() is called. This method is a
stub in Cmd; it exists to be overridden by subclasses.

postloop()
Hook method executed once when cmdloop() is about to return. This
method is a stub in Cmd; it exists to be overridden by subclasses.

Instances of Cmd subclasses have some public instance variables:

prompt
The prompt issued to solicit input.

identchars
The string of characters accepted for the command prefix.

lastcmd
The last nonempty command prefix seen.

intro
A string to issue as an intro or banner. May be overridden by giving the
cmdloop() method an argument.

doc_header
The header to issue if the help output has a section for documented
commands.

misc_header
The header to issue if the help output has a section for miscellaneous help
topics (that is, there are help_*() methods without corresponding do_*
() methods).

**undoc_header**
The header to issue if the help output has a section for undocumented commands (that is, there are do_*() methods without corresponding help_*() methods).

**ruler**
The character used to draw separator lines under the help-message headers. If empty, no ruler line is drawn. It defaults to ":=".

**use_rawinput**
A flag, defaulting to true. If true, cmdloop() uses raw_input() to display a prompt and read the next command; if false, sys.stdout.write() and sys.stdin.readline() are used. (This means that by importing readline, on systems that support it, the interpreter will automatically support Emacs-like line editing and command-history keystrokes.)
22.2 shlex -- Simple lexical analysis

New in version 1.5.2.

The shlex class makes it easy to write lexical analyzers for simple syntaxes resembling that of the UNIX shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

**Note:** The shlex module currently does not support Unicode input.

The shlex module defines the following functions:

```
split([s[, comments]])
```

Split the string s using shell-like syntax. If comments is False (the default), the parsing of comments in the given string will be disabled (setting the commenters member of the shlex instance to the empty string). This function operates in POSIX mode. New in version 2.3.

The shlex module defines the following class:

```
class shlex([instream[, infile[, posix]]])
```

A shlex instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with read() and readline() methods, or a string (strings are accepted since Python 2.3). If no argument is given, input will be taken from sys.stdin. The second optional argument is a filename string, which sets the initial value of the infile member. If the instream argument is omitted or equal to sys.stdin, this second argument defaults to `\stdin`. The posix argument was introduced in Python 2.3, and defines the operational mode. When posix is not true (default), the shlex instance will operate in compatibility mode. When operating in POSIX mode, shlex will try to be as close as possible to the POSIX shell parsing rules. See section [22.2.1](#).
See Also:

Module **ConfigParser**:
Parser for configuration files similar to the Windows .ini files.

Subsections

- [22.2.1 shlex Objects](#)
- [22.2.2 Parsing Rules](#)

Python Library Reference

Previous: [22.1.1 Cmd Objects](#) Up: [22. Program Frameworks](#) Next: [22.2.1 shlex Objects](#)

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
22.2.1 shlex Objects

A shlex instance has the following methods:

**get_token()**
Return a token. If tokens have been stacked using `push_token()`, pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, `self.eof` is returned (the empty string ("")) in non-POSIX mode, and None in POSIX mode).

**push_token( str)**
Push the argument onto the token stack.

**read_token()**
Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

**sourcehook( filename)**
When shlex detects a source request (see source below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as `sys.stdin`), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles `#include "file.h"`).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with `open()` called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)
This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding `close' hook, but a shlex instance will call the close() method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the push_source() and pop_source() methods.

**push_source(stream[, filename])**

Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the sourcehook method. New in version 2.1.

**pop_source()**

Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream. New in version 2.1.

**error_leader([file[, line]])**

This method generates an error message leader in the format of a UNIX C compiler error label; the format is "%s", line %d: ', where the "%s" is replaced with the name of the current source file and the "%d" with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage shlex users to generate error messages in the standard, parseable format understood by Emacs and other UNIX tools.

Instances of shlex subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

**commenters**

The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just "#" by default.
**wordchars**
The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore.

**whitespace**
Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

**escape**
Characters that will be considered as escape. This will be only used in POSIX mode, and includes just "\" by default. New in version 2.3.

**quotes**
Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

**escapedquotes**
Characters in quotes that will interpret escape characters defined in escape. This is only used in POSIX mode, and includes just "\" by default. New in version 2.3.

**whitespace_split**
If True, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with shlex, getting tokens in a similar way to shell arguments. New in version 2.3.

**infile**
The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

**instream**
The input stream from which this shlex instance is reading characters.

**source**
This member is None by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the "source"
keyword in various shells. That is, the immediately following token will opened as a filename and input taken from that stream until EOF, at which point the close() method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

**debug**

If this member is numeric and 1 or more, a shlex instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

**lineno**

Source line number (count of newlines seen so far plus one).

**token**

The token buffer. It may be useful to examine this when catching exceptions.

**eof**

Token used to determine end of file. This will be set to the empty string (''), in non-POSIX mode, and to None in POSIX mode. New in version 2.3.
22.2.2 Parsing Rules

When operating in non-POSIX mode, shlex will try to obey to the following rules.

- Quote characters are not recognized within words (Do"Not"Separate is parsed as the single word Do"Not"Separate);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words ("Do"Separate is parsed as "Do" and Separate);
- If whitespace_split is False, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is True, shlex will only split words in whitespaces;
- EOF is signaled with an empty string ('');
- It's not possible to parse empty strings, even if quoted.

When operating in POSIX mode, shlex will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words ("Do"Not"Separate" is parsed as the single word DoNotSeparate);
- Non-quoted escape characters (e.g. "\") preserve the literal value of the next character that follows;
- Enclosing characters in quotes which are not part of escapedquotes (e.g. "'" ) preserve the literal value of all characters within the quotes;
- Enclosing characters in quotes which are part of escapedquotes (e.g. "\"" ) preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in escape. The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.
- EOF is signaled with a None value;
- Quoted empty strings (' ') are allowed;
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
23. Development Tools

The modules described in this chapter help you write software. For example, the pydoc module takes a module and generates documentation based on the module's contents. The doctest and unittest modules contains frameworks for writing unit tests that automatically exercise code and verify that the expected output is produced.

The list of modules described in this chapter is:

- **pydoc**
  Documentation generator and online help system.
- **doctest**
  A framework for verifying interactive Python examples.
- **unittest**
  Unit testing framework for Python.
- **test**
  Regression tests package containing the testing suite for Python.
- **test.test_support**
  Support for Python regression tests.

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
23.1 pydoc -- Documentation generator and online help system

New in version 2.1.

The pydoc module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a Web browser, or saved to HTML files.

The built-in function help() invokes the online help system in the interactive interpreter, which uses pydoc to generate its documentation as text on the console. The same text documentation can also be viewed from outside the Python interpreter by running pydoc as a script at the operating system's command prompt. For example, running

```
    pydoc sys
```

at a shell prompt will display documentation on the sys module, in a style similar to the manual pages shown by the UNIX man command. The argument to pydoc can be the name of a function, module, or package, or a dotted reference to a class, method, or function within a module or module in a package. If the argument to pydoc looks like a path (that is, it contains the path separator for your operating system, such as a slash in UNIX), and refers to an existing Python source file, then documentation is produced for that file.

Specifying a -w flag before the argument will cause HTML documentation to be written out to a file in the current directory, instead of displaying text on the console.

Specifying a -k flag before the argument will search the synopsis lines of all available modules for the keyword given as the argument, again in a manner similar to the UNIX man command. The synopsis line of a module is the first line of its documentation string.

You can also use pydoc to start an HTTP server on the local machine that will serve documentation to visiting Web browsers. pydoc -p 1234 will start a HTTP server on port 1234, allowing you to browse the documentation at
http://localhost:1234/ in your preferred Web browser. `pydoc -g` will start the server and additionally bring up a small Tkinter-based graphical interface to help you search for documentation pages.

When `pydoc` generates documentation, it uses the current environment and path to locate modules. Thus, invoking `pydoc spam` documents precisely the version of the module you would get if you started the Python interpreter and typed "import spam".

Module docs for core modules are assumed to reside in http://www.python.org/doc/current/lib/. This can be overridden by setting the PYTHONDOCS environment variable to a different URL or to a local directory containing the Library Reference Manual pages.

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
23.2 doctest -- Test interactive Python examples

The `doctest` module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. There are several common ways to use doctest:

- To check that a module's docstrings are up-to-date by verifying that all interactive examples still work as documented.
- To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.
- To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of "literate testing" or "executable documentation".

Here's a complete but small example module:

```python
"""
This is the "example" module.

The example module supplies one function, factorial(). For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    If the result is small enough to fit in an int, return an int.
    Else return a long.
"

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> [factorial(long(n)) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    2652528598121910586363084800000000L
    >>> factorial(30L)
    2652528598121910586363084800000000L
    >>> factorial(-1)
```

```
Traceback (most recent call last):
...
ValueError: n must be >= 0

Factorials of floats are OK, but the float must be an exact integer:
>>> factorial(30.1)
Traceback (most recent call last):
...
ValueError: n must be exact integer
>>> factorial(30.0)
2652528598121910586363084800000000L

It must also not be ridiculously large:
>>> factorial(1e100)
Traceback (most recent call last):
...
OverflowError: n too large
""

import math
if not n >= 0:
    raise ValueError("n must be >= 0")
if math.floor(n) != n:
    raise ValueError("n must be exact integer")
if n+1 == n:  # catch a value like 1e300
    raise OverflowError("n too large")
result = 1
factor = 2
while factor <= n:
    result *= factor
    factor += 1
return result

def _test():
    import doctest
doctest.testmod()

    if __name__ == "__main__":
        _test()

If you run example.py directly from the command line, doctest works its magic:

$ python example.py
$

There's no output! That's normal, and it means all the examples worked. Pass -v to the script, and doctest prints a detailed log of what it's trying, and prints a summary at the end:
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
Trying:
    [factorial(long(n)) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok

And so on, eventually ending with:

Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
      ...
    OverflowError: n too large
ok
1 items had no tests:  
    __main__.test  
2 items passed all tests:  
    1 tests in __main__  
    8 tests in __main__.factorial  
9 tests in 3 items.  
9 passed and 0 failed.  
Test passed.  
$

That's all you need to know to start making productive use of doctest! Jump in. The following sections provide full details. Note that there are many examples of doctests in the standard Python test suite and libraries. Especially useful examples can be found in the standard test file Lib/test/test_doctest.py.

---

**Subsections**

- 23.2.1 Simple Usage: Checking Examples in Docstrings
- 23.2.2 Simple Usage: Checking Examples in a Text File
- 23.2.3 How It Works
  - 23.2.3.1 Which Docstrings Are Examined?
23.2.3.2 How are Docstring Examples Recognized?
23.2.3.3 What's the Execution Context?
23.2.3.4 What About Exceptions?
23.2.3.5 Option Flags and Directives
23.2.3.6 Warnings

- 23.2.4 Basic API
- 23.2.5 Unittest API
- 23.2.6 Advanced API
  - 23.2.6.1 DocTest Objects
  - 23.2.6.2 Example Objects
  - 23.2.6.3 DocTestFinder objects
  - 23.2.6.4 DocTestParser objects
  - 23.2.6.5 DocTestRunner objects
  - 23.2.6.6 OutputChecker objects

- 23.2.7 Debugging
- 23.2.8 Soapbox
23.2.1 Simple Usage: Checking Examples in Docstrings

The simplest way to start using doctest (but not necessarily the way you'll continue to do it) is to end each module M with:

```python
def _test():
    import doctest
doctest.testmod()

if __name__ == "__main__":
    _test()
```

doctest then examines docstrings in module M.

Running the module as a script causes the examples in the docstrings to get executed and verified:

```bash
python M.py
```

This won't display anything unless an example fails, in which case the failing example(s) and the cause(s) of the failure(s) are printed to stdout, and the final line of output is "***Test Failed*** N failures.", where N is the number of examples that failed.

Run it with the -v switch instead:

```bash
python M.py -v
```

and a detailed report of all examples tried is printed to standard output, along with assorted summaries at the end.

You can force verbose mode by passing `verbose=True` to `testmod()`, or prohibit it by passing `verbose=False`. In either of those cases, `sys.argv` is not examined by `testmod()` (so passing `-v` or not has no effect).

For more information on `testmod()`, see section 23.2.4.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
23.2.2 Simple Usage: Checking Examples in a Text File

Another simple application of doctest is testing interactive examples in a text file. This can be done with the `testfile()` function:

```python
import doctest
doctest.testfile("example.txt")
```

That short script executes and verifies any interactive Python examples contained in the file `example.txt`. The file content is treated as if it were a single giant docstring; the file doesn't need to contain a Python program! For example, perhaps `example.txt` contains this:

```restructuredtext
The ``example`` module

Using ``factorial``

This is an example text file in reStructuredText format. First impo
``factorial`` from the ``example`` module:

```python
>>> from example import factorial
```

Now use it:

```python
>>> factorial(6)
120
```

Running `doctest.testfile("example.txt")` then finds the error in this documentation:

```plaintext
File "./example.txt", line 14, in example.txt
Failed example:
  factorial(6)
Expected:
  120
Got:
  720
```

As with `testmod()`, `testfile()` won't display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to stdout, using the same format as `testmod()`.
By default, `testfile()` looks for files in the calling module's directory. See section 23.2.4 for a description of the optional arguments that can be used to tell it to look for files in other locations.

Like `testmod()`, `testfile()`'s verbosity can be set with the `-v` command-line switch or with the optional keyword argument `verbose`.

For more information on `testfile()`, see section 23.2.4.
23.2.3 How It Works

This section examines in detail how doctest works: which docstrings it looks at, how it finds interactive examples, what execution context it uses, how it handles exceptions, and how option flags can be used to control its behavior. This is the information that you need to know to write doctest examples; for information about actually running doctest on these examples, see the following sections.

Subsections

- [23.2.3.1 Which Docstrings Are Examined?](#)
- [23.2.3.2 How are Docstring Examples Recognized?](#)
- [23.2.3.3 What's the Execution Context?](#)
- [23.2.3.4 What About Exceptions?](#)
- [23.2.3.5 Option Flags and Directives](#)
- [23.2.3.6 Warnings](#)
23.2.4 Basic API

The functions `testmod()` and `testfile()` provide a simple interface to `doctest` that should be sufficient for most basic uses. For a less formal introduction to these two functions, see sections 23.2.1 and 23.2.2.

\[
\text{testfile(filename[, module_relative][, name][, package][, globs][, verbose][, report][, optionflags][, extraglobs][, raise_on_error][, parser][, encoding])}
\]

All arguments except `filename` are optional, and should be specified in keyword form.

Test examples in the file named `filename`. Return "(failure_count, test_count)".

Optional argument `module_relative` specifies how the filename should be interpreted:

- If `module_relative` is `True` (the default), then `filename` specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the `package` argument is specified, then it is relative to that package. To ensure OS-independence, `filename` should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If `module_relative` is `False`, then `filename` specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument `name` gives the name of the test; by default, or if `None`, `os.path.basename(filename)` is used.

Optional argument `package` is a Python package or the name of a Python package whose directory should be used as the base directory for a module-relative filename. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an
error to specify package if module_relative is False.

Optional argument globs gives a dict to be used as the globals when executing examples. A new shallow copy of this dict is created for the doctest, so its examples start with a clean slate. By default, or if None, a new empty dict is used.

Optional argument extraglobs gives a dict merged into the globals used to execute examples. This works like dict.update(): if globs and extraglobs have a common key, the associated value in extraglobs appears in the combined dict. By default, or if None, no extra globs are used. This is an advanced feature that allows parameterization of doctests. For example, a doctest can be written for a base class, using a generic name for the class, then reused to test any number of subclasses by passing an extraglobs dict mapping the generic name to the subclass to be tested.

Optional argument verbose prints lots of stuff if true, and prints only failures if false; by default, or if None, it's true if and only if '-v' is in sys.argv.

Optional argument report prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else the summary is very brief (in fact, empty if all tests passed).

Optional argument optionflags or's together option flags. See section 23.2.3.

Optional argument raise_on_error defaults to false. If true, an exception is raised upon the first failure or unexpected exception in an example. This allows failures to be post-mortem debugged. Default behavior is to continue running examples.

Optional argument parser specifies a DocTestParser (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., DocTestParser()).

Optional argument encoding specifies an encoding that should be used to convert the file to unicode.

New in version 2.4.
Changed in version 2.5: The parameter *encoding* was added.

```

testmod([m][, name][, globs][, verbose][, report][, optionflags][, extraglobs][, raise_on_error][, exclude_empty])
```

All arguments are optional, and all except for *m* should be specified in keyword form.

Test examples in docstrings in functions and classes reachable from module *m* (or module *__main__* if *m* is not supplied or is *None*), starting with *m.__doc__*.

Also test examples reachable from dict *m.__test__*, if it exists and is not *None*. *m.__test__* maps names (strings) to functions, classes and strings; function and class docstrings are searched for examples; strings are searched directly, as if they were docstrings.

Only docstrings attached to objects belonging to module *m* are searched.

Return "(failure_count, test_count)".

Optional argument *name* gives the name of the module; by default, or if *None, m.__name__* is used.

Optional argument *exclude_empty* defaults to false. If true, objects for which no doctests are found are excluded from consideration. The default is a backward compatibility hack, so that code still using `doctest.master.summarize()` in conjunction with `testmod()` continues to get output for objects with no tests. The *exclude_empty* argument to the newer `DocTestFinder` constructor defaults to true.

Optional arguments *extraglobs, verbose, report, optionflags, raise_on_error*, and *globs* are the same as for function `testfile()` above, except that *globs* defaults to *m.__dict__*.

Changed in version 2.3: The parameter *optionflags* was added.

Changed in version 2.4: The parameters *extraglobs, raise_on_error* and *exclude_empty* were added.
Changed in version 2.5: The optional argument \textit{isprivate}, deprecated in 2.4, was removed.

There's also a function to run the doctests associated with a single object. This function is provided for backward compatibility. There are no plans to deprecate it, but it's rarely useful:

\begin{verbatim}
run_docstring_examples(f, globs[, verbose][, name][, compileflags][, optionflags])
\end{verbatim}

Test examples associated with object \textit{f}; for example, \textit{f} may be a module, function, or class object.

A shallow copy of dictionary argument \textit{globs} is used for the execution context.

Optional argument \textit{name} is used in failure messages, and defaults to "NoName".

If optional argument \textit{verbose} is true, output is generated even if there are no failures. By default, output is generated only in case of an example failure.

Optional argument \textit{compileflags} gives the set of flags that should be used by the Python compiler when running the examples. By default, or if \textit{None}, flags are deduced corresponding to the set of future features found in \textit{globs}.

Optional argument \textit{optionflags} works as for function \textit{testfile()} above.
23.2.5 Unittest API

As your collection of doctest'ed modules grows, you'll want a way to run all their doctests systematically. Prior to Python 2.4, doctest had a barely documented Tester class that supplied a rudimentary way to combine doctests from multiple modules. Tester was feeble, and in practice most serious Python testing frameworks build on the unittest module, which supplies many flexible ways to combine tests from multiple sources. So, in Python 2.4, doctest’s Tester class is deprecated, and doctest provides two functions that can be used to create unittest test suites from modules and text files containing doctests. These test suites can then be run using unittest test runners:

```python
import unittest
import doctest
import my_module_with_doctests, and_another

suite = unittest.TestSuite()
for mod in my_module_with_doctests, and_another:
    suite.addTest(doctest.DocTestSuite(mod))
runner = unittest.TextTestRunner()
runner.run(suite)
```

There are two main functions for creating unittest.TestSuite instances from text files and modules with doctests:

```
DocFileSuite([module_relative][, package][, setUp][, tearDown][, globs][, optionflags][, parser][, encoding])
```

Convert doctest tests from one or more text files to a unittest.TestSuite.

The returned unittest.TestSuite is to be run by the unittest framework and runs the interactive examples in each file. If an example in any file fails, then the synthesized unit test fails, and a failureException exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Pass one or more paths (as strings) to text files to be examined.
Options may be provided as keyword arguments:

Optional argument `module_relative` specifies how the filenames in `paths` should be interpreted:

- If `module_relative` is `True` (the default), then each filename specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the `package` argument is specified, then it is relative to that package. To ensure OS-independence, each filename should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If `module_relative` is `False`, then each filename specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument `package` is a Python package or the name of a Python package whose directory should be used as the base directory for module-relative filenames. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify `package` if `module_relative` is `False`.

Optional argument `setUp` specifies a set-up function for the test suite. This is called before running the tests in each file. The `setUp` function will be passed a `DocTest` object. The `setUp` function can access the test globals as the `globs` attribute of the test passed.

Optional argument `tearDown` specifies a tear-down function for the test suite. This is called after running the tests in each file. The `tearDown` function will be passed a `DocTest` object. The `tearDown` function can access the test globals as the `globs` attribute of the test passed.

Optional argument `globs` is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, `globs` is a new empty dictionary.

Optional argument `optionflags` specifies the default doctest options for the tests, created by or-ing together individual option flags. See section 23.2.3. See function `set_unittest_reportflags()` below for a better way
to set reporting options.

Optional argument *parser* specifies a *DocTestParser* (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., *DocTestParser*).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

New in version 2.4.

Changed in version 2.5: The global **file** was added to the globals provided to doctests loaded from a text file using *DocFileSuite()*.

Changed in version 2.5: The parameter *encoding* was added.

**DocTestSuite(***module***[, **globs**][, **extraglobs**][, **test_finder**][, **setUp**][, **tearDown**][, **checker**])**

Convert doctest tests for a module to a unittest.*TestSuite.*

The returned unittest.*TestSuite* is to be run by the unittest framework and runs each doctest in the module. If any of the doctests fail, then the synthesized unit test fails, and a failureException exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Optional argument *module* provides the module to be tested. It can be a module object or a (possibly dotted) module name. If not specified, the module calling this function is used.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is a new empty dictionary.

Optional argument *extraglobs* specifies an extra set of global variables, which is merged into *globs*. By default, no extra globals are used.

Optional argument *test_finder* is the *DocTestFinder* object (or a drop-in replacement) that is used to extract doctests from the module.
Optional arguments setUp, tearDown, and optionflags are the same as for function DocFileSuite() above.

New in version 2.3.

Changed in version 2.4: The parameters globs, extraglobs, test_finder, setUp, tearDown, and optionflags were added; this function now uses the same search technique as testmod().

Under the covers, DocTestSuite() creates a unittest.TestSuite out of doctest.DocTestCase instances, and DocTestCase is a subclass of unittest.TestCase. DocTestCase isn't documented here (it's an internal detail), but studying its code can answer questions about the exact details of unittest integration.

Similarly, DocFileSuite() creates a unittest.TestSuite out of doctest.DocFileCase instances, and DocFileCase is a subclass of DocTestCase.

So both ways of creating a unittest.TestSuite run instances of DocTestCase. This is important for a subtle reason: when you run doctest functions yourself, you can control the doctest options in use directly, by passing option flags to doctest functions. However, if you're writing a unittest framework, unittest ultimately controls when and how tests get run. The framework author typically wants to control doctest reporting options (perhaps, e.g., specified by command line options), but there's no way to pass options through unittest to doctest test runners.

For this reason, doctest also supports a notion of doctest reporting flags specific to unittest support, via this function:

**set_unittest_reportflags(flags)**

Set the doctest reporting flags to use.

Argument flags or's together option flags. See section 23.2.3. Only "reporting flags" can be used.

This is a module-global setting, and affects all future doctests run by module unittest: the runTest() method of DocTestCase looks at
the option flags specified for the test case when the DocTestCase instance was constructed. If no reporting flags were specified (which is the typical and expected case), doctest's unittest reporting flags are or'ed into the option flags, and the option flags so augmented are passed to the DocTestRunner instance created to run the doctest. If any reporting flags were specified when the DocTestCase instance was constructed, doctest's unittest reporting flags are ignored.

The value of the unittest reporting flags in effect before the function was called is returned by the function.

New in version 2.4.
23.2.6 Advanced API

The basic API is a simple wrapper that's intended to make doctest easy to use. It is fairly flexible, and should meet most users' needs; however, if you require more fine-grained control over testing, or wish to extend doctest's capabilities, then you should use the advanced API.

The advanced API revolves around two container classes, which are used to store the interactive examples extracted from doctest cases:

- **Example**: A single python statement, paired with its expected output.
- **DocTest**: A collection of Examples, typically extracted from a single docstring or text file.

Additional processing classes are defined to find, parse, and run, and check doctest examples:

- **DocTestFinder**: Finds all docstrings in a given module, and uses a **DocTestParser** to create a **DocTest** from every docstring that contains interactive examples.
- **DocTestParser**: Creates a **DocTest** object from a string (such as an object's docstring).
- **DocTestRunner**: Executes the examples in a **DocTest**, and uses an **OutputChecker** to verify their output.
- **OutputChecker**: Compares the actual output from a doctest example with the expected output, and decides whether they match.

The relationships among these processing classes are summarized in the following diagram:

```
+--------+          +--------+          +--------+
| module | --> DocTestFinder --> | DocTest | --> DocTestRunner --> results
|--------+                |--------+                |--------+
        |                  ^    |                  ^    | (printed)
        |                  |    |                  |
        |                  |    |                  |
        |                  |    |                  |
          v                  v                  v
          DocTestParser    Example    OutputChecker
```

```
23.2.7 Debugging

Doctest provides several mechanisms for debugging doctest examples:

- Several functions convert doctests to executable Python programs, which can be run under the Python debugger, pdb.
- The DebugRunner class is a subclass of DocTestRunner that raises an exception for the first failing example, containing information about that example. This information can be used to perform post-mortem debugging on the example.
- The unittest cases generated by DocTestSuite() support the debug() method defined by unittest.TestCase.
- You can add a call to pdb.set_trace() in a doctest example, and you'll drop into the Python debugger when that line is executed. Then you can inspect current values of variables, and so on. For example, suppose a.py contains just this module docstring:

    """
    >>> def f(x):
    ...     g(x*2)
    >>> def g(x):
    ...     print x+3
    ...     import pdb; pdb.set_trace()
    >>> f(3)
    9
    """

Then an interactive Python session may look like this:

    >>> import a, doctest
    >>> doctest.testmod(a)
    --Return--
    > <doctest a[1]>(3)g()--None
    --> import pdb; pdb.set_trace()
     (Pdb) list
     1    def g(x):
     2     print x+3
     3   --> import pdb; pdb.set_trace()
    [EOF]
     (Pdb) print x
    6
     (Pdb) step
    --Return--
    > <doctest a[0]>(2)f()--None
Changed in version 2.4: The ability to use `pdb.set_trace()` usefully inside doctests was added.

Functions that convert doctests to Python code, and possibly run the synthesized code under the debugger:

**script_from_examples(s)**

Convert text with examples to a script.

Argument *s* is a string containing doctest examples. The string is converted to a Python script, where doctest examples in *s* are converted to regular code, and everything else is converted to Python comments. The generated script is returned as a string. For example,

```python
import doctest
print doctest.script_from_examples(r""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print x+y
    3

"")
```

displays:

```plaintext
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print x+y
# Expected:
```
This function is used internally by other functions (see below), but can also be useful when you want to transform an interactive Python session into a Python script.

New in version 2.4.

**testsource**( *module, name*)

Convert the doctest for an object to a script.

Argument *module* is a module object, or dotted name of a module, containing the object whose doctests are of interest. Argument *name* is the name (within the module) of the object with the doctests of interest. The result is a string, containing the object's docstring converted to a Python script, as described for **script_from_examples()** above. For example, if module *a.py* contains a top-level function *f()* , then

```python
import a, doctest
print doctest.testsource(a, "a.f")
```

prints a script version of function *f()*'s docstring, with doctests converted to code, and the rest placed in comments.

New in version 2.3.

**debug**( *module, name*, [ *pm* ])

Debug the doctests for an object.

The *module* and *name* arguments are the same as for function **testsource()** above. The synthesized Python script for the named object's docstring is written to a temporary file, and then that file is run under the control of the Python debugger, **pdb**.

A shallow copy of *module*.

```
''
```

is used for both local and global execution context.

Optional argument *pm* controls whether post-mortem debugging is used. If *pm* has a true value, the script file is run directly, and the debugger gets involved only if the script terminates via raising an unhandled exception. If
it does, then post-mortem debugging is invoked, via `pdb.post_mortem()`, passing the traceback object from the unhandled exception. If `pm` is not specified, or is false, the script is run under the debugger from the start, via passing an appropriate `execfile()` call to `pdb.run()`.

New in version 2.3.

Changed in version 2.4: The `pm` argument was added.

```python
def debug_src(src, pm=None, globs=None)
    Debug the doctests in a string.

    This is like function `debug()` above, except that a string containing doctest examples is specified directly, via the `src` argument.

    Optional argument `pm` has the same meaning as in function `debug()` above.

    Optional argument `globs` gives a dictionary to use as both local and global execution context. If not specified, or `None`, an empty dictionary is used. If specified, a shallow copy of the dictionary is used.

    New in version 2.4.
```

The `DebugRunner` class, and the special exceptions it may raise, are of most interest to testing framework authors, and will only be sketched here. See the source code, and especially `DebugRunner`'s docstring (which is a doctest!) for more details:

```python
class DebugRunner([checker][, verbose][, optionflags])
    A subclass of `DocTestRunner` that raises an exception as soon as a failure is encountered. If an unexpected exception occurs, an `UnexpectedException` exception is raised, containing the test, the example, and the original exception. If the output doesn't match, then a `DocTestFailure` exception is raised, containing the test, the example, and the actual output.
```
For information about the constructor parameters and methods, see the
documentation for `DocTestRunner` in section 23.2.6.

There are two exceptions that may be raised by `DebugRunner` instances:

**exception DocTestFailure** *(test, example, got)*

An exception thrown by `DocTestRunner` to signal that a doctest
eexample's actual output did not match its expected output. The constructor
arguments are used to initialize the member variables of the same names.

`DocTestFailure` defines the following member variables:

**test**

The `DocTest` object that was being run when the example failed.

**example**

The `Example` that failed.

**got**

The example's actual output.

**exception UnexpectedException** *(test, example, exc_info)*

An exception thrown by `DocTestRunner` to signal that a doctest
eexample raised an unexpected exception. The constructor arguments are
used to initialize the member variables of the same names.

`UnexpectedException` defines the following member variables:

**test**

The `DocTest` object that was being run when the example failed.

**example**

The `Example` that failed.

**exc_info**

A tuple containing information about the unexpected exception, as returned
by `sys.exc_info()`.
Soapbox

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
23.2.8 Soapbox

As mentioned in the introduction, `doctest` has grown to have three primary uses:

1. Checking examples in docstrings.
2. Regression testing.
3. Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There's an art to this that needs to be learned--it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. I'm still amazed at how often one of my `doctest` examples stops working after a "harmless" change.

Doctest also makes an excellent tool for regression testing, especially if you don't skimp on explanatory text. By interleaving prose and examples, it becomes much easier to keep track of what's actually being tested, and why. When a test fails, good prose can make it much easier to figure out what the problem is, and how it should be fixed. It's true that you could write extensive comments in code-based testing, but few programmers do. Many have found that using doctest approaches instead leads to much clearer tests. Perhaps this is simply because doctest makes writing prose a little easier than writing code, while writing comments in code is a little harder. I think it goes deeper than just that: the natural attitude when writing a doctest-based test is that you want to explain the fine points of your software, and illustrate them with examples. This in turn naturally leads to test files that start with the simplest features, and logically progress to complications and edge cases. A coherent narrative is the result, instead of a collection of isolated functions that test isolated bits of functionality seemingly at random. It's a different attitude, and produces different results, blurring the distinction between testing and explaining.
Regression testing is best confined to dedicated objects or files. There are several options for organizing tests:

- Write text files containing test cases as interactive examples, and test the files using `testfile()` or `DocFileSuite()`. This is recommended, although is easiest to do for new projects, designed from the start to use doctest.
- Define functions named `_regrtest_topic` that consist of single docstrings, containing test cases for the named topics. These functions can be included in the same file as the module, or separated out into a separate test file.
- Define a `__test__` dictionary mapping from regression test topics to docstrings containing test cases.
23.3 `unittest` -- Unit testing framework

New in version 2.1.

The Python unit testing framework, sometimes referred to as `PyUnit`, is a Python language version of JUnit, by Kent Beck and Erich Gamma. JUnit is, in turn, a Java version of Kent's Smalltalk testing framework. Each is the de facto standard unit testing framework for its respective language.

`unittest` supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. The `unittest` module provides classes that make it easy to support these qualities for a set of tests.

To achieve this, `unittest` supports some important concepts:

**test fixture**

A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

**test case**

A test case is the smallest unit of testing. It checks for a specific response to a particular set of inputs. `unittest` provides a base class, `TestCase`, which may be used to create new test cases.

**test suite**

A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

**test runner**

A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of
executing the tests.

The test case and test fixture concepts are supported through the `TestCase` and `FunctionTestCase` classes; the former should be used when creating new tests, and the latter can be used when integrating existing test code with a `unittest`-driven framework. When building test fixtures using `TestCase`, the `setUp()` and `tearDown()` methods can be overridden to provide initialization and cleanup for the fixture. With `FunctionTestCase`, existing functions can be passed to the constructor for these purposes. When the test is run, the fixture initialization is run first; if it succeeds, the cleanup method is run after the test has been executed, regardless of the outcome of the test. Each instance of the `TestCase` will only be used to run a single test method, so a new fixture is created for each test.

Test suites are implemented by the `TestSuite` class. This class allows individual tests and test suites to be aggregated; when the suite is executed, all tests added directly to the suite and in `"child"` test suites are run.

A test runner is an object that provides a single method, `run()`, which accepts a `TestCase` or `TestSuite` object as a parameter, and returns a result object. The class `TestResult` is provided for use as the result object. `unittest` provides the `TextTestRunner` as an example test runner which reports test results on the standard error stream by default. Alternate runners can be implemented for other environments (such as graphical environments) without any need to derive from a specific class.

**See Also:**

Module [doctest](#)

Another test-support module with a very different flavor.

*Simple Smalltalk Testing: With Patterns*

Kent Beck's original paper on testing frameworks using the pattern shared by `unittest`.
• 23.3.1 Basic example
• 23.3.2 Organizing test code
• 23.3.3 Re-using old test code
• 23.3.4 Classes and functions
• 23.3.5 TestCase Objects
• 23.3.6 TestSuite Objects
• 23.3.7 TestResult Objects
• 23.3.8 TestLoader Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
23.3.1 Basic example

The unittest module provides a rich set of tools for constructing and running tests. This section demonstrates that a small subset of the tools suffice to meet the needs of most users.

Here is a short script to test three functions from the `random` module:

```python
import random
import unittest

class TestSequenceFunctions(unittest.TestCase):
    def setUp(self):
        self.seq = range(10)

    def testshuffle(self):
        # make sure the shuffled sequence does not lose any elements
        random.shuffle(self.seq)
        self.seq.sort()
        self.assertEqual(self.seq, range(10))

    def testchoice(self):
        element = random.choice(self.seq)
        self.assert_(element in self.seq)

    def testsample(self):
        self.assertRaises(ValueError, random.sample, self.seq, 20)
        for element in random.sample(self.seq, 5):
            self.assert_(element in self.seq)

if __name__ == '__main__':
    unittest.main()
```

A testcase is created by subclassing `unittest.TestCase`. The three individual tests are defined with methods whose names start with the letters "test". This naming convention informs the test runner about which methods represent tests.

The crux of each test is a call to `assertEqual()` to check for an expected result; `assert_()` to verify a condition; or `assertRaises()` to verify that an expected exception gets raised. These methods are used instead of the `assert` statement so the test runner can accumulate all test results and produce a report.
When a `setUp()` method is defined, the test runner will run that method prior to each test. Likewise, if a `tearDown()` method is defined, the test runner will invoke that method after each test. In the example, `setUp()` was used to create a fresh sequence for each test.

The final block shows a simple way to run the tests. `unittest.main()` provides a command line interface to the test script. When run from the command line, the above script produces an output that looks like this:

```
... ran testload.000s
OK
```

Instead of `unittest.main()`, there are other ways to run the tests with a finer level of control, less terse output, and no requirement to be run from the command line. For example, the last two lines may be replaced with:

```python
suite = unittest.TestLoader().loadTestsFromTestCase(TestSequenceFunctions)
unittest.TextTestRunner(verbosity=2).run(suite)
```

Running the revised script from the interpreter or another script produces the following output:

```
testchoice (__main__.TestSequenceFunctions) ... ok
testsample (__main__.TestSequenceFunctions) ... ok
testshuffle (__main__.TestSequenceFunctions) ... ok
```

```
Ran 3 tests in 0.110s
OK
```

The above examples show the most commonly used `unittest` features which are sufficient to meet many everyday testing needs. The remainder of the documentation explores the full feature set from first principles.
23.3.2 Organizing test code

The basic building blocks of unit testing are *test cases* -- single scenarios that must be set up and checked for correctness. In `unittest`, test cases are represented by instances of `unittest`'s `TestCase` class. To make your own test cases you must write subclasses of `TestCase`, or use `FunctionTestCase`.

An instance of a `TestCase`-derived class is an object that can completely run a single test method, together with optional set-up and tidy-up code.

The testing code of a `TestCase` instance should be entirely self contained, such that it can be run either in isolation or in arbitrary combination with any number of other test cases.

The simplest `TestCase` subclass will simply override the `runTest()` method in order to perform specific testing code:

```python
import unittest
class DefaultWidgetSizeTestCase(unittest.TestCase):
    def runTest(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50), 'incorrect default size')
```

Note that in order to test something, we use the one of the `assert*()` or `fail*()` methods provided by the `TestCase` base class. If the test fails, an exception will be raised, and `unittest` will identify the test case as a *failure*. Any other exceptions will be treated as *errors*. This helps you identify where the problem is: *failures* are caused by incorrect results - a 5 where you expected a 6. *Errors* are caused by incorrect code - e.g., a `TypeError` caused by an incorrect function call.

The way to run a test case will be described later. For now, note that to construct an instance of such a test case, we call its constructor without arguments:

```python
testCase = DefaultWidgetSizeTestCase()
```

Now, such test cases can be numerous, and their set-up can be repetitive. In the
above case, constructing a Widget in each of 100 Widget test case subclasses would mean unsightly duplication.

Luckily, we can factor out such set-up code by implementing a method called setUp(), which the testing framework will automatically call for us when we run the test:

```python
import unittest

class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

class DefaultWidgetSizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.failUnless(self.widget.size() == (50,50), 'incorrect default size')

class WidgetResizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.widget.resize(100,150)
        self.failUnless(self.widget.size() == (100,150), 'wrong size after resize')
```

If the setUp() method raises an exception while the test is running, the framework will consider the test to have suffered an error, and the runTest() method will not be executed.

Similarly, we can provide a tearDown() method that tidies up after the runTest() method has been run:

```python
import unittest

class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
        self.widget = None
```

If setUp() succeeded, the tearDown() method will be run whether runTest() succeeded or not.

Such a working environment for the testing code is called a fixture.
Often, many small test cases will use the same fixture. In this case, we would end up subclassing `SimpleWidgetTestCase` into many small one-method classes such as `DefaultWidgetSizeTestCase`. This is time-consuming and discouraging, so in the same vein as JUnit, `unittest` provides a simpler mechanism:

```python
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
        self.widget = None

    def testDefaultSize(self):
        self.failUnless(self.widget.size() == (50,50),
                        'incorrect default size')

    def testResize(self):
        self.widget.resize(100,150)
        self.failUnless(self.widget.size() == (100,150),
                        'wrong size after resize')
```

Here we have not provided a `runTest()` method, but have instead provided two different test methods. Class instances will now each run one of the `test*` methods, with `self.widget` created and destroyed separately for each instance. When creating an instance we must specify the test method it is to run. We do this by passing the method name in the constructor:

```python
defaultSizeTestCase = WidgetTestCase('testDefaultSize')
resizeTestCase = WidgetTestCase('testResize')
```

Test case instances are grouped together according to the features they test. `unittest` provides a mechanism for this: the `test suite`, represented by `unittest's TestSuite` class:

```python
widgetTestSuite = unittest.TestSuite()
widgetTestSuite.addTest(WidgetTestCase('testDefaultSize'))
widgetTestSuite.addTest(WidgetTestCase('testResize'))
```

For the ease of running tests, as we will see later, it is a good idea to provide in each test module a callable object that returns a pre-built test suite:
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('testDefaultSize'))
    suite.addTest(WidgetTestCase('testResize'))
    return suite

or even:

def suite():
    tests = ['testDefaultSize', 'testResize']
    return unittest.TestSuite(map(WidgetTestCase, tests))

Since it is a common pattern to create a TestCase subclass with many similarly named test functions, unittest provides a TestLoader class that can be used to automate the process of creating a test suite and populating it with individual tests. For example,

    suite = unittest.TestLoader().loadTestsFromTestCase(WidgetTestCase)

will create a test suite that will run
WidgetTestCase.testDefaultSize() and
WidgetTestCase.testResize. TestLoader uses the 'test' method name prefix to identify test methods automatically.

Note that the order in which the various test cases will be run is determined by sorting the test function names with the built-in cmp() function.

Often it is desirable to group suites of test cases together, so as to run tests for the whole system at once. This is easy, since TestSuite instances can be added to a TestSuite just as TestCase instances can be added to a TestSuite:

    suite1 = module1.TheTestSuite()
    suite2 = module2.TheTestSuite()
    alltests = unittest.TestSuite([suite1, suite2])

You can place the definitions of test cases and test suites in the same modules as the code they are to test (such as widget.py), but there are several advantages to placing the test code in a separate module, such as test_widget.py:

- The test module can be run standalone from the command line.
- The test code can more easily be separated from shipped code.
• There is less temptation to change test code to fit the code it tests without a good reason.
• Test code should be modified much less frequently than the code it tests.
• Tested code can be refactored more easily.
• Tests for modules written in C must be in separate modules anyway, so why not be consistent?
• If the testing strategy changes, there is no need to change the source code.
23.3.3 Re-using old test code

Some users will find that they have existing test code that they would like to run from unittest, without converting every old test function to a TestCase subclass.

For this reason, unittest provides a FunctionTestCase class. This subclass of TestCase can be used to wrap an existing test function. Set-up and tear-down functions can also be provided.

Given the following test function:

```python
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

one can create an equivalent test case instance as follows:

```python
testcase = unittest.FunctionTestCase(testSomething)
```

If there are additional set-up and tear-down methods that should be called as part of the test case's operation, they can also be provided like so:

```python
testcase = unittest.FunctionTestCase(testSomething,
    setUp=makeSomethingDB,
    tearDown=deleteSomethingDB)
```

To make migrating existing test suites easier, unittest supports tests raising AssertionError to indicate test failure. However, it is recommended that you use the explicit TestCase.fail*() and TestCase.assert*() methods instead, as future versions of unittest may treat AssertionError differently.

**Note:** Even though FunctionTestCase can be used to quickly convert an existing test base over to a unittest-based system, this approach is not recommended. Taking the time to set up proper TestCase subclasses will make future test refactoring infinitely easier.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
23.3.4 Classes and functions

class **TestCase** ([`methodName`])
Instances of the `TestCase` class represent the smallest testable units in the `unittest` universe. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the test, and methods that the test code can use to check for and report various kinds of failure.

Each instance of `TestCase` will run a single test method: the method named `methodName`. If you remember, we had an earlier example that went something like this:

```python
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('testDefaultSize'))
    suite.addTest(WidgetTestCase('testResize'))
    return suite
```

Here, we create two instances of `WidgetTestCase`, each of which runs a single test.

`methodName` defaults to 'runTest'.

class **FunctionTestCase** ([`testFunc`, `setUp`, `tearDown`, `description`])
This class implements the portion of the `TestCase` interface which allows the test runner to drive the test, but does not provide the methods which test code can use to check and report errors. This is used to create test cases using legacy test code, allowing it to be integrated into a `unittest`-based test framework.

class **TestSuite** ([`tests`])
This class represents an aggregation of individual tests cases and test suites. The class presents the interface needed by the test runner to allow it to be run as any other test case. Running a `TestSuite` instance is the same as iterating over the suite, running each test individually.
If tests is given, it must be an iterable of individual test cases or other test suites that will be used to build the suite initially. Additional methods are provided to add test cases and suites to the collection later on.

class TestLoader()
    This class is responsible for loading tests according to various criteria and returning them wrapped in a TestSuite. It can load all tests within a given module or TestCase subclass.

class TestResult()
    This class is used to compile information about which tests have succeeded and which have failed.

defaultTestLoader
    Instance of the TestLoader class intended to be shared. If no customization of the TestLoader is needed, this instance can be used instead of repeatedly creating new instances.

class TextTestRunner([stream[, descriptions[, verbosity]]]])
    A basic test runner implementation which prints results on standard error. It has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations.

main([module[, defaultTest[, argv[, testRunner[, testRunner]]]]]])
    A command-line program that runs a set of tests; this is primarily for making test modules conveniently executable. The simplest use for this function is to include the following line at the end of a test script:

        if __name__ == '__main__':
            unittest.main()

In some cases, the existing tests may have been written using the doctest module. If so, that module provides a DocTestSuite class that can automatically build unittest.TestSuite instances from the existing doctest-based tests. New in version 2.3.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
TestSuite Objects
23.3.5 Test Case Objects

Each `TestCase` instance represents a single test, but each concrete subclass may be used to define multiple tests -- the concrete class represents a single test fixture. The fixture is created and cleaned up for each test case.

`TestCase` instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

Methods in the first group (running the test) are:

`setUp()`  
Method called to prepare the test fixture. This is called immediately before calling the test method; any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

`tearDown()`  
Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception raised by this method will be considered an error rather than a test failure. This method will only be called if the `setUp()` succeeds, regardless of the outcome of the test method. The default implementation does nothing.

`run([result])`  
Run the test, collecting the result into the test result object passed as `result`. If `result` is omitted or `None`, a temporary result object is created (by calling the `defaultTestCase()` method) and used; this result object is not returned to `run()`'s caller.

The same effect may be had by simply calling the `TestCase` instance.

`debug()`
Run the test without collecting the result. This allows exceptions raised by the test to be propagated to the caller, and can be used to support running tests under a debugger.

The test code can use any of the following methods to check for and report failures.

```python
assert_(expr[, msg])
failUnless(expr[, msg])

Signal a test failure if `expr` is false; the explanation for the error will be `msg` if given, otherwise it will be `None`.

assertEqual(first, second[, msg])
failUnlessEqual(first, second[, msg])

Test that `first` and `second` are equal. If the values do not compare equal, the test will fail with the explanation given by `msg`, or `None`. Note that using `failUnlessEqual()` improves upon doing the comparison as the first parameter to `failUnless()`: the default value for `msg` can be computed to include representations of both `first` and `second`.

assertNotEqual(first, second[, msg])
failIfEqual(first, second[, msg])

Test that `first` and `second` are not equal. If the values do compare equal, the test will fail with the explanation given by `msg`, or `None`. Note that using `failIfEqual()` improves upon doing the comparison as the first parameter to `failUnless()` is that the default value for `msg` can be computed to include representations of both `first` and `second`.

assertAlmostEqual(first, second[, places[, msg]])
failUnlessAlmostEqual(first, second[, places[, msg]])

Test that `first` and `second` are approximately equal by computing the difference, rounding to the given number of `places`, and comparing to zero. Note that comparing a given number of decimal places is not the same as comparing a given number of significant digits. If the values do not compare equal, the test will fail with the explanation given by `msg`, or
None.

assertNotAlmostEqual(first, second[, places[, msg]])

failIfAlmostEqual(first, second[, places[, msg]])

Test that first and second are not approximately equal by computing the difference, rounding to the given number of places, and comparing to zero. Note that comparing a given number of decimal places is not the same as comparing a given number of significant digits. If the values do not compare equal, the test will fail with the explanation given by msg, or None.

assertRaises(exception, callable,...)

failUnlessRaises(exception, callable,...)

Test that an exception is raised when callable is called with any positional or keyword arguments that are also passed to assertRaises(). The test passes if exception is raised, is an error if another exception is raised, or fails if no exception is raised. To catch any of a group of exceptions, a tuple containing the exception classes may be passed as exception.

failIf(expr[, msg])

The inverse of the failUnless() method is the failIf() method. This signals a test failure if expr is true, with msg or None for the error message.

fail([msg])

Signals a test failure unconditionally, with msg or None for the error message.

failureException

This class attribute gives the exception raised by the test() method. If a test framework needs to use a specialized exception, possibly to carry additional information, it must subclass this exception in order to "play fair" with the framework. The initial value of this attribute is AssertionError.

Testing frameworks can use the following methods to collect information on the test:
**countTestCases()**

Return the number of tests represented by this test object. For `TestCase` instances, this will always be 1.

**defaultTestResult()**

Return an instance of the test result class that should be used for this test case class (if no other result instance is provided to the `run()` method).

For `TestCase` instances, this will always be an instance of `TestResult`; subclasses of `TestCase` should override this as necessary.

**id()**

Return a string identifying the specific test case. This is usually the full name of the test method, including the module and class name.

**shortDescription()**

Returns a one-line description of the test, or `None` if no description has been provided. The default implementation of this method returns the first line of the test method's docstring, if available, or `None`. 
23.3.6 TestSuite Objects

TestSuite objects behave much like testCase objects, except they do not actually implement a test. Instead, they are used to aggregate tests into groups of tests that should be run together. Some additional methods are available to add tests to TestSuite instances:

**addTest**(test)
Add a TestCase or TestSuite to the suite.

**addTests**(tests)
Add all the tests from an iterable of TestCase and TestSuite instances to this test suite.

This is equivalent to iterating over tests, calling addTest() for each element.

TestSuite shares the following methods withTestCase:

**run**(result)
Run the tests associated with this suite, collecting the result into the test result object passed as result. Note that unlike testCase.run(), TestSuite.run() requires the result object to be passed in.

**debug**()
Run the tests associated with this suite without collecting the result. This allows exceptions raised by the test to be propagated to the caller and can be used to support running tests under a debugger.

**countTestCases**()
Return the number of tests represented by this test object, including all individual tests and sub-suites.

In the typical usage of a TestSuite object, the run() method is invoked by a TestRunner rather than by the end-user test harness.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
23.3.7 TestResult Objects

A TestResult object stores the results of a set of tests. The TestCase and TestSuite classes ensure that results are properly recorded; test authors do not need to worry about recording the outcome of tests.

Testing frameworks built on top of unittest may want access to the TestResult object generated by running a set of tests for reporting purposes; a TestResult instance is returned by the TestRunner.run() method for this purpose.

TestResult instances have the following attributes that will be of interest when inspecting the results of running a set of tests:

**errors**
A list containing 2-tuples of TestCase instances and strings holding formatted tracebacks. Each tuple represents a test which raised an unexpected exception. Changed in version 2.2: Contains formatted tracebacks instead of sys.exc_info() results.

**failures**
A list containing 2-tuples of TestCase instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the TestCase.fail*() or TestCase.assert*() methods. Changed in version 2.2: Contains formatted tracebacks instead of sys.exc_info() results.

**testsRun**
The total number of tests run so far.

**wasSuccessful()**
Returns True if all tests run so far have passed, otherwise returns False.

**stop()**
This method can be called to signal that the set of tests being run should be aborted by setting the TestResult's shouldStop attribute to True. TestRunner objects should respect this flag and return without running
any additional tests.

For example, this feature is used by the TextTestRunner class to stop the test framework when the user signals an interrupt from the keyboard. Interactive tools which provide TestRunner implementations can use this in a similar manner.

The following methods of the TestResult class are used to maintain the internal data structures, and may be extended in subclasses to support additional reporting requirements. This is particularly useful in building tools which support interactive reporting while tests are being run.

startTest( test)
Called when the test case test is about to be run.

The default implementation simply increments the instance's testsRun counter.

stopTest( test)
Called after the test case test has been executed, regardless of the outcome.

The default implementation does nothing.

addError( test, err)
Called when the test case test raises an unexpected exception err is a tuple of the form returned by sys.exc_info(): (type, value, traceback).

The default implementation appends (test, err) to the instance's errors attribute.

addFailure( test, err)
Called when the test case test signals a failure. err is a tuple of the form returned by sys.exc_info(): (type, value, traceback).

The default implementation appends (test, err) to the instance's failures attribute.

addSuccess( test)
Called when the test case *test* succeeds.

The default implementation does nothing.
23.3.8 TestLoader Objects

The TestLoader class is used to create test suites from classes and modules. Normally, there is no need to create an instance of this class; the unittest module provides an instance that can be shared as unittest.defaultTestLoader. Using a subclass or instance, however, allows customization of some configurable properties.

TestLoader objects have the following methods:

**loadTestsFromTestCase(testCaseClass)**
Return a suite of all tests cases contained in the TestCase-derived testCaseClass.

**loadTestsFromModule(module)**
Return a suite of all tests cases contained in the given module. This method searches module for classes derived from TestCase and creates an instance of the class for each test method defined for the class.

**Warning:** While using a hierarchy of TestCase-derived classes can be convenient in sharing fixtures and helper functions, defining test methods on base classes that are not intended to be instantiated directly does not play well with this method. Doing so, however, can be useful when the fixtures are different and defined in subclasses.

**loadTestsFromName(name[, module])**
Return a suite of all tests cases given a string specifier.

The specifier name is a ``dotted name" that may resolve either to a module, a test case class, a test method within a test case class, a TestSuite instance, or a callable object which returns a TestCase or TestSuite instance. These checks are applied in the order listed here; that is, a method on a possible test case class will be picked up as ``a test method within a test case class", rather than ``a callable object".

For example, if you have a module SampleTests containing a
TestCase-derived class SampleTestCase with three test methods (test_one(), test_two(), and test_three()), the specifier 'SampleTests.SampleTestCase' would cause this method to return a suite which will run all three test methods. Using the specifier 'SampleTests.SampleTestCase.test_two' would cause it to return a test suite which will run only the test_two() test method. The specifier can refer to modules and packages which have not been imported; they will be imported as a side-effect.

The method optionally resolves name relative to the given module.

loadTestsFromNames(names[, module])
Similar to loadTestsFromName(), but takes a sequence of names rather than a single name. The return value is a test suite which supports all the tests defined for each name.

getTestCaseNames(testCaseClass)
Return a sorted sequence of method names found within testCaseClass; this should be a subclass of TestCase.

The following attributes of a TestLoader can be configured either by subclassing or assignment on an instance:

testMethodPrefix
String giving the prefix of method names which will be interpreted as test methods. The default value is 'test'.

This affects getTestCaseNames() and all the loadTestsFrom*() methods.

sortTestMethodsUsing
Function to be used to compare method names when sorting them in getTestCaseNames() and all the loadTestsFrom*() methods. The default value is the built-in cmp() function; the attribute can also be set to None to disable the sort.

suiteClass
Callable object that constructs a test suite from a list of tests. No methods
on the resulting object are needed. The default value is the TestSuite class.

This affects all the loadTestsFrom*() methods.
23.4 test -- Regression tests package for Python

The test package contains all regression tests for Python as well as the modules test.test_support and test.regrtest. test.test_support is used to enhance your tests while test.regrtest drives the testing suite.

Each module in the test package whose name starts with "test_" is a testing suite for a specific module or feature. All new tests should be written using the unittest module; using unittest is not required but makes the tests more flexible and maintenance of the tests easier. Some older tests are written to use doctest and a ``traditional'' testing style; these styles of tests will not be covered.

See Also:

Module [unittest](#): Writing PyUnit regression tests.

Module [doctest](#): Tests embedded in documentation strings.

Subsections

- 23.4.1 Writing Unit Tests for the test package
- 23.4.2 Running tests using test.regrtest

Python Library Reference

Previous: 23.3.8 TestLoader Objects Up: 23. Development Tools Next: 23.4.1 Writing Unit Tests
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
23.4.1 Writing Unit Tests for the test package

It is preferred that tests for the test package use the unittest module and follow a few guidelines. One is to name the test module by starting it with "test_" and end it with the name of the module being tested. The test methods in the test module should start with "test_" and end with a description of what the method is testing. This is needed so that the methods are recognized by the test driver as test methods. Also, no documentation string for the method should be included. A comment (such as "# Tests function returns only True or False") should be used to provide documentation for test methods. This is done because documentation strings get printed out if they exist and thus what test is being run is not stated.

A basic boilerplate is often used:

```python
import unittest
from test import test_support

class MyTestCase1(unittest.TestCase):
    # Only use setUp() and tearDown() if necessary
    def setUp(self):
        ... code to execute in preparation for tests ...
    def tearDown(self):
        ... code to execute to clean up after tests ...
    def test_feature_one(self):
        # Test feature one.
        ... testing code ...
    def test_feature_two(self):
        # Test feature two.
        ... testing code ...
    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...
    ... more test classes ...

def test_main():
    test_support.run_unittest(MyTestCase1,
```
This boilerplate code allows the testing suite to be run by `test.regrtest` as well as on its own as a script.

The goal for regression testing is to try to break code. This leads to a few guidelines to be followed:

- The testing suite should exercise all classes, functions, and constants. This includes not just the external API that is to be presented to the outside world but also "private" code.
- Whitebox testing (examining the code being tested when the tests are being written) is preferred. Blackbox testing (testing only the published user interface) is not complete enough to make sure all boundary and edge cases are tested.
- Make sure all possible values are tested including invalid ones. This makes sure that not only all valid values are acceptable but also that improper values are handled correctly.
- Exhaust as many code paths as possible. Test where branching occurs and thus tailor input to make sure as many different paths through the code are taken.
- Add an explicit test for any bugs discovered for the tested code. This will make sure that the error does not crop up again if the code is changed in the future.
- Make sure to clean up after your tests (such as close and remove all temporary files).
- If a test is dependent on a specific condition of the operating system then verify the condition already exists before attempting the test.
- Import as few modules as possible and do it as soon as possible. This minimizes external dependencies of tests and also minimizes possible anomalous behavior from side-effects of importing a module.
- Try to maximize code reuse. On occasion, tests will vary by something as small as what type of input is used. Minimize code duplication by subclassing a basic test class with a class that specifies the input:

```python
class TestFuncAcceptsSequences(unittest.TestCase):
```
func = mySuperWhammyFunction

def test_func(self):
    self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequences):
    arg = [1,2,3]

class AcceptStrings(TestFuncAcceptsSequences):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequences):
    arg = (1,2,3)

See Also:

*Test Driven Development*

A book by Kent Beck on writing tests before code.
23.4.2 Running tests using test.regrtest

test.regrtest can be used as a script to drive Python's regression test suite. Running the script by itself automatically starts running all regression tests in the test package. It does this by finding all modules in the package whose name starts with "test_", importing them, and executing the function test_main() if present. The names of tests to execute may also be passed to the script. Specifying a single regression test (python regrtest.py test_spam.py) will minimize output and only print whether the test passed or failed and thus minimize output.

Running test.regrtest directly allows what resources are available for tests to use to be set. You do this by using the -u command-line option. Run python regrtest.py -uall to turn on all resources; specifying all as an option for -u enables all possible resources. If all but one resource is desired (a more common case), a comma-separated list of resources that are not desired may be listed after all. The command python regrtest.py -uall,-audio,-largefile will run test.regrtest with all resources except the audio and largefile resources. For a list of all resources and more command-line options, run python regrtest.py -h.

Some other ways to execute the regression tests depend on what platform the tests are being executed on. On UNIX, you can run make test at the top-level directory where Python was built. On Windows, executing rt.bat from your PCBuild directory will run all regression tests.
23.5 test.test_support -- Utility functions for tests

The test.test_support module provides support for Python's regression tests.

This module defines the following exceptions:

**exception TestFailed**
Exception to be raised when a test fails.

**exception TestSkipped**
Subclass of TestFailed. Raised when a test is skipped. This occurs when a needed resource (such as a network connection) is not available at the time of testing.

**exception ResourceDenied**
Subclass of TestSkipped. Raised when a resource (such as a network connection) is not available. Raised by the requires() function.

The test.test_support module defines the following constants:

**verbose**
True when verbose output is enabled. Should be checked when more detailed information is desired about a running test. verbose is set by test.regrtest.

**have_unicode**
True when Unicode support is available.

**is_jython**
True if the running interpreter is Jython.

**TESTFN**
Set to the path that a temporary file may be created at. Any temporary that is created should be closed and unlinked (removed).
The `test.test_support` module defines the following functions:

**forget**( `module_name` )
Removes the module named `module_name` from `sys.modules` and deletes any byte-compiled files of the module.

**is_resource_enabled**( `resource` )
Returns `True` if `resource` is enabled and available. The list of available resources is only set when `test.regrtest` is executing the tests.

**requires**( `resource[, msg]` )
Raises `ResourceDenied` if `resource` is not available. `msg` is the argument to `ResourceDenied` if it is raised. Always returns true if called by a function whose `__name__` is '__main__'. Used when tests are executed by `test.regrtest`.

**findfile**( `filename` )
Return the path to the file named `filename`. If no match is found `filename` is returned. This does not equal a failure since it could be the path to the file.

**run_unittest**( `*classes` )
Execute `unittest.TestCase` subclasses passed to the function. The function scans the classes for methods starting with the prefix "test_" and executes the tests individually. This is the preferred way to execute tests.

**run_suite**( `suite[, testclass]` )
Execute the `unittest.TestSuite` instance `suite`. The optional argument `testclass` accepts one of the test classes in the suite so as to print out more detailed information on where the testing suite originated from.
24. The Python Debugger

The module `pdb` defines an interactive source code debugger for Python programs. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.

The debugger is extensible -- it is actually defined as the class `Pdb`. This is currently undocumented but easily understood by reading the source. The extension interface uses the modules `bdb` (undocumented) and `cmd`.

The debugger's prompt is "(Pdb) ". Typical usage to run a program under control of the debugger is:

```python
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

`pdb.py` can also be invoked as a script to debug other scripts. For example:

```bash
python -m pdb myscript.py
```

When invoked as a script, `pdb` will automatically enter post-mortem debugging if the program being debugged exits abnormally. After post-mortem debugging (or after normal exit of the program), `pdb` will restart the program. Automatic restarting preserves `pdb`'s state (such as breakpoints) and in most cases is more useful than quitting the debugger upon program's exit. New in version 2.4: Restarting post-mortem behavior added.

Typical usage to inspect a crashed program is:

```python
>>> import pdb
>>> import mymodule
```
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print spam
NameError: spam

>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print spam
(Pdb)

The module defines the following functions; each enters the debugger in a slightly different way:

run( statement[, globals[, locals]])

Execute the statement (given as a string) under debugger control. The debugger prompt appears before any code is executed; you can set breakpoints and type "continue", or you can step through the statement using "step" or "next" (all these commands are explained below). The optional globals and locals arguments specify the environment in which the code is executed; by default the dictionary of the module __main__ is used. (See the explanation of the exec statement or the eval() built-in function.)

runeval( expression[, globals[, locals]])

Evaluate the expression (given as a string) under debugger control. When runeval() returns, it returns the value of the expression. Otherwise this function is similar to run().

runcall( function[, argument, ...])

Call the function (a function or method object, not a string) with the given arguments. When runcall() returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

set_trace()

Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails).
**post_mortem**(*traceback*)

Enter post-mortem debugging of the given *traceback* object.

**pm()**
Enter post-mortem debugging of the traceback found in
`sys.last_traceback`.

---

**Subsections**

- [24.1 Debugger Commands](#)
- [24.2 How It Works](#)

---

*Release 2.5, documentation updated on 19th September, 2006.*
See [About this document...](#) for information on suggesting changes.
24.1 Debugger Commands

The debugger recognizes the following commands. Most commands can be abbreviated to one or two letters; e.g. "h(elp)" means that either "h" or "help" can be used to enter the help command (but not "he" or "hel", nor "H" or "Help" or "HELP"). Arguments to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets ([ ]") in the command syntax; the square brackets must not be typed. Alternatives in the command syntax are separated by a vertical bar ("| ").

Entering a blank line repeats the last command entered. Exception: if the last command was a "list" command, the next 11 lines are listed.

Commands that the debugger doesn't recognize are assumed to be Python statements and are executed in the context of the program being debugged. Python statements can also be prefixed with an exclamation point ("! "). This is a powerful way to inspect the program being debugged; it is even possible to change a variable or call a function. When an exception occurs in such a statement, the exception name is printed but the debugger's state is not changed.

Multiple commands may be entered on a single line, separated by ";;". (A single ";" is not used as it is the separator for multiple commands in a line that is passed to the Python parser.) No intelligence is applied to separating the commands; the input is split at the first ";;" pair, even if it is in the middle of a quoted string.

The debugger supports aliases. Aliases can have parameters which allows one a certain level of adaptability to the context under examination.

If a file .pdbrc exists in the user's home directory or in the current directory, it is read in and executed as if it had been typed at the debugger prompt. This is particularly useful for aliases. If both files exist, the one in the home directory is read first and aliases defined there can be overridden by the local file.

h(elp) [command]

Without argument, print the list of available commands. With a command as
argument, print help about that command. "help pdb" displays the full documentation file; if the environment variable PAGER is defined, the file is piped through that command instead. Since the command argument must be an identifier, "help exec" must be entered to get help on the "!" command.

w(here)

Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.

d(own)

Move the current frame one level down in the stack trace (to a newer frame).

u(p)

Move the current frame one level up in the stack trace (to an older frame).

b(reak) [[filename:]lineno | function[, condition]]

With a lineno argument, set a break there in the current file. With a function argument, set a break at the first executable statement within that function. The line number may be prefixed with a filename and a colon, to specify a breakpoint in another file (probably one that hasn't been loaded yet). The file is searched on sys.path. Note that each breakpoint is assigned a number to which all the other breakpoint commands refer.

If a second argument is present, it is an expression which must evaluate to true before the breakpoint is honored.

Without argument, list all breaks, including for each breakpoint, the number of times that breakpoint has been hit, the current ignore count, and the associated condition if any.

tbreak [[filename:]lineno | function[, condition]]

Temporary breakpoint, which is removed automatically when it is first hit.
The arguments are the same as break.

\texttt{clear} \hspace{1em} \texttt{[bpnumber \{bpnumber \ldots\}]}

With a space separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks (but first ask confirmation).

\texttt{disable} \hspace{1em} \texttt{[bpnumber \{bpnumber \ldots\}]}

Disables the breakpoints given as a space separated list of breakpoint numbers. Disabling a breakpoint means it cannot cause the program to stop execution, but unlike clearing a breakpoint, it remains in the list of breakpoints and can be (re-)enabled.

\texttt{enable} \hspace{1em} \texttt{[bpnumber \{bpnumber \ldots\}]}

Enables the breakpoints specified.

\texttt{ignore} \hspace{1em} \texttt{bpnumber \{count\}}

Sets the ignore count for the given breakpoint number. If count is omitted, the ignore count is set to 0. A breakpoint becomes active when the ignore count is zero. When non-zero, the count is decremented each time the breakpoint is reached and the breakpoint is not disabled and any associated condition evaluates to true.

\texttt{condition} \hspace{1em} \texttt{bpnumber \{condition\}}

Condition is an expression which must evaluate to true before the breakpoint is honored. If condition is absent, any existing condition is removed; i.e., the breakpoint is made unconditional.

\texttt{commands} \hspace{1em} \texttt{[bpnumber]}

Specify a list of commands for breakpoint number \texttt{bpnumber}. The commands themselves appear on the following lines. Type a line containing just 'end' to terminate the commands. An example:

\begin{verbatim}
(Pdb) commands 1
(com) print some_variable
\end{verbatim}
To remove all commands from a breakpoint, type commands and follow it immediately with end; that is, give no commands.

With no bpnumber argument, commands refers to the last breakpoint set.

You can use breakpoint commands to start your program up again. Simply use the continue command, or step, or any other command that resumes execution.

Specifying any command resuming execution (currently continue, step, next, return, jump, quit and their abbreviations) terminates the command list (as if that command was immediately followed by end). This is because any time you resume execution (even with a simple next or step), you may encounter another breakpoint-which could have its own command list, leading to ambiguities about which list to execute.

If you use the 'silent' command in the command list, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the other commands print anything, you see no sign that the breakpoint was reached.

New in version 2.5.

s(tep)

Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).

n(ext)

Continue execution until the next line in the current function is reached or it returns. (The difference between "next" and "step" is that "step" stops inside a called function, while "next" executes called functions at (nearly) full speed, only stopping at the next line in the current function.)

r(et)
Continue execution until the current function returns.

c(ont(inue))

Continue execution, only stop when a breakpoint is encountered.

j(ump) lineno

Set the next line that will be executed. Only available in the bottom-most frame. This lets you jump back and execute code again, or jump forward to skip code that you don't want to run.

It should be noted that not all jumps are allowed -- for instance it is not possible to jump into the middle of a for loop or out of a finally clause.

l(ist) [first[, last]]

List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With one argument, list 11 lines around at that line. With two arguments, list the given range; if the second argument is less than the first, it is interpreted as a count.

a(rgs)

Print the argument list of the current function.

p expression

Evaluate the expression in the current context and print its value. Note: "print" can also be used, but is not a debugger command -- this executes the Python print statement.

pp expression

Like the "p" command, except the value of the expression is pretty-printed using the pprint module.

alias [name [command]]
Creates an alias called *name* that executes *command*. The command must *not* be enclosed in quotes. Replaceable parameters can be indicated by "%1", "%2", and so on, while "%*" is replaced by all the parameters. If no command is given, the current alias for *name* is shown. If no arguments are given, all aliases are listed.

Aliases may be nested and can contain anything that can be legally typed at the pdb prompt. Note that internal pdb commands *can* be overridden by aliases. Such a command is then hidden until the alias is removed. Aliasing is recursively applied to the first word of the command line; all other words in the line are left alone.

As an example, here are two useful aliases (especially when placed in the .pdbrc file):

```
#Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print "%1.",k,="",%1.__dict__[k]
#Print instance variables in self
alias ps pi self
```

**unalias name**

Deletes the specified alias.

**[!]statement**

Execute the (one-line) *statement* in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command. To set a global variable, you can prefix the assignment command with a "global" command on the same line, e.g.:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

**q(uit)**

Quit from the debugger. The program being executed is aborted.
How It Works

Release 2.5, documentation updated on 19th September, 2006.  
See About this document... for information on suggesting changes.
24.2 How It Works

Some changes were made to the interpreter:

- `sys.settrace(func)` sets the global trace function
- there can also be a local trace function (see later)

Trace functions have three arguments: `frame`, `event`, and `arg`. `frame` is the current stack frame. `event` is a string: 'call', 'line', 'return', 'exception', 'c_call', 'c_return', or 'c_exception'. `arg` depends on the event type.

The global trace function is invoked (with `event` set to 'call') whenever a new local scope is entered; it should return a reference to the local trace function to be used that scope, or `None` if the scope shouldn't be traced.

The local trace function should return a reference to itself (or to another function for further tracing in that scope), or `None` to turn off tracing in that scope.

Instance methods are accepted (and very useful!) as trace functions.

The events have the following meaning:

'call'
A function is called (or some other code block entered). The global trace function is called; `arg` is `None`; the return value specifies the local trace function.

'line'
The interpreter is about to execute a new line of code (sometimes multiple line events on one line exist). The local trace function is called; `arg` is `None`; the return value specifies the new local trace function.

'return'
A function (or other code block) is about to return. The local trace function is called; `arg` is the value that will be returned. The trace function's return value is ignored.
'exception'
An exception has occurred. The local trace function is called; arg is a triple (exception, value, traceback); the return value specifies the new local trace function.

'c_call'
A C function is about to be called. This may be an extension function or a builtin. arg is the C function object.

'c_return'
A C function has returned. arg is None.

'c_exception'
A C function has thrown an exception. arg is None.

Note that as an exception is propagated down the chain of callers, an 'exception' event is generated at each level.

For more information on code and frame objects, refer to the Python Reference Manual.
25. The Python Profilers

Copyright © 1994, by InfoSeek Corporation, all rights reserved.

Written by James Roskind.\(^{25.1}\)

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose (subject to the restriction in the following sentence) without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of InfoSeek not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. This permission is explicitly restricted to the copying and modification of the software to remain in Python, compiled Python, or other languages (such as C) wherein the modified or derived code is exclusively imported into a Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

The profiler was written after only programming in Python for 3 weeks. As a result, it is probably clumsy code, but I don't know for sure yet 'cause I'm a beginner :-). I did work hard to make the code run fast, so that profiling would be a reasonable thing to do. I tried not to repeat code fragments, but I'm sure I did some stuff in really awkward ways at times. Please send suggestions for improvements to: jar@netscape.com. I won't promise any support. ...but I'd appreciate the feedback.
Footnotes

... Roskind. 

Updated and converted to LaTeX by Guido van Rossum. Further updated by Armin Rigo to integrate the documentation for the new cProfile module of Python 2.5.

Subsections

- **25.1 Introduction to the profilers**
- **25.2 Instant User's Manual**
- **25.3 What Is Deterministic Profiling?**
- **25.4 Reference Manual - profile and cProfile**
  - **25.4.1 The Stats Class**
- **25.5 Limitations**
- **25.6 Calibration**
- **25.7 Extensions -- Deriving Better Profilers**
- **25.8 hotshot -- High performance logging profiler**
  - **25.8.1 Profile Objects**
  - **25.8.2 Using hotshot data**
  - **25.8.3 Example Usage**
- **25.9 timeit -- Measure execution time of small code snippets**
  - **25.9.1 Command Line Interface**
  - **25.9.2 Examples**
- **25.10 trace -- Trace or track Python statement execution**
  - **25.10.1 Command Line Usage**
  - **25.10.2 Programming Interface**

Previous: **24.2 How It Works**  Up: **Python Library Reference**  Next: **25.1 Introduction to the**
25.1 Introduction to the profilers

A profiler is a program that describes the run time performance of a program, providing a variety of statistics. This documentation describes the profiler functionality provided in the modules `profile` and `pstats`. This profiler provides deterministic profiling of any Python programs. It also provides a series of report generation tools to allow users to rapidly examine the results of a profile operation.

The Python standard library provides three different profilers:

1. `profile`, a pure Python module, described in the sequel. Copyright © 1994, by InfoSeek Corporation. Changed in version 2.4: also reports the time spent in calls to built-in functions and methods.

2. `cProfile`, a module written in C, with a reasonable overhead that makes it suitable for profiling long-running programs. Based on `lsprof`, contributed by Brett Rosen and Ted Czotter. New in version 2.5.

3. `hotshot`, a C module focusing on minimizing the overhead while profiling, at the expense of long data post-processing times. Changed in version 2.5: the results should be more meaningful than in the past: the timing core contained a critical bug.

The `profile` and `cProfile` modules export the same interface, so they are mostly interchangeable; `cProfile` has a much lower overhead but is not so far as well-tested and might not be available on all systems. `cProfile` is really a compatibility layer on top of the internal `_lsprof` module. The `hotshot` module is reserved to specialized usages.
25.2 Instant User's Manual

This section is provided for users that "don't want to read the manual." It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile an application with a main entry point of `foo()`, you would add the following to your module:

```python
import cProfile
cProfile.run('foo()')
```

(Use `profile` instead of `cProfile` if the latter is not available on your system.)

The above action would cause `foo()` to be run, and a series of informative lines (the profile) to be printed. The above approach is most useful when working with the interpreter. If you would like to save the results of a profile into a file for later examination, you can supply a file name as the second argument to the `run()` function:

```python
import cProfile
cProfile.run('foo()', 'fooprof')
```

The file `cProfile.py` can also be invoked as a script to profile another script. For example:

```bash
python -m cProfile myscript.py
```

cProfile.py accepts two optional arguments on the command line:

```
cProfile.py [-o output_file] [-s sort_order]
```

-s only applies to standard output (-o is not supplied). Look in the Stats documentation for valid sort values.

When you wish to review the profile, you should use the methods in the `pstats` module. Typically you would load the statistics data as follows:

```python
import pstats
```
\[ p = \text{pstats.Stats('fooprof')} \]

The class `Stats` (the above code just created an instance of this class) has a variety of methods for manipulating and printing the data that was just read into `p`. When you ran `cProfile.run()` above, what was printed was the result of three method calls:

\[ p\.\text{strip_dirs()}.\text{sort_stats(-1)}.\text{print_stats()} \]

The first method removed the extraneous path from all the module names. The second method sorted all the entries according to the standard module/line/name string that is printed. The third method printed out all the statistics. You might try the following sort calls:

\[ p\.\text{sort_stats('name')} \]
\[ p\.\text{print_stats()} \]

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

\[ p\.\text{sort_stats('cumulative')}\.\text{print_stats(10)} \]

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

\[ p\.\text{sort_stats('time')}\.\text{print_stats(10)} \]

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

\[ p\.\text{sort_stats('file')}\.\text{print_stats('__init__')} \]

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with `__init__` in them). As one final example, you could try:
p.sort_stats('time', 'cum').print_stats(.5, 'init')

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: ".5") of its original size, then only lines containing init are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (p is still sorted according to the last criteria) do:

    p.print_callers(.5, 'init')

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following functions do:

    p.print_callees()
    p.add('fooprof')

Invoked as a script, the pstats module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using cmd) and interactive help.
25.3 What Is Deterministic Profiling?

Deterministic profiling is meant to reflect the fact that all function call, function return, and exception events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, statistical profiling (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a hook (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify "hot loops" that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.
25.4 Reference Manual - profile and cProfile

The primary entry point for the profiler is the global function `profile.run()` (resp. `cProfile.run()`). It is typically used to create any profile information. The reports are formatted and printed using methods of the class `pstats.Stats`. The following is a description of all of these standard entry points and functions. For a more in-depth view of some of the code, consider reading the later section on Profiler Extensions, which includes discussion of how to derive ``better'' profilers from the classes presented, or reading the source code for these modules.

```
run(command[, filename])
```

This function takes a single argument that has can be passed to the `exec` statement, and an optional file name. In all cases this routine attempts to `exec` its first argument, and gather profiling statistics from the execution. If no file name is present, then this function automatically prints a simple profiling report, sorted by the standard name string (file/line/function-name) that is presented in each line. The following is a typical output from such a call:

```
2706 function calls (2004 primitive calls) in 4.504 CPU sec
Ordered by: standard name

ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
2       0.006    0.003   0.953    0.477  pobject.py:75(save_objects)
43/3    0.533    0.012   0.749    0.250  pobject.py:99(evaluate)
...
```

The first line indicates that 2706 calls were monitored. Of those calls, 2004 were primitive. We define primitive to mean that the call was not induced via recursion. The next line: `Ordered by: standard name`, indicates that the text string in the far right column was used to sort the output. The column headings include:
ncalls
   for the number of calls,

tottime
   for the total time spent in the given function (and excluding time made
   in calls to sub-functions),

percall
   is the quotient of tottime divided by ncalls

cumtime
   is the total time spent in this and all subfunctions (from invocation till
   exit). This figure is accurate even for recursive functions.

percall
   is the quotient of cumtime divided by primitive calls

filename:lineno(function)
   provides the respective data of each function

   When there are two numbers in the first column (for example, "43/3"),
   then the latter is the number of primitive calls, and the former is the actual
   number of calls. Note that when the function does not recurse, these two
   values are the same, and only the single figure is printed.

runctx( command, globals, locals[, filename])
   This function is similar to run(), with added arguments to supply the
   globals and locals dictionaries for the command string.

Analysis of the profiler data is done using the Stats class.

Note: The Stats class is defined in the pstats module.

class Stats( filename[, stream=sys.stdout[, ...]])
   This class constructor creates an instance of a "statistics object" from a
   filename (or set of filenames). Stats objects are manipulated by methods,
   in order to print useful reports. You may specify an alternate output stream
   by giving the keyword argument, stream.
The file selected by the above constructor must have been created by the corresponding version of profile or cProfile. To be specific, there is no file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers.

If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing Stats object, the add() method can be used.

Changed in version 2.5: The stream parameter was added.
25.4.1 The Stats Class

Stats objects have the following methods:

**strip_dirs()**
This method for the Stats class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a ``random'' order, as it was just after object initialization and loading. If `strip_dirs()` causes two function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

**add(filename[, ...])**
This method of the Stats class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of `profile.run()` or `cProfile.run()`. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

**dump_stats(filename)**
Save the data loaded into the Stats object to a file named `filename`. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the `profile.Profile` and `cProfile.Profile` classes. New in version 2.3.

**sort_stats(key[, ...])**
This method modifies the Stats object by sorting it according to the supplied criteria. The argument is typically a string identifying the basis of a sort (example: 'time' or 'name').

When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, `sort_stats('name', 'file')` will sort all the entries
according to their function name, and resolve all ties (identical function names) by sorting by file name.

Abbreviations can be used for any key names, as long as the abbreviation is unambiguous. The following are the keys currently defined:

<table>
<thead>
<tr>
<th>Valid Arg</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'calls'</td>
<td>call count</td>
</tr>
<tr>
<td>'cumulative'</td>
<td>cumulative time</td>
</tr>
<tr>
<td>'file'</td>
<td>file name</td>
</tr>
<tr>
<td>'module'</td>
<td>file name</td>
</tr>
<tr>
<td>'pcalls'</td>
<td>primitive call count</td>
</tr>
<tr>
<td>'line'</td>
<td>line number</td>
</tr>
<tr>
<td>'name'</td>
<td>function name</td>
</tr>
<tr>
<td>'nfl'</td>
<td>name/file/line</td>
</tr>
<tr>
<td>'stdname'</td>
<td>standard name</td>
</tr>
<tr>
<td>'time'</td>
<td>internal time</td>
</tr>
</tbody>
</table>

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between 'nfl' and 'stdname' is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, 'nfl' does a numeric compare of the line numbers. In fact, sort_stats('nfl') is the same as sort_stats('name', 'file', 'line').

For backward-compatibility reasons, the numeric arguments -1, 0, 1, and 2 are permitted. They are interpreted as 'stdname', 'calls', 'time', and 'cumulative' respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

reverse_order()

This method for the Stats class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is
properly selected based on the sort key of choice.

**print_stats([restriction, ...])**

This method for the Stats class prints out a report as described in the `profile.run()` definition.

The order of the printing is based on the last `sort_stats()` operation done on the object (subject to caveats in `add()` and `strip_dirs()`).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a regular expression (to pattern match the standard name that is printed; as of Python 1.5b1, this uses the Perl-style regular expression syntax defined by the `re` module). If several restrictions are provided, then they are applied sequentially. For example:

```
  print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename .*foo:. In contrast, the command:

```
  print_stats('foo:', .1)
```

would limit the list to all functions having file names .*foo:, and then proceed to only print the first 10% of them.

**print_callers([restriction, ...])**

This method for the Stats class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by `print_stats()`, and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats:

- With `profile`, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.
• With cProfile, each caller is preceded by three numbers: the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

**print_callees([restriction, ...])**

This method for the Stats class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the print_callers() method.
25.5 Limitations

One limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying ``clock'' is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate than the underlying clock. If enough measurements are taken, then the ```error'' will tend to average out. Unfortunately, removing this first error induces a second source of error.

The second problem is that it ```takes a while'' from when an event is dispatched until the profiler's call to get the time actually gets the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock's value was obtained (and then squirreled away), until the user's code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (less than one clock tick), but it can accumulate and become very significant.

The problem is more important with profile than with the lower-overhead cProfile. For this reason, profile provides a means of calibrating itself for a given platform so that this error can be probabilistically (on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-). ) Do not be alarmed by negative numbers in the profile. They should only appear if you have calibrated your profiler, and the results are actually better than without calibration.
25.6 Calibration

The profiler of the profile module subtracts a constant from each event handling time to compensate for the overhead of calling the time function, and socking away the results. By default, the constant is 0. The following procedure can be used to obtain a better constant for a given platform (see discussion in section Limitations above).

```python
import profile
pr = profile.Profile()
for i in range(5):
    print pr.calibrate(10000)
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on an 800 MHz Pentium running Windows 2000, and using Python's time.clock() as the timer, the magical number is about 12.5e-6.

The object of this exercise is to get a fairly consistent result. If your computer is very fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

When you have a consistent answer, there are three ways you can use it:

```python
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will "less often" show up as negative in profile statistics.
Prior to Python 2.2, it was necessary to edit the profiler source code to embed the bias as a literal number. You still can, but that method is no longer described, because no longer needed.
25.7 Extensions -- Deriving Better Profilers

The Profile class of both modules, profile and cProfile, were written so that derived classes could be developed to extend the profiler. The details are not described here, as doing this successfully requires an expert understanding of how the Profile class works internally. Study the source code of the module carefully if you want to pursue this.

If all you want to do is change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the Profile class constructor:

```python
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func()`.

**profile.Profile**

`your_time_func()` should return a single number, or a list of numbers whose sum is the current time (like what `os.times()` returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose. For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (`os.times()` is pretty bad, as it returns a tuple of floating point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

**cProfile.Profile**

`your_time_func()` should return a single number. If it returns plain integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For example, if `your_integer_time_func()` returns times measured in thousands of
seconds, you would construct the `Profile` instance as follows:

```python
pr = profile.Profile(your_integer_time_func, 0.001)
```

As the `cProfile.Profile` class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal `_lsprof` module.
25.8.1 Profile Objects
25.8 hotshot -- High performance logging profiler

New in version 2.2.

This module provides a nicer interface to the _hotshot C module. Hotshot is a replacement for the existing profile module. As it's written mostly in C, it should result in a much smaller performance impact than the existing profile module.

**Note:** The hotshot module focuses on minimizing the overhead while profiling, at the expense of long data post-processing times. For common usages it is recommended to use cProfile instead. hotshot is not maintained and might be removed from the standard library in the future.

Changed in version 2.5: the results should be more meaningful than in the past: the timing core contained a critical bug.

**Warning:** The hotshot profiler does not yet work well with threads. It is useful to use an unthreaded script to run the profiler over the code you're interested in measuring if at all possible.

```python
class Profile( logfile[, lineevents[, linetimings]])
```

The profiler object. The argument `logfile` is the name of a log file to use for logged profile data. The argument `lineevents` specifies whether to generate events for every source line, or just on function call/return. It defaults to 0 (only log function call/return). The argument `linetimings` specifies whether to record timing information. It defaults to 1 (store timing information).

**Subsections**
• 25.8.1 Profile Objects
• 25.8.2 Using hotshot data
• 25.8.3 Example Usage

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
25.8.1 Profile Objects

Profile objects have the following methods:

\textbf{addinfo}(key, value)

Add an arbitrary labelled value to the profile output.

\textbf{close}()

Close the logfile and terminate the profiler.

\textbf{fileno}()

Return the file descriptor of the profiler's log file.

\textbf{run}(cmd)

Profile an \texttt{exec}-compatible string in the script environment. The globals from the \texttt{\_main\_} module are used as both the globals and locals for the script.

\textbf{runcall}(func, *args, **keywords)

Profile a single call of a callable. Additional positional and keyword arguments may be passed along; the result of the call is returned, and exceptions are allowed to propagate cleanly, while ensuring that profiling is disabled on the way out.

\textbf{runctx}(cmd, globals, locals)

Evaluate an \texttt{exec}-compatible string in a specific environment. The string is compiled before profiling begins.

\textbf{start}()

Start the profiler.

\textbf{stop}()

Stop the profiler.
25.8.2 Using hotshot data

New in version 2.2.

This module loads hotshot profiling data into the standard \texttt{pstats} Stats objects.

\texttt{load( filename)}

Load hotshot data from \texttt{filename}. Returns an instance of the \texttt{pstats.Stats} class.

\textbf{See Also:}

Module \texttt{profile}:
The profile module's \texttt{Stats} class.
25.8.3 Example Usage

Note that this example runs the python `"benchmark"` pystones. It can take some time to run, and will produce large output files.

>>> import hotshot, hotshot.stats, test.pystone
>>> prof = hotshot.Profile("stones.prof")
>>> benchtime, stones = prof.runcall(test.pystone.pystones)
>>> prof.close()
>>> stats = hotshot.stats.load("stones.prof")
>>> stats.strip_dirs()
>>> stats.sort_stats('time', 'calls')
>>> stats.print_stats(20)

850004 function calls in 10.090 CPU seconds

Ordered by: internal time, call count

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1      3.295    3.295    10.090   10.090 pystone.py:79(Proc0)
150000  1.315    0.000    1.315    0.000 pystone.py:203(Proc7)
50000   1.313    0.000    1.463    0.000 pystone.py:229(Func2)
```

...
25.9 timeit -- Measure execution time of small code snippets

New in version 2.3.

This module provides a simple way to time small bits of Python code. It has both command line as well as callable interfaces. It avoids a number of common traps for measuring execution times. See also Tim Peters' introduction to the "Algorithms" chapter in the Python Cookbook, published by O'Reilly.

The module defines the following public class:

class Timer([stmt='pass' [, setup='pass' [, timer=<timer function>]]])

Class for timing execution speed of small code snippets.

The constructor takes a statement to be timed, an additional statement used for setup, and a timer function. Both statements default to 'pass'; the timer function is platform-dependent (see the module doc string). The statements may contain newlines, as long as they don't contain multi-line string literals.

To measure the execution time of the first statement, use the timeit() method. The repeat() method is a convenience to call timeit() multiple times and return a list of results.

print_exc([file=None])

Helper to print a traceback from the timed code.

Typical use:

t = Timer(...)  # outside the try/except
try:
    t.timeit(...)  # or t.repeat(...)
except:
    t.print_exc()
The advantage over the standard traceback is that source lines in the compiled template will be displayed. The optional file argument directs where the traceback is sent; it defaults to `sys.stderr`.

```
repeat([repeat=3 [ number=1000000]])
```

Call `timeit()` a few times.

This is a convenience function that calls the `timeit()` repeatedly, returning a list of results. The first argument specifies how many times to call `timeit()`. The second argument specifies the `number` argument for `timeit()`.

---

**Note:** It's tempting to calculate mean and standard deviation from the result vector and report these. However, this is not very useful. In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python's speed, but by other processes interfering with your timing accuracy. So the `min()` of the result is probably the only number you should be interested in. After that, you should look at the entire vector and apply common sense rather than statistics.

```
timeit([number=1000000])
```

Time `number` executions of the main statement. This executes the setup statement once, and then returns the time it takes to execute the main statement a number of times, measured in seconds as a float. The argument is the number of times through the loop, defaulting to one million. The main statement, the setup statement and the timer function to be used are passed to the constructor.

---

**Note:** By default, `timeit()` temporarily turns off garbage collection during the timing. The advantage of this approach is that it makes independent timings more comparable. This disadvantage is that GC may be an
important component of the performance of the function being measured. If so, GC can be re-enabled as the first statement in the setup string. For example:

```python
timeit.Timer('for i in xrange(10): oct(i)', 'gc.enable()').timeit()
```
25.9.1 Command Line Interface

When called as a program from the command line, the following form is used:

```
python timeit.py [-n N] [-r N] [-s S] [-t] [-c] [-h] [statement ...]
```

where the following options are understood:

- **-n N**/ **--number=N**
  how many times to execute 'statement'

- **-r N**/ **--repeat=N**
  how many times to repeat the timer (default 3)

- **-s S**/ **--setup=S**
  statement to be executed once initially (default 'pass')

- **-t**/ **--time**
  use `time.time()` (default on all platforms but Windows)

- **-c**/ **--clock**
  use `time.clock()` (default on Windows)

- **-v**/ **--verbose**
  print raw timing results; repeat for more digits precision

- **-h**/ **--help**
  print a short usage message and exit

A multi-line statement may be given by specifying each line as a separate statement argument; indented lines are possible by enclosing an argument in quotes and using leading spaces. Multiple -s options are treated similarly.

If -n is not given, a suitable number of loops is calculated by trying successive powers of 10 until the total time is at least 0.2 seconds.

The default timer function is platform dependent. On Windows, `time.clock()` has microsecond granularity but `time.time()`'s granularity is 1/60th of a second; on UNIX, `time.clock()` has 1/100th of a second granularity and `time.time()` is much more precise. On either platform, the default timer functions measure wall clock time, not the CPU time. This means that other processes running on the same computer may interfere with the timing. The best thing to do when accurate timing is necessary is to repeat the timing a few times and use the best time. The -r option is good for this; the
default of 3 repetitions is probably enough in most cases. On UNIX, you can use \texttt{time.clock()} to measure CPU time.

\begin{quote}
\textbf{Note:} There is a certain baseline overhead associated with executing a pass statement. The code here doesn't try to hide it, but you should be aware of it. The baseline overhead can be measured by invoking the program without arguments.
\end{quote}

The baseline overhead differs between Python versions! Also, to fairly compare older Python versions to Python 2.3, you may want to use Python's \texttt{-O} option for the older versions to avoid timing \texttt{SET\_LINENO} instructions.
25.9.2 Examples

Here are two example sessions (one using the command line, one using the module interface) that compare the cost of using `hasattr()` vs. `try/except` to test for missing and present object attributes.

```python
>>> import timeit
>>> s = """
... try:
...     str.__nonzero__
... except AttributeError:
...     pass
... """
>>> t = timeit.Timer(stmt=s)
>>> print "%.2f usec/pass" % (1000000 * t.timeit(number=100000)/1000)
17.09 usec/pass
>>> s = """
... if hasattr(str, '__nonzero__'):
...     pass
... """
>>> t = timeit.Timer(stmt=s)
>>> print "%.2f usec/pass" % (1000000 * t.timeit(number=100000)/1000)
4.85 usec/pass
>>> s = """
... try:
...     int.__nonzero__
... except AttributeError:
...     pass
... """
>>> t = timeit.Timer(stmt=s)
>>> print "%.2f usec/pass" % (1000000 * t.timeit(number=100000)/1000)
1.97 usec/pass
>>> s = """
... if hasattr(int, '__nonzero__'):
...     pass
... """
>>> t = timeit.Timer(stmt=s)
>>> print "%.2f usec/pass" % (1000000 * t.timeit(number=100000)/1000)
3.15 usec/pass
```

To give the `timeit` module access to functions you define, you can pass a
setup parameter which contains an import statement:

```python
def test():
    "Stupid test function"
    L = []
    for i in range(100):
        L.append(i)

if __name__=='__main__':
    from timeit import Timer
    t = Timer("test()", "from __main__ import test")
    print t.timeit()
```

Python Library Reference

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
25.10 trace -- Trace or track Python statement execution

The trace module allows you to trace program execution, generate annotated statement coverage listings, print caller/callee relationships and list functions executed during a program run. It can be used in another program or from the command line.

Subsections

- 25.10.1 Command Line Usage
- 25.10.2 Programming Interface

Previous: 25.9.2 Examples  Up: 25. The Python Profilers  Next: 25.10.1 Command Line Usage

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
25.10.1 Command Line Usage

The `trace` module can be invoked from the command line. It can be as simple as

```
python -m trace --count somefile.py ...
```

The above will generate annotated listings of all Python modules imported during the execution of `somefile.py`.

The following command-line arguments are supported:

---trace, -t
Display lines as they are executed.

--count, -c
Produce a set of annotated listing files upon program completion that shows how many times each statement was executed.

--report, -r
Produce an annotated list from an earlier program run that used the --count and --file arguments.

--no-report, -R
Do not generate annotated listings. This is useful if you intend to make several runs with --count then produce a single set of annotated listings at the end.

--listfuncs, -l
List the functions executed by running the program.

--trackcalls, -T
Generate calling relationships exposed by running the program.

--file, -f
Name a file containing (or to contain) counts.

--coverdir, -C
Name a directory in which to save annotated listing files.
--missing, -m
When generating annotated listings, mark lines which were not executed with `>>>>>>>'.

--summary, -s
When using --count or --report, write a brief summary to stdout for each file processed.

--ignore-module
Ignore the named module and its submodules (if it is a package). May be given multiple times.

--ignore-dir
Ignore all modules and packages in the named directory and subdirectories. May be given multiple times.
25.10.2 Programming Interface

```
[count=1, trace=1, countfuncs=0, countcallers=0,

class Trace( ignoremods=(), ignoredirs=(), infile=None,
            outfile=None))]
```

Create an object to trace execution of a single statement or expression. All parameters are optional. `count` enables counting of line numbers. `trace` enables line execution tracing. `countfuncs` enables listing of the functions called during the run. `countcallers` enables call relationship tracking. `ignoremods` is a list of modules or packages to ignore. `ignoredirs` is a list of directories whose modules or packages should be ignored. `infile` is the file from which to read stored count information. `outfile` is a file in which to write updated count information.

```
run( cmd)
```

Run `cmd` under control of the Trace object with the current tracing parameters.

```
runcxtx( cmd[, globals=None[, locals=None]])
```

Run `cmd` under control of the Trace object with the current tracing parameters in the defined global and local environments. If not defined, `globals` and `locals` default to empty dictionaries.

```
runcfunc( func, *args, **kwds)
```

Call `func` with the given arguments under control of the `Trace` object with the current tracing parameters.

This is a simple example showing the use of this module:

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to
# do tracing or line-counting or both.
tracer = trace.Trace(    ignoredirs=[sys.prefix, sys.exec_prefix],
        trace=0,    count=1)
```
# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in /tmp
r = tracer.results()
r.write_results(show_missing=True, coverdir="/tmp")
26. Python Runtime Services

The modules described in this chapter provide a wide range of services related to the Python interpreter and its interaction with its environment. Here's an overview:

- `sys` Access system-specific parameters and functions.
- `__builtin__` The module that provides the built-in namespace.
- `__main__` The environment where the top-level script is run.
- `warnings` Issue warning messages and control their disposition.
- `contextlib` Utilities for `with`-statement contexts.
- `atexit` Register and execute cleanup functions.
- `traceback` Print or retrieve a stack traceback.
- `__future__` Future statement definitions
- `gc` Interface to the cycle-detecting garbage collector.
- `inspect` Extract information and source code from live objects.
- `site` A standard way to reference site-specific modules.
- `user` A standard way to reference user-specific modules.
- `fpectl` Provide control for floating point exception handling.
26.1 sys -- System-specific parameters and functions

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

argv
The list of command line arguments passed to a Python script. `argv[0]` is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the `-c` command line option to the interpreter, `argv[0]` is set to the string '-c'. If no script name was passed to the Python interpreter, `argv` has zero length.

byteorder
An indicator of the native byte order. This will have the value 'big' on big-endian (most-significant byte first) platforms, and 'little' on little-endian (least-significant byte first) platforms. New in version 2.0.

subversion
A triple (repo, branch, version) representing the Subversion information of the Python interpreter. `repo` is the name of the repository, 'CPython'. `branch` is a string of one of the forms 'trunk', 'branches/name' or 'tags/name'. `version` is the output of `svnversion`, if the interpreter was built from a Subversion checkout; it contains the revision number (range) and possibly a trailing 'M' if there were local modifications. If the tree was exported (or `svnversion` was not available), it is the revision of `Include/patchlevel.h` if the branch is a tag. Otherwise, it is `None`. New in version 2.5.

builtin_module_names
A tuple of strings giving the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way -- `modules.keys()` only lists the imported modules.)

copyright
A string containing the copyright pertaining to the Python interpreter.

$current_frames()$
Return a dictionary mapping each thread's identifier to the topmost stack frame currently active in that thread at the time the function is called. Note that functions in the `traceback` module can build the call stack given such a frame.

This is most useful for debugging deadlock: this function does not require the deadlocked threads' cooperation, and such threads' call stacks are frozen for as long as they remain deadlocked. The frame returned for a non-deadlocked thread may bear no relationship to that thread's current activity by the time calling code examines the frame.

This function should be used for internal and specialized purposes only. New in version 2.5.

$dllhandle$
Integer specifying the handle of the Python DLL. Availability: Windows.

displayhook($value$)
If $value$ is not `None`, this function prints it to `sys.stdout`, and saves it in `__builtin__._displayhook`.

`sys.displayhook` is called on the result of evaluating an expression entered in an interactive Python session. The display of these values can be customized by assigning another one-argument function to `sys.displayhook`.

excepthook($type$, $value$, $traceback$)
This function prints out a given traceback and exception to `sys.stderr`.

When an exception is raised and uncaught, the interpreter calls `sys.excepthook` with three arguments, the exception class, exception instance, and a traceback object. In an interactive session this happens just before control is returned to the prompt; in a Python program this happens just before the program exits. The handling of such top-level exceptions can be customized by assigning another three-argument function to
sys.excepthook.

__displayhook__
__excepthook__

These objects contain the original values of displayhook and excepthook at the start of the program. They are saved so that displayhook and excepthook can be restored in case they happen to get replaced with broken objects.

exc_info()

This function returns a tuple of three values that give information about the exception that is currently being handled. The information returned is specific both to the current thread and to the current stack frame. If the current stack frame is not handling an exception, the information is taken from the calling stack frame, or its caller, and so on until a stack frame is found that is handling an exception. Here, "handling an exception" is defined as "executing or having executed an except clause." For any stack frame, only information about the most recently handled exception is accessible.

If no exception is being handled anywhere on the stack, a tuple containing three None values is returned. Otherwise, the values returned are (type, value, traceback). Their meaning is: type gets the exception type of the exception being handled (a class object); value gets the exception parameter (its associated value or the second argument to raise, which is always a class instance if the exception type is a class object); traceback gets a traceback object (see the Reference Manual) which encapsulates the call stack at the point where the exception originally occurred.

If exc_clear() is called, this function will return three None values until either another exception is raised in the current thread or the execution stack returns to a frame where another exception is being handled.

Warning: Assigning the traceback return value to a local variable in a function that is handling an exception will cause a circular reference. This will prevent anything referenced by a local variable in the same function or by the traceback from being garbage collected. Since most functions don't need access to the traceback, the best solution is to use something like exc_type, value = sys.exc_info()[2] to extract only the
exception type and value. If you do need the traceback, make sure to delete it after use (best done with a try ... finally statement) or to call exc_info() in a function that does not itself handle an exception. Note: Beginning with Python 2.2, such cycles are automatically reclaimed when garbage collection is enabled and they become unreachable, but it remains more efficient to avoid creating cycles.

exc_clear()
This function clears all information relating to the current or last exception that occurred in the current thread. After calling this function, exc_info() will return three None values until another exception is raised in the current thread or the execution stack returns to a frame where another exception is being handled.

This function is only needed in only a few obscure situations. These include logging and error handling systems that report information on the last or current exception. This function can also be used to try to free resources and trigger object finalization, though no guarantee is made as to what objects will be freed, if any. New in version 2.3.

exc_type
exc_value
exc_traceback
Deprecated since release 1.5. Use exc_info() instead.

Since they are global variables, they are not specific to the current thread, so their use is not safe in a multi-threaded program. When no exception is being handled, exc_type is set to None and the other two are undefined.

exec_prefix
A string giving the site-specific directory prefix where the platform-dependent Python files are installed; by default, this is also '/usr/local'. This can be set at build time with the --exec-prefix argument to the configure script. Specifically, all configuration files (e.g. the pyconfig.h header file) are installed in the directory exec_prefix + '/lib/pythonversion/config', and shared library modules are installed in exec_prefix + '/lib/pythonversion/lib-dynload', where version is equal to version[:3].
executable
A string giving the name of the executable binary for the Python interpreter, on systems where this makes sense.

exit([arg])
Exit from Python. This is implemented by raising the SystemExit exception, so cleanup actions specified by finally clauses of try statements are honored, and it is possible to intercept the exit attempt at an outer level. The optional argument arg can be an integer giving the exit status (defaulting to zero), or another type of object. If it is an integer, zero is considered "successful termination" and any nonzero value is considered "abnormal termination" by shells and the like. Most systems require it to be in the range 0-127, and produce undefined results otherwise. Some systems have a convention for assigning specific meanings to specific exit codes, but these are generally underdeveloped; Unix programs generally use 2 for command line syntax errors and 1 for all other kind of errors. If another type of object is passed, None is equivalent to passing zero, and any other object is printed to sys.stderr and results in an exit code of 1. In particular, sys.exit("some error message") is a quick way to exit a program when an error occurs.

exitfunc
This value is not actually defined by the module, but can be set by the user (or by a program) to specify a clean-up action at program exit. When set, it should be a parameterless function. This function will be called when the interpreter exits. Only one function may be installed in this way; to allow multiple functions which will be called at termination, use the atexit module. Note: The exit function is not called when the program is killed by a signal, when a Python fatal internal error is detected, or when os._exit() is called.
Deprecated since release 2.4. Use atexit instead.

getcheckinterval()
Return the interpreter's "check interval"; see setcheckinterval(). New in version 2.3.

defaultencoding()
Return the name of the current default string encoding used by the Unicode
implementation. New in version 2.0.

getdlopenflags()

Return the current value of the flags that are used for dlopen() calls. The flag constants are defined in the dl and DLFCN modules. Availability: UNIX. New in version 2.2.

getfilesystemencoding()

Return the name of the encoding used to convert Unicode filenames into system file names, or None if the system default encoding is used. The result value depends on the operating system:

- On Windows 9x, the encoding is `mbcs`.
- On Mac OS X, the encoding is `utf-8`.
- On UNIX, the encoding is the user's preference according to the result of nl_langinfo(CODESET), or None if the nl_langinfo(CODESET) failed.
- On Windows NT+, file names are Unicode natively, so no conversion is performed. getfilesystemencoding() still returns 'mbcs', as this is the encoding that applications should use when they explicitly want to convert Unicode strings to byte strings that are equivalent when used as file names.

New in version 2.3.

getrefcount(object)

Return the reference count of the object. The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to getrefcount().

getrecursionlimit()

Return the current value of the recursion limit, the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. It can be set by setrecursionlimit().

_getframe([depth])

Return a frame object from the call stack. If optional integer depth is given, return the frame object that many calls below the top of the stack. If that is
deeper than the call stack, \texttt{ValueError} is raised. The default for \textit{depth} is zero, returning the frame at the top of the call stack.

This function should be used for internal and specialized purposes only.

\textbf{getwindowsversion()}  
Return a tuple containing five components, describing the Windows version currently running. The elements are \textit{major}, \textit{minor}, \textit{build}, \textit{platform}, and \textit{text}. \textit{text} contains a string while all other values are integers.

\textit{platform} may be one of the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Win32s on Windows 3.1</td>
</tr>
<tr>
<td>1</td>
<td>Windows 95/98/ME</td>
</tr>
<tr>
<td>2</td>
<td>Windows NT/2000/XP</td>
</tr>
<tr>
<td>3</td>
<td>Windows CE</td>
</tr>
</tbody>
</table>

This function wraps the Win32 \texttt{GetVersionEx()} function; see the Microsoft documentation for more information about these fields.

Availability: Windows. New in version 2.3.

\textbf{hexversion}  
The version number encoded as a single integer. This is guaranteed to increase with each version, including proper support for non-production releases. For example, to test that the Python interpreter is at least version 1.5.2, use:

```
if sys.hexversion >= 0x010502f0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

This is called "\texttt{hexversion}" since it only really looks meaningful when viewed as the result of passing it to the built-in \texttt{hex()} function. The \texttt{version_info} value may be used for a more human-friendly encoding of the same information. New in version 1.5.2.
last_type
last_value
last_traceback
These three variables are not always defined; they are set when an exception is not handled and the interpreter prints an error message and a stack traceback. Their intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error. (Typical use is "import pdb; pdb.pm()" to enter the post-mortem debugger; see chapter 24, "The Python Debugger," for more information.)

The meaning of the variables is the same as that of the return values from exc_info() above. (Since there is only one interactive thread, threadsafety is not a concern for these variables, unlike for exc_type etc.)

maxint
The largest positive integer supported by Python's regular integer type. This is at least 2**31-1. The largest negative integer is -maxint-1 -- the asymmetry results from the use of 2's complement binary arithmetic.

maxunicode
An integer giving the largest supported code point for a Unicode character. The value of this depends on the configuration option that specifies whether Unicode characters are stored as UCS-2 or UCS-4.

modules
This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. Note that removing a module from this dictionary is not the same as calling reload() on the corresponding module object.

path
A list of strings that specifies the search path for modules. Initialized from the environment variable PYTHONPATH, plus an installation-dependent default.

As initialized upon program startup, the first item of this list, path[0], is the directory containing the script that was used to invoke the Python interpreter. If the script directory is not available (e.g. if the interpreter is
invoked interactively or if the script is read from standard input), path[0] is the empty string, which directs Python to search modules in the current directory first. Notice that the script directory is inserted before the entries inserted as a result of PYTHONPATH.

A program is free to modify this list for its own purposes.

Changed in version 2.3: Unicode strings are no longer ignored.

**platform**
This string contains a platform identifier, e.g. 'sunos5' or 'linux1'. This can be used to append platform-specific components to path, for instance.

**prefix**
A string giving the site-specific directory prefix where the platform independent Python files are installed; by default, this is the string '/usr/local'. This can be set at build time with the --prefix argument to the configure script. The main collection of Python library modules is installed in the directory prefix + '/lib/pythonversion' while the platform independent header files (all except pyconfig.h) are stored in prefix + '/include/pythonversion', where version is equal to version[:3].

**ps1**
**ps2**
Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter is in interactive mode. Their initial values in this case are '>>> ' and '... '. If a non-string object is assigned to either variable, its str() is re-evaluated each time the interpreter prepares to read a new interactive command; this can be used to implement a dynamic prompt.

**setcheckinterval**( interval)
Set the interpreter's `check interval". This integer value determines how often the interpreter checks for periodic things such as thread switches and signal handlers. The default is 100, meaning the check is performed every 100 Python virtual instructions. Setting it to a larger value may increase performance for programs using threads. Setting it to a value <= 0 checks
every virtual instruction, maximizing responsiveness as well as overhead.

**setdefaultencoding**( *name*)
Set the current default string encoding used by the Unicode implementation. If *name* does not match any available encoding, LookupError is raised. This function is only intended to be used by the [site](https://docs.python.org/3/library/site.html) module implementation and, where needed, by [sitecustomize](https://docs.python.org/3/library/site.html#site.sitecustomize). Once used by the [site](https://docs.python.org/3/library/site.html) module, it is removed from the [sys](https://docs.python.org/3/library/sys.html) module's namespace. New in version 2.0.

**setdlopenflags**( *n*)
Set the flags used by the interpreter for dlopen() calls, such as when the interpreter loads extension modules. Among other things, this will enable a lazy resolving of symbols when importing a module, if called as [sys.setdlopenflags](https://docs.python.org/3/library/sys.html#sys.setdlopenflags)(0). To share symbols across extension modules, call as [sys.setdlopenflags](https://docs.python.org/3/library/sys.html#sys.setdlopenflags)(dl.RTLD_NOW | dl.RTLD_GLOBAL). Symbolic names for the flag modules can be either found in the d1 module, or in the DLFCN module. If DLFCN is not available, it can be generated from /usr/include/dlfcn.h using the [h2py](https://h2py.readthedocs.io/en/latest/) script. Availability: UNIX. New in version 2.2.

**setprofile**( *profilefunc*)
Set the system's profile function, which allows you to implement a Python source code profiler in Python. See chapter 25 for more information on the Python profiler. The system's profile function is called similarly to the system's trace function (see [settrace()](https://docs.python.org/3/library/sys.html#sys.settrace)), but it isn't called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return None.

**setrecursionlimit**( *limit*)
Set the maximum depth of the Python interpreter stack to *limit*. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.
The highest possible limit is platform-dependent. A user may need to set the limit higher when she has a program that requires deep recursion and a platform that supports a higher limit. This should be done with care, because a too-high limit can lead to a crash.

`settrace(tracefunc)`
Set the system's trace function, which allows you to implement a Python source code debugger in Python. See section 24.2, "How It Works," in the chapter on the Python debugger. The function is thread-specific; for a debugger to support multiple threads, it must be registered using `settrace()` for each thread being debugged. **Note:** The `settrace()` function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

`settscdump(on_flag)`
Activate dumping of VM measurements using the Pentium timestamp counter, if `on_flag` is true. Deactivate these dumps if `on_flag` is off. The function is available only if Python was compiled with `--with-tsc`. To understand the output of this dump, read `Python/ceval.c` in the Python sources. New in version 2.4.

`stdin`  
`stdout`  
`stderr`  

File objects corresponding to the interpreter's standard input, output and error streams. `stdin` is used for all interpreter input except for scripts but including calls to `input()` and `raw_input()`. `stdout` is used for the output of `print` and expression statements and for the prompts of `input()` and `raw_input()`. The interpreter's own prompts and (almost all of) its error messages go to `stderr`. `stdout` and `stderr` needn't be built-in file objects: any object is acceptable as long as it has a `write()` method that takes a string argument. (Changing these objects doesn't affect the standard I/O streams of processes executed by `os.popen()`, `os.system()` or the `exec*()` family of functions in the `os` module.)

`stdin`
These objects contain the original values of \texttt{stdin}, \texttt{stderr} and \texttt{stdout} at the start of the program. They are used during finalization, and could be useful to restore the actual files to known working file objects in case they have been overwritten with a broken object.

**tracebacklimit**

When this variable is set to an integer value, it determines the maximum number of levels of traceback information printed when an unhandled exception occurs. The default is \texttt{1000}. When set to \texttt{0} or less, all traceback information is suppressed and only the exception type and value are printed.

**version**

A string containing the version number of the Python interpreter plus additional information on the build number and compiler used. It has a value of the form \texttt{'version (#build_number, build_date, build_time) [compiler]'}. The first three characters are used to identify the version in the installation directories (where appropriate on each platform). An example:

```python
>>> import sys
>>> sys.version
'1.5.2 (#0 Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)]'
```

**api_version**

The C API version for this interpreter. Programmers may find this useful when debugging version conflicts between Python and extension modules. New in version 2.3.

**version_info**

A tuple containing the five components of the version number: \texttt{major}, \texttt{minor}, \texttt{micro}, \texttt{releaselevel}, and \texttt{serial}. All values except \texttt{releaselevel} are integers; the release level is \texttt{'alpha'}, \texttt{'beta'}, \texttt{'candidate'}, or \texttt{'final'}. The \texttt{version_info} value corresponding to the Python version 2.0 is \texttt{(2, 0, 0, 'final', 0)}. New in version 2.0.

**warnoptions**

This is an implementation detail of the warnings framework; do not modify this value. Refer to the \texttt{warnings} module for more information on the
warnings framework.

**winver**

The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the first three characters of *version*. It is provided in the *sys* module for informational purposes; modifying this value has no effect on the registry keys used by Python. Availability: Windows.

### See Also:

Module **site**:

This describes how to use `.pth` files to extend `sys.path`. 
26.2 __builtin__ -- Built-in objects

This module provides direct access to all `built-in' identifiers of Python; for example, __builtin__.open is the full name for the built-in function open(). See chapter 2, ``Built-in Objects.''

This module is not normally accessed explicitly by most applications, but can be useful in modules that provide objects with the same name as a built-in value, but in which the built-in of that name is also needed. For example, in a module that wants to implement an open() function that wraps the built-in open(), this module can be used directly:

```python
import __builtin__

def open(path):
    f = __builtin__.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to upper-case.''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

As an implementation detail, most modules have the name __builtins__ (note the "s") made available as part of their globals. The value of __builtins__ is normally either this module or the value of this module's __dict__ attribute. Since this is an implementation detail, it may not be used by alternate implementations of Python.
26.3 __main__ -- Top-level script environment

This module represents the (otherwise anonymous) scope in which the interpreter's main program executes -- commands read either from standard input, from a script file, or from an interactive prompt. It is this environment in which the idiomatic ``conditional script'' stanza causes a script to run:

```python
if __name__ == "__main__":
    main()
```

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
Warning Categories
26.4 warnings -- Warning control

New in version 2.1.

Warning messages are typically issued in situations where it is useful to alert the user of some condition in a program, where that condition (normally) doesn't warrant raising an exception and terminating the program. For example, one might want to issue a warning when a program uses an obsolete module.

Python programmers issue warnings by calling the `warn()` function defined in this module. (C programmers use `PyErr_Warn()`; see the Python/C API Reference Manual for details).

Warning messages are normally written to `sys.stderr`, but their disposition can be changed flexibly, from ignoring all warnings to turning them into exceptions. The disposition of warnings can vary based on the warning category (see below), the text of the warning message, and the source location where it is issued. Repetitions of a particular warning for the same source location are typically suppressed.

There are two stages in warning control: first, each time a warning is issued, a determination is made whether a message should be issued or not; next, if a message is to be issued, it is formatted and printed using a user-settable hook.

The determination whether to issue a warning message is controlled by the warning filter, which is a sequence of matching rules and actions. Rules can be added to the filter by calling `filterwarnings()` and reset to its default state by calling `resetwarnings()`.

The printing of warning messages is done by calling `showwarning()`, which may be overridden; the default implementation of this function formats the message by calling `formatwarning()`, which is also available for use by custom implementations.

### Subsections

- [26.4.1 Warning Categories](#)
26.4.2 The Warnings Filter
26.4.3 Available Functions

Warning Categories

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
26.4.1 Warning Categories

There are a number of built-in exceptions that represent warning categories. This categorization is useful to be able to filter out groups of warnings. The following warnings category classes are currently defined:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warning</td>
<td>This is the base class of all warning category classes. It is a subclass of Exception.</td>
</tr>
<tr>
<td>UserWarning</td>
<td>The default category for warn().</td>
</tr>
<tr>
<td>DeprecationWarning</td>
<td>Base category for warnings about deprecated features.</td>
</tr>
<tr>
<td>SyntaxWarning</td>
<td>Base category for warnings about dubious syntactic features.</td>
</tr>
<tr>
<td>RuntimeWarning</td>
<td>Base category for warnings about dubious runtime features.</td>
</tr>
<tr>
<td>FutureWarning</td>
<td>Base category for warnings about constructs that will change semantically in the future.</td>
</tr>
<tr>
<td>PendingDeprecationWarning</td>
<td>Base category for warnings about features that will be deprecated in the future (ignored by default).</td>
</tr>
<tr>
<td>ImportWarning</td>
<td>Base category for warnings triggered during the process of importing a module (ignored by default).</td>
</tr>
<tr>
<td>UnicodeWarning</td>
<td>Base category for warnings related to Unicode.</td>
</tr>
</tbody>
</table>

While these are technically built-in exceptions, they are documented here, because conceptually they belong to the warnings mechanism.

User code can define additional warning categories by subclassing one of the standard warning categories. A warning category must always be a subclass of the Warning class.
26.4.2 The Warnings Filter

The warnings filter controls whether warnings are ignored, displayed, or turned into errors (raising an exception).

Conceptually, the warnings filter maintains an ordered list of filter specifications; any specific warning is matched against each filter specification in the list in turn until a match is found; the match determines the disposition of the match. Each entry is a tuple of the form \((action, message, category, module, lineno)\), where:

- \textit{action} is one of the following strings:

<table>
<thead>
<tr>
<th>Value</th>
<th>Disposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;error&quot;</td>
<td>turn matching warnings into exceptions</td>
</tr>
<tr>
<td>&quot;ignore&quot;</td>
<td>never print matching warnings</td>
</tr>
<tr>
<td>&quot;always&quot;</td>
<td>always print matching warnings</td>
</tr>
<tr>
<td>&quot;default&quot;</td>
<td>print the first occurrence of matching warnings for each location where the warning is issued</td>
</tr>
<tr>
<td>&quot;module&quot;</td>
<td>print the first occurrence of matching warnings for each module where the warning is issued</td>
</tr>
<tr>
<td>&quot;once&quot;</td>
<td>print only the first occurrence of matching warnings, regardless of location</td>
</tr>
</tbody>
</table>

- \textit{message} is a string containing a regular expression that the warning message must match (the match is compiled to always be case-insensitive)

- \textit{category} is a class (a subclass of \texttt{Warning}) of which the warning category must be a subclass in order to match

- \textit{module} is a string containing a regular expression that the module name must match (the match is compiled to be case-sensitive)

- \textit{lineno} is an integer that the line number where the warning occurred must match, or \(0\) to match all line numbers

Since the \texttt{Warning} class is derived from the built-in \texttt{Exception} class, to turn a warning into an error we simply raise \texttt{category(message)}. 
The warnings filter is initialized by `-W` options passed to the Python interpreter command line. The interpreter saves the arguments for all `-W` options without interpretation in `sys.warnoptions`; the `warnings` module parses these when it is first imported (invalid options are ignored, after printing a message to `sys.stderr`).

The warnings that are ignored by default may be enabled by passing `-Wd` to the interpreter. This enables default handling for all warnings, including those that are normally ignored by default. This is particular useful for enabling `ImportWarning` when debugging problems importing a developed package. `ImportWarning` can also be enabled explicitly in Python code using:

```
warnings.simplefilter('default', ImportWarning)
```

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
26.4.3 Available Functions

**warn** (*message[, category[, stacklevel]]*)

Issue a warning, or maybe ignore it or raise an exception. The *category* argument, if given, must be a warning category class (see above); it defaults to `UserWarning`. Alternatively *message* can be a `Warning` instance, in which case *category* will be ignored and `message.__class__` will be used. In this case the message text will be `str(message)`. This function raises an exception if the particular warning issued is changed into an error by the warnings filter see above. The *stacklevel* argument can be used by wrapper functions written in Python, like this:

```python
    def deprecation(message):
        warnings.warn(message, DeprecationWarning, stacklevel=2)
```

This makes the warning refer to `deprecation()`'s caller, rather than to the source of `deprecation()` itself (since the latter would defeat the purpose of the warning message).

**warn_explicit** (*message, category, filename, lineno[, module[, registry[, module_globals]]]*)

This is a low-level interface to the functionality of `warn()`, passing in explicitly the message, category, filename and line number, and optionally the module name and the registry (which should be the __warningregistry__ dictionary of the module). The module name defaults to the filename with `.py` stripped; if no registry is passed, the warning is never suppressed. *message* must be a string and *category* a subclass of `Warning` or *message* may be a `Warning` instance, in which case *category* will be ignored.

*module_globals*, if supplied, should be the global namespace in use by the code for which the warning is issued. (This argument is used to support displaying source for modules found in zipfiles or other non-filesystem import sources, and was added in Python 2.5.)

**showwarning** (*message, category, filename, lineno[, file]*)
Write a warning to a file. The default implementation calls `formatwarning(message, category, filename, lineno)` and writes the resulting string to `file`, which defaults to `sys.stderr`. You may replace this function with an alternative implementation by assigning to `warnings.showwarning`.

`formatwarning(message, category, filename, lineno)` Format a warning the standard way. This returns a string which may contain embedded newlines and ends in a newline.

`filterwarnings(action[, message[, category[, module[, lineno[, append]]]]])` Insert an entry into the list of warnings filters. The entry is inserted at the front by default; if `append` is true, it is inserted at the end. This checks the types of the arguments, compiles the message and module regular expressions, and inserts them as a tuple in the list of warnings filters. Entries closer to the front of the list override entries later in the list, if both match a particular warning. Omitted arguments default to a value that matches everything.

`simplefilter(action[, category[, lineno[, append]]])` Insert a simple entry into the list of warnings filters. The meaning of the function parameters is as for `filterwarnings()`, but regular expressions are not needed as the filter inserted always matches any message in any module as long as the category and line number match.

`resetwarnings()` Reset the warnings filter. This discards the effect of all previous calls to `filterwarnings()`, including that of the `-W` command line options and calls to `simplefilter()`.
26.5 contextlib -- Utilities for with-statement contexts.

New in version 2.5.

This module provides utilities for common tasks involving the with statement.

Functions provided:

contextmanager(func)

This function is a decorator that can be used to define a factory function for with statement context managers, without needing to create a class or separate __enter__() and __exit__() methods.

A simple example (this is not recommended as a real way of generating HTML!):

```python
from __future__ import with_statement
from contextlib import contextmanager

@contextmanager
def tag(name):
    print "<%s>" % name
    yield
    print "</%s>" % name

>>> with tag("h1"):
...     print "foo"
...
<h1>
foo
</h1>
```

The function being decorated must return a generator-iterator when called. This iterator must yield exactly one value, which will be bound to the targets in the with statement’s as clause, if any.

At the point where the generator yields, the block nested in the with statement is executed. The generator is then resumed after the block is exited. If an unhandled exception occurs in the block, it is reraised inside
the generator at the point where the yield occurred. Thus, you can use a `try...except...finally` statement to trap the error (if any), or ensure that some cleanup takes place. If an exception is trapped merely in order to log it or to perform some action (rather than to suppress it entirely), the generator must reraise that exception. Otherwise the generator context manager will indicate to the `with` statement that the exception has been handled, and execution will resume with the statement immediately following the `with` statement.

**nested**(`mgr1[, mgr2[, ...]]`)  
Combine multiple context managers into a single nested context manager.

Code like this:

```python
from contextlib import nested

with nested(A, B, C) as (X, Y, Z):
    do_something()
```

is equivalent to this:

```python
with A as X:
    with B as Y:
        with C as Z:
            do_something()
```

Note that if the `__exit__()` method of one of the nested context managers indicates an exception should be suppressed, no exception information will be passed to any remaining outer context managers. Similarly, if the `__exit__()` method of one of the nested managers raises an exception, any previous exception state will be lost; the new exception will be passed to the `__exit__()` methods of any remaining outer context managers. In general, `__exit__()` methods should avoid raising exceptions, and in particular they should not re-raise a passed-in exception.

**closing**(`thing`)  
Return a context manager that closes `thing` upon completion of the block. This is basically equivalent to:

```python
from contextlib import contextmanager

@contextmanager
```
```python
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

And lets you write code like this:

```python
from __future__ import with_statement
from contextlib import closing
import codecs

with closing(urllib.urlopen('http://www.python.org')) as page:
    for line in page:
        print line
```

without needing to explicitly close `page`. Even if an error occurs, `page.close()` will be called when the `with` block is exited.

See Also:

**PEP 0343, The "with" statement**

The specification, background, and examples for the Python `with` statement.
26.6 atexit -- Exit handlers

New in version 2.0.

The `atexit` module defines a single function to register cleanup functions. Functions thus registered are automatically executed upon normal interpreter termination.

Note: the functions registered via this module are not called when the program is killed by a signal, when a Python fatal internal error is detected, or when `os._exit()` is called.

This is an alternate interface to the functionality provided by the `sys.exitfunc` variable.

Note: This module is unlikely to work correctly when used with other code that sets `sys.exitfunc`. In particular, other core Python modules are free to use `atexit` without the programmer's knowledge. Authors who use `sys.exitfunc` should convert their code to use `atexit` instead. The simplest way to convert code that sets `sys.exitfunc` is to import `atexit` and register the function that had been bound to `sys.exitfunc`.

```python
register(func[, *args[, **kargs]])
```

Register `func` as a function to be executed at termination. Any optional arguments that are to be passed to `func` must be passed as arguments to `register()`.

At normal program termination (for instance, if `sys.exit()` is called or the main module's execution completes), all functions registered are called in last in, first out order. The assumption is that lower level modules will normally be imported before higher level modules and thus must be cleaned up later.

If an exception is raised during execution of the exit handlers, a traceback is printed (unless `SystemExit` is raised) and the exception information is saved. After all exit handlers have had a chance to run the last exception to
be raised is re-raised.

See Also:

Module **readline**:
Useful example of `atexit` to read and write `readline` history files.

Subsections

- [26.6.1 atexit Example](#)

---

*Python Library Reference*

Previous: [26.5 contextlib](#) Up: [26. Python Runtime Services](#) Next: [26.6.1 atexit Example](#)

*Release 2.5, documentation updated on 19th September, 2006.*
See *About this document...* for information on suggesting changes.
26.6.1 atexit Example

The following simple example demonstrates how a module can initialize a counter from a file when it is imported and save the counter's updated value automatically when the program terminates without relying on the application making an explicit call into this module at termination.

```python
try:
    _count = int(open("/tmp/counter").read())
except IOError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    open("/tmp/counter", "w").write("%d" % _count)

import atexit
atexit.register(savecounter)
```

Positional and keyword arguments may also be passed to `register()` to be passed along to the registered function when it is called:

```python
def goodbye(name, adjective):
    print 'Goodbye, %s, it was %s to meet you.' % (name, adjective)

import atexit
atexit.register(goodbye, 'Donny', 'nice')

# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```
26.7 traceback -- Print or retrieve a stack traceback

This module provides a standard interface to extract, format and print stack traces of Python programs. It exactly mimics the behavior of the Python interpreter when it prints a stack trace. This is useful when you want to print stack traces under program control, such as in a ``wrapper'' around the interpreter.

The module uses traceback objects -- this is the object type that is stored in the variables `sys.exc_traceback` (deprecated) and `sys.last_traceback` and returned as the third item from `sys.exc_info()`.

The module defines the following functions:

**print_tb**(traceback[, limit[, file]])

Print up to limit stack trace entries from traceback. If limit is omitted or None, all entries are printed. If file is omitted or None, the output goes to `sys.stderr`; otherwise it should be an open file or file-like object to receive the output.

**print_exception**(type, value, traceback[, limit[, file]])

Print exception information and up to limit stack trace entries from traceback to file. This differs from `print_tb()` in the following ways: (1) if traceback is not None, it prints a header "Traceback (most recent call last):"; (2) it prints the exception type and value after the stack trace; (3) if type is SyntaxError and value has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

**print_exc**([limit[, file]])

This is a shorthand for `print_exception(sys.exc_type, sys.exc_value, sys.exc_traceback, limit, file)`. (In fact, it uses `sys.exc_info()` to retrieve the same information in a thread-safe
way instead of using the deprecated variables.)

**format_exc**([*limit]*)

This is like `print_exc(limit)` but returns a string instead of printing to a file. New in version 2.4.

**print_last**([*limit[, file]*])

This is a shorthand for `print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file)`.

**print_stack**([*f[, limit[, file]*]])

This function prints a stack trace from its invocation point. The optional *f* argument can be used to specify an alternate stack frame to start. The optional *limit* and *file* arguments have the same meaning as for `print_exception()`.

**extract_tb**(*traceback[, limit]*)

Return a list of up to *limit* ``pre-processed'' stack trace entries extracted from the traceback object `traceback`. It is useful for alternate formatting of stack traces. If *limit* is omitted or `None`, all entries are extracted. A ``pre-processed'' stack trace entry is a quadruple (`filename`, *line number*, *function name*, *text`) representing the information that is usually printed for a stack trace. The *text* is a string with leading and trailing whitespace stripped; if the source is not available it is `None`.

**extract_stack**([*f[, limit]*])

Extract the raw traceback from the current stack frame. The return value has the same format as for `extract_tb()`. The optional *f* and *limit* arguments have the same meaning as for `print_stack()`.

**format_list**(*list*)

Given a list of tuples as returned by `extract_tb()` or `extract_stack()`, return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not
None.

**format_exception_only**(type, value)
Format the exception part of a traceback. The arguments are the exception type and value such as given by `sys.last_type` and `sys.last_value`. The return value is a list of strings, each ending in a newline. Normally, the list contains a single string; however, for `SyntaxError` exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. The message indicating which exception occurred is the always last string in the list.

**format_exception**(type, value, tb[, limit])
Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to `print_exception()`.

The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does `print_exception()`.

**format_tb**(tb[, limit])
A shorthand for `format_list(extract_tb(tb, limit))`.

**format_stack**([f[, limit]])
A shorthand for `format_list(extract_stack(f, limit))`.

**tb_lineno**(tb)
This function returns the current line number set in the traceback object. This function was necessary because in versions of Python prior to 2.3 when the `-O` flag was passed to Python the `tb.tb_lineno` was not updated correctly. This function has no use in versions past 2.3.

---

Subsections

- [26.7.1 Traceback Example](#)
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
26.7.1 Traceback Example

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the code module.

```python
import sys, traceback

def run_user_code(envdir):
    source = raw_input(">>> ")
    try:
        exec source in envdir
    except:
        print "Exception in user code:"
        print '-'*60
        traceback.print_exc(file=sys.stdout)
        print '-'*60

    envdir = {}
    while 1:
        run_user_code(envdir)
```

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
26.8 __future__ -- Future statement definitions

__future__ is a real module, and serves three purposes:

- To avoid confusing existing tools that analyze import statements and expect to find the modules they're importing.

- To ensure that future_statements run under releases prior to 2.1 at least yield runtime exceptions (the import of __future__ will fail, because there was no module of that name prior to 2.1).

- To document when incompatible changes were introduced, and when they will be -- or were -- made mandatory. This is a form of executable documentation, and can be inspected programatically via importing __future__ and examining its contents.

Each statement in __future__.py is of the form:

```python
FeatureName = "_Feature(" OptionalRelease "," MandatoryRelease ","
CompilerFlag ")"
```

where, normally, OptionalRelease is less than MandatoryRelease, and both are 5-tuples of the same form as sys.version_info:

```python
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
PY_MINOR_VERSION, # the 1; an int
PY_MICRO_VERSION, # the 0; an int
PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; st
PY_RELEASE_SERIAL # the 3; an int
)
```

OptionalRelease records the first release in which the feature was accepted.

In the case of a MandatoryRelease that has not yet occurred, MandatoryRelease predicts the release in which the feature will become part of the language.
Else *MandatoryRelease* records when the feature became part of the language; in releases at or after that, modules no longer need a future statement to use the feature in question, but may continue to use such imports.

*MandatoryRelease* may also be *None*, meaning that a planned feature got dropped.

Instances of class *_Feature* have two corresponding methods, *getOptionalRelease()* and *getMandatoryRelease()*.

*CompilerFlag* is the (bitfield) flag that should be passed in the fourth argument to the builtin function *compile()* to enable the feature in dynamically compiled code. This flag is stored in the *compiler_flag* attribute on *_Feature* instances.

No feature description will ever be deleted from `__future__`.

---

*Python Library Reference*


*Release 2.5, documentation updated on 19th September, 2006.*

*See About this document... for information on suggesting changes.*
26.9 gc -- Garbage Collector interface

This module provides an interface to the optional garbage collector. It provides the ability to disable the collector, tune the collection frequency, and set debugging options. It also provides access to unreachable objects that the collector found but cannot free. Since the collector supplements the reference counting already used in Python, you can disable the collector if you are sure your program does not create reference cycles. Automatic collection can be disabled by calling `gc.disable()`. To debug a leaking program call `gc.set_debug(gc.DEBUG_LEAK)`. Notice that this includes `gc.DEBUG_SAVEALL`, causing garbage-collected objects to be saved in `gc.garbage` for inspection.

The `gc` module provides the following functions:

- **enable()**
  
  Enable automatic garbage collection.

- **disable()**
  
  Disable automatic garbage collection.

- **isenabled()**
  
  Returns true if automatic collection is enabled.

- **collect([generation])**
  
  With no arguments, run a full collection. The optional argument `generation` may be an integer specifying which generation to collect (from 0 to 2). A `ValueError` is raised if the generation number is invalid. The number of unreachable objects found is returned.

  Changed in version 2.5: The optional `generation` argument was added.

- **set_debug(flags)**
  
  Set the garbage collection debugging flags. Debugging information will be
written to \texttt{sys.stderr}. See below for a list of debugging flags which can be combined using bit operations to control debugging.

\textbf{get\_debug}()
\begin{flushleft}
Return the debugging flags currently set.
\end{flushleft}

\textbf{get\_objects}()
\begin{flushleft}
Returns a list of all objects tracked by the collector, excluding the list returned. New in version 2.2.
\end{flushleft}

\textbf{set\_threshold} \texttt{(threshold0[, threshold1[, threshold2]])}
\begin{flushleft}
Set the garbage collection thresholds (the collection frequency). Setting \texttt{threshold0} to zero disables collection.
\end{flushleft}

The GC classifies objects into three generations depending on how many collection sweeps they have survived. New objects are placed in the youngest generation (generation 0). If an object survives a collection it is moved into the next older generation. Since generation 2 is the oldest generation, objects in that generation remain there after a collection. In order to decide when to run, the collector keeps track of the number object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds \texttt{threshold0}, collection starts. Initially only generation 0 is examined. If generation 0 has been examined more than \texttt{threshold1} times since generation 1 has been examined, then generation 1 is examined as well. Similarly, \texttt{threshold2} controls the number of collections of generation 1 before collecting generation 2.

\textbf{get\_count}()
\begin{flushleft}
Return the current collection counts as a tuple of \texttt{(count0, count1, count2)}. New in version 2.5.
\end{flushleft}

\textbf{get\_threshold}()
\begin{flushleft}
Return the current collection thresholds as a tuple of \texttt{(threshold0, threshold1, threshold2)}.
\end{flushleft}

\textbf{get\_referrers} \texttt{(\*objs)}
Return the list of objects that directly refer to any of objs. This function will only locate those containers which support garbage collection; extension types which do refer to other objects but do not support garbage collection will not be found.

Note that objects which have already been dereferenced, but which live in cycles and have not yet been collected by the garbage collector can be listed among the resulting referrers. To get only currently live objects, call `collect()` before calling `get_referrers()`.

Care must be taken when using objects returned by `get_referrers()` because some of them could still be under construction and hence in a temporarily invalid state. Avoid using `get_referrers()` for any purpose other than debugging.

New in version 2.2.

```python
get_referents(*objs)
```

Return a list of objects directly referred to by any of the arguments. The referents returned are those objects visited by the arguments’ C-level `tp_traverse` methods (if any), and may not be all objects actually directly reachable. `tp_traverse` methods are supported only by objects that support garbage collection, and are only required to visit objects that may be involved in a cycle. So, for example, if an integer is directly reachable from an argument, that integer object may or may not appear in the result list.

New in version 2.3.

The following variable is provided for read-only access (you can mutate its value but should not rebind it):

```python
garbage
```

A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects). By default, this list contains only objects with `__del__()` methods. Objects that have `__del__()` methods and are part of a reference cycle cause the entire reference cycle to be uncollectable, including objects not necessarily in the cycle but reachable only from it. Python doesn't collect such cycles automatically because, in
general, it isn't possible for Python to guess a safe order in which to run the __del__() methods. If you know a safe order, you can force the issue by examining the garbage list, and explicitly breaking cycles due to your objects within the list. Note that these objects are kept alive even so by virtue of being in the garbage list, so they should be removed from garbage too. For example, after breaking cycles, do del gc.garbage[:] to empty the list. It's generally better to avoid the issue by not creating cycles containing objects with __del__() methods, and garbage can be examined in that case to verify that no such cycles are being created.

If DEBUG_SAVEALL is set, then all unreachable objects will be added to this list rather than freed.

The following constants are provided for use with set_debug():

**DEBUG_STATS**
Print statistics during collection. This information can be useful when tuning the collection frequency.

**DEBUG_COLLECTABLE**
Print information on collectable objects found.

**DEBUG_UNCOLLECTABLE**
Print information of uncollectable objects found (objects which are not reachable but cannot be freed by the collector). These objects will be added to the garbage list.

**DEBUG_INSTANCES**
When DEBUG_COLLECTABLE or DEBUG_UNCOLLECTABLE is set, print information about instance objects found.

**DEBUG_OBJECTS**
When DEBUG_COLLECTABLE or DEBUG_UNCOLLECTABLE is set, print information about objects other than instance objects found.

**DEBUG_SAVEALL**
When set, all unreachable objects found will be appended to garbage rather than being freed. This can be useful for debugging a leaking program.
DEBUG_LEAK

The debugging flags necessary for the collector to print information about a leaking program (equal to DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_INSTANCES | DEBUG_OBJECTS | DEBUG_SAVEALL).

Footnotes

... methods. 26.1

Prior to Python 2.2, the list contained all instance objects in unreachable cycles, not only those with __del__() methods.
26.10 inspect -- Inspect live objects

New in version 2.1.

The inspect module provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects. For example, it can help you examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

There are four main kinds of services provided by this module: type checking, getting source code, inspecting classes and functions, and examining the interpreter stack.

Subsections

- 26.10.1 Types and members
- 26.10.2 Retrieving source code
- 26.10.3 Classes and functions
- 26.10.4 The interpreter stack

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
## 26.10.1 Types and members

The `getmembers()` function retrieves the members of an object such as a class or module. The eleven functions whose names begin with "is" are mainly provided as convenient choices for the second argument to `getmembers()`. They also help you determine when you can expect to find the following special attributes:

<table>
<thead>
<tr>
<th>Type</th>
<th>Attribute</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>module</td>
<td><code>__doc__</code></td>
<td>documentation string</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>__file__</code></td>
<td>filename (missing for built-in modules)</td>
<td></td>
</tr>
<tr>
<td>class</td>
<td><code>__doc__</code></td>
<td>documentation string</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>__module__</code></td>
<td>name of module in which this class was defined</td>
<td></td>
</tr>
<tr>
<td>method</td>
<td><code>__doc__</code></td>
<td>documentation string</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>__name__</code></td>
<td>name with which this method was defined</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>im_class</code></td>
<td>class object that asked for this method</td>
<td>(1)</td>
</tr>
<tr>
<td></td>
<td><code>im_func</code></td>
<td>function object containing implementation of method</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>im_self</code></td>
<td>instance to which this method is bound, or None</td>
<td></td>
</tr>
<tr>
<td>function</td>
<td><code>__doc__</code></td>
<td>documentation string</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>__name__</code></td>
<td>name with which this function was defined</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>func_code</code></td>
<td>code object containing compiled function bytecode</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>func_defaults</code></td>
<td>tuple of any default values for arguments</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>func_doc</code></td>
<td>(same as <code>__doc__</code>)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>func_globals</code></td>
<td>global namespace in which this function was defined</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>func_name</code></td>
<td>(same as <code>__name__</code>)</td>
<td></td>
</tr>
<tr>
<td>traceback</td>
<td><code>tb_frame</code></td>
<td>frame object at this level</td>
<td></td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>tb_lasti</code></td>
<td>index of last attempted instruction in bytecode</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>tb_lineno</code></td>
<td>current line number in Python source code</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>tb_next</code></td>
<td>next inner traceback object (called by this level)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>frame</code></td>
<td>next outer frame object (this frame's caller)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>f_back</code></td>
<td>built-in namespace seen by this frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>f_builtins</code></td>
<td>code object being executed in this frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>f_code</code></td>
<td>traceback if raised in this frame, or None</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>f_exc_type</code></td>
<td>exception type if raised in this frame, or None</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>f_exc_value</code></td>
<td>exception value if raised in this frame, or None</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>f_globals</code></td>
<td>global namespace seen by this frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>f_lasti</code></td>
<td>local namespace seen by this frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>f_lineno</code></td>
<td>index of last attempted instruction in bytecode</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>f_locals</code></td>
<td>current line number in Python source code</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>f_restricted</code></td>
<td>0 or 1 if frame is in restricted execution mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>f_trace</code></td>
<td>tracing function for this frame, or None</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>code</code></td>
<td>number of arguments (not including *, ** args)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>co_argcount</code></td>
<td>string of raw compiled bytecode</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>co_consts</code></td>
<td>tuple of constants used in the bytecode</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>co_filename</code></td>
<td>name of file in which this code object was created</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>co_firstlineno</code></td>
<td>number of first line in Python source code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_flags</td>
<td>bitmap: 1=optimized</td>
<td>2=newlocals</td>
<td>4=*arg</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>co_lnotab</td>
<td>encoded mapping of line numbers to bytecode indices</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_name</td>
<td>name with which this code object was defined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_names</td>
<td>tuple of names of local variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_nlocals</td>
<td>number of local variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_stacksize</td>
<td>virtual machine stack space required</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_varnames</td>
<td>tuple of names of arguments and local variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>builtin</td>
<td><strong>doc</strong></td>
<td>documentation string</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>name</strong></td>
<td>original name of this function or method</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>self</strong></td>
<td>instance to which a method is bound, or None</td>
<td></td>
</tr>
</tbody>
</table>

Note:

(1) Changed in version 2.2: im_class used to refer to the class that defined the method.

getmembers( object[, predicate])

Return all the members of an object in a list of (name, value) pairs sorted by name. If the optional predicate argument is supplied, only members for which the predicate returns a true value are included.

getmoduleinfo( path)

Return a tuple of values that describe how Python will interpret the file identified by path if it is a module, or None if it would not be identified as a module. The return tuple is (name, suffix, mode, mtype), where name is the name of the module without the name of any enclosing package, suffix is the trailing part of the file name (which may not be a dot-delimited extension), mode is the open() mode that would be used ('r' or 'rb'), and mtype is an integer giving the type of the module. mtype will have a value which can be compared to the constants defined in the imp
module; see the documentation for that module for more information on
module types.

getmodulename( path)
    Return the name of the module named by the file path, without including
the names of enclosing packages. This uses the same algorithm as the
interpreter uses when searching for modules. If the name cannot be matched
according to the interpreter's rules, None is returned.

ismodule( object)
    Return true if the object is a module.

iscool (object)
    Return true if the object is a class.

ismethod( object)
    Return true if the object is a method.

isfunction( object)
    Return true if the object is a Python function or unnamed (lambda)
function.

istraceback( object)
    Return true if the object is a traceback.

isframe( object)
    Return true if the object is a frame.

isceode( object)
    Return true if the object is a code.

isbuiltin( object)
    Return true if the object is a built-in function.

isroutine( object)
    Return true if the object is a user-defined or built-in function or method.
ismethoddescriptor(object)
Return true if the object is a method descriptor, but not if ismethod() or isclass() or isfunction() are true.

This is new as of Python 2.2, and, for example, is true of int.__add__. An object passing this test has a __get__ attribute but not a __set__ attribute, but beyond that the set of attributes varies. __name__ is usually sensible, and __doc__ often is.

Methods implemented via descriptors that also pass one of the other tests return false from the ismethoddescriptor() test, simply because the other tests promise more - you can, e.g., count on having the im_func attribute (etc) when an object passes ismethod().

isdatadescriptor(object)
Return true if the object is a data descriptor.

Data descriptors have both a __get__ and a __set__ attribute. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have __name__ and __doc__ attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed. New in version 2.3.

isgetsetdescriptor(object)
Return true if the object is a getset descriptor.

getsets are attributes defined in extension modules via PyGetSetDef structures. For Python implementations without such types, this method will always return False. New in version 2.5.

ismemberdescriptor(object)
Return true if the object is a member descriptor.

Member descriptors are attributes defined in extension modules via PyMemberDef structures. For Python implementations without such types, this method will always return False. New in version 2.5.
Release 2.5, documentation updated on 19th September, 2006.  
See About this document... for information on suggesting changes.
26.10.2 Retrieving source code

`getdoc(object)`
Get the documentation string for an object. All tabs are expanded to spaces. To clean up docstrings that are indented to line up with blocks of code, any whitespace than can be uniformly removed from the second line onwards is removed.

`getcomments(object)`
Return in a single string any lines of comments immediately preceding the object's source code (for a class, function, or method), or at the top of the Python source file (if the object is a module).

`getfile(object)`
Return the name of the (text or binary) file in which an object was defined. This will fail with a `TypeError` if the object is a built-in module, class, or function.

`getmodule(object)`
Try to guess which module an object was defined in.

`getsourcefile(object)`
Return the name of the Python source file in which an object was defined. This will fail with a `TypeError` if the object is a built-in module, class, or function.

`getsourcelines(object)`
Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An `IOError` is raised if the source code cannot be retrieved.

`getsource(object)`
Return the text of the source code for an object. The argument may be a
module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An IOError is raised if the source code cannot be retrieved.
26.10.3 Classes and functions

getclasstree(classes[, unique])

Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the unique argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

getargspec(func)

Get the names and default values of a function's arguments. A tuple of four things is returned: (args, varargs, varkw, defaults). args is a list of the argument names (it may contain nested lists). varargs and varkw are the names of the * and ** arguments or None. defaults is a tuple of default argument values or None if there are no default arguments; if this tuple has n elements, they correspond to the last n elements listed in args.

getargvalues(frame)

Get information about arguments passed into a particular frame. A tuple of four things is returned: (args, varargs, varkw, locals). args is a list of the argument names (it may contain nested lists). varargs and varkw are the names of the * and ** arguments or None. locals is the locals dictionary of the given frame.

formatargspec(args[, varargs, varkw, defaults, formatarg, formatvarargs, formatvarkw, formatvalue, join])

Format a pretty argument spec from the four values returned by getargspec(). The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

formatargvalues(args[, varargs, varkw, locals, formatarg, formatvarargs, formatvarkw, formatvalue, join])
Format a pretty argument spec from the four values returned by `getargvalues()`. The `format*` arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

`getmro(cls)`

Return a tuple of class `cls`'s base classes, including `cls`, in method resolution order. No class appears more than once in this tuple. Note that the method resolution order depends on `cls`'s type. Unless a very peculiar user-defined metatype is in use, `cls` will be the first element of the tuple.
26.10.4 The interpreter stack

When the following functions return ``frame records,'' each record is a tuple of six items: the frame object, the filename, the line number of the current line, the function name, a list of lines of context from the source code, and the index of the current line within that list.

**Warning:** Keeping references to frame objects, as found in the first element of the frame records these functions return, can cause your program to create reference cycles. Once a reference cycle has been created, the lifespan of all objects which can be accessed from the objects which form the cycle can become much longer even if Python's optional cycle detector is enabled. If such cycles must be created, it is important to ensure they are explicitly broken to avoid the delayed destruction of objects and increased memory consumption which occurs.

Though the cycle detector will catch these, destruction of the frames (and local variables) can be made deterministic by removing the cycle in a `finally` clause. This is also important if the cycle detector was disabled when Python was compiled or using `gc.disable()`. For example:

```python
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

The optional `context` argument supported by most of these functions specifies the number of lines of context to return, which are centered around the current line.

**getframeinfo**( `frame[, context]` )

Get information about a frame or traceback object. A 5-tuple is returned, the last five elements of the frame's frame record.
getouterframes( frame[, context])

Get a list of frame records for a frame and all outer frames. These frames represent the calls that lead to the creation of frame. The first entry in the returned list represents frame; the last entry represents the outermost call on frame's stack.

getinnerframes( traceback[, context])

Get a list of frame records for a traceback's frame and all inner frames. These frames represent calls made as a consequence of frame. The first entry in the list represents traceback; the last entry represents where the exception was raised.

currentframe()

Return the frame object for the caller's stack frame.

stack([context])

Return a list of frame records for the caller's stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.

trace([context])

Return a list of frame records for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.
26.11 site -- Site-specific configuration hook

This module is automatically imported during initialization. The automatic import can be suppressed using the interpreter's -S option.

Importing this module will append site-specific paths to the module search path.

It starts by constructing up to four directories from a head and a tail part. For the head part, it uses sys.prefix and sys.exec_prefix; empty heads are skipped. For the tail part, it uses the empty string and then lib/site-packages (on Windows) or lib/python2.5/site-packages and then lib/site-python (on UNIX and Macintosh). For each of the distinct head-tail combinations, it sees if it refers to an existing directory, and if so, adds it to sys.path and also inspects the newly added path for configuration files.

A path configuration file is a file whose name has the form package.pth and exists in one of the four directories mentioned above; its contents are additional items (one per line) to be added to sys.path. Non-existing items are never added to sys.path, but no check is made that the item refers to a directory (rather than a file). No item is added to sys.path more than once. Blank lines and lines beginning with # are skipped. Lines starting with import are executed.

For example, suppose sys.prefix and sys.exec_prefix are set to /usr/local. The Python 2.5 library is then installed in /usr/local/lib/python2.5 (where only the first three characters of sys.version are used to form the installation path name). Suppose this has a subdirectory /usr/local/lib/python2.5/site-packages with three subsubdirectories, foo, bar and spam, and two path configuration files, foo.pth and bar.pth. Assume foo.pth contains the following:

```plaintext
# foo package configuration
foo
bar
bletch
```
and `bar.pth` contains:

```
# bar package configuration
bar
```

Then the following directories are added to `sys.path`, in this order:

```
/usr/local/lib/python2.3/site-packages/bar
/usr/local/lib/python2.3/site-packages/foo
```

Note that `bletch` is omitted because it doesn't exist; the `bar` directory precedes the `foo` directory because `bar.pth` comes alphabetically before `foo.pth`; and `spam` is omitted because it is not mentioned in either path configuration file.

After these path manipulations, an attempt is made to import a module named `sitecustomize`, which can perform arbitrary site-specific customizations. If this import fails with an `ImportError` exception, it is silently ignored.

Note that for some non-UNIX systems, `sys.prefix` and `sys.exec_prefix` are empty, and the path manipulations are skipped; however the import of `sitecustomize` is still attempted.
26.12 user -- User-specific configuration hook

As a policy, Python doesn't run user-specified code on startup of Python programs. (Only interactive sessions execute the script specified in the PYTHONSTARTUP environment variable if it exists).

However, some programs or sites may find it convenient to allow users to have a standard customization file, which gets run when a program requests it. This module implements such a mechanism. A program that wishes to use the mechanism must execute the statement

```python
import user
```

The `user` module looks for a file `.pythonrc.py` in the user's home directory and if it can be opened, executes it (using `execfile()`) in its own (the module `user`'s) global namespace. Errors during this phase are not caught; that's up to the program that imports the `user` module, if it wishes. The home directory is assumed to be named by the HOME environment variable; if this is not set, the current directory is used.

The user's `.pythonrc.py` could conceivably test for `sys.version` if it wishes to do different things depending on the Python version.

A warning to users: be very conservative in what you place in your `.pythonrc.py` file. Since you don't know which programs will use it, changing the behavior of standard modules or functions is generally not a good idea.

A suggestion for programmers who wish to use this mechanism: a simple way to let users specify options for your package is to have them define variables in their `.pythonrc.py` file that you test in your module. For example, a module `spam` that has a verbosity level can look for a variable `user.spam_verbose`, as follows:

```python
import user

verbose = bool(getattr(user, "spam_verbose", 0))
```
(The three-argument form of `getattr()` is used in case the user has not defined `spam_verbose` in their `.pythonrc.py` file.)

Programs with extensive customization needs are better off reading a program-specific customization file.

Programs with security or privacy concerns should not import this module; a user can easily break into a program by placing arbitrary code in the `.pythonrc.py` file.

Modules for general use should not import this module; it may interfere with the operation of the importing program.

**See Also:**

Module `site`:

Site-wide customization mechanism.
26.13 fpectl -- Floating point exception control

Availability: **Unix.**

Most computers carry out floating point operations in conformance with the so-called IEEE-754 standard. On any real computer, some floating point operations produce results that cannot be expressed as a normal floating point value. For example, try

```python
>>> import math
>>> math.exp(1000)
inf
>>> math.exp(1000) / math.exp(1000)
nan
```

(The example above will work on many platforms. DEC Alpha may be one exception.) "Inf" is a special, non-numeric value in IEEE-754 that stands for "infinity", and "nan" means "not a number." Note that, other than the non-numeric results, nothing special happened when you asked Python to carry out those calculations. That is in fact the default behaviour prescribed in the IEEE-754 standard, and if it works for you, stop reading now.

In some circumstances, it would be better to raise an exception and stop processing at the point where the faulty operation was attempted. The `fpectl` module is for use in that situation. It provides control over floating point units from several hardware manufacturers, allowing the user to turn on the generation of **SIGFPE** whenever any of the IEEE-754 exceptions Division by Zero, Overflow, or Invalid Operation occurs. In tandem with a pair of wrapper macros that are inserted into the C code comprising your python system, **SIGFPE** is trapped and converted into the Python **FloatingPointError** exception.

The `fpectl` module defines the following functions and may raise the given exception:

```python
turnon_sigfpe()
```

    Turn on the generation of **SIGFPE**, and set up an appropriate signal
handler.

**turnoff_sigfpe()**
Reset default handling of floating point exceptions.

**exception FloatingPointError**
After `turnon_sigfpe()` has been executed, a floating point operation that raises one of the IEEE-754 exceptions Division by Zero, Overflow, or Invalid operation will in turn raise this standard Python exception.

---

**Subsections**

- [26.13.1 Example](#)
- [26.13.2 Limitations and other considerations](#)

---

**Python Library Reference**


Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
26.13.1 Example

The following example demonstrates how to start up and test operation of the fpectl module.

```python
>>> import fpectl
>>> import fpetest
>>> fpectl.turnon_sigfpe()
>>> fpetest.test()
overflow    PASS
FloatingPointError: Overflow

div by 0    PASS
FloatingPointError: Division by zero
[  more output from test elided ]
>>> import math
>>> math.exp(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
FloatingPointError: in math_1
```


Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
26.13.2 Limitations and other considerations

Setting up a given processor to trap IEEE-754 floating point errors currently requires custom code on a per-architecture basis. You may have to modify `fpectl` to control your particular hardware.

Conversion of an IEEE-754 exception to a Python exception requires that the wrapper macros `PyFPE_START_PROTECT` and `PyFPE_END_PROTECT` be inserted into your code in an appropriate fashion. Python itself has been modified to support the `fpectl` module, but many other codes of interest to numerical analysts have not.

The `fpectl` module is not thread-safe.

See Also:

Some files in the source distribution may be interesting in learning more about how this module operates. The include file `include/pyfpe.h` discusses the implementation of this module at some length. `Modules/fpetestmodule.c` gives several examples of use. Many additional examples can be found in `Objects/floatobject.c`. 

---

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
27. Custom Python Interpreters

The modules described in this chapter allow writing interfaces similar to Python's interactive interpreter. If you want a Python interpreter that supports some special feature in addition to the Python language, you should look at the code module. (The codeop module is lower-level, used to support compiling a possibly-incomplete chunk of Python code.)

The full list of modules described in this chapter is:

- **code** Base classes for interactive Python interpreters.
- **codeop** Compile (possibly incomplete) Python code.
27.1 code -- Interpreter base classes

The code module provides facilities to implement read-eval-print loops in Python. Two classes and convenience functions are included which can be used to build applications which provide an interactive interpreter prompt.

class InteractiveInterpreter([locals])

This class deals with parsing and interpreter state (the user's namespace); it does not deal with input buffering or prompting or input file naming (the filename is always passed in explicitly). The optional locals argument specifies the dictionary in which code will be executed; it defaults to a newly created dictionary with key '__name__' set to '__console__' and key '__doc__' set to None.

class InteractiveConsole([locals[, filename]])

Closely emulate the behavior of the interactive Python interpreter. This class builds on InteractiveInterpreter and adds prompting using the familiar sys.ps1 and sys.ps2, and input buffering.

interact([banner[, readfunc[, local]]])

Convenience function to run a read-eval-print loop. This creates a new instance of InteractiveConsole and sets readfunc to be used as the raw_input() method, if provided. If local is provided, it is passed to the InteractiveConsole constructor for use as the default namespace for the interpreter loop. The interact() method of the instance is then run with banner passed as the banner to use, if provided. The console object is discarded after use.

compile_command(source[, filename[, symbol]])

This function is useful for programs that want to emulate Python's interpreter main loop (a.k.a. the read-eval-print loop). The tricky part is to determine when the user has entered an incomplete command that can be completed by entering more text (as opposed to a complete command or a syntax error). This function almost always makes the same decision as the real interpreter main loop.
source is the source string; filename is the optional filename from which source was read, defaulting to '<input>'; and symbol is the optional grammar start symbol, which should be either 'single' (the default) or 'eval'.

Returns a code object (the same as compile(source, filename, symbol)) if the command is complete and valid; None if the command is incomplete; raises SyntaxError if the command is complete and contains a syntax error, or raises OverflowError or ValueError if the command contains an invalid literal.
27.1.1 Interactive Interpreter Objects

**runsource**(*source[, filename[, symbol]])

Compile and run some source in the interpreter. Arguments are the same as for compile_command(); the default for filename is '&lt;input&gt;', and for symbol is 'single'. One several things can happen:

- The input is incorrect; compile_command() raised an exception (SyntaxError or OverflowError). A syntax traceback will be printed by calling the showsyntaxerror() method. runsource() returns False.

- The input is incomplete, and more input is required; compile_command() returned None. runsource() returns True.

- The input is complete; compile_command() returned a code object. The code is executed by calling the runcode() (which also handles run-time exceptions, except for SystemExit). runsource() returns False.

The return value can be used to decide whether to use sys.ps1 or sys.ps2 to prompt the next line.

**runcode**(*code*)

Execute a code object. When an exception occurs, showtraceback() is called to display a traceback. All exceptions are caught except SystemExit, which is allowed to propagate.

A note about KeyboardInterrupt: this exception may occur elsewhere in this code, and may not always be caught. The caller should be prepared to deal with it.

**showsyntaxerror**([filename])

Display the syntax error that just occurred. This does not display a stack trace because there isn't one for syntax errors. If filename is given, it is
stuffed into the exception instead of the default filename provided by
Python's parser, because it always uses '<string>' when reading from a
string. The output is written by the write() method.

**showtraceback()**
Display the exception that just occurred. We remove the first stack item
because it is within the interpreter object implementation. The output is
written by the write() method.

**write(data)**
Write a string to the standard error stream (sys.stderr). Derived classes
should override this to provide the appropriate output handling as needed.
27.1.2 Interactive Console Objects

The `InteractiveConsole` class is a subclass of `InteractiveInterpreter`, and so offers all the methods of the interpreter objects as well as the following additions.

**interact([banner])**
Closely emulate the interactive Python console. The optional banner argument specify the banner to print before the first interaction; by default it prints a banner similar to the one printed by the standard Python interpreter, followed by the class name of the console object in parentheses (so as not to confuse this with the real interpreter - since it's so close!).

**push(line)**
Push a line of source text to the interpreter. The line should not have a trailing newline; it may have internal newlines. The line is appended to a buffer and the interpreter's `runsource()` method is called with the concatenated contents of the buffer as source. If this indicates that the command was executed or invalid, the buffer is reset; otherwise, the command is incomplete, and the buffer is left as it was after the line was appended. The return value is `True` if more input is required, `False` if the line was dealt with in some way (this is the same as `runsource()`).

**resetbuffer()**
Remove any unhandled source text from the input buffer.

**raw_input([prompt])**
Write a prompt and read a line. The returned line does not include the trailing newline. When the user enters the EOF key sequence, `EOFError` is raised. The base implementation uses the built-in function `raw_input()`; a subclass may replace this with a different implementation.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
27.2 codeop -- Compile Python code

The codeop module provides utilities upon which the Python read-eval-print loop can be emulated, as is done in the code module. As a result, you probably don't want to use the module directly; if you want to include such a loop in your program you probably want to use the code module instead.

There are two parts to this job:

1. Being able to tell if a line of input completes a Python statement: in short, telling whether to print `>>>` or `...` next.
2. Remembering which future statements the user has entered, so subsequent input can be compiled with these in effect.

The codeop module provides a way of doing each of these things, and a way of doing them both.

To do just the former:

\[\text{compile_command}(\text{source}[\text{, filename}[\text{, symbol}]])\]

Tries to compile source, which should be a string of Python code and return a code object if source is valid Python code. In that case, the filename attribute of the code object will be filename, which defaults to '<<input>>'. Returns None if source is not valid Python code, but is a prefix of valid Python code.

If there is a problem with source, an exception will be raised. SyntaxError is raised if there is invalid Python syntax, and OverflowError or ValueError if there is an invalid literal.

The symbol argument determines whether source is compiled as a statement ('single', the default) or as an expression ('eval'). Any other value will cause ValueError to be raised.

Caveat: It is possible (but not likely) that the parser stops parsing with a successful outcome before reaching the end of the source; in this case, trailing symbols may be ignored instead of causing an error. For example, a
backslash followed by two newlines may be followed by arbitrary garbage. This will be fixed once the API for the parser is better.

class Compile()
Instances of this class have __call__() methods identical in signature to the built-in function compile(), but with the difference that if the instance compiles program text containing a __future__ statement, the instance 'remembers' and compiles all subsequent program texts with the statement in force.

class CommandCompiler()
Instances of this class have __call__() methods identical in signature to compile_command(); the difference is that if the instance compiles program text containing a __future__ statement, the instance 'remembers' and compiles all subsequent program texts with the statement in force.

A note on version compatibility: the Compile and CommandCompiler are new in Python 2.2. If you want to enable the future-tracking features of 2.2 but also retain compatibility with 2.1 and earlier versions of Python you can either write

```python
try:
    from codeop import CommandCompiler
    compile_command = CommandCompiler()
    del CommandCompiler
except ImportError:
    from codeop import compile_command
```

which is a low-impact change, but introduces possibly unwanted global state into your program, or you can write:

```python
try:
    from codeop import CommandCompiler
except ImportError:
    def CommandCompiler():
        from codeop import compile_command
        return compile_command
```

and then call CommandCompiler every time you need a fresh compiler object.
28. Restricted Execution

**Warning:** In Python 2.3 these modules have been disabled due to various known and not readily fixable security holes. The modules are still documented here to help in reading old code that uses the `re` and `Bastion` modules.

Restricted execution is the basic framework in Python that allows for the segregation of trusted and untrusted code. The framework is based on the notion that trusted Python code (a supervisor) can create a `padded cell` (or environment) with limited permissions, and run the untrusted code within this cell. The untrusted code cannot break out of its cell, and can only interact with sensitive system resources through interfaces defined and managed by the trusted code. The term "restricted execution" is favored over "safe-Python" since true safety is hard to define, and is determined by the way the restricted environment is created. Note that the restricted environments can be nested, with inner cells creating subcells of lesser, but never greater, privilege.

An interesting aspect of Python's restricted execution model is that the interfaces presented to untrusted code usually have the same names as those presented to trusted code. Therefore no special interfaces need to be learned to write code designed to run in a restricted environment. And because the exact nature of the padded cell is determined by the supervisor, different restrictions can be imposed, depending on the application. For example, it might be deemed "safe" for untrusted code to read any file within a specified directory, but never to write a file. In this case, the supervisor may redefine the built-in `open()` function so that it raises an exception whenever the `mode` parameter is `'w'`. It might also perform a `chroot()`-like operation on the `filename` parameter, such that root is always relative to some safe "sandbox" area of the filesystem. In this case, the untrusted code would still see an built-in `open()` function in its environment, with the same calling interface. The semantics would be identical too, with `IOError`s being raised when the supervisor determined that an unallowable parameter is being used.

The Python run-time determines whether a particular code block is executing in
restricted execution mode based on the identity of the __builtins__ object in its global variables: if this is (the dictionary of) the standard __builtin__ module, the code is deemed to be unrestricted, else it is deemed to be restricted.

Python code executing in restricted mode faces a number of limitations that are designed to prevent it from escaping from the padded cell. For instance, the function object attribute func_globals and the class and instance object attribute __dict__ are unavailable.

Two modules provide the framework for setting up restricted execution environments:

rexec Basic restricted execution framework.
Bastion Providing restricted access to objects.

See Also:

Grail Home Page Grail, an Internet browser written in Python, uses these modules to support Python applets. More information on the use of Python’s restricted execution mode in Grail is available on the Web site.
28.1 rexec -- Restricted execution framework

Changed in version 2.3: Disabled module.

Warning: The documentation has been left in place to help in reading old code that uses the module.

This module contains the RExec class, which supports r_eval(), r_execfile(), r_exec(), and r_import() methods, which are restricted versions of the standard Python functions eval(), execfile() and the exec and import statements. Code executed in this restricted environment will only have access to modules and functions that are deemed safe; you can subclass RExec to add or remove capabilities as desired.

Warning: While the rexec module is designed to perform as described below, it does have a few known vulnerabilities which could be exploited by carefully written code. Thus it should not be relied upon in situations requiring ``production ready'' security. In such situations, execution via sub-processes or very careful ``cleansing'' of both code and data to be processed may be necessary. Alternatively, help in patching known rexec vulnerabilities would be welcomed.

Note: The RExec class can prevent code from performing unsafe operations like reading or writing disk files, or using TCP/IP sockets. However, it does not protect against code using extremely large amounts of memory or processor time.

class RExec([hooks[, verbose]])
Returns an instance of the RExec class.

*hooks* is an instance of the RHooks class or a subclass of it. If it is omitted or None, the default RHooks class is instantiated. Whenever the rexec module searches for a module (even a built-in one) or reads a module's code, it doesn't actually go out to the file system itself. Rather, it calls methods of an RHooks instance that was passed to or created by its constructor. (Actually, the RExec object doesn't make these calls -- they are made by a module loader object that's part of the RExec object. This allows another level of flexibility, which can be useful when changing the mechanics of import within the restricted environment.)

By providing an alternate RHooks object, we can control the file system accesses made to import a module, without changing the actual algorithm that controls the order in which those accesses are made. For instance, we could substitute an RHooks object that passes all filesystem requests to a file server elsewhere, via some RPC mechanism such as ILU. Grail's applet loader uses this to support importing applets from a URL for a directory.

If *verbose* is true, additional debugging output may be sent to standard output.

It is important to be aware that code running in a restricted environment can still call the `sys.exit()` function. To disallow restricted code from exiting the interpreter, always protect calls that cause restricted code to run with a `try/except` statement that catches the SystemExit exception. Removing the `sys.exit()` function from the restricted environment is not sufficient -- the restricted code could still use `raise SystemExit`. Removing SystemExit is not a reasonable option; some library code makes use of this and would break were it not available.

**See Also:**

*Grail Home Page*

Grail is a Web browser written entirely in Python. It uses the rexec module as a foundation for supporting Python applets, and can be used as an example usage of this module.
28.1.1 RExec Objects

RExec instances support the following methods:

r_eval(code)
   code must either be a string containing a Python expression, or a compiled
code object, which will be evaluated in the restricted environment's
__main__ module. The value of the expression or code object will be
returned.

r_exec(code)
   code must either be a string containing one or more lines of Python code, or
a compiled code object, which will be executed in the restricted
environment's __main__ module.

r_execfile(filename)
   Execute the Python code contained in the file filename in the restricted
environment's __main__ module.

Methods whose names begin with "s_" are similar to the functions beginning
with "r_", but the code will be granted access to restricted versions of the
standard I/O streams sys.stdin, sys.stderr, and sys.stdout.

s_eval(code)
   code must be a string containing a Python expression, which will be
evaluated in the restricted environment.

s_exec(code)
   code must be a string containing one or more lines of Python code, which
will be executed in the restricted environment.

s_execfile(code)
   Execute the Python code contained in the file filename in the restricted
environment.

RExec objects must also support various methods which will be implicitly
called by code executing in the restricted environment. Overriding these methods in a subclass is used to change the policies enforced by a restricted environment.

**r_import(module, globals[, locals[, fromlist]]))**
Import the module `module`, raising an `ImportError` exception if the module is considered unsafe.

**r_open(filename[, mode[, bufsize]])**
Method called when `open()` is called in the restricted environment. The arguments are identical to those of `open()`, and a file object (or a class instance compatible with file objects) should be returned. `RExec`'s default behaviour is allow opening any file for reading, but forbidding any attempt to write a file. See the example below for an implementation of a less restrictive `r_open()`.

**r_reload(module)**
Reload the module object `module`, re-parsing and re-initializing it.

**r_unload(module)**
Unload the module object `module` (remove it from the restricted environment's `sys.modules` dictionary).

And their equivalents with access to restricted standard I/O streams:

**s_import(module, globals[, locals[, fromlist]]))**
Import the module `module`, raising an `ImportError` exception if the module is considered unsafe.

**s_reload(module)**
Reload the module object `module`, re-parsing and re-initializing it.

**s_unload(module)**
Unload the module object `module`. 
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
28.1.2 Defining restricted environments

The `RExec` class has the following class attributes, which are used by the `__init__()` method. Changing them on an existing instance won't have any effect; instead, create a subclass of `RExec` and assign them new values in the class definition. Instances of the new class will then use those new values. All these attributes are tuples of strings.

**nok_builtin_names**
Contains the names of built-in functions which will *not* be available to programs running in the restricted environment. The value for `RExec` is (`'open', 'reload', '__import__'`). (This gives the exceptions, because by far the majority of built-in functions are harmless. A subclass that wants to override this variable should probably start with the value from the base class and concatenate additional forbidden functions -- when new dangerous built-in functions are added to Python, they will also be added to this module.)

**ok_builtin_modules**
Contains the names of built-in modules which can be safely imported. The value for `RExec` is (`'audioop', 'array', 'binascii', 'cmath', 'errno', 'imageop', 'marshal', 'math', 'md5', 'operator', 'parser', 'regex', 'select', 'sha', '_sre', 'strop', 'struct', 'time'`). A similar remark about overriding this variable applies -- use the value from the base class as a starting point.

**ok_path**
Contains the directories which will be searched when an `import` is performed in the restricted environment. The value for `RExec` is the same as `sys.path` (at the time the module is loaded) for unrestricted code.

**ok_posix_names**
Contains the names of the functions in the `os` module which will be available to programs running in the restricted environment. The value for `RExec` is (`'error', 'fstat', 'listdir', 'lstat', 'readlink', 'stat', 'times', 'uname', 'getpid',`
'getppid', 'getcwd', 'getuid', 'getgid',
'geteuid', 'getegid').

**ok_sys_names**
Contains the names of the functions and variables in the `sys` module which will be available to programs running in the restricted environment. The value for RExec is ('ps1', 'ps2', 'copyright', 'version', 'platform', 'exit', 'maxint').

**ok_file_types**
Contains the file types from which modules are allowed to be loaded. Each file type is an integer constant defined in the `imp` module. The meaningful values are PY_SOURCE, PYCompiled, and C_EXTENSION. The value for RExec is (C_EXTENSION, PY_SOURCE). Adding PYCompiled in subclasses is not recommended; an attacker could exit the restricted execution mode by putting a forged byte-compiled file (.pyc) anywhere in your file system, for example by writing it to /tmp or uploading it to the /incoming directory of your public FTP server.
28.1.3 An example

Let us say that we want a slightly more relaxed policy than the standard RExec class. For example, if we're willing to allow files in /tmp to be written, we can subclass the RExec class:

```python
class TmpWriterRExec(rexec.RExec):
    def r_open(self, file, mode='r', buf=-1):
        if mode in ('r', 'rb'):
            pass
        elif mode in ('w', 'wb', 'a', 'ab'):
            # check filename : must begin with /tmp/
            if file[:5]!='/tmp/':
                raise IOError, "can't write outside /tmp"
            elif (string.find(file, '/../') >= 0 or
                  file[:3] == '../' or file[-3:] == '/../'):
                raise IOError, "'..' in filename forbidden"
        else:
            raise IOError, "Illegal open() mode"
        return open(file, mode, buf)
```

Notice that the above code will occasionally forbid a perfectly valid filename; for example, code in the restricted environment won't be able to open a file called /tmp/foo/../bar. To fix this, the r_open() method would have to simplify the filename to /tmp/bar, which would require splitting apart the filename and performing various operations on it. In cases where security is at stake, it may be preferable to write simple code which is sometimes overly restrictive, instead of more general code that is also more complex and may harbor a subtle security hole.
28.2 Bastion -- Restricting access to objects

Changed in version 2.3: Disabled module.

**Warning:** The documentation has been left in place to help in reading old code that uses the module.

According to the dictionary, a bastion is ``a fortified area or position'', or ``something that is considered a stronghold.'' It's a suitable name for this module, which provides a way to forbid access to certain attributes of an object. It must always be used with the `reexec` module, in order to allow restricted-mode programs access to certain safe attributes of an object, while denying access to other, unsafe attributes.

**Bastion**(*object*, *filter*, *name*, *class*)

Protect the object `object`, returning a bastion for the object. Any attempt to access one of the object's attributes will have to be approved by the `filter` function; if the access is denied an `AttributeError` exception will be raised.

If present, `filter` must be a function that accepts a string containing an attribute name, and returns true if access to that attribute will be permitted; if `filter` returns false, the access is denied. The default filter denies access to any function beginning with an underscore ("_"). The bastion's string representation will be "<Bastion for name>" if a value for `name` is provided; otherwise, "repr(object)" will be used.

`class`, if present, should be a subclass of `BastionClass`; see the code in `bastion.py` for the details. Overriding the default `BastionClass` will rarely be required.

**class BastionClass**(*getfunc*, *name*)
Class which actually implements bastion objects. This is the default class used by `Bastion()`. The `getfunc` parameter is a function which returns the value of an attribute which should be exposed to the restricted execution environment when called with the name of the attribute as the only parameter. `name` is used to construct the `repr()` of the `BastionClass` instance.
29. Importing Modules

The modules described in this chapter provide new ways to import other Python modules and hooks for customizing the import process.

The full list of modules described in this chapter is:

- **imp**: Access the implementation of the `import` statement.
- **zipimport**: Support for importing Python modules from ZIP archives.
- **pkgutil**: Utilities to support extension of packages.
- **modulefinder**: Find modules used by a script.
- **runpy**: Locate and execute Python modules as scripts.

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
29.1 imp -- Access the import internals

This module provides an interface to the mechanisms used to implement the import statement. It defines the following constants and functions:

get_magic()

Return the magic string value used to recognize byte-compiled code files (.pyc files). (This value may be different for each Python version.)

get_suffixes()

Return a list of triples, each describing a particular type of module. Each triple has the form (suffix, mode, type), where suffix is a string to be appended to the module name to form the filename to search for, mode is the mode string to pass to the built-in open() function to open the file (this can be 'r' for text files or 'rb' for binary files), and type is the file type, which has one of the values PY_SOURCE, PY_COMPILED, or C_EXTENSION, described below.

find_module(name[, path])

Try to find the module name on the search path path. If path is a list of directory names, each directory is searched for files with any of the suffixes returned by get_suffixes() above. Invalid names in the list are silently ignored (but all list items must be strings). If path is omitted or None, the list of directory names given by sys.path is searched, but first it searches a few special places: it tries to find a built-in module with the given name (C_BUILTIN), then a frozen module (PY_FROZEN), and on some systems some other places are looked in as well (on the Mac, it looks for a resource (PY_RESOURCE); on Windows, it looks in the registry which may point to a specific file).

If search is successful, the return value is a triple (file, pathname, description) where file is an open file object positioned at the beginning, pathname is the pathname of the file found, and description is a triple as
contained in the list returned by `get_suffixes()` describing the kind of module found. If the module does not live in a file, the returned `file` is `None`, `filename` is the empty string, and the `description` tuple contains empty strings for its suffix and mode; the module type is as indicated in parentheses above. If the search is unsuccessful, `ImportError` is raised. Other exceptions indicate problems with the arguments or environment.

This function does not handle hierarchical module names (names containing dots). In order to find `P.M`, that is, submodule `M` of package `P`, use `find_module()` and `load_module()` to find and load package `P`, and then use `find_module()` with the `path` argument set to `P.__path__`. When `P` itself has a dotted name, apply this recipe recursively.

`load_module(name, file, filename, description)`

Load a module that was previously found by `find_module()` (or by an otherwise conducted search yielding compatible results). This function does more than importing the module: if the module was already imported, it is equivalent to a `reload()`! The `name` argument indicates the full module name (including the package name, if this is a submodule of a package). The `file` argument is an open file, and `filename` is the corresponding file name; these can be `None` and `'`, respectively, when the module is not being loaded from a file. The `description` argument is a tuple, as would be returned by `get_suffixes()`, describing what kind of module must be loaded.

If the load is successful, the return value is the module object; otherwise, an exception (usually `ImportError`) is raised.

**Important:** the caller is responsible for closing the `file` argument, if it was not `None`, even when an exception is raised. This is best done using a `try` ... `finally` statement.

`new_module(name)`

Return a new empty module object called `name`. This object is not inserted in `sys.modules`.

`lock_held()`
Return True if the import lock is currently held, else False. On platforms without threads, always return False.

On platforms with threads, a thread executing an import holds an internal lock until the import is complete. This lock blocks other threads from doing an import until the original import completes, which in turn prevents other threads from seeing incomplete module objects constructed by the original thread while in the process of completing its import (and the imports, if any, triggered by that).

**acquire_lock()**
Acquires the interpreter's import lock for the current thread. This lock should be used by import hooks to ensure thread-safety when importing modules. On platforms without threads, this function does nothing. New in version 2.3.

**release_lock()**
Release the interpreter's import lock. On platforms without threads, this function does nothing. New in version 2.3.

The following constants with integer values, defined in this module, are used to indicate the search result of find_module().

**PY_SOURCE**
The module was found as a source file.

**PY_COMPILED**
The module was found as a compiled code object file.

**C_EXTENSION**
The module was found as dynamically loadable shared library.

**PYRESOURCE**
The module was found as a Mac OS 9 resource. This value can only be returned on a Mac OS 9 or earlier Macintosh.

**PKG_DIRECTORY**
The module was found as a package directory.
**C_BUILTIN**
The module was found as a built-in module.

**PY_FROZEN**
The module was found as a frozen module (see init_frozen()).

The following constant and functions are obsolete; their functionality is available through find_module() or load_module(). They are kept around for backward compatibility:

**SEARCH_ERROR**
Unused.

**init_builtin**(name)
Initialize the built-in module called name and return its module object. If the module was already initialized, it will be initialized again. A few modules cannot be initialized twice -- attempting to initialize these again will raise an ImportError exception. If there is no built-in module called name, None is returned.

**init_frozen**(name)
Initialize the frozen module called name and return its module object. If the module was already initialized, it will be initialized again. If there is no frozen module called name, None is returned. (Frozen modules are modules written in Python whose compiled byte-code object is incorporated into a custom-built Python interpreter by Python's freeze utility. See Tools/freezefile for now.)

**is_builtin**(name)
Return 1 if there is a built-in module called name which can be initialized again. Return -1 if there is a built-in module called name which cannot be initialized again (see init builtin()). Return 0 if there is no built-in module called name.

**is_frozen**(name)
Return True if there is a frozen module (see init_frozen()) called name, or False if there is no such module.
**load_compiled**(*name, pathname, [file]*)

Load and initialize a module implemented as a byte-compiled code file and return its module object. If the module was already initialized, it will be initialized *again*. The *name* argument is used to create or access a module object. The *pathname* argument points to the byte-compiled code file. The *file* argument is the byte-compiled code file, open for reading in binary mode, from the beginning. It must currently be a real file object, not a user-defined class emulating a file.

**load_dynamic**(*name, pathname[, file]*)

Load and initialize a module implemented as a dynamically loadable shared library and return its module object. If the module was already initialized, it will be initialized *again*. Some modules don't like that and may raise an exception. The *pathname* argument must point to the shared library. The *name* argument is used to construct the name of the initialization function: an external C function called "initname()" in the shared library is called. The optional *file* argument is ignored. (Note: using shared libraries is highly system dependent, and not all systems support it.)

**load_source**(*name, pathname[, file]*)

Load and initialize a module implemented as a Python source file and return its module object. If the module was already initialized, it will be initialized *again*. The *name* argument is used to create or access a module object. The *pathname* argument points to the source file. The *file* argument is the source file, open for reading as text, from the beginning. It must currently be a real file object, not a user-defined class emulating a file. Note that if a properly matching byte-compiled file (with suffix .pyc or .pyo) exists, it will be used instead of parsing the given source file.

**class NullImporter**(*path_string*)

The NullImporter type is a [PEP 302](https://www.python.org/dev/peps/pep-0302) import hook that handles non-directory path strings by failing to find any modules. Calling this type with an existing directory or empty string raises `ImportError`. Otherwise, a NullImporter instance is returned.

Python adds instances of this type to `sys.path_importer_cache` for any path entries that are not directories and are not handled by any other
path hooks on `sys.path_hooks`. Instances have only one method:

```python
find_module(fullname [, path])
```
This method always returns `None`, indicating that the requested module could not be found.

New in version 2.5.

---

**Subsections**

- [29.1.1 Examples](#)

---

**Python Library Reference**

Previous: [29. Importing Modules](#) Up: [29. Importing Modules](#) Next: [29.1.1 Examples](#)

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
29.1.1 Examples

The following function emulates what was the standard import statement up to Python 1.4 (no hierarchical module names). (This *implementation* wouldn't work in that version, since `find_module()` has been extended and `load_module()` has been added in 1.4.)

```python
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
        return sys.modules[name]
    except KeyError:
        pass

    # If any of the following calls raises an exception,
    # there's a problem we can't handle -- let the caller handle it.

    fp, pathname, description = imp.find_module(name)

    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        # Since we may exit via an exception, close fp explicitly.
        if fp:
            fp.close()
```

A more complete example that implements hierarchical module names and includes a `reload()` function can be found in the module `knee`. The `knee` module can be found in `Demo/imputil/` in the Python source distribution.
29.2 zipimport -- Import modules from Zip archives

New in version 2.3.

This module adds the ability to import Python modules (*.py, *.py[co]) and packages from ZIP-format archives. It is usually not needed to use the `zipimport` module explicitly; it is automatically used by the builtin `import` mechanism for `sys.path` items that are paths to ZIP archives.

Typically, `sys.path` is a list of directory names as strings. This module also allows an item of `sys.path` to be a string naming a ZIP file archive. The ZIP archive can contain a subdirectory structure to support package imports, and a path within the archive can be specified to only import from a subdirectory. For example, the path `/tmp/example.zip/lib/` would only import from the `lib/` subdirectory within the archive.

Any files may be present in the ZIP archive, but only files `.py` and `.py[co]` are available for import. ZIP import of dynamic modules (.pyd, .so) is disallowed. Note that if an archive only contains `.py` files, Python will not attempt to modify the archive by adding the corresponding `.pyc` or `.pyo` file, meaning that if a ZIP archive doesn't contain `.pyc` files, importing may be rather slow.

Using the built-in `reload()` function will fail if called on a module loaded from a ZIP archive; it is unlikely that `reload()` would be needed, since this would imply that the ZIP has been altered during runtime.

The available attributes of this module are:

**exception Zip ImportError**
Exception raised by zipimporter objects. It's a subclass of `ImportError`, so it can be caught as `ImportError`, too.

**class zipimporter**
The class for importing ZIP files. See `}`zipimporter Objects" (section 29.2.1) for constructor details.
See Also:

PKZIP Application Note
  Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

PEP 0273, Import Modules from Zip Archives
  Written by James C. Ahlstrom, who also provided an implementation. Python 2.3 follows the specification in PEP 273, but uses an implementation written by Just van Rossum that uses the import hooks described in PEP 302.

PEP 0302, New Import Hooks
  The PEP to add the import hooks that help this module work.

---

Subsections

- 29.2.1 zipimporter Objects
- 29.2.2 Examples

---

Python Library Reference

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
29.2.1 zipimporter Objects

class zipimporter(archivepath)
    Create a new zipimporter instance. archivepath must be a path to a zipfile. ZipImportError is raised if archivepath doesn't point to a valid ZIP archive.

find_module(fullname[, path])
    Search for a module specified by fullname. fullname must be the fully qualified (dotted) module name. It returns the zipimporter instance itself if the module was found, or None if it wasn't. The optional path argument is ignored--it's there for compatibility with the importer protocol.

get_code(fullname)
    Return the code object for the specified module. Raise ZipImportError if the module couldn't be found.

get_data(pathname)
    Return the data associated with pathname. Raise IOError if the file wasn't found.

get_source(fullname)
    Return the source code for the specified module. Raise ZipImportError if the module couldn't be found, return None if the archive does contain the module, but has no source for it.

is_package(fullname)
    Return True if the module specified by fullname is a package. Raise ZipImportError if the module couldn't be found.

load_module(fullname)
    Load the module specified by fullname. fullname must be the fully qualified (dotted) module name. It returns the imported module, or raises ZipImportError if it wasn't found.
29.2.2 Examples

Here is an example that imports a module from a ZIP archive - note that the `zipimport` module is not explicitly used.

```
$ unzip -l /tmp/example.zip
Archive: /tmp/example.zip
   Length   Date   Time    Name
        ----  ----   ----    ----
   8467  11-26-02 22:30    jwzthreading.py

8467        1 file

$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, '/tmp/example.zip')  # Add .zip file to front
>>> import jwzthreading
>>> jwzthreading.__file__
'/tmp/example.zip/jwzthreading.py'
```
29.3 **pkgutil -- Package extension utility**

New in version 2.3.

This module provides a single function:

**extend_path(path, name)**

Extend the search path for the modules which comprise a package. Intended use is to place the following code in a package's __init__.py:

```python
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

This will add to the package's __path__ all subdirectories of directories on sys.path named after the package. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

It also looks for *.pkg files beginning where * matches the name argument. This feature is similar to *.pth files (see the site module for more information), except that it doesn't special-case lines starting with import. A *.pkg file is trusted at face value: apart from checking for duplicates, all entries found in a *.pkg file are added to the path, regardless of whether they exist on the filesystem. (This is a feature.)

If the input path is not a list (as is the case for frozen packages) it is returned unchanged. The input path is not modified; an extended copy is returned. Items are only appended to the copy at the end.

It is assumed that sys.path is a sequence. Items of sys.path that are not (Unicode or 8-bit) strings referring to existing directories are ignored. Unicode items on sys.path that cause errors when used as filenames may cause this function to raise an exception (in line with os.path.isdir() behavior).
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
29.4 modulefinder -- Find modules used by a script

New in version 2.3.

This module provides a ModuleFinder class that can be used to determine the set of modules imported by a script. modulefinder.py can also be run as a script, giving the filename of a Python script as its argument, after which a report of the imported modules will be printed.

AddPackagePath( pkg_name, path)

Record that the package named pkg_name can be found in the specified path.

ReplacePackage( oldname, newname)

Allows specifying that the module named oldname is in fact the package named newname. The most common usage would be to handle how the _xmlplus package replaces the xml package.

class ModuleFinder( [path=None, debug=0, excludes=[], replace_paths= []])

This class provides run_script() and report() methods to determine the set of modules imported by a script. path can be a list of directories to search for modules; if not specified, sys.path is used. debug sets the debugging level; higher values make the class print debugging messages about what it's doing. excludes is a list of module names to exclude from the analysis. replace_paths is a list of (oldpath, newpath) tuples that will be replaced in module paths.

report()

Print a report to standard output that lists the modules imported by the script and their paths, as well as modules that are missing or seem to be missing.
run_script(pathname)

Analyze the contents of the pathname file, which must contain Python code.
29.5 runpy -- Locating and executing Python modules.

New in version 2.5.

The runpy module is used to locate and run Python modules without importing them first. Its main use is to implement the -m command line switch that allows scripts to be located using the Python module namespace rather than the filesystem.

When executed as a script, the module effectively operates as follows:

```python
del sys.argv[0]  # Remove the runpy module from the arguments
run_module(sys.argv[0], run_name="__main__", alter_sys=True)
```

The runpy module provides a single function:

```python
run_module(mod_name[, init_globals][, run_name][, alter_sys])
```

Execute the code of the specified module and return the resulting module globals dictionary. The module's code is first located using the standard import mechanism (refer to PEP 302 for details) and then executed in a fresh module namespace.

The optional dictionary argument init_globals may be used to pre-populate the globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by the run_module function.

The special global variables __name__, __file__, __loader__ and __builtins__ are set in the globals dictionary before the module code is executed.

__name__ is set to run_name if this optional argument is supplied, and the mod_name argument otherwise.
__loader__ is set to the PEP 302 module loader used to retrieve the code for the module (This loader may be a wrapper around the standard import mechanism).

__file__ is set to the name provided by the module loader. If the loader does not make filename information available, this variable is set to None.

__builtins__ is automatically initialised with a reference to the top level namespace of the __builtin__ module.

If the argument alter_sys is supplied and evaluates to True, then sys.argv[0] is updated with the value of __file__ and sys.modules[__name__] is updated with a temporary module object for the module being executed. Both sys.argv[0] and sys.modules[__name__] are restored to their original values before the function returns.

Note that this manipulation of sys is not thread-safe. Other threads may see the partially initialised module, as well as the altered list of arguments. It is recommended that the sys module be left alone when invoking this function from threaded code.

See Also:

PEP 338, Executing modules as scripts
PEP written and implemented by Nick Coghlan.
30. Python Language Services

Python provides a number of modules to assist in working with the Python language. These modules support tokenizing, parsing, syntax analysis, bytecode disassembly, and various other facilities.

These modules include:

- **parser**: Access parse trees for Python source code.
- **symbol**: Constants representing internal nodes of the parse tree.
- **token**: Constants representing terminal nodes of the parse tree.
- **keyword**: Test whether a string is a keyword in Python.
- **tokenizer**: Lexical scanner for Python source code.
- **tabnanny**: Tool for detecting white space related problems in Python source files in a directory tree.
- **pyclbr**: Supports information extraction for a Python class browser.
- **py_compile**: Compile Python source files to byte-code files.
- **compileall**: Tools for byte-compiling all Python source files in a directory tree.
- **dis**: Disassembler for Python byte code.
- **pickletools**: Contains extensive comments about the pickle protocols and pickle-machine opcodes, as well as some useful functions.
- **distutils**: Support for building and installing Python modules into an existing Python installation.
30.1 parser -- Access Python parse trees

The parser module provides an interface to Python's internal parser and byte-code compiler. The primary purpose for this interface is to allow Python code to edit the parse tree of a Python expression and create executable code from this. This is better than trying to parse and modify an arbitrary Python code fragment as a string because parsing is performed in a manner identical to the code forming the application. It is also faster.

There are a few things to note about this module which are important to making use of the data structures created. This is not a tutorial on editing the parse trees for Python code, but some examples of using the parser module are presented.

Most importantly, a good understanding of the Python grammar processed by the internal parser is required. For full information on the language syntax, refer to the Python Language Reference. The parser itself is created from a grammar specification defined in the file Grammar/Grammar in the standard Python distribution. The parse trees stored in the AST objects created by this module are the actual output from the internal parser when created by the expr() or suite() functions, described below. The AST objects created by sequence2ast() faithfully simulate those structures. Be aware that the values of the sequences which are considered "correct" will vary from one version of Python to another as the formal grammar for the language is revised. However, transporting code from one Python version to another as source text will always allow correct parse trees to be created in the target version, with the only restriction being that migrating to an older version of the interpreter will not support more recent language constructs. The parse trees are not typically compatible from one version to another, whereas source code has always been forward-compatible.

Each element of the sequences returned by ast2list() or ast2tuple() has a simple form. Sequences representing non-terminal elements in the grammar always have a length greater than one. The first element is an integer which identifies a production in the grammar. These integers are given symbolic names in the C header file Include/graminit.h and the Python module symbol1.
Each additional element of the sequence represents a component of the production as recognized in the input string: these are always sequences which have the same form as the parent. An important aspect of this structure which should be noted is that keywords used to identify the parent node type, such as the keyword `if` in an `if_stmt`, are included in the node tree without any special treatment. For example, the `if` keyword is represented by the tuple `(1, 'if')`, where 1 is the numeric value associated with all `NAME` tokens, including variable and function names defined by the user. In an alternate form returned when line number information is requested, the same token might be represented as `(1, 'if', 12)`, where the 12 represents the line number at which the terminal symbol was found.

Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the `if` keyword above is representative. The various types of terminal symbols are defined in the C header file `Include/token.h` and the Python module `token`.

The AST objects are not required to support the functionality of this module, but are provided for three purposes: to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python list or tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple ``wrapper`` class may be created in Python to hide the use of AST objects.

The parser module defines functions for a few distinct purposes. The most important purposes are to create AST objects and to convert AST objects to other representations such as parse trees and compiled code objects, but there are also functions which serve to query the type of parse tree represented by an AST object.

**See Also:**

Module `symbol`:
Useful constants representing internal nodes of the parse tree.

Module `token`:
Useful constants representing leaf nodes of the parse tree and functions for testing node values.

Subsections

- 30.1.1 Creating AST Objects
- 30.1.2 Converting AST Objects
- 30.1.3 Queries on AST Objects
- 30.1.4 Exceptions and Error Handling
- 30.1.5 AST Objects
- 30.1.6 Examples
  - 30.1.6.1 Emulation of compile()
  - 30.1.6.2 Information Discovery

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
30.1.1 Creating AST Objects

AST objects may be created from source code or from a parse tree. When creating an AST object from source, different functions are used to create the 'eval' and 'exec' forms.

expr(source)

The expr() function parses the parameter source as if it were an input to "compile(source, 'file.py', 'eval')". If the parse succeeds, an AST object is created to hold the internal parse tree representation, otherwise an appropriate exception is thrown.

suite(source)

The suite() function parses the parameter source as if it were an input to "compile(source, 'file.py', 'exec')". If the parse succeeds, an AST object is created to hold the internal parse tree representation, otherwise an appropriate exception is thrown.

sequence2ast(sequence)

This function accepts a parse tree represented as a sequence and builds an internal representation if possible. If it can validate that the tree conforms to the Python grammar and all nodes are valid node types in the host version of Python, an AST object is created from the internal representation and returned to the called. If there is a problem creating the internal representation, or if the tree cannot be validated, a ParserError exception is thrown. An AST object created this way should not be assumed to compile correctly; normal exceptions thrown by compilation may still be initiated when the AST object is passed to compileast(). This may indicate problems not related to syntax (such as a MemoryError exception), but may also be due to constructs such as the result of parsing del f(0), which escapes the Python parser but is checked by the bytecode compiler.

Sequences representing terminal tokens may be represented as either two-element lists of the form (1, 'name') or as three-element lists of the form (1, 'name', 56). If the third element is present, it is assumed to
be a valid line number. The line number may be specified for any subset of the terminal symbols in the input tree.

**tuple2ast**(*sequence*)

This is the same function as `sequence2ast()`. This entry point is maintained for backward compatibility.
30.1.2 Converting AST Objects

AST objects, regardless of the input used to create them, may be converted to parse trees represented as list- or tuple- trees, or may be compiled into executable code objects. Parse trees may be extracted with or without line numbering information.

\texttt{ast2list( ast[, line\_info])}

This function accepts an AST object from the caller in \texttt{ast} and returns a Python list representing the equivalent parse tree. The resulting list representation can be used for inspection or the creation of a new parse tree in list form. This function does not fail so long as memory is available to build the list representation. If the parse tree will only be used for inspection, \texttt{ast2tuple()} should be used instead to reduce memory consumption and fragmentation. When the list representation is required, this function is significantly faster than retrieving a tuple representation and converting that to nested lists.

If \texttt{line\_info} is true, line number information will be included for all terminal tokens as a third element of the list representing the token. Note that the line number provided specifies the line on which the token \textit{ends}. This information is omitted if the flag is false or omitted.

\texttt{ast2tuple( ast[, line\_info])}

This function accepts an AST object from the caller in \texttt{ast} and returns a Python tuple representing the equivalent parse tree. Other than returning a tuple instead of a list, this function is identical to \texttt{ast2list()}. If \texttt{line\_info} is true, line number information will be included for all terminal tokens as a third element of the list representing the token. This information is omitted if the flag is false or omitted.

\texttt{compileast( ast[, filename  =  '<ast>'])}

The Python byte compiler can be invoked on an AST object to produce code objects which can be used as part of an \texttt{exec} statement or a call to the built-in \texttt{eval()} function. This function provides the interface to the
compiler, passing the internal parse tree from \textit{ast} to the parser, using the source file name specified by the \textit{filename} parameter. The default value supplied for \textit{filename} indicates that the source was an AST object.

Compiling an AST object may result in exceptions related to compilation; an example would be a \texttt{SyntaxError} caused by the parse tree for \texttt{del f(\theta)}: this statement is considered legal within the formal grammar for Python but is not a legal language construct. The \texttt{SyntaxError} raised for this condition is actually generated by the Python byte-compiler normally, which is why it can be raised at this point by the \texttt{parser} module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.
30.1.3 Queries on AST Objects

Two functions are provided which allow an application to determine if an AST was created as an expression or a suite. Neither of these functions can be used to determine if an AST was created from source code via `expr()` or `suite()` or from a parse tree via `sequence2ast()`.

**isexpr (ast)**

When `ast` represents an 'eval' form, this function returns true, otherwise it returns false. This is useful, since code objects normally cannot be queried for this information using existing built-in functions. Note that the code objects created by `compileast()` cannot be queried like this either, and are identical to those created by the built-in `compile()` function.

**issuite (ast)**

This function mirrors `isexpr()` in that it reports whether an AST object represents an 'exec' form, commonly known as a ``suite.'' It is not safe to assume that this function is equivalent to "not isexpr(ast)", as additional syntactic fragments may be supported in the future.
30.1.4 Exceptions and Error Handling

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

**exception ParserError**

Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built in SyntaxError thrown during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a sequence causing the failure from a parse tree passed to sequence2ast() and an explanatory string. Calls to sequence2ast() need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.

Note that the functions compileast(), expr(), and suite() may throw exceptions which are normally thrown by the parsing and compilation process. These include the built in exceptions MemoryError, OverflowError, SyntaxError, and SystemError. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.
30.1.5 AST Objects

Ordered and equality comparisons are supported between AST objects. Pickling of AST objects (using the pickle module) is also supported.

ASTType
The type of the objects returned by expr(), suite() and sequence2ast().

AST objects have the following methods:

compile([filename])
Same as compileast(ast, filename).

isexpr()
Same as isexpr(ast).

issuite()
Same as issuite(ast).

tolist([line_info])
Same as ast2list(ast, line_info).

totuple([line_info])
Same as ast2tuple(ast, line_info).
30.1.6 Examples

The parser modules allows operations to be performed on the parse tree of Python source code before the bytecode is generated, and provides for inspection of the parse tree for information gathering purposes. Two examples are presented. The simple example demonstrates emulation of the `compile()` built-in function and the complex example shows the use of a parse tree for information discovery.

Subsections

- 30.1.6.1 Emulation of `compile()`
- 30.1.6.2 Information Discovery
30.2 symbol -- Constants used with Python parse trees

This module provides constants which represent the numeric values of internal nodes of the parse tree. Unlike most Python constants, these use lower-case names. Refer to the file Grammar/Grammar in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

This module also provides one additional data object:

**sym_name**

Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.

See Also:

Module **parser**: The second example for the **parser** module shows how to use the **symbol** module.

---

Release 2.5, documentation updated on 19th September, 2006. 
See **About this document...** for information on suggesting changes.
30.3 token -- Constants used with Python parse trees

This module provides constants which represent the numeric values of leaf nodes of the parse tree (terminal tokens). Refer to the file Grammar/Grammar in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

This module also provides one data object and some functions. The functions mirror definitions in the Python C header files.

**tok_name**
Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.

**ISTERMINAL**(x)
Return true for terminal token values.

**ISNONTERMINAL**(x)
Return true for non-terminal token values.

**ISEOF**(x)
Return true if x is the marker indicating the end of input.

### See Also:
Module [parser](#)
The second example for the [parser](#) module shows how to use the [symbol](#) module.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
30.4 keyword -- Testing for Python keywords

This module allows a Python program to determine if a string is a keyword.

**iskeyword**(*s*)

Return true if *s* is a Python keyword.

**kwlist**

Sequence containing all the keywords defined for the interpreter. If any keywords are defined to only be active when particular __future__ statements are in effect, these will be included as well.
30.5 tokenize -- Tokenizer for Python source

The `tokenize` module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing ``pretty-printers," including colorizers for on-screen displays.

The primary entry point is a generator:

```
generate_tokens( readline)
```

The `generate_tokens()` generator requires one argument, `readline`, which must be a callable object which provides the same interface as the `readline()` method of built-in file objects (see section 3.9). Each call to the function should return one line of input as a string.

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple (`srow`, `scol`) of ints specifying the row and column where the token begins in the source; a 2-tuple (`erow`, `ecol`) of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed is the `logical` line; continuation lines are included. New in version 2.2.

An older entry point is retained for backward compatibility:

```
tokenize( readline[, tokeneater])
```

The `tokenize()` function accepts two parameters: one representing the input stream, and one providing an output mechanism for `tokenize()`.

The first parameter, `readline`, must be a callable object which provides the same interface as the `readline()` method of built-in file objects (see section 3.9). Each call to the function should return one line of input as a string. Alternately, `readline` may be a callable object that signals completion by raising `StopIteration`. Changed in version 2.5: Added `StopIteration` support.
The second parameter, *tokeneater*, must also be a callable object. It is called once for each token, with five arguments, corresponding to the tuples generated by `generate_tokens()`.

All constants from the `token` module are also exported from `tokenize`, as are two additional token type values that might be passed to the `tokeneater` function by `tokenize()`:

**COMMENT**
Token value used to indicate a comment.

**NL**
Token value used to indicate a non-terminating newline. The NEWLINE token indicates the end of a logical line of Python code; NL tokens are generated when a logical line of code is continued over multiple physical lines.

Another function is provided to reverse the tokenization process. This is useful for creating tools that tokenize a script, modify the token stream, and write back the modified script.

**untokenize( iterable)**
Converts tokens back into Python source code. The *iterable* must return sequences with at least two elements, the token type and the token string. Any additional sequence elements are ignored.

The reconstructed script is returned as a single string. The result is guaranteed to tokenize back to match the input so that the conversion is lossless and round-trips are assured. The guarantee applies only to the token type and token string as the spacing between tokens (column positions) may change. New in version 2.5.

Example of a script re-writer that transforms float literals into Decimal objects:

```python
def decistmt(s):
    """Substitute Decimals for floats in a string of statements."

    >>> from decimal import Decimal
    >>> s = 'print +21.3e-5*- .1234/81.7'
    >>> decistmt(s)
    "print +Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7')"```
>>> exec(s)
-3.21716034272e-007
>>> exec(decistmt(s))
-3.217160342717258261933904529E-7

""
result = []
g = generate_tokens(StringIO(s).readline)  # tokenize the string
for toknum, tokval, _, _, _ in g:
    if toknum == NUMBER and '.' in tokval:  # replace NUMBER tok
        result.extend([
            (NAME, 'Decimal'),
            (OP, '('),
            (STRING, repr(tokval)),
            (OP, ')')
        ])  
else:
    result.append((toknum, tokval))
return untokenize(result)

---

Python Library Reference

Previous: 30.4 keyword Up: 30. Python Language Services Next: 30.6 tabnanny

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
30.6 tabnanny -- Detection of ambiguous indentation

For the time being this module is intended to be called as a script. However it is possible to import it into an IDE and use the function check() described below.

**Warning:** The API provided by this module is likely to change in future releases; such changes may not be backward compatible.

```python
def check(file_or_dir):
    if file_or_dir is a directory and not a symbolic link, then recursively descend the directory tree named by file_or_dir, checking all .py files along the way. If file_or_dir is an ordinary Python source file, it is checked for whitespace related problems. The diagnostic messages are written to standard output using the print statement.
```

**verbose**
Flag indicating whether to print verbose messages. This is incremented by the -v option if called as a script.

**filename_only**
Flag indicating whether to print only the filenames of files containing whitespace related problems. This is set to true by the -q option if called as a script.

**exception NannyNag**
Raised by token eater() if detecting an ambiguous indent. Captured and handled in check().

```python
def token eater(type, token, start, end, line):
    This function is used by check() as a callback parameter to the function tokenize.tokenize().
```
See Also:

Module **tokenize**:  
Lexical scanner for Python source code.
30.7 pyc1br -- Python class browser support

The pyc1br can be used to determine some limited information about the classes, methods and top-level functions defined in a module. The information provided is sufficient to implement a traditional three-pane class browser. The information is extracted from the source code rather than by importing the module, so this module is safe to use with untrusted source code. This restriction makes it impossible to use this module with modules not implemented in Python, including many standard and optional extension modules.

readmodule( module[, path])

Read a module and return a dictionary mapping class names to class descriptor objects. The parameter module should be the name of a module as a string; it may be the name of a module within a package. The path parameter should be a sequence, and is used to augment the value of sys.path, which is used to locate module source code.

readmodule_ex( module[, path])

Like readmodule(), but the returned dictionary, in addition to mapping class names to class descriptor objects, also maps top-level function names to function descriptor objects. Moreover, if the module being read is a package, the key '__path__' in the returned dictionary has as its value a list which contains the package search path.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
30.7.1 Class Descriptor Objects

The class descriptor objects used as values in the dictionary returned by readmodule() and readmodule_ex() provide the following data members:

**module**
The name of the module defining the class described by the class descriptor.

**name**
The name of the class.

**super**
A list of class descriptors which describe the immediate base classes of the class being described. Classes which are named as superclasses but which are not discoverable by readmodule() are listed as a string with the class name instead of class descriptors.

**methods**
A dictionary mapping method names to line numbers.

**file**
Name of the file containing the class statement defining the class.

**lineno**
The line number of the class statement within the file named by file.
30.7.2 Function Descriptor Objects

The function descriptor objects used as values in the dictionary returned by `readmodule_ex()` provide the following data members:

- **module**
  The name of the module defining the function described by the function descriptor.

- **name**
  The name of the function.

- **file**
  Name of the file containing the `def` statement defining the function.

- **lineno**
  The line number of the `def` statement within the file named by `file`.

Python Library Reference

Previous: 30.7.1 Class Descriptor Objects  Up: 30.7 pyclbr  Next: 30.8 py_compile

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
30.8 py_compile -- Compile Python source files

The py_compile module provides a function to generate a byte-code file from a source file, and another function used when the module source file is invoked as a script.

Though not often needed, this function can be useful when installing modules for shared use, especially if some of the users may not have permission to write the byte-code cache files in the directory containing the source code.

**exception PyCompileError**

Exception raised when an error occurs while attempting to compile the file.

**compile**(*file*, *cfile*, *dfile*, *doraise*)

Compile a source file to byte-code and write out the byte-code cache file. The source code is loaded from the file name `file`. The byte-code is written to `cfile`, which defaults to `file + 'c'` ("o" if optimization is enabled in the current interpreter). If `dfile` is specified, it is used as the name of the source file in error messages instead of `file`. If `doraise` is true, a PyCompileError is raised when an error is encountered while compiling `file`. If `doraise` is false (the default), an error string is written to `sys.stderr`, but no exception is raised.

**main**(*args*)

Compile several source files. The files named in `args` (or on the command line, if `args` is not specified) are compiled and the resulting bytecode is cached in the normal manner. This function does not search a directory structure to locate source files; it only compiles files named explicitly.

When this module is run as a script, the `main()` is used to compile all the files named on the command line.

**See Also:**
Module **compileall**: Utilities to compile all Python source files in a directory tree.
30.9 compileall -- Byte-compile Python libraries

This module provides some utility functions to support installing Python libraries. These functions compile Python source files in a directory tree, allowing users without permission to write to the libraries to take advantage of cached byte-code files.

The source file for this module may also be used as a script to compile Python sources in directories named on the command line or in sys.path.

**compile_dir** *(dir[, maxlevels[, ddir[, force[, rx[, quiet]]]]])*  
Recursively descend the directory tree named by *dir*, compiling all .py files along the way. The *maxlevels* parameter is used to limit the depth of the recursion; it defaults to 10. If *ddir* is given, it is used as the base path from which the filenames used in error messages will be generated. If *force* is true, modules are re-compiled even if the timestamps are up to date.

If *rx* is given, it specifies a regular expression of file names to exclude from the search; that expression is searched for in the full path.

If *quiet* is true, nothing is printed to the standard output in normal operation.

**compile_path** *( [skip_curdir[, maxlevels[, force]]] )*  
Byte-compile all the .py files found along sys.path. If *skip_curdir* is true (the default), the current directory is not included in the search. The *maxlevels* and *force* parameters default to 0 and are passed to the **compile_dir()** function.

To force a recompile of all the .py files in the Lib/ subdirectory and all its subdirectories:

```python
import compileall
compileall.compile_dir('Lib/', force=True)
```
# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile('/\.[.]svn'), force=True)

See Also:

Module **py_compile**:

Byte-compile a single source file.
30.10 dis -- Disassembler for Python byte code

The dis module supports the analysis of Python byte code by disassembling it. Since there is no Python assembler, this module defines the Python assembly language. The Python byte code which this module takes as an input is defined in the file Include/opcode.h and used by the compiler and the interpreter.

Example: Given the function myfunc:

```python
def myfunc(alist):
    return len(alist)
```

the following command can be used to get the disassembly of `myfunc()`:

```python
>>> dis.dis(myfunc)
  2          0 LOAD_GLOBAL              0 (len)
  3          0 LOAD_FAST               0 (alist)
  6 CALL_FUNCTION                      1
  9 RETURN_VALUE
```

(The ``2'' is a line number).

The dis module defines the following functions and constants:

```python
dis([bytesource])
```

Disassemble the bytesource object. bytesource can denote either a module, a class, a method, a function, or a code object. For a module, it disassembles all functions. For a class, it disassembles all methods. For a single code sequence, it prints one line per byte code instruction. If no object is provided, it disassembles the last traceback.

```python
distb([tb])
```

Disassembles the top-of-stack function of a traceback, using the last traceback if none was passed. The instruction causing the exception is indicated.
disassemble(code[, lasti])
Disassembles a code object, indicating the last instruction if lasti was provided. The output is divided in the following columns:

1. the line number, for the first instruction of each line
2. the current instruction, indicated as "\-\-\>",
3. a labelled instruction, indicated with "\>>",
4. the address of the instruction,
5. the operation code name,
6. operation parameters, and
7. interpretation of the parameters in parentheses.

The parameter interpretation recognizes local and global variable names, constant values, branch targets, and compare operators.

disco(code[, lasti])
A synonym for disassemble. It is more convenient to type, and kept for compatibility with earlier Python releases.

opname
Sequence of operation names, indexable using the byte code.

opmap
Dictionary mapping byte codes to operation names.

cmp_op
Sequence of all compare operation names.

hasconst
Sequence of byte codes that have a constant parameter.

hasfree
Sequence of byte codes that access a free variable.

hasname
Sequence of byte codes that access an attribute by name.

hasjrel
Sequence of byte codes that have a relative jump target.
hasjabs
Sequence of byte codes that have an absolute jump target.

haslocal
Sequence of byte codes that access a local variable.

hascompare
Sequence of byte codes of Boolean operations.

Subsections

- 30.10.1 Python Byte Code Instructions
30.10.1 Python Byte Code Instructions

The Python compiler currently generates the following byte code instructions.

**STOP_CODE**
Indicates end-of-code to the compiler, not used by the interpreter.

**NOP**
Do nothing code. Used as a placeholder by the bytecode optimizer.

**POP_TOP**
Removes the top-of-stack (TOS) item.

**ROT_TWO**
Swaps the two top-most stack items.

**ROT_THREE**
Lifts second and third stack item one position up, moves top down to position three.

**ROT_FOUR**
Lifts second, third and forth stack item one position up, moves top down to position four.

**DUP_TOP**
Duplicates the reference on top of the stack.

Unary Operations take the top of the stack, apply the operation, and push the result back on the stack.

**UNARY_POSITIVE**
Implements TOS = +TOS.

**UNARY_NEGATIVE**
Implements TOS = -TOS.

**UNARY_NOT**
Implements TOS = not TOS.
UNARY_CONVERT
   Implements TOS = `TOS`.

UNARY_INVERT
   Implements TOS = ~TOS.

GET_ITER
   Implements TOS = iter(TOS).

Binary operations remove the top of the stack (TOS) and the second top-most stack item (TOS1) from the stack. They perform the operation, and put the result back on the stack.

BINARY_POWER
   Implements TOS = TOS1 ** TOS.

BINARY_MULTIPLY
   Implements TOS = TOS1 * TOS.

BINARY_DIVIDE
   Implements TOS = TOS1 / TOS when from __future__ import division is not in effect.

BINARY_FLOOR_DIVIDE
   Implements TOS = TOS1 // TOS.

BINARY_TRUE_DIVIDE
   Implements TOS = TOS1 / TOS when from __future__ import division is in effect.

BINARY_MODULO
   Implements TOS = TOS1 % TOS.

BINARY_ADD
   Implements TOS = TOS1 + TOS.

BINARY_SUBTRACT
   Implements TOS = TOS1 - TOS.

BINARY_SUBSCR
Implements \( \text{TOS} = \text{TOS1}[\text{TOS}] \).

**BINARY_LSHIFT**
Implements \( \text{TOS} = \text{TOS1} \ll \text{TOS} \).

**BINARY_RSHIFT**
Implements \( \text{TOS} = \text{TOS1} \gg \text{TOS} \).

**BINARY_AND**
Implements \( \text{TOS} = \text{TOS1} & \text{TOS} \).

**BINARY_XOR**
Implements \( \text{TOS} = \text{TOS1} ^ \text{TOS} \).

**BINARY_OR**
Implements \( \text{TOS} = \text{TOS1} | \text{TOS} \).

In-place operations are like binary operations, in that they remove TOS and TOS1, and push the result back on the stack, but the operation is done in-place when TOS1 supports it, and the resulting TOS may be (but does not have to be) the original TOS1.

**INPLACE_POWER**
Implements in-place \( \text{TOS} = \text{TOS1} ** \text{TOS} \).

**INPLACE_MULTIPLY**
Implements in-place \( \text{TOS} = \text{TOS1} * \text{TOS} \).

**INPLACE_DIVIDE**
Implements in-place \( \text{TOS} = \text{TOS1} / \text{TOS} \) when \text{from \_\_future\_} \text{import \ division} \) is not in effect.

**INPLACE_FLOOR_DIVIDE**
Implements in-place \( \text{TOS} = \text{TOS1} // \text{TOS} \).

**INPLACE_TRUE_DIVIDE**
Implements in-place \( \text{TOS} = \text{TOS1} / \text{TOS} \) when \text{from \_\_future\_} \text{import \ division} \) is in effect.

**INPLACE_MODULO**
Implements in-place \( \text{TOS} = \text{TOS1} \mod \text{TOS} \).

**INPLACE_ADD**
Implements in-place \( \text{TOS} = \text{TOS1} + \text{TOS} \).

**INPLACE_SUBTRACT**
Implements in-place \( \text{TOS} = \text{TOS1} - \text{TOS} \).

**INPLACE_LSHIFT**
Implements in-place \( \text{TOS} = \text{TOS1} \ll \text{TOS} \).

**INPLACE_RSHIFT**
Implements in-place \( \text{TOS} = \text{TOS1} \gg \text{TOS} \).

**INPLACE_AND**
Implements in-place \( \text{TOS} = \text{TOS1} \& \text{TOS} \).

**INPLACE_XOR**
Implements in-place \( \text{TOS} = \text{TOS1} \^ \text{TOS} \).

**INPLACE_OR**
Implements in-place \( \text{TOS} = \text{TOS1} \mid \text{TOS} \).

The slice opcodes take up to three parameters.

**SLICE+0**
Implements \( \text{TOS} = \text{TOS}[\cdot:] \).

**SLICE+1**
Implements \( \text{TOS} = \text{TOS1}[\text{TOS}:] \).

**SLICE+2**
Implements \( \text{TOS} = \text{TOS1}[\cdot:\text{TOS}] \).

**SLICE+3**
Implements \( \text{TOS} = \text{TOS2}[\text{TOS1}:	ext{TOS}] \).

Slice assignment needs even an additional parameter. As any statement, they put nothing on the stack.
STORE_SLICE+0
  Implements TOS[:] = TOS1.

STORE_SLICE+1
  Implements TOS1[TOS:] = TOS2.

STORE_SLICE+2
  Implements TOS1[:TOS] = TOS2.

STORE_SLICE+3
  Implements TOS2[TOS1:TOS] = TOS3.

DELETE_SLICE+0
  Implements del TOS[:].

DELETE_SLICE+1
  Implements del TOS1[TOS:].

DELETE_SLICE+2
  Implements del TOS1[:TOS].

DELETE_SLICE+3
  Implements del TOS2[TOS1:TOS].

STORE_SUBSCR
  Implements TOS1[TOS] = TOS2.

DELETE_SUBSCR
  Implements del TOS1[TOS].

Miscellaneous opcodes.

PRINT_EXPR
  Implements the expression statement for the interactive mode. TOS is removed from the stack and printed. In non-interactive mode, an expression statement is terminated with POP_STACK.

PRINT_ITEM
  Prints TOS to the file-like object bound to sys.stdout. There is one such instruction for each item in the print statement.
PRINT_ITEM_TO
Like PRINT_ITEM, but prints the item second from TOS to the file-like object at TOS. This is used by the extended print statement.

PRINT_NEWLINE
Prints a new line on sys.stdout. This is generated as the last operation of a print statement, unless the statement ends with a comma.

PRINT_NEWLINE_TO
Like PRINT_NEWLINE, but prints the new line on the file-like object on the TOS. This is used by the extended print statement.

BREAK_LOOP
Terminates a loop due to a break statement.

CONTINUE_LOOP  target
Continues a loop due to a continue statement. target is the address to jump to (which should be a FOR_ITER instruction).

LIST_APPEND
Calls list.append(TOS1, TOS). Used to implement list comprehensions.

LOAD_LOCALS
Pushes a reference to the locals of the current scope on the stack. This is used in the code for a class definition: After the class body is evaluated, the locals are passed to the class definition.

RETURN_VALUE
Returns with TOS to the caller of the function.

YIELD_VALUE
Pops TOS and yields it from a generator.

IMPORT_STAR
Loads all symbols not starting with "_" directly from the module TOS to the local namespace. The module is popped after loading all names. This opcode implements from module import *.
**EXEC_STMT**
Implements `exec TOS2, TOS1, TOS`. The compiler fills missing optional parameters with `None`.

**POP_BLOCK**
Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.

**END_FINALLY**
Terminates a `finally` clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

**BUILD_CLASS**
Creates a new class object. TOS is the methods dictionary, TOS1 the tuple of the names of the base classes, and TOS2 the class name.

All of the following opcodes expect arguments. An argument is two bytes, with the more significant byte last.

**STORE_NAME**  `namei`
Implements `name = TOS`. `namei` is the index of `name` in the attribute `co_names` of the code object. The compiler tries to use `STORE_LOCAL` or `STORE_GLOBAL` if possible.

**DELETE_NAME**  `namei`
Implements `del name`, where `namei` is the index into `co_names` attribute of the code object.

**UNPACK_SEQUENCE**  `count`
Unpacks TOS into `count` individual values, which are put onto the stack right-to-left.

**DUP_TOPX**  `count`
Duplicate `count` items, keeping them in the same order. Due to implementation limits, `count` should be between 1 and 5 inclusive.

**STORE_ATTR**  `namei`
Implements `TOS.name = TOS1`, where `namei` is the index of name in
co_names.

**DELETE_ATTR**  *namei*
Implements `del TOS.name`, using `namei` as index into `co_names`.

**STORE_GLOBAL**  *namei*
Works as **STORE_NAME**, but stores the name as a global.

**DELETE_GLOBAL**  *namei*
Works as **DELETE_NAME**, but deletes a global name.

**LOAD_CONST**  *consti*
Pushes "co_consts[*consti]" onto the stack.

**LOAD_NAME**  *namei*
Pushes the value associated with "co_names[*namei]" onto the stack.

**BUILD_TUPLE**  *count*
Creates a tuple consuming *count* items from the stack, and pushes the resulting tuple onto the stack.

**BUILD_LIST**  *count*
Works as **BUILD_TUPLE**, but creates a list.

**BUILD_MAP**  *zero*
Pushes a new empty dictionary object onto the stack. The argument is ignored and set to zero by the compiler.

**LOAD_ATTR**  *namei*
Replaces TOS with `getattr(TOS, co_names[*namei])`.

**COMPARE_OP**  *opname*
Performs a Boolean operation. The operation name can be found in `cmp_op[*opname]`.

**IMPORT_NAME**  *namei*
Imports the module `co_names[*namei]`. The module object is pushed onto the stack. The current namespace is not affected: for a proper import statement, a subsequent **STORE_FAST** instruction modifies the namespace.
**IMPORT_FROM**  *namei*
Loads the attribute `co_names[*namei*]` from the module found in TOS. The resulting object is pushed onto the stack, to be subsequently stored by a `STORE_FAST` instruction.

**JUMP_FORWARD**  *delta*
Increments byte code counter by `delta`.

**JUMP_IF_TRUE**  *delta*
If TOS is true, increment the byte code counter by `delta`. TOS is left on the stack.

**JUMP_IF_FALSE**  *delta*
If TOS is false, increment the byte code counter by `delta`. TOS is not changed.

**JUMP_ABSOLUTE**  *target*
Set byte code counter to `target`.

**FOR_ITER**  *delta*
TOS is an iterator. Call its `next()` method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted TOS is popped, and the byte code counter is incremented by `delta`.

**LOAD_GLOBAL**  *namei*
 Loads the global named `co_names[*namei*]` onto the stack.

**SETUP_LOOP**  *delta*
Pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of `delta` bytes.

**SETUP_EXCEPT**  *delta*
Pushes a try block from a try-except clause onto the block stack. `delta` points to the first except block.

**SETUP_FINALLY**  *delta*
Pushes a try block from a try-except clause onto the block stack. `delta` points to the finally block.
LOAD_FAST  var_num
    Pushes a reference to the local co_varnames[var_num] onto the stack.

STORE_FAST  var_num
    Stores TOS into the local co_varnames[var_num].

DELETE_FAST  var_num
    Deletes local co_varnames[var_num].

LOAD_CLOSURE  i
    Pushes a reference to the cell contained in slot i of the cell and free variable storage. The name of the variable is co_cellvars[i] if i is less than the length of co_cellvars. Otherwise it is co_freevars[i - len(co_cellvars)].

LOAD_DEREF  i
    Loads the cell contained in slot i of the cell and free variable storage. Pushes a reference to the object the cell contains on the stack.

STORE_DEREF  i
    Stores TOS into the cell contained in slot i of the cell and free variable storage.

SET_LINENO  lineno
    This opcode is obsolete.

RAISE_VARARGS  argc
    Raises an exception. argc indicates the number of parameters to the raise statement, ranging from 0 to 3. The handler will find the traceback as TOS2, the parameter as TOS1, and the exception as TOS.

CALL_FUNCTION  argc
    Calls a function. The low byte of argc indicates the number of positional parameters, the high byte the number of keyword parameters. On the stack, the opcode finds the keyword parameters first. For each keyword argument, the value is on top of the key. Below the keyword parameters, the positional parameters are on the stack, with the right-most parameter on top. Below the parameters, the function object to call is on the stack.
MAKE_FUNCTION  argc
Pushes a new function object on the stack. TOS is the code associated with the function. The function object is defined to have argc default parameters, which are found below TOS.

MAKE_CLOSURE  argc
Creates a new function object, sets its func_closure slot, and pushes it on the stack. TOS is the code associated with the function. If the code object has N free variables, the next N items on the stack are the cells for these variables. The function also has argc default parameters, where are found before the cells.

BUILD_SLICE  argc
Pushes a slice object on the stack. argc must be 2 or 3. If it is 2, slice(TOS1, TOS) is pushed; if it is 3, slice(TOS2, TOS1, TOS) is pushed. See the slice() built-in function for more information.

EXTENDED_ARG  ext
Prefixes any opcode which has an argument too big to fit into the default two bytes. ext holds two additional bytes which, taken together with the subsequent opcode's argument, comprise a four-byte argument, ext being the two most-significant bytes.

CALL_FUNCTION_VAR  argc
Calls a function. argc is interpreted as in CALL_FUNCTION. The top element on the stack contains the variable argument list, followed by keyword and positional arguments.

CALL_FUNCTION_KW  argc
Calls a function. argc is interpreted as in CALL_FUNCTION. The top element on the stack contains the keyword arguments dictionary, followed by explicit keyword and positional arguments.

CALL_FUNCTION_VAR_KW  argc
Calls a function. argc is interpreted as in CALL_FUNCTION. The top element on the stack contains the keyword arguments dictionary, followed by the variable-arguments tuple, followed by explicit keyword and positional arguments.
**HAVE_ARGUMENT**

This is not really an opcode. It identifies the dividing line between opcodes which don't take arguments `< HAVE_ARGUMENT` and those which do `>= HAVE_ARGUMENT`. 

---

**Python Library Reference**

Previous: 30.10 dis Up: 30.10 dis Next: 30.11 pickletools

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
30.11 pickletools -- Tools for pickle developers.

New in version 2.3.

This module contains various constants relating to the intimate details of the pickle module, some lengthy comments about the implementation, and a few useful functions for analyzing pickled data. The contents of this module are useful for Python core developers who are working on the pickle and cPickle implementations; ordinary users of the pickle module probably won't find the pickletools module relevant.

dis(pickle[, out=None, memo=None, indentlevel=4])

Outputs a symbolic disassembly of the pickle to the file-like object out, defaulting to sys.stdout. pickle can be a string or a file-like object. memo can be a Python dictionary that will be used as the pickle's memo; it can be used to perform disassemblies across multiple pickles created by the same pickler. Successive levels, indicated by MARK opcodes in the stream, are indented by indentlevel spaces.

genops(pickle)

Provides an iterator over all of the opcodes in a pickle, returning a sequence of (opcode, arg, pos) triples. opcode is an instance of an OpcodeInfo class; arg is the decoded value, as a Python object, of the opcode's argument; pos is the position at which this opcode is located. pickle can be a string or a file-like object.
30.12 distutils -- Building and installing Python modules

The distutils package provides support for building and installing additional modules into a Python installation. The new modules may be either 100%-pure Python, or may be extension modules written in C, or may be collections of Python packages which include modules coded in both Python and C.

This package is discussed in two separate documents which are included in the Python documentation package. To learn about distributing new modules using the distutils facilities, read Distributing Python Modules; this includes documentation needed to extend distutils. To learn about installing Python modules, whether or not the author made use of the distutils package, read Installing Python Modules.

See Also:

Distributing Python Modules
   The manual for developers and packagers of Python modules. This describes how to prepare distutils-based packages so that they may be easily installed into an existing Python installation.

Installing Python Modules
   An `administrators` manual which includes information on installing modules into an existing Python installation. You do not need to be a Python programmer to read this manual.
See About this document... for information on suggesting changes.
31. Python compiler package

The Python compiler package is a tool for analyzing Python source code and generating Python bytecode. The compiler contains libraries to generate an abstract syntax tree from Python source code and to generate Python bytecode from the tree.

The \texttt{compiler} package is a Python source to bytecode translator written in Python. It uses the built-in parser and standard \texttt{parser} module to generated a concrete syntax tree. This tree is used to generate an abstract syntax tree (AST) and then Python bytecode.

The full functionality of the package duplicates the builtin compiler provided with the Python interpreter. It is intended to match its behavior almost exactly. Why implement another compiler that does the same thing? The package is useful for a variety of purposes. It can be modified more easily than the builtin compiler. The AST it generates is useful for analyzing Python source code.

This chapter explains how the various components of the \texttt{compiler} package work. It blends reference material with a tutorial.

The following modules are part of the \texttt{compiler} package:

- \texttt{compiler}
- \texttt{compiler.ast}
- \texttt{compiler.visitor}
31.1 The basic interface

The top-level of the package defines four functions. If you import `compiler`, you will get these functions and a collection of modules contained in the package.

`parse` \((buf)\)

Returns an abstract syntax tree for the Python source code in \(buf\). The function raises `SyntaxError` if there is an error in the source code. The return value is a `compiler.ast.Module` instance that contains the tree.

`parseFile` \((path)\)

Return an abstract syntax tree for the Python source code in the file specified by \(path\). It is equivalent to `parse(open(path).read())`.

`walk` \((ast, visitor[, verbose])\)

Do a pre-order walk over the abstract syntax tree \(ast\). Call the appropriate method on the \(visitor\) instance for each node encountered.

`compile` \((source, filename, mode, flags=\text{None}, dont_inherit=\text{None})\)

Compile the string \(source\), a Python module, statement or expression, into a code object that can be executed by the `exec` statement or `eval()` function. This function is a replacement for the built-in `compile()` function.

The \texttt{filename} will be used for run-time error messages.

The \texttt{mode} must be 'exec' to compile a module, 'single' to compile a single (interactive) statement, or 'eval' to compile an expression.

The \texttt{flags} and \texttt{dont_inherit} arguments affect future-related statements, but are not supported yet.

`compileFile` \((source)\)

Compiles the file \(source\) and generates a .pyc file.

The `compiler` package contains the following modules: \texttt{ast}, \texttt{consts},
future, misc, pyassem, pycodegen, symbols, transformer, and visitor.
31.2 Limitations

There are some problems with the error checking of the compiler package. The interpreter detects syntax errors in two distinct phases. One set of errors is detected by the interpreter's parser, the other set by the compiler. The compiler package relies on the interpreter's parser, so it gets the first phases of error checking for free. It implements the second phase itself, and that implementation is incomplete. For example, the compiler package does not raise an error if a name appears more than once in an argument list: `def f(x, x): ...`

A future version of the compiler should fix these problems.
31.3 Python Abstract Syntax

The compiler.ast module defines an abstract syntax for Python. In the abstract syntax tree, each node represents a syntactic construct. The root of the tree is Module object.

The abstract syntax offers a higher level interface to parsed Python source code. The parser module and the compiler written in C for the Python interpreter use a concrete syntax tree. The concrete syntax is tied closely to the grammar description used for the Python parser. Instead of a single node for a construct, there are often several levels of nested nodes that are introduced by Python's precedence rules.

The abstract syntax tree is created by the compiler.transformer module. The transformer relies on the builtin Python parser to generate a concrete syntax tree. It generates an abstract syntax tree from the concrete tree.

The transformer module was created by Greg Stein and Bill Tutt for an experimental Python-to-C compiler. The current version contains a number of modifications and improvements, but the basic form of the abstract syntax and of the transformer are due to Stein and Tutt.

Subsections

- 31.3.1 AST Nodes
- 31.3.2 Assignment nodes
- 31.3.3 Examples
31.3.1 AST Nodes

The `compiler.ast` module is generated from a text file that describes each node type and its elements. Each node type is represented as a class that inherits from the abstract base class `compiler.ast.Node` and defines a set of named attributes for child nodes.

```python
class Node()
```

The `Node` instances are created automatically by the parser generator. The recommended interface for specific `Node` instances is to use the public attributes to access child nodes. A public attribute may be bound to a single node or to a sequence of nodes, depending on the `Node` type. For example, the `bases` attribute of the `Class` node, is bound to a list of base class nodes, and the `doc` attribute is bound to a single node.

Each `Node` instance has a `lineno` attribute which may be `None`. XXX Not sure what the rules are for which nodes will have a useful lineno.

All `Node` objects offer the following methods:

```python
getChildren()
```

Returns a flattened list of the child nodes and objects in the order they occur. Specifically, the order of the nodes is the order in which they appear in the Python grammar. Not all of the children are `Node` instances. The names of functions and classes, for example, are plain strings.

```python
getChildNodes()
```

Returns a flattened list of the child nodes in the order they occur. This method is like `getChildren()`, except that it only returns those children that are `Node` instances.

Two examples illustrate the general structure of `Node` classes. The `while` statement is defined by the following grammar production:

```
while_stmt:   "while" expression ":" suite
["else" ":" suite]
```
The While node has three attributes: `test`, `body`, and `else_`. (If the natural name for an attribute is also a Python reserved word, it can't be used as an attribute name. An underscore is appended to the word to make it a legal identifier, hence `else_` instead of `else`.)

The if statement is more complicated because it can include several tests.

```python
def if_stmt: 'if' test ' ':' suite ('elif' test ':' suite)* ['else' ':' suite]
```

The If node only defines two attributes: `tests` and `else_`. The `tests` attribute is a sequence of test expression, consequent body pairs. There is one pair for each `if/elif` clause. The first element of the pair is the test expression. The second elements is a Stmt node that contains the code to execute if the test is true.

The `getChildren()` method of If returns a flat list of child nodes. If there are three `if/elif` clauses and no `else` clause, then `getChildren()` will return a list of six elements: the first test expression, the first Stmt, the second text expression, etc.

The following table lists each of the Node subclasses defined in `compiler.ast` and each of the public attributes available on their instances. The values of most of the attributes are themselves Node instances or sequences of instances. When the value is something other than an instance, the type is noted in the comment. The attributes are listed in the order in which they are returned by `getChildren()` and `getChildNodes()`.

<table>
<thead>
<tr>
<th>Node type</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td><code>left</code></td>
<td>left operand</td>
</tr>
<tr>
<td></td>
<td><code>right</code></td>
<td>right operand</td>
</tr>
<tr>
<td>And</td>
<td><code>nodes</code></td>
<td>list of operands</td>
</tr>
<tr>
<td>AssAttr</td>
<td><code>expr</code></td>
<td><code>attribute as target of assignment</code></td>
</tr>
<tr>
<td></td>
<td><code>attrname</code></td>
<td>expression on the left-hand side of the dot</td>
</tr>
<tr>
<td></td>
<td><code>flags</code></td>
<td>XXX</td>
</tr>
<tr>
<td>AssList</td>
<td><code>nodes</code></td>
<td>list of list elements being assigned to</td>
</tr>
<tr>
<td>AssName</td>
<td><code>name</code></td>
<td>name being assigned to</td>
</tr>
<tr>
<td>flags</td>
<td>XXX</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>AssTuple</td>
<td>nodes</td>
<td>list of tuple elements being assigned to</td>
</tr>
<tr>
<td>Assert</td>
<td>test</td>
<td>the expression to be tested</td>
</tr>
<tr>
<td></td>
<td>fail</td>
<td>the value of the AssertionError</td>
</tr>
<tr>
<td>Assign</td>
<td>nodes</td>
<td>a list of assignment targets, one per equal sign</td>
</tr>
<tr>
<td></td>
<td>expr</td>
<td>the value being assigned</td>
</tr>
<tr>
<td>AugAssign</td>
<td>node</td>
<td></td>
</tr>
<tr>
<td></td>
<td>op</td>
<td></td>
</tr>
<tr>
<td></td>
<td>expr</td>
<td></td>
</tr>
<tr>
<td>Backquote</td>
<td>expr</td>
<td></td>
</tr>
<tr>
<td>Bitand</td>
<td>nodes</td>
<td></td>
</tr>
<tr>
<td>Bitor</td>
<td>nodes</td>
<td></td>
</tr>
<tr>
<td>Bitxor</td>
<td>nodes</td>
<td></td>
</tr>
<tr>
<td>Break</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallFunc</td>
<td>node</td>
<td>expression for the callee</td>
</tr>
<tr>
<td></td>
<td>args</td>
<td>a list of arguments</td>
</tr>
<tr>
<td></td>
<td>star_args</td>
<td>the extended *-arg value</td>
</tr>
<tr>
<td></td>
<td>dstar_args</td>
<td>the extended **-arg value</td>
</tr>
<tr>
<td>Class</td>
<td>name</td>
<td>the name of the class, a string</td>
</tr>
<tr>
<td></td>
<td>bases</td>
<td>a list of base classes</td>
</tr>
<tr>
<td></td>
<td>doc</td>
<td>doc string, a string or None</td>
</tr>
<tr>
<td></td>
<td>code</td>
<td>the body of the class statement</td>
</tr>
<tr>
<td>Compare</td>
<td>expr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ops</td>
<td></td>
</tr>
<tr>
<td>Const</td>
<td>value</td>
<td></td>
</tr>
<tr>
<td>Continue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decorators</td>
<td>nodes</td>
<td>List of function decorator expressions</td>
</tr>
<tr>
<td>Dict</td>
<td>items</td>
<td></td>
</tr>
<tr>
<td>Discard</td>
<td>expr</td>
<td></td>
</tr>
<tr>
<td>Div</td>
<td>left</td>
<td></td>
</tr>
<tr>
<td></td>
<td>right</td>
<td></td>
</tr>
<tr>
<td>Ellipsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>node</td>
<td></td>
</tr>
<tr>
<td>Token</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Exec</td>
<td><code>expr</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>locals</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>globals</code></td>
<td></td>
</tr>
<tr>
<td>FloorDiv</td>
<td><code>left</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>right</code></td>
<td></td>
</tr>
<tr>
<td>For</td>
<td><code>assign</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>list</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>body</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>else_</code></td>
<td></td>
</tr>
<tr>
<td>From</td>
<td><code>modname</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>names</code></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td><code>decorators</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>Decorators</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>or None</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>name</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>name used in def, a string</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>argnames</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>list of argument names, as strings</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>defaults</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>list of default values</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>flags</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>xxx</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>doc</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>doc string, a string or None</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>code</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>the body of the function</code></td>
<td></td>
</tr>
<tr>
<td>GenExpr</td>
<td><code>code</code></td>
<td></td>
</tr>
<tr>
<td>GenExprFor</td>
<td><code>assign</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>iter</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>ifs</code></td>
<td></td>
</tr>
<tr>
<td>GenExprIf</td>
<td><code>test</code></td>
<td></td>
</tr>
<tr>
<td>GenExprInner</td>
<td><code>expr</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>quals</code></td>
<td></td>
</tr>
<tr>
<td>Getattr</td>
<td><code>expr</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>attrname</code></td>
<td></td>
</tr>
<tr>
<td>Global</td>
<td><code>names</code></td>
<td></td>
</tr>
<tr>
<td>If</td>
<td><code>tests</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>else_</code></td>
<td></td>
</tr>
<tr>
<td>Import</td>
<td><code>names</code></td>
<td></td>
</tr>
<tr>
<td>Invert</td>
<td><code>expr</code></td>
<td></td>
</tr>
<tr>
<td>Keyword</td>
<td><code>name</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>expr</code></td>
<td></td>
</tr>
<tr>
<td>Lambda</td>
<td>argnames</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>defaults</td>
<td></td>
</tr>
<tr>
<td></td>
<td>flags</td>
<td></td>
</tr>
<tr>
<td></td>
<td>code</td>
<td></td>
</tr>
<tr>
<td>LeftShift</td>
<td>left</td>
<td></td>
</tr>
<tr>
<td></td>
<td>right</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td>nodes</td>
<td></td>
</tr>
<tr>
<td>ListComp</td>
<td>expr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>qquals</td>
<td></td>
</tr>
<tr>
<td>ListCompFor</td>
<td>assign</td>
<td></td>
</tr>
<tr>
<td></td>
<td>list</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ifs</td>
<td></td>
</tr>
<tr>
<td>ListCompIf</td>
<td>test</td>
<td></td>
</tr>
<tr>
<td>Mod</td>
<td>left</td>
<td></td>
</tr>
<tr>
<td></td>
<td>right</td>
<td></td>
</tr>
<tr>
<td>Module</td>
<td>doc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>node</td>
<td></td>
</tr>
<tr>
<td></td>
<td>body of the module, a Stmt</td>
<td></td>
</tr>
<tr>
<td>Mul</td>
<td>left</td>
<td></td>
</tr>
<tr>
<td></td>
<td>right</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>Not</td>
<td>expr</td>
<td></td>
</tr>
<tr>
<td>Or</td>
<td>nodes</td>
<td></td>
</tr>
<tr>
<td>Pass</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power</td>
<td>left</td>
<td></td>
</tr>
<tr>
<td></td>
<td>right</td>
<td></td>
</tr>
<tr>
<td>Print</td>
<td>nodes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dest</td>
<td></td>
</tr>
<tr>
<td>Printnl</td>
<td>nodes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dest</td>
<td></td>
</tr>
<tr>
<td>Raise</td>
<td>expr1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>expr2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>expr3</td>
<td></td>
</tr>
<tr>
<td>Return</td>
<td>value</td>
<td></td>
</tr>
<tr>
<td>RightShift</td>
<td>left</td>
<td></td>
</tr>
<tr>
<td>Syntax</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------</td>
<td></td>
</tr>
<tr>
<td>Slice</td>
<td>expr, flags, lower, upper</td>
<td></td>
</tr>
<tr>
<td>Sliceobj</td>
<td>nodes, list of statements</td>
<td></td>
</tr>
<tr>
<td>Stmt</td>
<td>nodes</td>
<td></td>
</tr>
<tr>
<td>Sub</td>
<td>left, right</td>
<td></td>
</tr>
<tr>
<td>Subscript</td>
<td>expr, flags, subs</td>
<td></td>
</tr>
<tr>
<td>TryExcept</td>
<td>body, handlers, else_</td>
<td></td>
</tr>
<tr>
<td>TryFinally</td>
<td>body, final</td>
<td></td>
</tr>
<tr>
<td>Tuple</td>
<td>nodes</td>
<td></td>
</tr>
<tr>
<td>UnaryAdd</td>
<td>expr</td>
<td></td>
</tr>
<tr>
<td>UnarySub</td>
<td>expr</td>
<td></td>
</tr>
<tr>
<td>While</td>
<td>test, body, else_</td>
<td></td>
</tr>
<tr>
<td>With</td>
<td>expr, vars, body</td>
<td></td>
</tr>
<tr>
<td>Yield</td>
<td>value</td>
<td></td>
</tr>
</tbody>
</table>
31.3.2 Assignment nodes

There is a collection of nodes used to represent assignments. Each assignment statement in the source code becomes a single Assign node in the AST. The nodes attribute is a list that contains a node for each assignment target. This is necessary because assignment can be chained, e.g. \( a = b = 2 \). Each Node in the list will be one of the following classes: AssAttr, AssList, AssName, or AssTuple.

Each target assignment node will describe the kind of object being assigned to: AssName for a simple name, e.g. \( a = 1 \). AssAttr for an attribute assigned, e.g. \( a.x = 1 \). AssList and AssTuple for list and tuple expansion respectively, e.g. \( a, b, c = a\_tuple \).

The target assignment nodes also have a flags attribute that indicates whether the node is being used for assignment or in a delete statement. The AssName is also used to represent a delete statement, e.g. \( \text{del } x \).

When an expression contains several attribute references, an assignment or delete statement will contain only one AssAttr node - for the final attribute reference. The other attribute references will be represented as Getattr nodes in the expr attribute of the AssAttr instance.
31.3.3 Examples

This section shows several simple examples of ASTs for Python source code. The examples demonstrate how to use the `parse()` function, what the repr of an AST looks like, and how to access attributes of an AST node.

The first module defines a single function. Assume it is stored in `/tmp/doublelib.py`.

"""This is an example module.

This is the docstring.
"""

def double(x):
    "Return twice the argument"
    return x * 2

In the interactive interpreter session below, I have reformatted the long AST reprs for readability. The AST reprs use unqualified class names. If you want to create an instance from a repr, you must import the class names from the `compiler.ast` module.

```python
>>> import compiler
>>> mod = compiler.parseFile("/tmp/doublelib.py")
>>> mod
Module('This is an example module.

This is the docstring.
',
    Stmt([Function(None, 'double', ['x'], [], 0,
    'Return twice the argument',
    Stmt([Return(Mul((Name('x'), Const(2))))]))]))
>>> from compiler.ast import *
>>> Module('This is an example module.

This is the docstring.
',
    Stmt([Function(None, 'double', ['x'], [], 0,
    'Return twice the argument',
    Stmt([Return(Mul((Name('x'), Const(2))))]))]))
>>> mod.doc
'This is an example module.

This is the docstring.'
>>> for node in mod.node.nodes:
...     print node
...     print
...     Function(None, 'double', ['x'], [], 0, 'Return twice the argument',
        Stmt([Return(Mul((Name('x'), Const(2))))]))
```
>>> func = mod.node.nodes[0]
>>> func.code
Stmt([Return(Mul((Name('x'), Const(2))))])
31.4 Using Visitors to Walk ASTs

The visitor pattern is ... The compiler package uses a variant on the visitor pattern that takes advantage of Python's introspection features to eliminate the need for much of the visitor's infrastructure.

The classes being visited do not need to be programmed to accept visitors. The visitor need only define visit methods for classes it is specifically interested in; a default visit method can handle the rest.

XXX The magic visit() method for visitors.

\[
\text{walk}(\text{tree}, \text{visitor}[\text{, verbose}])
\]

class ASTVisitor()

The ASTVisitor is responsible for walking over the tree in the correct order. A walk begins with a call to preorder(). For each node, it checks the visitor argument to preorder() for a method named `visitNodeType,' where NodeType is the name of the node's class, e.g. for a While node a visitWhile() would be called. If the method exists, it is called with the node as its first argument.

The visitor method for a particular node type can control how child nodes are visited during the walk. The ASTVisitor modifies the visitor argument by adding a visit method to the visitor; this method can be used to visit a particular child node. If no visitor is found for a particular node type, the default() method is called.

ASTVisitor objects have the following methods:

XXX describe extra arguments

\[
\text{default}(\text{node}[\text{, ...}])
\]

\[
\text{dispatch}(\text{node}[\text{, ...}])
\]
preorder(\ tree, \ visitor)
31.5 Bytecode Generation

The code generator is a visitor that emits bytecodes. Each visit method can call the `emit()` method to emit a new bytecode. The basic code generator is specialized for modules, classes, and functions. An assembler converts that emitted instructions to the low-level bytecode format. It handles things like generator of constant lists of code objects and calculation of jump offsets.
32. Abstract Syntax Trees

New in version 2.5.

The _ast module helps Python applications to process trees of the Python abstract syntax grammar. The Python compiler currently provides read-only access to such trees, meaning that applications can only create a tree for a given piece of Python source code; generating byte code from a (potentially modified) tree is not supported. The abstract syntax itself might change with each Python release; this module helps to find out programmatically what the current grammar looks like.

An abstract syntax tree can be generated by passing _ast.PyCF_ONLY_AST as a flag to the compile builtin function. The result will be a tree of objects whose classes all inherit from _ast.AST.

The actual classes are derived from the Parser/Python.asdl file, which is reproduced below. There is one class defined for each left-hand side symbol in the abstract grammar (for example, _ast.stmt or _ast.expr). In addition, there is one class defined for each constructor on the right-hand side; these classes inherit from the classes for the left-hand side trees. For example, _ast.BinOp inherits from _ast.expr. For production rules with alternatives (aka "sums"), the left-hand side class is abstract: only instances of specific constructor nodes are ever created.

Each concrete class has an attribute _fields which gives the names of all child nodes.

Each instance of a concrete class has one attribute for each child node, of the type as defined in the grammar. For example, _ast.BinOp instances have an attribute left of type _ast.expr. Instances of _ast.expr and _ast.stmt subclasses also have lineno and col_offset attributes. The lineno is the line number of source text (1 indexed so the first line is line 1) and the col_offset is the utf8 byte offset of the first token that generated the node. The utf8 offset is recorded because the parser uses utf8 internally.

If these attributes are marked as optional in the grammar (using a question
mark), the value might be None. If the attributes can have zero-or-more values (marked with an asterisk), the values are represented as Python lists.

Subsections

- **32.1 Abstract Grammar**

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
32.1 Abstract Grammar

The module defines a string constant __version__ which is the decimal subversion revision number of the file shown below.

The abstract grammar is currently defined as follows:

```python
-- ASDL's five builtin types are identifier, int, string, object, bool
module Python version "$Revision: 43614 "$ {
    mod = Module(stmt* body)
    | Interactive(stmt* body)
    | Expression(expr body)
    -- not really an actual node but useful in Jython's type
    | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                        stmt* body, expr* decorators)
    | ClassDef(identifier name, expr* bases, stmt* body)
    | Return(expr? value)
    | Delete(expr* targets)
    | Assign(expr* targets, expr value)
    | AugAssign(expr target, operator op, expr value)
    -- not sure if bool is allowed, can always use int
    | Print(expr? dest, expr* values, bool nl)
    -- use 'orelse' because else is a keyword in target language
    | For(expr target, expr iter, stmt* body, stmt* orelse)
    | While(expr test, stmt* body, stmt* orelse)
    | If(expr test, stmt* body, stmt* orelse)
    | With(expr context_expr, expr? optional_vars, stmt* body)
    -- 'type' is a bad name
    | Raise(expr? type, expr? inst, expr? tback)
    | TryExcept(stmt* body, excepthandler* handlers, stmt*)
    | TryFinally(stmt* body, stmt* finalbody)
    | Assert(expr test, expr? msg)
    | Import(alias* names)
    | ImportFrom(identifier module, alias* names, int? level)
    -- Doesn't capture requirement that locals must be
    -- defined if globals is
```
-- still supports use as a function!
| Exec(expr body, expr? globals, expr? locals)

| Global(identifier* names)
| Expr(expr value)
| Pass | Break | Continue

-- XXX Jython will be different
-- col_offset is the byte offset in the utf8 string th
attributes (int lineno, int col_offset)

-- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
| BinOp(expr left, operator op, expr right)
| UnaryOp(unaryop op, expr operand)
| Lambda(arguments args, expr body)
| IfExp(expr test, expr body, expr orelse)
| Dict(expr* keys, expr* values)
| ListComp(expr elt, comprehension* generators)
| GeneratorExp(expr elt, comprehension* generators)

-- the grammar constrains where yield expressions can o
| Yield(expr? value)
-- need sequences for compare to distinguish between
-- x < 4 < 3 and (x < 4) < 3
| Compare(expr left, cmpop* ops, expr* comparators)
| Call(expr func, expr* args, keyword* keywords,
| expr? starargs, expr? kwargs)
| Repr(expr value)
| Num(object n) -- a number as a PyObject.
| Str(string s) -- need to specify raw, unicode, etc?
-- other literals? bools?

-- the following expression can appear in assignment co
| Attribute(expr value, identifier attr, expr_context ctx)
| Subscript(expr value, slice slice, expr_context ctx)
| Name(identifier id, expr_context ctx)
| List(expr* elts, expr_context ctx)
| Tuple(expr *elts, expr_context ctx)

-- col_offset is the byte offset in the utf8 string th
attributes (int lineno, int col_offset)

expr_context = Load | Store | Del | AugLoad | AugStore | Par

slice = Ellipsis | Slice(expr? lower, expr? upper, expr? step
| ExtSlice(slice* dims)
| Index(expr value)

boolop = And | Or

operator = Add | Sub | Mult | Div | Mod | Pow | LShift
| RShift | BitOr | BitXor | BitAnd | FloorDiv
unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In |

comprehension = (expr target, expr iter, expr* ifs)

-- not sure what to call the first argument for raise and ex
-- TODO(jhylton): Figure out if there is a better way to han
--         lineno and col_offset fields, particularly
--        ast is exposed to Python.
excepthandler = (expr? type, expr? name, stmt* body, int lin
        int col_offset)

arguments = (expr* args, identifier? vararg,
        identifier? kwarg, expr* defaults)

-- keyword arguments supplied to call
keyword = (identifier arg, expr value)

-- import name with optional 'as' alias.
alias = (identifier name, identifier? asname)

}
33. Miscellaneous Services

The modules described in this chapter provide miscellaneous services that are available in all Python versions. Here's an overview:

**formatter**

Generic output formatter and device interface.

---

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
33.1.1 The Formatter Interface
33.1 formatter -- Generic output formatting

This module supports two interface definitions, each with multiple implementations. The *formatter* interface is used by the *HTMLParser* class of the *htmllib* module, and the *writer* interface is required by the formatter interface.

Formatter objects transform an abstract flow of formatting events into specific output events on writer objects. Formatters manage several stack structures to allow various properties of a writer object to be changed and restored; writers need not be able to handle relative changes nor any sort of ``change back'' operation. Specific writer properties which may be controlled via formatter objects are horizontal alignment, font, and left margin indentations. A mechanism is provided which supports providing arbitrary, non-exclusive style settings to a writer as well. Additional interfaces facilitate formatting events which are not reversible, such as paragraph separation.

Writer objects encapsulate device interfaces. Abstract devices, such as file formats, are supported as well as physical devices. The provided implementations all work with abstract devices. The interface makes available mechanisms for setting the properties which formatter objects manage and inserting data into the output.

Subsections

- 33.1.1 The Formatter Interface
- 33.1.2 Formatter Implementations
- 33.1.3 The Writer Interface
- 33.1.4 Writer Implementations

Previous: 33. Miscellaneous Services  Up: 33. Miscellaneous Services  Next: 33.1.1 The Formatter Interface
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
Implementations
33.1.1 The Formatter Interface

Interfaces to create formatters are dependent on the specific formatter class being instantiated. The interfaces described below are the required interfaces which all formatters must support once initialized.

One data element is defined at the module level:

**AS_IS**
Value which can be used in the font specification passed to the `push_font()` method described below, or as the new value to any other `push_property()` method. Pushing the AS_IS value allows the corresponding `pop_property()` method to be called without having to track whether the property was changed.

The following attributes are defined for formatter instance objects:

**writer**
The writer instance with which the formatter interacts.

**end_paragraph**( `blanklines` )
Close any open paragraphs and insert at least `blanklines` before the next paragraph.

**add_line_break**( )
Add a hard line break if one does not already exist. This does not break the logical paragraph.

**add_hor_rule**( `*args`, `**kw` )
Insert a horizontal rule in the output. A hard break is inserted if there is data in the current paragraph, but the logical paragraph is not broken. The arguments and keywords are passed on to the writer's `send_line_break()` method.

**add_flow_data**( `data` )
Provide data which should be formatted with collapsed whitespace. White space from preceding and successive calls to
add_flowing_data() is considered as well when the whitespace collapse is performed. The data which is passed to this method is expected to be word-wrapped by the output device. Note that any word-wrapping still must be performed by the writer object due to the need to rely on device and font information.

add_literal_data(data)

Provide data which should be passed to the writer unchanged. Whitespace, including newline and tab characters, are considered legal in the value of data.

add_label_data(format, counter)

Insert a label which should be placed to the left of the current left margin. This should be used for constructing bulleted or numbered lists. If the format value is a string, it is interpreted as a format specification for counter, which should be an integer. The result of this formatting becomes the value of the label; if format is not a string it is used as the label value directly. The label value is passed as the only argument to the writer's send_label_data() method. Interpretation of non-string label values is dependent on the associated writer.

Format specifications are strings which, in combination with a counter value, are used to compute label values. Each character in the format string is copied to the label value, with some characters recognized to indicate a transform on the counter value. Specifically, the character "1" represents the counter value formatter as an Arabic number, the characters "A" and "a" represent alphabetic representations of the counter value in upper and lower case, respectively, and "I" and "i" represent the counter value in Roman numerals, in upper and lower case. Note that the alphabetic and roman transforms require that the counter value be greater than zero.

flush_softspace()

Send any pending whitespace buffered from a previous call to add_flowing_data() to the associated writer object. This should be called before any direct manipulation of the writer object.

push_alignment(align)

Push a new alignment setting onto the alignment stack. This may be AS_IS
if no change is desired. If the alignment value is changed from the previous setting, the writer's `new_alignment()` method is called with the `align` value.

**pop_alignment()**
Restore the previous alignment.

**push_font((size, italic, bold, teletype))**
Change some or all font properties of the writer object. Properties which are not set to `AS_IS` are set to the values passed in while others are maintained at their current settings. The writer's `new_font()` method is called with the fully resolved font specification.

**pop_font()**
Restore the previous font.

**push_margin(margin)**
Increase the number of left margin indentations by one, associating the logical tag `margin` with the new indentation. The initial margin level is 0. Changed values of the logical tag must be true values; false values other than `AS_IS` are not sufficient to change the margin.

**pop_margin()**
Restore the previous margin.

**push_style(*styles)**
Push any number of arbitrary style specifications. All styles are pushed onto the styles stack in order. A tuple representing the entire stack, including `AS_IS` values, is passed to the writer's `new_styles()` method.

**pop_style([n = 1])**
Pop the last n style specifications passed to `push_style()`. A tuple representing the revised stack, including `AS_IS` values, is passed to the writer's `new_styles()` method.

**set_spacing(spacing)**
Set the spacing style for the writer.
assert_line_data([flag = 1])

Inform the formatter that data has been added to the current paragraph out-of-band. This should be used when the writer has been manipulated directly. The optional flag argument can be set to false if the writer manipulations produced a hard line break at the end of the output.
33.1.2 Formatter Implementations

Two implementations of formatter objects are provided by this module. Most applications may use one of these classes without modification or subclassing.

```python
class NullFormatter([writer])
    A formatter which does nothing. If writer is omitted, a NullWriter instance is created. No methods of the writer are called by NullFormatter instances. Implementations should inherit from this class if implementing a writer interface but don't need to inherit any implementation.
```

class AbstractFormatter(writer)
    The standard formatter. This implementation has demonstrated wide applicability to many writers, and may be used directly in most circumstances. It has been used to implement a full-featured World Wide Web browser.
```
33.1.3 The Writer Interface

Interfaces to create writers are dependent on the specific writer class being instantiated. The interfaces described below are the required interfaces which all writers must support once initialized. Note that while most applications can use the AbstractFormatter class as a formatter, the writer must typically be provided by the application.

`flush()`
Flush any buffered output or device control events.

`new_alignment(align)`
Set the alignment style. The `align` value can be any object, but by convention is a string or `None`, where `None` indicates that the writer's `"preferred"` alignment should be used. Conventional `align` values are 'left', 'center', 'right', and 'justify'.

`new_font(font)`
Set the font style. The value of `font` will be `None`, indicating that the device's default font should be used, or a tuple of the form `(size, italic, bold, teletype)`. Size will be a string indicating the size of font that should be used; specific strings and their interpretation must be defined by the application. The `italic`, `bold`, and `teletype` values are Boolean values specifying which of those font attributes should be used.

`new_margin(margin, level)`
Set the margin level to the integer `level` and the logical tag to `margin`. Interpretation of the logical tag is at the writer's discretion; the only restriction on the value of the logical tag is that it not be a false value for non-zero values of `level`.

`new_spacing(spacing)`
Set the spacing style to `spacing`.

`new_styles(styles)`
Set additional styles. The `styles` value is a tuple of arbitrary values; the
value AS_IS should be ignored. The styles tuple may be interpreted either as a set or as a stack depending on the requirements of the application and writer implementation.

**send_line_break()**
Break the current line.

**send_paragraph( blankline)**
Produce a paragraph separation of at least blankline blank lines, or the equivalent. The blankline value will be an integer. Note that the implementation will receive a call to send_line_break() before this call if a line break is needed; this method should not include ending the last line of the paragraph. It is only responsible for vertical spacing between paragraphs.

**send_hor_rule( *args, **kw)**
Display a horizontal rule on the output device. The arguments to this method are entirely application- and writer-specific, and should be interpreted with care. The method implementation may assume that a line break has already been issued via send_line_break().

**send_flowing_data( data)**
Output character data which may be word-wrapped and re-flowed as needed. Within any sequence of calls to this method, the writer may assume that spans of multiple whitespace characters have been collapsed to single space characters.

**send_literal_data( data)**
Output character data which has already been formatted for display. Generally, this should be interpreted to mean that line breaks indicated by newline characters should be preserved and no new line breaks should be introduced. The data may contain embedded newline and tab characters, unlike data provided to the send_formatted_data() interface.

**send_label_data( data)**
Set data to the left of the current left margin, if possible. The value of data is not restricted; treatment of non-string values is entirely application- and
writer-dependent. This method will only be called at the beginning of a line.
33.1.4 Writer Implementations

Three implementations of the writer object interface are provided as examples by this module. Most applications will need to derive new writer classes from the NullWriter class.

**class NullWriter()**

A writer which only provides the interface definition; no actions are taken on any methods. This should be the base class for all writers which do not need to inherit any implementation methods.

**class AbstractWriter()**

A writer which can be used in debugging formatters, but not much else. Each method simply announces itself by printing its name and arguments on standard output.

**class DumbWriter([file, maxcol = 72])**

Simple writer class which writes output on the file object passed in as *file* or, if *file* is omitted, on standard output. The output is simply word-wrapped to the number of columns specified by *maxcol*. This class is suitable for reflowing a sequence of paragraphs.
34. SGI IRIX Specific Services

The modules described in this chapter provide interfaces to features that are unique to SGI's IRIX operating system (versions 4 and 5).

**al**  Audio functions on the SGI.

**AL**  Constants used with the `al` module.

**cd**  Interface to the CD-ROM on Silicon Graphics systems.

**f1**  FORMS library for applications with graphical user interfaces.

**FL**  Constants used with the `f1` module.

**flp**  Functions for loading stored FORMS designs.

**fm**  *Font Manager* interface for SGI workstations.

**gl**  Functions from the Silicon Graphics *Graphics Library*.

**DEVICE**  Constants used with the `gl` module.

**GL**  Constants used with the `gl` module.

**imgfile**  Support for SGI imglib files.

**jpeg**  Read and write image files in compressed JPEG format.
34.1 al -- Audio functions on the SGI

Availability: IRIX.

This module provides access to the audio facilities of the SGI Indy and Indigo workstations. See section 3A of the IRIX man pages for details. You'll need to read those man pages to understand what these functions do! Some of the functions are not available in IRIX releases before 4.0.5. Again, see the manual to check whether a specific function is available on your platform.

All functions and methods defined in this module are equivalent to the C functions with "AL" prefixed to their name.

Symbolic constants from the C header file <audio.h> are defined in the standard module AL, see below.

**Warning:** The current version of the audio library may dump core when bad argument values are passed rather than returning an error status. Unfortunately, since the precise circumstances under which this may happen are undocumented and hard to check, the Python interface can provide no protection against this kind of problems. (One example is specifying an excessive queue size -- there is no documented upper limit.)

The module defines the following functions:

**openport**( name, direction[, config])

The name and direction arguments are strings. The optional config argument is a configuration object as returned by newconfig(). The return value is an audio port object; methods of audio port objects are described below.

**newconfig()**

The return value is a new audio configuration object; methods of audio configuration objects are described below.

**queryparams**( device)
The device argument is an integer. The return value is a list of integers containing the data returned by \texttt{ALqueryparams()}.

\textbf{getparams}( \textit{device, list} )

The \textit{device} argument is an integer. The list argument is a list such as returned by \texttt{queryparams()}; it is modified in place (!).

\textbf{setparams}( \textit{device, list} )

The \textit{device} argument is an integer. The \textit{list} argument is a list such as returned by \texttt{queryparams()}.
34.1.1 Configuration Objects

Configuration objects returned by `newconfig()` have the following methods:

`getqueue(size)`
Return the queue size.

`setqueue(size)`
Set the queue size.

`getwidth()`
Get the sample width.

`setwidth(width)`
Set the sample width.

`getchannels()`
Get the channel count.

`setchannels(nchannels)`
Set the channel count.

`getsampfmt()`
Get the sample format.

`setsampfmt(sampfmt)`
Set the sample format.

`getfloatmax()`
Get the maximum value for floating sample formats.

`setfloatmax(floatmax)`
Set the maximum value for floating sample formats.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
34.1.2 Port Objects

Port objects, as returned by openport(), have the following methods:

**closeport()**
Close the port.

**getfd()**
Return the file descriptor as an int.

**getfilled()**
Return the number of filled samples.

**getfillable()**
Return the number of fillable samples.

**readsamps( nsamples)**
Read a number of samples from the queue, blocking if necessary. Return the data as a string containing the raw data, (e.g., 2 bytes per sample in big-endian byte order (high byte, low byte) if you have set the sample width to 2 bytes).

**writesamps( samples)**
Write samples into the queue, blocking if necessary. The samples are encoded as described for the readsamps() return value.

**getfillpoint()**
Return the `fill point'.

**setfillpoint( fillpoint)**
Set the `fill point'.

**getconfig()**
Return a configuration object containing the current configuration of the port.
setconfig( *config*)
   Set the configuration from the argument, a configuration object.

getstatus( *list*)
   Get status information on last error.
34.2 AL -- Constants used with the al module

Availability: IRIX.

This module defines symbolic constants needed to use the built-in module al (see above); they are equivalent to those defined in the C header file <audio.h> except that the name prefix "AL_" is omitted. Read the module source for a complete list of the defined names. Suggested use:

```python
import al
from AL import *
```

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
34.3 cd -- CD-ROM access on SGI systems

Availability: **IRIX**.

This module provides an interface to the Silicon Graphics CD library. It is available only on Silicon Graphics systems.

The way the library works is as follows. A program opens the CD-ROM device with `open()` and creates a parser to parse the data from the CD with `createparser()`. The object returned by `open()` can be used to read data from the CD, but also to get status information for the CD-ROM device, and to get information about the CD, such as the table of contents. Data from the CD is passed to the parser, which parses the frames, and calls any callback functions that have previously been added.

An audio CD is divided into *tracks* or *programs* (the terms are used interchangeably). Tracks can be subdivided into *indices*. An audio CD contains a *table of contents* which gives the starts of the tracks on the CD. Index 0 is usually the pause before the start of a track. The start of the track as given by the table of contents is normally the start of index 1.

Positions on a CD can be represented in two ways. Either a frame number or a tuple of three values, minutes, seconds and frames. Most functions use the latter representation. Positions can be both relative to the beginning of the CD, and to the beginning of the track.

Module cd defines the following functions and constants:

**createparser()**
Create and return an opaque parser object. The methods of the parser object are described below.

**msftoframe(minutes, seconds, frames)**
Converts a *minutes, seconds, frames* tuple representing time in absolute time code into the corresponding CD frame number.
open([device[, mode]])
Open the CD-ROM device. The return value is an opaque player object; methods of the player object are described below. The device is the name of the SCSI device file, e.g. '/dev/scsi/sc0d4l0', or None. If omitted or None, the hardware inventory is consulted to locate a CD-ROM drive. The mode, if not omitted, should be the string 'r'.

The module defines the following variables:

**exception error**
Exception raised on various errors.

**DATASIZE**
The size of one frame's worth of audio data. This is the size of the audio data as passed to the callback of type audio.

**BLOCKSIZE**
The size of one uninterpreted frame of audio data.

The following variables are states as returned by getstatus():

**READY**
The drive is ready for operation loaded with an audio CD.

**NODISC**
The drive does not have a CD loaded.

**CDROM**
The drive is loaded with a CD-ROM. Subsequent play or read operations will return I/O errors.

**ERROR**
An error occurred while trying to read the disc or its table of contents.

**PLAYING**
The drive is in CD player mode playing an audio CD through its audio jacks.

**PAUSED**
The drive is in CD layer mode with play paused.

STILL

The equivalent of PAUSED on older (non 3301) model Toshiba CD-ROM drives. Such drives have never been shipped by SGI.

audio
pnum
index
ptime
atime
catalog
ident
control

Integer constants describing the various types of parser callbacks that can be set by the addcallback() method of CD parser objects (see below).

Subsections

- 34.3.1 Player Objects
- 34.3.2 Parser Objects
34.3.1 Player Objects

Player objects (returned by `open()`) have the following methods:

**allowremoval()**
Unlocks the eject button on the CD-ROM drive permitting the user to eject the caddy if desired.

**bestreadsize()**
Returns the best value to use for the `num_frames` parameter of the `reada()` method. Best is defined as the value that permits a continuous flow of data from the CD-ROM drive.

**close()**
Frees the resources associated with the player object. After calling `close()`, the methods of the object should no longer be used.

**eject()**
Ejects the caddy from the CD-ROM drive.

**getstatus()**
Returns information pertaining to the current state of the CD-ROM drive. The returned information is a tuple with the following values: `state`, `track`, `rtime`, `atime`, `ttme`, `first`, `last`, `scsi_audio`, `cur_block`. `rtime` is the time relative to the start of the current track; `atime` is the time relative to the beginning of the disc; `ttme` is the total time on the disc. For more information on the meaning of the values, see the man page `CDgetstatus(3dm)`. The value of `state` is one of the following: `ERROR`, `NODISC`, `READY`, `PLAYING`, `PAUSED`, `STILL`, or `CDROM`.

**gettrackinfo( track)**
Returns information about the specified track. The returned information is a tuple consisting of two elements, the start time of the track and the duration of the track.

**msftoblock( min, sec, frame)**
Converts a minutes, seconds, frames triple representing a time in absolute time code into the corresponding logical block number for the given CD-ROM drive. You should use msftostream() rather than msftoblock() for comparing times. The logical block number differs from the frame number by an offset required by certain CD-ROM drives.

**play**(start, play)
Starts playback of an audio CD in the CD-ROM drive at the specified track. The audio output appears on the CD-ROM drive's headphone and audio jacks (if fitted). Play stops at the end of the disc. start is the number of the track at which to start playing the CD; if play is 0, the CD will be set to an initial paused state. The method togglepause() can then be used to commence play.

**playabs**(minutes, seconds, frames, play)
Like play(), except that the start is given in minutes, seconds, and frames instead of a track number.

**playtrack**(start, play)
Like play(), except that playing stops at the end of the track.

**playtrackabs**(track, minutes, seconds, frames, play)
Like play(), except that playing begins at the specified absolute time and ends at the end of the specified track.

**preventremoval()**
Locks the eject button on the CD-ROM drive thus preventing the user from arbitrarily ejecting the caddy.

**readda**(num_frames)
Reads the specified number of frames from an audio CD mounted in the CD-ROM drive. The return value is a string representing the audio frames. This string can be passed unaltered to the parseframe() method of the parser object.

**seek**(minutes, seconds, frames)
Sets the pointer that indicates the starting point of the next read of digital
audio data from a CD-ROM. The pointer is set to an absolute time code location specified in minutes, seconds, and frames. The return value is the logical block number to which the pointer has been set.

**seekblock**(*block*)

Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified logical block number. The return value is the logical block number to which the pointer has been set.

**seektrack**(*track*)

Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified track. The return value is the logical block number to which the pointer has been set.

**stop**()

Stops the current playing operation.

**togglepause**()

Pauses the CD if it is playing, and makes it play if it is paused.
34.3.2 Parser Objects

Parser objects (returned by `createparser()`) have the following methods:

**addcallback** *(type, func, arg)*

Adds a callback for the parser. The parser has callbacks for eight different types of data in the digital audio data stream. Constants for these types are defined at the cd module level (see above). The callback is called as follows: `func(arg, type, data)`, where *arg* is the user supplied argument, *type* is the particular type of callback, and *data* is the data returned for this *type* of callback. The type of the data depends on the *type* of callback as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>audio</td>
<td>String which can be passed unmodified to <code>al.writesamps()</code></td>
</tr>
<tr>
<td>pnum</td>
<td>Integer giving the program (track) number.</td>
</tr>
<tr>
<td>index</td>
<td>Integer giving the index number.</td>
</tr>
<tr>
<td>ptime</td>
<td>Tuple consisting of the program time in minutes, seconds, and frames.</td>
</tr>
<tr>
<td>atime</td>
<td>Tuple consisting of the absolute time in minutes, seconds, and frames.</td>
</tr>
<tr>
<td>catalog</td>
<td>String of 13 characters, giving the catalog number of the CD.</td>
</tr>
<tr>
<td>ident</td>
<td>String of 12 characters, giving the ISRC identification number of the recording. The string consists of two characters country code, three characters owner code, two characters giving the year, and five characters giving a serial number.</td>
</tr>
<tr>
<td>control</td>
<td>Integer giving the control bits from the CD subcode data</td>
</tr>
</tbody>
</table>

**deletparser()**

Deletes the parser and frees the memory it was using. The object should not be used after this call. This call is done automatically when the last reference to the object is removed.
**parseframe**(*frame*)

Parses one or more frames of digital audio data from a CD such as returned by `readda()` . It determines which subcodes are present in the data. If these subcodes have changed since the last frame, then `parseframe()` executes a callback of the appropriate type passing to it the subcode data found in the frame. Unlike the C function, more than one frame of digital audio data can be passed to this method.

**removecallback**(*type*)

Removes the callback for the given *type*.

**resetparser**()

Resets the fields of the parser used for tracking subcodes to an initial state. `resetparser()` should be called after the disc has been changed.
34.4 f1 -- FORMS library for graphical user interfaces

Availability: IRIX.

This module provides an interface to the FORMS Library by Mark Overmars. The source for the library can be retrieved by anonymous ftp from host "ftp.cs.ruu.nl", directory SGI/FORMS. It was last tested with version 2.0b.

Most functions are literal translations of their C equivalents, dropping the initial "f1_" from their name. Constants used by the library are defined in module f1_described below.

The creation of objects is a little different in Python than in C: instead of the `current form' maintained by the library to which new FORMS objects are added, all functions that add a FORMS object to a form are methods of the Python object representing the form. Consequently, there are no Python equivalents for the C functions f1_addto_form() and f1_end_form(), and the equivalent of f1_bgn_form() is called f1_make_form().

Watch out for the somewhat confusing terminology: FORMS uses the word object for the buttons, sliders etc. that you can place in a form. In Python, `object' means any value. The Python interface to FORMS introduces two new Python object types: form objects (representing an entire form) and FORMS objects (representing one button, slider etc.). Hopefully this isn't too confusing.

There are no `free objects' in the Python interface to FORMS, nor is there an easy way to add object classes written in Python. The FORMS interface to GL event handling is available, though, so you can mix FORMS with pure GL windows.

Please note: importing f1 implies a call to the GL function foreground() and to the FORMS routine f1_init().

Subsections
- **34.4.1 Functions Defined in Module \texttt{f1}**
- **34.4.2 Form Objects**
- **34.4.3 FORMS Objects**

*Python Library Reference*

*Previous: 34.3.2 Parser Objects  Up: 34. SGI IRIX Specific  Next: 34.4.1 Functions Defined in*

*Release 2.5, documentation updated on 19th September, 2006.*
*See About this document... for information on suggesting changes.*
34.4.1 Functions Defined in Module fl

Module fl defines the following functions. For more information about what they do, see the description of the equivalent C function in the FORMS documentation:

**make_form**(type, width, height)
Create a form with given type, width and height. This returns a form object, whose methods are described below.

**do_forms()**
The standard FORMS main loop. Returns a Python object representing the FORMS object needing interaction, or the special value FL.EVENT.

**check_forms()**
Check for FORMS events. Returns what do_forms() above returns, or None if there is no event that immediately needs interaction.

**set_event_call_back**(function)
Set the event callback function.

**set_graphics_mode**(rgbmode, doublebuffering)
Set the graphics modes.

**get_rgbmode()**
Return the current rgb mode. This is the value of the C global variable fl_rgbmode.

**show_message**(str1, str2, str3)
Show a dialog box with a three-line message and an OK button.

**show_question**(str1, str2, str3)
Show a dialog box with a three-line message and YES and NO buttons. It returns 1 if the user pressed YES, 0 if NO.
**show_choice**(*str1, str2, str3, but1[, but2[, but3]])

Show a dialog box with a three-line message and up to three buttons. It returns the number of the button clicked by the user (1, 2 or 3).

**show_input**(*prompt, default*)

Show a dialog box with a one-line prompt message and text field in which the user can enter a string. The second argument is the default input string. It returns the string value as edited by the user.

**show_file_selector**(*message, directory, pattern, default*)

Show a dialog box in which the user can select a file. It returns the absolute filename selected by the user, or None if the user presses Cancel.

**get_directory**()

**get_pattern**()

**get_filename**()

These functions return the directory, pattern and filename (the tail part only) selected by the user in the last **show_file_selector**() call.

**qdevice**(*dev*)

**unqdevice**(*dev*)

**isqueued**(*dev*)

**qtest**(*dev*)

**qread**()

**qreset**()

**qenter**(*dev, val*)

**get_mouse**()

**tie**(*button, valuator1, valuator2*)

These functions are the FORMS interfaces to the corresponding GL functions. Use these if you want to handle some GL events yourself when using **fl.do_events()**. When a GL event is detected that FORMS cannot handle, **fl.do_forms()** returns the special value **FL.EVENT** and you should call **fl.qread()** to read the event from the queue. Don't use the equivalent GL functions!
color()
mapcolor()
getmcolor()

See the description in the FORMS documentation of `fl_color()`, `fl_mapcolor()` and `fl_getmcolor()`.
34.4.2 Form Objects

Form objects (returned by `make_form()` above) have the following methods. Each method corresponds to a C function whose name is prefixed with "fl_"; and whose first argument is a form pointer; please refer to the official FORMS documentation for descriptions.

All the `add_*()` methods return a Python object representing the FORMS object. Methods of FORMS objects are described below. Most kinds of FORMS object also have some methods specific to that kind; these methods are listed here.

```python
show_form(placement, bordertype, name)
    Show the form.

hide_form()
    Hide the form.

redraw_form()
    Redraw the form.

set_form_position(x, y)
    Set the form's position.

freeze_form()
    Freeze the form.

unfreeze_form()
    Unfreeze the form.

activate_form()
    Activate the form.

deactivate_form()
    Deactivate the form.
```
bgn_group()
Begin a new group of objects; return a group object.

del_group()
End the current group of objects.

find_first()
Find the first object in the form.

find_last()
Find the last object in the form.

add_box( type, x, y, w, h, name)
Add a box object to the form. No extra methods.

add_text( type, x, y, w, h, name)
Add a text object to the form. No extra methods.

add_clock( type, x, y, w, h, name)
Add a clock object to the form.
Method: get_clock().

add_button( type, x, y, w, h, name)
Add a button object to the form.
Methods: get_button(), set_button().

add_lightbutton( type, x, y, w, h, name)
Add a lightbutton object to the form.
Methods: get_button(), set_button().

add_roundbutton( type, x, y, w, h, name)
Add a roundbutton object to the form.
Methods: get_button(), set_button().

add_slider( type, x, y, w, h, name)
Add a slider object to the form.
Methods: set_slider_value(), get_slider_value(),
set_slider_bounds(), get_slider_bounds(),
set_slider_return(), set_slider_size(),
set_slider_precision(), set_slider_step().

add_valslider( type, x, y, w, h, name)
Add a valslider object to the form.
Methods: set_slider_value(), get_slider_value(),
set_slider_bounds(), get_slider_bounds(),
set_slider_return(), set_slider_size(),
set_slider_precision(), set_slider_step().

add_dial( type, x, y, w, h, name)
Add a dial object to the form.
Methods: set_dial_value(), get_dial_value(),
set_dial_bounds(), get_dial_bounds().

add_positioner( type, x, y, w, h, name)
Add a positioner object to the form.
Methods: set_positioner_xvalue(),
set_positioner_yvalue(), set_positioner_xbounds(),
set_positioner_ybounds(), get_positioner_xvalue(),
get_positioner_yvalue(), get_positioner_xbounds(),
get_positioner_ybounds().

add_counter( type, x, y, w, h, name)
Add a counter object to the form.
Methods: set_counter_value(), get_counter_value(),
set_counter_bounds(), set_counter_step(),
set_counter_precision(), set_counter_return().

add_input( type, x, y, w, h, name)
Add a input object to the form.
Methods: set_input(), get_input(), set_input_color(),
set_input_return().

add_menu( type, x, y, w, h, name)
Add a menu object to the form.
Methods: set_menu(), get_menu(), addto_menu().

add_choice( type, x, y, w, h, name)
Add a choice object to the form.
Methods: set_choice(), get_choice(), clear_choice(),
addto_choice(), replace_choice(), delete_choice(),
get_choice_text(), set_choice_fontsize(),
set_choice_fontstyle().

add_browser( type, x, y, w, h, name)
Add a browser object to the form.
Methods: set_browser_topline(), clear_browser(),
add_browser_line(), addto_browser(),
insert_browser_line(), delete_browser_line(),
replace_browser_line(), get_browser_line(),
load_browser(), get_browser_maxline(),
select_browser_line(), deselect_browser_line(),
deselect_browser(), isselected_browser_line(),
get_browser(), set_browser_fontsize(),
set_browser_fontstyle(), set_browser_specialkey().

add_timer( type, x, y, w, h, name)
Add a timer object to the form.
Methods: set_timer(), get_timer().

Form objects have the following data attributes; see the FORMS documentation:

<table>
<thead>
<tr>
<th>Name</th>
<th>C Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>window</td>
<td>int (read-only)</td>
<td>GL window id</td>
</tr>
<tr>
<td>w</td>
<td>float</td>
<td>form width</td>
</tr>
<tr>
<td>h</td>
<td>float</td>
<td>form height</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
<td>form x origin</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
<td>form y origin</td>
</tr>
<tr>
<td>deactivated</td>
<td>int</td>
<td>nonzero if form is deactivated</td>
</tr>
<tr>
<td>visible</td>
<td>int</td>
<td>nonzero if form is visible</td>
</tr>
<tr>
<td>frozen</td>
<td>int</td>
<td>nonzero if form is frozen</td>
</tr>
</tbody>
</table>
doublebuf | int | nonzero if double buffering on

Python Library Reference

Previous: 34.4.1 Functions Defined in  Up: 34.4.2  Next: 34.4.3 FORMS Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
34.4.3 FORMS Objects

Besides methods specific to particular kinds of FORMS objects, all FORMS objects also have the following methods:

```python
set_call_back(function, argument)
```
Set the object's callback function and argument. When the object needs interaction, the callback function will be called with two arguments: the object, and the callback argument. (FORMS objects without a callback function are returned by `fl.do_forms()` or `fl.check_forms()` when they need interaction.) Call this method without arguments to remove the callback function.

```python
delete_object()
```
Delete the object.

```python
show_object()
```
Show the object.

```python
hide_object()
```
Hide the object.

```python
redraw_object()
```
Redraw the object.

```python
freeze_object()
```
Freeze the object.

```python
unfreeze_object()
```
Unfreeze the object.

FORMS objects have these data attributes; see the FORMS documentation:

<table>
<thead>
<tr>
<th>Name</th>
<th>C Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>objclass</td>
<td>int (read-only)</td>
<td>object class</td>
</tr>
<tr>
<td>type</td>
<td>int (read-only)</td>
<td>object type</td>
</tr>
<tr>
<td>Variable</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>boxtype</td>
<td>int</td>
<td>box type</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
<td>x origin</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
<td>y origin</td>
</tr>
<tr>
<td>w</td>
<td>float</td>
<td>width</td>
</tr>
<tr>
<td>h</td>
<td>float</td>
<td>height</td>
</tr>
<tr>
<td>col1</td>
<td>int</td>
<td>primary color</td>
</tr>
<tr>
<td>col2</td>
<td>int</td>
<td>secondary color</td>
</tr>
<tr>
<td>align</td>
<td>int</td>
<td>alignment</td>
</tr>
<tr>
<td>lcol</td>
<td>int</td>
<td>label color</td>
</tr>
<tr>
<td>lsize</td>
<td>float</td>
<td>label font size</td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td>label string</td>
</tr>
<tr>
<td>lstyle</td>
<td>int</td>
<td>label style</td>
</tr>
<tr>
<td>pushed</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>focus</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>belowmouse</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>frozen</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>active</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>input</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>visible</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>radio</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>automatic</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
</tbody>
</table>
34.5 FL -- Constants used with the fl module

Availability: IRIX.

This module defines symbolic constants needed to use the built-in module fl (see above); they are equivalent to those defined in the C header file <forms.h> except that the name prefix "FL_" is omitted. Read the module source for a complete list of the defined names. Suggested use:

```python
import fl
from FL import *
```
34.6 f1p -- Functions for loading stored FORMS designs

Availability: IRIX.

This module defines functions that can read form definitions created by the `form designer' (fdesign) program that comes with the FORMS library (see module f1 above).

For now, see the file f1p.doc in the Python library source directory for a description.

XXX A complete description should be inserted here!
34.7 fm -- Font Manager interface

Availability: IRIX.

This module provides access to the IRIS Font Manager library. It is available only on Silicon Graphics machines. See also: 4Sight User's Guide, section 1, chapter 5: ``Using the IRIS Font Manager."

This is not yet a full interface to the IRIS Font Manager. Among the unsupported features are: matrix operations; cache operations; character operations (use string operations instead); some details of font info; individual glyph metrics; and printer matching.

It supports the following operations:

**init()**
Initialization function. Calls fmInit(). It is normally not necessary to call this function, since it is called automatically the first time the fm module is imported.

**findfont(fontname)**
Return a font handle object. Calls fmFindFont(fontname).

**enumerate()**
Returns a list of available font names. This is an interface to fmenumerate().

**prstr(string)**
Render a string using the current font (see the setfont() font handle method below). Calls fmPrstr(string).

**setpath(string)**
Sets the font search path. Calls fmSetpath(string). (XXX Does not work!?!)

**fontpath()**
Returns the current font search path.

Font handle objects support the following operations:

**scalefont** *(factor)*

Returns a handle for a scaled version of this font. Calls `fmscalefont(fh, factor)`.

**setfont()**

Makes this font the current font. Note: the effect is undone silently when the font handle object is deleted. Calls `fmsetfont(fh)`.

**getfontname()**

Returns this font's name. Calls `fmgetfontname(fh)`.

**getcomment()**

Returns the comment string associated with this font. Raises an exception if there is none. Calls `fmgetcomment(fh)`.

**getfontinfo()**

Returns a tuple giving some pertinent data about this font. This is an interface to `fmgetfontinfo()`. The returned tuple contains the following numbers: *(printermatched, fixed_width, xorig, yorig, xsize, ysize, height, nglyphs)*.

**getstrwidth** *(string)*

Returns the width, in pixels, of *string* when drawn in this font. Calls `fmgetstrwidth(fh, string)`.
34.8 g1 -- Graphics Library interface

Availability: IRIX.

This module provides access to the Silicon Graphics Graphics Library. It is available only on Silicon Graphics machines.

**Warning:** Some illegal calls to the GL library cause the Python interpreter to dump core. In particular, the use of most GL calls is unsafe before the first window is opened.

The module is too large to document here in its entirety, but the following should help you to get started. The parameter conventions for the C functions are translated to Python as follows:

- All (short, long, unsigned) int values are represented by Python integers.
- All float and double values are represented by Python floating point numbers. In most cases, Python integers are also allowed.
- All arrays are represented by one-dimensional Python lists. In most cases, tuples are also allowed.
- All string and character arguments are represented by Python strings, for instance, `winopen('Hi There!')` and `rotate(900, 'z')`.
- All (short, long, unsigned) integer arguments or return values that are only used to specify the length of an array argument are omitted. For example, the C call
  
  `lmdef(deftype, index, np, props)`

  is translated to Python as
  
  `lmdef(deftype, index, props)`

- Output arguments are omitted from the argument list; they are transmitted as function return values instead. If more than one value must be returned, the return value is a tuple. If the C function has both a regular return value (that is not omitted because of the previous rule) and an output argument, the return value comes first in the tuple. Examples: the C call
  
  `lmdef(deftype, index, props)`
getmcolor(i, &red, &green, &blue)

is translated to Python as

red, green, blue = getmcolor(i)

The following functions are non-standard or have special argument conventions:

**varray( argument)**
Equivalent to but faster than a number of v3d() calls. The argument is a list (or tuple) of points. Each point must be a tuple of coordinates \((x, y, z)\) or \((x, y)\). The points may be 2- or 3-dimensional but must all have the same dimension. Float and int values may be mixed however. The points are always converted to 3D double precision points by assuming \(z = 0.0\) if necessary (as indicated in the man page), and for each point v3d() is called.

**narray()**
Equivalent to but faster than a number of n3f and v3f calls. The argument is an array (list or tuple) of pairs of normals and points. Each pair is a tuple of a point and a normal for that point. Each point or normal must be a tuple of coordinates \((x, y, z)\). Three coordinates must be given. Float and int values may be mixed. For each pair, n3f() is called for the normal, and then v3f() is called for the point.

**vnarray()**
Similar to narray() but the pairs have the point first and the normal second.

**nurbssurface( s_k, t_k, ctl, s_ord, t_ord, type)**
Defines a nurbs surface. The dimensions of \(ctl[]\) are computed as follows: \([\text{len}(s_k) - s_{ord}], [\text{len}(t_k) - t_{ord}]\).

**nurbscurve( knots, ctlpoints, order, type)**
Defines a nurbs curve. The length of ctlpoints is \(\text{len}(knots) - order\).

**pwlcurve( points, type)**
Defines a piecewise-linear curve. points is a list of points. type must be
pick\((n)\)
select\((n)\)

The only argument to these functions specifies the desired size of the pick or select buffer.

endpick()
endselect()

These functions have no arguments. They return a list of integers representing the used part of the pick/select buffer. No method is provided to detect buffer overrun.

Here is a tiny but complete example GL program in Python:

```python
import gl, GL, time

def main():
    gl.foreground()
    gl.prefposition(500, 900, 500, 900)
    w = gl.winopen('CrissCross')
    gl.ortho2(0.0, 400.0, 0.0, 400.0)
    gl.color(GL.WHITE)
    gl.clear()
    gl.color(GL.RED)
    gl.bgnline()
    gl.v2f(0.0, 0.0)
    gl.v2f(400.0, 400.0)
    gl.endline()
    gl.bgnline()
    gl.v2f(400.0, 0.0)
    gl.v2f(0.0, 400.0)
    gl.endline()
    time.sleep(5)

main()
```

See Also:

*PyOpenGL: The Python OpenGL Binding*

An interface to OpenGL is also available; see information about the PyOpenGL project online at [http://pyopengl.sourceforge.net/](http://pyopengl.sourceforge.net/). This
may be a better option if support for SGI hardware from before about 1996 is not required.
Previous: 34.8 gl  Up: 34. SGI IRIX Specific  Next: 34.10 GL
34.9 DEVICE -- Constants used with the gl module

Availability: IRIX.

This module defines the constants used by the Silicon Graphics Graphics Library that C programmers find in the header file <gl/device.h>. Read the module source file for details.
34.10 GL -- Constants used with the gl module

Availability: IRIX.

This module contains constants used by the Silicon Graphics Graphics Library from the C header file <g1/g1.h>. Read the module source file for details.
34.11 imgfile -- Support for SGI imglib files

Availability: IRIX.

The `imgfile` module allows Python programs to access SGI imglib image files (also known as .rgb files). The module is far from complete, but is provided anyway since the functionality that there is enough in some cases. Currently, colormap files are not supported.

The module defines the following variables and functions:

**exception error**
This exception is raised on all errors, such as unsupported file type, etc.

**getsizes(file)**
This function returns a tuple \((x, y, z)\) where \(x\) and \(y\) are the size of the image in pixels and \(z\) is the number of bytes per pixel. Only 3 byte RGB pixels and 1 byte greyscale pixels are currently supported.

**read(file)**
This function reads and decodes the image on the specified file, and returns it as a Python string. The string has either 1 byte greyscale pixels or 4 byte RGBA pixels. The bottom left pixel is the first in the string. This format is suitable to pass to `gl.lrectwrite()`, for instance.

**readscaled(file, x, y, filter[, blur])**
This function is identical to read but it returns an image that is scaled to the given \(x\) and \(y\) sizes. If the `filter` and `blur` parameters are omitted scaling is done by simply dropping or duplicating pixels, so the result will be less than perfect, especially for computer-generated images.

Alternatively, you can specify a filter to use to smooth the image after scaling. The filter forms supported are 'impulse', 'box', 'triangle', 'quadratic' and 'gaussian'. If a filter is specified
blur is an optional parameter specifying the blurriness of the filter. It defaults to 1.0.

readscaled() makes no attempt to keep the aspect ratio correct, so that is the users' responsibility.

ttob(flag)
This function sets a global flag which defines whether the scan lines of the image are read or written from bottom to top (flag is zero, compatible with SGI GL) or from top to bottom (flag is one, compatible with X). The default is zero.

write(file, data, x, y, z)
This function writes the RGB or greyscale data in data to image file file. x and y give the size of the image, z is 1 for 1 byte greyscale images or 3 for RGB images (which are stored as 4 byte values of which only the lower three bytes are used). These are the formats returned by gl.lrectread().
34.12 jpeg -- Read and write JPEG files

Availability: IRIX.

The module jpeg provides access to the jpeg compressor and decompressor written by the Independent JPEG Group (IJG). JPEG is a standard for compressing pictures; it is defined in ISO 10918. For details on JPEG or the Independent JPEG Group software refer to the JPEG standard or the documentation provided with the software.

A portable interface to JPEG image files is available with the Python Imaging Library (PIL) by Fredrik Lundh. Information on PIL is available at http://www.pythonware.com/products/pil/.

The jpeg module defines an exception and some functions.

**exception error**
Exception raised by `compress()` and `decompress()` in case of errors.

**compress(data, w, h, b)**
Treat data as a pixmap of width w and height h, with b bytes per pixel. The data is in SGI GL order, so the first pixel is in the lower-left corner. This means that `gl.lrectread()` return data can immediately be passed to `compress()`. Currently only 1 byte and 4 byte pixels are allowed, the former being treated as greyscale and the latter as RGB color. `compress()` returns a string that contains the compressed picture, in JFIF format.

**decompress(data)**
Data is a string containing a picture in JFIF format. It returns a tuple `(data, width, height, bytesperpixel)`. Again, the data is suitable to pass to `gl.lrectwrite()`.

**setoption(name, value)**
Set various options. Subsequent `compress()` and `decompress()` calls will use these options. The following options are available:

<table>
<thead>
<tr>
<th>Option</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>'forcegray'</td>
<td>Force output to be grayscale, even if input is RGB.</td>
</tr>
<tr>
<td>'quality'</td>
<td>Set the quality of the compressed image to a value between 0 and 100 (default is 75). This only affects compression.</td>
</tr>
<tr>
<td>'optimize'</td>
<td>Perform Huffman table optimization. Takes longer, but results in smaller compressed image. This only affects compression.</td>
</tr>
<tr>
<td>'smooth'</td>
<td>Perform inter-block smoothing on uncompressed image. Only useful for low-quality images. This only affects decompression.</td>
</tr>
</tbody>
</table>

**See Also:**

*JPEG Still Image Data Compression Standard*

The canonical reference for the JPEG image format, by Pennebaker and Mitchell.

*Information Technology - Digital Compression and Coding of Continuous-tone Still Images - Requirements and Guidelines*

The ISO standard for JPEG is also published as ITU T.81. This is available online in PDF form.
35. SunOS Specific Services

The modules described in this chapter provide interfaces to features that are unique to SunOS 5 (also known as Solaris version 2).

Subsections

- 35.1 sunaudiodev -- Access to Sun audio hardware
  - 35.1.1 Audio Device Objects
- 35.2 SUNAUDIODEV -- Constants used with sunaudiodev

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
35.1 sunaudiodev -- Access to Sun audio hardware

Availability: SunOS.

This module allows you to access the Sun audio interface. The Sun audio hardware is capable of recording and playing back audio data in u-LAW format with a sample rate of 8K per second. A full description can be found in the audio(7I) manual page.

The module SUNAUDIODEV defines constants which may be used with this module.

This module defines the following variables and functions:

**exception error**
This exception is raised on all errors. The argument is a string describing what went wrong.

**open( mode)**
This function opens the audio device and returns a Sun audio device object. This object can then be used to do I/O on. The *mode* parameter is one of 'r' for record-only access, 'w' for play-only access, 'rw' for both and 'control' for access to the control device. Since only one process is allowed to have the recorder or player open at the same time it is a good idea to open the device only for the activity needed. See audio(7I) for details.

As per the manpage, this module first looks in the environment variable AUDIODEV for the base audio device filename. If not found, it falls back to /dev/audio. The control device is calculated by appending ``ctl'' to the base audio device.

---

Subsections
35.1.1 Audio Device Objects

Previous: 35. SunOS Specific Services  Up: 35. SunOS Specific Services
Next: 35.1.1 Audio Device Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
35.1.1 Audio Device Objects

The audio device objects are returned by `open()` define the following methods (except control objects which only provide `getinfo()`, `setinfo()`, `fileno()`, and `drain()`):

**close()**
This method explicitly closes the device. It is useful in situations where deleting the object does not immediately close it since there are other references to it. A closed device should not be used again.

**fileno()**
Returns the file descriptor associated with the device. This can be used to set up SIGPOLL notification, as described below.

**drain()**
This method waits until all pending output is processed and then returns. Calling this method is often not necessary: destroying the object will automatically close the audio device and this will do an implicit drain.

**flush()**
This method discards all pending output. It can be used avoid the slow response to a user's stop request (due to buffering of up to one second of sound).

**getinfo()**
This method retrieves status information like input and output volume, etc. and returns it in the form of an audio status object. This object has no methods but it contains a number of attributes describing the current device status. The names and meanings of the attributes are described in `<sun/audioio.h>` and in the `audio(7I)` manual page. Member names are slightly different from their C counterparts: a status object is only a single structure. Members of the `play` substructure have "o_" prepended to their name and members of the `record` structure have "i_". So, the C member `play.sample_rate` is accessed as `o_sample_rate`, `record.gain` as `i_gain` and `monitor_gain` plainly as
monitor_gain.

**ibufcount()**
This method returns the number of samples that are buffered on the recording side, i.e. the program will not block on a **read()** call of so many samples.

**obufcount()**
This method returns the number of samples buffered on the playback side. Unfortunately, this number cannot be used to determine a number of samples that can be written without blocking since the kernel output queue length seems to be variable.

**read(size)**
This method reads size samples from the audio input and returns them as a Python string. The function blocks until enough data is available.

**setinfo(status)**
This method sets the audio device status parameters. The status parameter is an device status object as returned by **getinfo()** and possibly modified by the program.

**write(samples)**
Write is passed a Python string containing audio samples to be played. If there is enough buffer space free it will immediately return, otherwise it will block.

The audio device supports asynchronous notification of various events, through the SIGPOLL signal. Here's an example of how you might enable this in Python:

```python
def handle_sigpoll(signum, frame):
    print 'I got a SIGPOLL update'

import fcntl, signal, STROPTS

signal.signal(signal.SIGPOLL, handle_sigpoll)
fcntl.ioctl(audio_obj.fileno(), STROPTS.I_SETSIG, STROPTS.S_MSG)
```
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
35.2 SUNAUDIODEV -- Constants used with sunaudiodev

Availability: SunOS.

This is a companion module to sunaudiodev which defines useful symbolic constants like MIN_GAIN, MAX_GAIN, SPEAKER, etc. The names of the constants are the same names as used in the C include file <sun/audioio.h>, with the leading string "AUDIO_" stripped.
36. MS Windows Specific Services

This chapter describes modules that are only available on MS Windows platforms.

- **msilib**: Creation of Microsoft Installer files, and CAB files.
- **msvcrt**: Miscellaneous useful routines from the MS VC++ runtime.
- **_winreg**: Routines and objects for manipulating the Windows registry.
- **winsound**: Access to the sound-playing machinery for Windows.
36.1.1 Database Objects
36.1 msilib -- Read and write Microsoft Installer files

Availability: Windows.

New in version 2.5.

The msilib supports the creation of Microsoft Installer (.msi) files. Because these files often contain an embedded ``cabinet'' file (.cab), it also exposes an API to create CAB files. Support for reading .cab files is currently not implemented; read support for the .msi database is possible.

This package aims to provide complete access to all tables in an .msi file, therefore, it is a fairly low-level API. Two primary applications of this package are the distutils command bdist_msi, and the creation of Python installer package itself (although that currently uses a different version of msilib).

The package contents can be roughly split into four parts: low-level CAB routines, low-level MSI routines, higher-level MSI routines, and standard table structures.

**FCICreate**(cabname, files)
Create a new CAB file named cabname. files must be a list of tuples, each containing the name of the file on disk, and the name of the file inside the CAB file.

The files are added to the CAB file in the order they appear in the list. All files are added into a single CAB file, using the MSZIP compression algorithm.

Callbacks to Python for the various steps of MSI creation are currently not exposed.

**UUIDCreate()**
Return the string representation of a new unique identifier. This wraps the
Windows API functions UuidCreate and UuidToString.

**OpenDatabase**(path, persist)

Return a new database object by calling MsiOpenDatabase. path is the file name of the MSI file; persist can be one of the constants MSIDBOPEN_CREATEDIRECT, MSIDBOPEN_CREATE, MSIDBOPEN_DIRECT, MSIDBOPEN_READONLY, or MSIDBOPEN_TRANSACT, and may include the flag MSIDBOPEN_PATCHFILE. See the Microsoft documentation for the meaning of these flags; depending on the flags, an existing database is opened, or a new one created.

**CreateRecord**(count)

Return a new record object by calling MSICreateRecord. count is the number of fields of the record.

**init_database**(name, schema, ProductName, ProductCode, ProductVersion, Manufacturer)

Create and return a new database name, initialize it with schema, and set the properties ProductName, ProductCode, ProductVersion, and Manufacturer.

schema must be a module object containing tables and _Validation_records attributes; typically, msilib.schema should be used.

The database will contain just the schema and the validation records when this function returns.

**add_data**(database, records)

Add all records to database. records should be a list of tuples, each one containing all fields of a record according to the schema of the table. For optional fields, None can be passed.

Field values can be int or long numbers, strings, or instances of the Binary class.

**class Binary**(filename)
Represents entries in the Binary table; inserting such an object using `add_data` reads the file named `filename` into the table.

**add_tables(database, module)**
Add all table content from `module` to `database`. `module` must contain an attribute `tables` listing all tables for which content should be added, and one attribute per table that has the actual content.

This is typically used to install the sequence tables.

**add_stream(database, name, path)**
Add the file `path` into the `_Stream` table of `database`, with the stream name `name`.

**gen_uuid()**
Return a new UUID, in the format that MSI typically requires (i.e. in curly braces, and with all hexdigits in upper-case).

---

**See Also:**

- `FCICreateFile`
- `UuidCreate`
- `UuidToString`

---

**Subsections**

- [36.1.1 Database Objects](#)
- [36.1.2 View Objects](#)
- [36.1.3 Summary Information Objects](#)
- [36.1.4 Record Objects](#)
- [36.1.5 Errors](#)
- [36.1.6 CAB Objects](#)
- [36.1.7 Directory Objects](#)
- [36.1.8 Features](#)
• 36.1.9 GUI classes
• 36.1.10 Precomputed tables
36.1.1 Database Objects

OpenView( sql)

Return a view object, by calling MSIDatabaseOpenView. sql is the SQL statement to execute.

Commit()

Commit the changes pending in the current transaction, by calling MSIDatabaseCommit.

GetSummaryInformation( count)

Return a new summary information object, by calling MsiGetSummaryInformation. count is the maximum number of updated values.

See Also:

MSIOpenView
MSIDatabaseCommit
MSIGetSummaryInformation
### 36.1.2 View Objects

**Execute**([params=None])

Execute the SQL query of the view, through **MSIViewExecute**. *params* is an optional record describing actual values of the parameter tokens in the query.

**GetColumnInfo**(kind)

Return a record describing the columns of the view, through calling **MsiViewGetColumnInfo**. *kind* can be either **MSICOLINFO_NAMES** or **MSICOLINFO_TYPES**.

**Fetch**()

Return a result record of the query, through calling **MsiViewFetch**.

**Modify**(kind, data)

Modify the view, by calling **MsiViewModify**. *kind* can be one of **MSIMODIFY_SEEK**, **MSIMODIFY_REFRESH**, **MSIMODIFY_INSERT**, **MSIMODIFY_UPDATE**, **MSIMODIFY_ASSIGN**, **MSIMODIFY_REPLACE**, **MSIMODIFY_MERGE**, **MSIMODIFY_DELETE**, **MSIMODIFY_INSERT_TEMPORARY**, **MSIMODIFY_VALIDATE**, **MSIMODIFY_VALIDATE_NEW**, **MSIMODIFY_VALIDATE_FIELD**, or **MSIMODIFY_VALIDATE_DELETE**.

*data* must be a record describing the new data.

**Close**()

Close the view, through **MsiViewClose**.

---

**See Also:**

*MsiViewExecute*

**MSIViewGetColumnInfo**
36.1.3 Summary Information Objects

GetProperty( field)
Return a property of the summary, through MsiSummaryInfoGetProperty. field is the name of the property, and can be one of the constants PID_CODEPAGE, PID_TITLE, PID_SUBJECT, PID_AUTHOR, PID_KEYWORDS, PID_COMMENTS, PID_TEMPLATE, PID_LASTAUTHOR, PID_REVNUMBER, PID_LASTPRINTED, PID_CREATE_DTM, PID_LASTSAVE_DTM, PID_PAGECOUNT, PID_WORDCOUNT, PID_CHARCOUNT, PID_APPNAME, or PID_SECURITY.

GetPropertyCount()
Return the number of summary properties, through MsiSummaryInfoGetPropertyCount.

SetProperty( field, value)
Set a property through MsiSummaryInfoSetProperty. field can have the same values as inGetProperty, value is the new value of the property. Possible value types are integer and string.

Persist()
Write the modified properties to the summary information stream, using MsiSummaryInfoPersist.

See Also:

MsiSummaryInfoGetProperty
MsiSummaryInfoGetPropertyCount
MsiSummaryInfoSetProperty
MsiSummaryInfoPersist
Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
36.1.4 Record Objects

GetFieldCount()

Return the number of fields of the record, through MsiRecordGetFieldCount.

SetString(field, value)

Set field to value through MsiRecordSetString. field must be an integer; value a string.

SetStream(field, value)

Set field to the contents of the file named value, through MsiRecordSetStream. field must be an integer; value a string.

SetInteger(field, value)

Set field to value through MsiRecordSetInteger. Both field and value must be an integer.

ClearData()

Set all fields of the record to 0, through MsiRecordClearData.

See Also:

MsiRecordGetFieldCount

MsiRecordSetString

MsiRecordSetStream

MsiRecordSetInteger

MsiRecordClear
Errors

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
36.1.5 Errors

All wrappers around MSI functions raise `MsiError`; the string inside the exception will contain more detail.
36.1.6 CAB Objects

class CAB( name)

The class CAB represents a CAB file. During MSI construction, files will be added simultaneously to the Files table, and to a CAB file. Then, when all files have been added, the CAB file can be written, then added to the MSI file.

name is the name of the CAB file in the MSI file.

append( full, logical)

Add the file with the pathname full to the CAB file, under the name logical. If there is already a file named logical, a new file name is created.

Return the index of the file in the CAB file, and the new name of the file inside the CAB file.

append( database)

Generate a CAB file, add it as a stream to the MSI file, put it into the Media table, and remove the generated file from the disk.
36.1.7 Directory Objects

class Directory(database, cab, basedir, physical, logical, default, component, [componentflags])

Create a new directory in the Directory table. There is a current component at each point in time for the directory, which is either explicitly created through start_component, or implicitly when files are added for the first time. Files are added into the current component, and into the cab file. To create a directory, a base directory object needs to be specified (can be None), the path to the physical directory, and a logical directory name. default specifies the DefaultDir slot in the directory table. componentflags specifies the default flags that new components get.

start_component([component[, feature[, flags[, keyfile[, uuid]]]]])

Add an entry to the Component table, and make this component the current component for this directory. If no component name is given, the directory name is used. If no feature is given, the current feature is used. If no flags are given, the directory's default flags are used. If no keyfile is given, the KeyPath is left null in the Component table.

add_file(file[, src[, version[, language]]])

Add a file to the current component of the directory, starting a new one if there is no current component. By default, the file name in the source and the file table will be identical. If the src file is specified, it is interpreted relative to the current directory. Optionally, a version and a language can be specified for the entry in the File table.

glob(pattern[, exclude])

Add a list of files to the current component as specified in the glob pattern. Individual files can be excluded in the exclude list.

remove_pyc()

Remove .pyc/.pyo files on uninstall.
36.1.8 Features

```
class Feature(database, id, title, desc, display[, level=1[, parent[{}],
directory[, attributes=0]]]])
```

Add a new record to the Feature table, using the values id, parent.id, title, desc, display, level, directory, and attributes. The resulting feature object can be passed to the start_component method of Directory.

```
set_current()
```

Make this feature the current feature of msilib. New components are automatically added to the default feature, unless a feature is explicitly specified.

See Also:

Feature Table
36.1.9 GUI classes

msilib provides several classes that wrap the GUI tables in an MSI database. However, no standard user interface is provided; use bdist_msi to create MSI files with a user-interface for installing Python packages.

class Control( dlg, name)
   Base class of the dialog controls. dlg is the dialog object the control belongs to, and name is the control's name.

event( event, argument[ , condition = `1` [, ordering]])
   Make an entry into the ControlEvent table for this control.

mapping( event, attribute)
   Make an entry into the EventMapping table for this control.

condition( action, condition)
   Make an entry into the ControlCondition table for this control.

class RadioButtonGroup( dlg, name, property)
   Create a radio button control named name. property is the installer property that gets set when a radio button is selected.

add( name, x, y, width, height, text [, value])
   Add a radio button named name to the group, at the coordinates x, y, width, height, and with the label text. If value is omitted, it defaults to name.

class Dialog( db, name, x, y, w, h, attr, title, first, default, cancel)
   Return a new Dialog object. An entry in the Dialog table is made, with the specified coordinates, dialog attributes, title, name of the first, default, and cancel controls.

control( name, type, x, y, width, height, attributes, property, text, control_next, help)
Return a new Control object. An entry in the Control table is made with the specified parameters.

This is a generic method; for specific types, specialized methods are provided.

**text**(*name, x, y, width, height, attributes, text*)
Add and return a Text control.

**bitmap**(*name, x, y, width, height, text*)
Add and return a Bitmap control.

**line**(*name, x, y, width, height*)
Add and return a Line control.

**pushbutton**(*name, x, y, width, height, attributes, text, next_control*)
Add and return a PushButton control.

**radiogroup**(*name, x, y, width, height, attributes, property, text, next_control*)
Add and return a RadioButtonGroup control.

**checkbox**(*name, x, y, width, height, attributes, property, text, next_control*)
Add and return a CheckBox control.

See Also:

* [Dialog Table](#)
* [Control Table](#)
* [Control Types](#)
* [ControlCondition Table](#)
* [ControlEvent Table](#)
**EventMapping Table**

**RadioButton Table**

---

**Python Library Reference**

Previous: [36.1.8 Features](#) Up: [36.1 msilib](#) Next: [36.1.10 Precomputed tables](#)

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
36.1.10 Precomputed tables

msilib provides a few subpackages that contain only schema and table definitions. Currently, these definitions are based on MSI version 2.0.

**schema**
This is the standard MSI schema for MSI 2.0, with the `tables` variable providing a list of table definitions, and `_Validation_records` providing the data for MSI validation.

**sequence**
This module contains table contents for the standard sequence tables: `AdminExecuteSequence`, `AdminUISequence`, `AdvtExecuteSequence`, `InstallExecuteSequence`, and `InstallUISequence`.

**text**
This module contains definitions for the UIText and ActionText tables, for the standard installer actions.
36.2 msvcrtn - Useful routines from the MS VC++ runtime

Availability: Windows.

These functions provide access to some useful capabilities on Windows platforms. Some higher-level modules use these functions to build the Windows implementations of their services. For example, the `getpass` module uses this in the implementation of the `getpass()` function.

Further documentation on these functions can be found in the Platform API documentation.

---

Subsections

- 36.2.1 File Operations
- 36.2.2 Console I/O
- 36.2.3 Other Functions

---

Python Library Reference

Previous: 36.1.10 Precomputed tables Up: 36. MS Windows Specific Next: 36.2.1 File Operations

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
36.2.1 File Operations

**locking( fd, mode, nbytes)**
Lock part of a file based on file descriptor *fd* from the C runtime. Raises IOError on failure. The locked region of the file extends from the current file position for *nbytes* bytes, and may continue beyond the end of the file. *mode* must be one of the LK_* constants listed below. Multiple regions in a file may be locked at the same time, but may not overlap. Adjacent regions are not merged; they must be unlocked individually.

**LK_LOCK**
**LK_RLCK**
Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, IOError is raised.

**LK_NBLCK**
**LK_NBRLCK**
Locks the specified bytes. If the bytes cannot be locked, IOError is raised.

**LK_UNLCK**
Unlocks the specified bytes, which must have been previously locked.

**setmode( fd, flags)**
Set the line-end translation mode for the file descriptor *fd*. To set it to text mode, *flags* should be os.O_TEXT; for binary, it should be os.O_BINARY.

**open_osfhandle( handle, flags)**
Create a C runtime file descriptor from the file handle *handle*. The *flags* parameter should be a bit-wise OR of os.O_APPEND, os.O_RDONLY, and os.O_TEXT. The returned file descriptor may be used as a parameter to os.fdopen() to create a file object.

**get_osfhandle( fd)**
Return the file handle for the file descriptor `fd`. Raises `IOError` if `fd` is not recognized.
36.2.2 Console I/O

**kbhit()**
Return true if a keypress is waiting to be read.

**getch()**
Read a keypress and return the resulting character. Nothing is echoed to the console. This call will block if a keypress is not already available, but will not wait for Enter to be pressed. If the pressed key was a special function key, this will return '\000' or '\xe0'; the next call will return the keycode. The control-c keypress cannot be read with this function.

**getche()**
Similar to **getch()**, but the keypress will be echoed if it represents a printable character.

**putch**( char)
Print the character char to the console without buffering.

**ungetch**( char)
Cause the character char to be "pushed back" into the console buffer; it will be the next character read by **getch()** or **getche()**.

*Release 2.5, documentation updated on 19th September, 2006.*
See [About this document...](#) for information on suggesting changes.
36.2.3 Other Functions

heapmin()

Force the malloc() heap to clean itself up and return unused blocks to the operating system. This only works on Windows NT. On failure, this raises IOError.
36.3 _winreg - Windows registry access

Availability: Windows.

New in version 2.0.

These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a handle object is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them.

This module exposes a very low-level interface to the Windows registry; it is expected that in the future a new winreg module will be created offering a higher-level interface to the registry API.

This module offers the following functions:

**CloseKey**( *hkey*)

Closes a previously opened registry key. The *hkey* argument specifies a previously opened key.

Note that if *hkey* is not closed using this method (or via *handle.Close()*), it is closed when the *hkey* object is destroyed by Python.

**ConnectRegistry**( *computer_name*, *key*)

Establishes a connection to a predefined registry handle on another computer, and returns a *handle object*

*computer_name* is the name of the remote computer, of the form r"\\computername". If None, the local computer is used.

*key* is the predefined handle to connect to.

The return value is the handle of the opened key. If the function fails, an *EnvironmentError* exception is raised.
CreateKey( key, sub_key)

Creates or opens the specified key, returning a handle object

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that names the key this method opens or creates.

If key is one of the predefined keys, sub_key may be None. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an EnvironmentError exception is raised.

DeleteKey( key, sub_key)

Deletes the specified key.

key is an already open key, or any one of the predefined HKEY_* constants.

sub_key is a string that must be a subkey of the key identified by the key parameter. This value must not be None, and the key may not have subkeys.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an EnvironmentError exception is raised.

DeleteValue( key, value)

Removes a named value from a registry key.

key is an already open key, or one of the predefined HKEY_* constants.

value is a string that identifies the value to remove.

EnumKey( key, index)

Enumerates subkeys of an open registry key, returning a string.
key is an already open key, or any one of the predefined HKEY_* constants.

index is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until an EnvironmentError exception is raised, indicating, no more values are available.

EnumValue( key, index)
Enumerates values of an open registry key, returning a tuple.

key is an already open key, or any one of the predefined HKEY_* constants.

index is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until an EnvironmentError exception is raised, indicating no more values.

The result is a tuple of 3 items:

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A string that identifies the value name</td>
</tr>
<tr>
<td>1</td>
<td>An object that holds the value data, and whose type depends on the underlying registry type</td>
</tr>
<tr>
<td>2</td>
<td>An integer that identifies the type of the value data</td>
</tr>
</tbody>
</table>

FlushKey( key)
Writes all the attributes of a key to the registry.

key is an already open key, or one of the predefined HKEY_* constants.

It is not necessary to call RegFlushKey to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike CloseKey(), the FlushKey() method returns only when all the data has been written to the registry. An application should only call FlushKey() if it requires absolute certainty that registry changes are on disk.
If you don't know whether a `FlushKey()` call is required, it probably isn't.

**RegLoadKey(key, sub_key, file_name)**

Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

*key* is an already open key, or any of the predefined HKEY_* constants.

*sub_key* is a string that identifies the sub_key to load.

*file_name* is the name of the file to load registry data from. This file must have been created with the `SaveKey()` function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to LoadKey() fails if the calling process does not have the `SE_RESTORE_PRIVILEGE` privilege. Note that privileges are different than permissions - see the Win32 documentation for more details.

If *key* is a handle returned by `ConnectRegistry()`, then the path specified in *file_name* is relative to the remote computer.

The Win32 documentation implies *key* must be in the HKEY_USER or HKEY_LOCAL_MACHINE tree. This may or may not be true.

**OpenKey(key, sub_key[, res = 0][, sam = KEY_READ])**

Opens the specified key, returning a handle object

*key* is an already open key, or any one of the predefined HKEY_* constants.

*sub_key* is a string that identifies the sub_key to open.

*res* is a reserved integer, and must be zero. The default is zero.

*sam* is an integer that specifies an access mask that describes the desired security access for the key. Default is KEY_READ

The result is a new handle to the specified key.
If the function fails, `EnvironmentError` is raised.

**OpenKeyEx()**

The functionality of `OpenKeyEx()` is provided via `OpenKey()`, by the use of default arguments.

**QueryInfoKey** *(key)*

Returns information about a key, as a tuple.

`key` is an already open key, or one of the predefined `HKEY_*` constants.

The result is a tuple of 3 items:

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>An integer giving the number of sub keys this key has.</td>
</tr>
<tr>
<td>1</td>
<td>An integer giving the number of values this key has.</td>
</tr>
<tr>
<td>2</td>
<td>A long integer giving when the key was last modified (if available) as 100's of nanoseconds since Jan 1, 1600.</td>
</tr>
</tbody>
</table>

**QueryValue** *(key, sub_key)*

Retrieves the unnamed value for a key, as a string

`key` is an already open key, or one of the predefined `HKEY_*` constants.

`sub_key` is a string that holds the name of the subkey with which the value is associated. If this parameter is `None` or empty, the function retrieves the value set by the `SetValue()` method for the key identified by `key`.

Values in the registry have name, type, and data components. This method retrieves the data for a key's first value that has a NULL name. But the underlying API call doesn't return the type, Lame Lame Lame, DO NOT USE THIS!!

**QueryValueEx** *(key, value_name)*

Retrieves the type and data for a specified value name associated with an open registry key.

`key` is an already open key, or one of the predefined `HKEY_*` constants.
value_name is a string indicating the value to query.

The result is a tuple of 2 items:

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The value of the registry item.</td>
</tr>
<tr>
<td>1</td>
<td>An integer giving the registry type for this value.</td>
</tr>
</tbody>
</table>

**SaveKey( key, file_name)**
Saves the specified key, and all its subkeys to the specified file.

*key* is an already open key, or one of the predefined HKEY_* constants.

*file_name* is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the LoadKey(), ReplaceKey() or RestoreKey() methods.

If *key* represents a key on a remote computer, the path described by *file_name* is relative to the remote computer. The caller of this method must possess the SeBackupPrivilege security privilege. Note that privileges are different than permissions - see the Win32 documentation for more details.

This function passes NULL for security_attributes to the API.

**SetValue( key, sub_key, type, value)**
Associates a value with a specified key.

*key* is an already open key, or one of the predefined HKEY_* constants.

*sub_key* is a string that names the subkey with which the value is associated.

*type* is an integer that specifies the type of the data. Currently this must be REG_SZ, meaning only strings are supported. Use the SetValueEx() function for support for other data types.

*value* is a string that specifies the new value.
If the key specified by the *sub_key* parameter does not exist, the *SetValue* function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the *key* parameter must have been opened with *KEY_SET_VALUE* access.

**SetValueEx**(*key, value_name, reserved, type, value*)
Stores data in the value field of an open registry key.

*key* is an already open key, or one of the predefined HKEY_* constants.

*sub_key* is a string that names the subkey with which the value is associated.

*type* is an integer that specifies the type of the data. This should be one of the following constants defined in this module:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG_BINARY</td>
<td>Binary data in any form.</td>
</tr>
<tr>
<td>REG_DWORD</td>
<td>A 32-bit number.</td>
</tr>
<tr>
<td>REG_DWORD_LITTLE_ENDIAN</td>
<td>A 32-bit number in little-endian format.</td>
</tr>
<tr>
<td>REG_DWORD_BIG_ENDIAN</td>
<td>A 32-bit number in big-endian format.</td>
</tr>
<tr>
<td>REG_EXPAND_SZ</td>
<td>Null-terminated string containing references to environment variables (&quot;%PATH%&quot;)</td>
</tr>
<tr>
<td>REG_LINK</td>
<td>A Unicode symbolic link.</td>
</tr>
<tr>
<td>REG_MULTI_SZ</td>
<td>A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)</td>
</tr>
<tr>
<td>REG_NONE</td>
<td>No defined value type.</td>
</tr>
<tr>
<td>REG_RESOURCE_LIST</td>
<td>A device-driver resource list.</td>
</tr>
</tbody>
</table>
REG_SZ | A null-terminated string.

reserved can be anything - zero is always passed to the API.

value is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the key parameter must have been opened with KEY_SET_VALUE access.

To open the key, use the CreateKeyEx() or OpenKey() methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

Subsections

- 36.3.1 Registry Handle Objects
36.3.1 Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the \texttt{Close()} method on the object, or the \texttt{CloseKey()} function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for \texttt{\_\_nonzero\_()} - thus

\begin{verbatim}
    if handle:
        print "Yes"
\end{verbatim}

will print \texttt{Yes} if the handle is currently valid (has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in \texttt{int()} function), in which case the underlying Windows handle value is returned. You can also use the \texttt{Detach()} method to return the integer handle, and also disconnect the Windows handle from the handle object.

\textbf{Close()}

Closes the underlying Windows handle.

If the handle is already closed, no error is raised.

\textbf{Detach()}

Detaches the Windows handle from the handle object.

The result is an integer (or long on 64 bit Windows) that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.

After calling this function, the handle is effectively invalidated, but the
handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.
36.4 winsound -- Sound-playing interface for Windows

Availability: Windows.

New in version 1.5.2.

The winsound module provides access to the basic sound-playing machinery provided by Windows platforms. It includes functions and several constants.

**Beep** (*frequency, duration*)

Beep the PC's speaker. The *frequency* parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The *duration* parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, `RuntimeError` is raised. **Note:** Under Windows 95 and 98, the Windows `Beep()` function exists but is useless (it ignores its arguments). In that case Python simulates it via direct port manipulation (added in version 2.1). It's unknown whether that will work on all systems. New in version 1.6.

**PlaySound** (*sound, flags*)

Call the underlying `PlaySound()` function from the Platform API. The *sound* parameter may be a filename, audio data as a string, or None. Its interpretation depends on the value of *flags*, which can be a bit-wise ORed combination of the constants described below. If the system indicates an error, `RuntimeError` is raised.

**MessageBeep** ([*type=MB_OK]*)

Call the underlying `MessageBeep()` function from the Platform API. This plays a sound as specified in the registry. The *type* argument specifies which sound to play; possible values are -1, MB_ICONASTERISK, MB_ICONEXCLAMATION, MB_ICONHAND, MB_ICONQUESTION, and MB_OK, all described below. The value -1 produces a ``simple beep``; this is the final fallback if a sound cannot be played otherwise. New in version 2.3.
SND_FILENAME
The `sound` parameter is the name of a WAV file. Do not use with SND_ALIAS.

SND_ALIAS
The `sound` parameter is a sound association name from the registry. If the registry contains no such name, play the system default sound unless SND_NODEFAULT is also specified. If no default sound is registered, raise RuntimeError. Do not use with SND_FILENAME.

All Win32 systems support at least the following; most systems support many more:

<table>
<thead>
<tr>
<th>PlaySound() name</th>
<th>Corresponding Control Panel Sound name</th>
</tr>
</thead>
<tbody>
<tr>
<td>'SystemAsterisk'</td>
<td>Asterisk</td>
</tr>
<tr>
<td>'SystemExclamation'</td>
<td>Exclamation</td>
</tr>
<tr>
<td>'SystemExit'</td>
<td>Exit Windows</td>
</tr>
<tr>
<td>'SystemHand'</td>
<td>Critical Stop</td>
</tr>
<tr>
<td>'SystemQuestion'</td>
<td>Question</td>
</tr>
</tbody>
</table>

For example:

```python
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (but"
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

SND_LOOP
Play the sound repeatedly. The SND_ASYNC flag must also be used to avoid blocking. Cannot be used with SND_MEMORY.

SND_MEMORY
The `sound` parameter to PlaySound() is a memory image of a WAV file, as a string.

Note: This module does not support playing from a memory image
asynchronously, so a combination of this flag and SND_ASYNC will raise RuntimeError.

**SND_PURGE**
Stop playing all instances of the specified sound.

**SND_ASYNC**
Return immediately, allowing sounds to play asynchronously.

**SND_NODEFAULT**
If the specified sound cannot be found, do not play the system default sound.

**SND_NOSTOP**
Do not interrupt sounds currently playing.

**SND_NOWAIT**
Return immediately if the sound driver is busy.

**MB_ICONASTERISK**
Play the SystemDefault sound.

**MB_ICONEXCLAMATION**
Play the SystemExclamation sound.

**MB_ICONHAND**
Play the SystemHand sound.

**MB_ICONQUESTION**
Play the SystemQuestion sound.

**MB_OK**
Play the SystemDefault sound.
See *About this document*... for information on suggesting changes.
A. Undocumented Modules

Here's a quick listing of modules that are currently undocumented, but that should be documented. Feel free to contribute documentation for them! (Send via email to docs@python.org.)

The idea and original contents for this chapter were taken from a posting by Fredrik Lundh; the specific contents of this chapter have been substantially revised.

Subsections

- A.1 Frameworks
- A.2 Miscellaneous useful utilities
- A.3 Platform specific modules
- A.4 Multimedia
- A.5 Obsolete
- A.6 SGI-specific Extension modules

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
A.1 Frameworks

Frameworks tend to be harder to document, but are well worth the effort spent.

None at this time.
A.2 Miscellaneous useful utilities

Some of these are very old and/or not very robust; marked with ``hmm."

bdb

-- A generic Python debugger base class (used by pdb).

ihooks

-- Import hook support (for rexec; may become obsolete).
A.3 Platform specific modules

These modules are used to implement the `os.path` module, and are not documented beyond this mention. There's little need to document these.

**ntpath**
-- Implementation of `os.path` on Win32, Win64, WinCE, and OS/2 platforms.

**posixpath**
-- Implementation of `os.path` on POSIX.

**bsddb185**
-- Backwards compatibility module for systems which still use the Berkeley DB 1.85 module. It is normally only available on certain BSD UNIX-based systems. It should never be used directly.
A.4 Multimedia

audiodев
-- Platform-independent API for playing audio data.

linuxaudiodев
-- Play audio data on the Linux audio device. Replaced in Python 2.3 by the ossaudiodев module.

sunaudio
-- Interpret Sun audio headers (may become obsolete or a tool/demo).

toaiff
-- Convert "arbitrary" sound files to AIFF files; should probably become a tool or demo. Requires the external program sox.
A.5 Obsolete

These modules are not normally available for import; additional work must be done to make them available.

These extension modules written in C are not built by default. Under Unix, these must be enabled by uncommenting the appropriate lines in Modules/Setup in the build tree and either rebuilding Python if the modules are statically linked, or building and installing the shared object if using dynamically-loaded extensions.

**timing**

-- Measure time intervals to high resolution (use `time.clock()` instead).
A.6 SGI-specific Extension modules

The following are SGI specific, and may be out of touch with the current version of reality.

cl

-- Interface to the SGI compression library.

sv

-- Interface to the "simple video" board on SGI Indigo (obsolete hardware).
B. Reporting Bugs

Python is a mature programming language which has established a reputation for stability. In order to maintain this reputation, the developers would like to know of any deficiencies you find in Python or its documentation.

Before submitting a report, you will be required to log into SourceForge; this will make it possible for the developers to contact you for additional information if needed. It is not possible to submit a bug report anonymously.

All bug reports should be submitted via the Python Bug Tracker on SourceForge ([http://sourceforge.net/bugs/?group_id=5470](http://sourceforge.net/bugs/?group_id=5470)). The bug tracker offers a Web form which allows pertinent information to be entered and submitted to the developers.

The first step in filing a report is to determine whether the problem has already been reported. The advantage in doing so, aside from saving the developers time, is that you learn what has been done to fix it; it may be that the problem has already been fixed for the next release, or additional information is needed (in which case you are welcome to provide it if you can!). To do this, search the bug database using the search box on the left side of the page.

If the problem you're reporting is not already in the bug tracker, go back to the Python Bug Tracker ([http://sourceforge.net/bugs/?group_id=5470](http://sourceforge.net/bugs/?group_id=5470)). Select the "Submit a Bug" link at the top of the page to open the bug reporting form.

The submission form has a number of fields. The only fields that are required are the "Summary" and "Details" fields. For the summary, enter a very short description of the problem; less than ten words is good. In the Details field, describe the problem in detail, including what you expected to happen and what did happen. Be sure to include the version of Python you used, whether any extension modules were involved, and what hardware and software platform you were using (including version information as appropriate).

The only other field that you may want to set is the "Category" field, which allows you to place the bug report into a broad category (such as "Documentation" or "Library").
Each bug report will be assigned to a developer who will determine what needs to be done to correct the problem. You will receive an update each time action is taken on the bug.

See Also:

*How to Report Bugs Effectively*

Article which goes into some detail about how to create a useful bug report. This describes what kind of information is useful and why it is useful.

*Bug Writing Guidelines*

Information about writing a good bug report. Some of this is specific to the Mozilla project, but describes general good practices.
C. History and License

Subsections

- C.1 History of the software
- C.2 Terms and conditions for accessing or otherwise using Python
- C.3 Licenses and Acknowledgements for Incorporated Software
  - C.3.1 Mersenne Twister
  - C.3.2 Sockets
  - C.3.3 Floating point exception control
  - C.3.4 MD5 message digest algorithm
  - C.3.5 Asynchronous socket services
  - C.3.6 Cookie management
  - C.3.7 Profiling
  - C.3.8 Execution tracing
  - C.3.9 UUencode and UUdecode functions
  - C.3.10 XML Remote Procedure Calls

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

<table>
<thead>
<tr>
<th>Release</th>
<th>Derived from</th>
<th>Year</th>
<th>Owner</th>
<th>GPL compatible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.0 thru 1.2</td>
<td>n/a</td>
<td>1991-1995</td>
<td>CWI</td>
<td>yes</td>
</tr>
<tr>
<td>1.3 thru 1.5.2</td>
<td>1.2</td>
<td>1995-1999</td>
<td>CNRI</td>
<td>yes</td>
</tr>
<tr>
<td>1.6</td>
<td>1.5.2</td>
<td>2000</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.0</td>
<td>1.6</td>
<td>2000</td>
<td>BeOpen.com</td>
<td>no</td>
</tr>
<tr>
<td>1.6.1</td>
<td>1.6</td>
<td>2001</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>no</td>
</tr>
<tr>
<td>2.0.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.1</td>
<td>2.1+2.0.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2</td>
<td>2.1.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
</tbody>
</table>
Note: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.
C.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.5

1. This LICENSE AGREEMENT is between the Python Software Foundation (``PSF''), and the Individual or Organization (``Licensee'') accessing and otherwise using Python 2.5 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.5 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., ``Copyright © 2001-2006 Python Software Foundation; All Rights Reserved'' are retained in Python 2.5 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.5 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.5.

4. PSF is making Python 2.5 available to Licensee on an ``AS IS'' basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.5 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS
OF PYTHON 2.5 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.5, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.5, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com (``BeOpen''), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (``Licensee'') accessing and otherwise using this software in source or binary form and its associated documentation (``the Software '').

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an ``AS IS'' basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT
LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY
REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR
FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY
RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER
USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR
CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY
DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY
THEREOF.

5. This License Agreement will automatically terminate upon a material
breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
by the law of the State of California, excluding conflict of law provisions.
Nothing in this License Agreement shall be deemed to create any
relationship of agency, partnership, or joint venture between BeOpen and
Licensee. This License Agreement does not grant permission to use
BeOpen trademarks or trade names in a trademark sense to endorse or
promote products or services of Licensee, or any third party. As an
exception, the ``BeOpen Python'' logos available at
http://www.pythonlabs.com/logos.html may be used according to the
permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to
be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National
Research Initiatives, having an office at 1895 Preston White Drive, Reston,
VA 20191 (`CNRI"), and the Individual or Organization (`Licensee"
accessing and otherwise using Python 1.6.1 software in source or binary
form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI
hereby grants Licensee a nonexclusive, royalty-free, world-wide license to
reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., ``Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): ``Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an `AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual
property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the ``ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR
ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

Subsections

- C.3.1 Mersenne Twister
- C.3.2 Sockets
- C.3.3 Floating point exception control
- C.3.4 MD5 message digest algorithm
- C.3.5 Asynchronous socket services
- C.3.6 Cookie management
- C.3.7 Profiling
- C.3.8 Execution tracing
- C.3.9 UUencode and UUdecode functions
- C.3.10 XML Remote Procedure Calls

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
C.3.1 Mersenne Twister

The _random module includes code based on a download from http://www.math.keio.ac.jp/~matumoto/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26. Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed) or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura, All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS AND CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.
http://www.math.keio.ac.jp/matumoto/emt.html
email: matumoto@math.keio.ac.jp
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
C.3.2 Sockets

The socket module uses the functions, `getaddrinfo`, and `getnameinfo`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/about/index.html.

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
C.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

```
/               Copyright (c) 1996.
                The Regents of the University of California.
                All rights reserved.

Permission to use, copy, modify, and distribute this software any purpose without fee is hereby granted, provided that this tire notice is included in all copies of any software which i includes a copy or modification of this software and in copies of the supporting documentation for such software.

This work was produced at the University of California, Lawr Livermore National Laboratory under contract no. W-7405-EN between the U.S. Department of Energy and The Regents of University of California for the operation of UC LLNL.

DISCLAIMER

This software was prepared as an account of work sponsored b agency of the United States Government. Neither the United St Government nor the University of California nor any of their ployees, makes any warranty, express or implied, or assumes liability or responsibility for the accuracy, completeness usefulness of any information, apparatus, product, or pro disclosed, or represents that its use would not infr privately-owned rights. Reference herein to any specific com rcial products, process, or service by trade name, trademark manufacturer, or otherwise, does not necessarily constitute imply its endorsement, recommendation, or favoring by the Un States Government or the University of California. The views opinions of authors expressed herein do not necessarily stat reflect those of the United States Government or the Univer of California, and shall not be used for advertising or pro endorsement purposes.
```
C.3.4 MD5 message digest algorithm

The source code for the `md5` module contains the following notice:

```
Copyright (C) 1999, 2002 Aladdin Enterprises. All rights reserved

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damage arising from the use of this software.

Permission is granted to anyone to use this software for any purpose including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and not misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose text is available at
http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite (section A.5) but excluding the rest of Appendix A. It does not include any code or documentation that is identified in the RFC as being copyrighted.

The original and principal author of `md5.h` is L. Peter Deutsch <ghost@aladdin.com>. Other authors are noted in the change history that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed references to Ghostscript; clarified derivation from RFC 132
now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than added conditionalization for C++ compilation from Martin Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
```
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
C.3.5 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```
C.3.6 Cookie management

The Cookie module contains the following notice:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Timothy O'Malley not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
C.3.7 Profiling

The profile and pstats modules contain the following notice:

Copyright 1994, by InfoSeek Corporation, all rights reserved. Written by James Roskind

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose (subject to the restriction in the following sentence) without fee is hereby granted provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of InfoSeek not be used advertising or publicity pertaining to distribution of the software without specific, written prior permission. This permission is explicitly restricted to the copying and modification of the software to remain in Python, compiled Python, or other languages (such as C wherein the modified or derived code is exclusively imported into a Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
C.3.8 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rig
erres... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights res
```

Permission to use, copy, modify, and distribute this Python software
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all co
and that both that copyright notice and this permission notice appe
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertai
distribution of the software without specific, written prior permis
C.3.9 UUencode and UUdecode functions

The uu module contains the following notice:

Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. Th
  version is still 5 times faster, though.
- Arguments more compliant with python standard
C.3.10 XML Remote Procedure Calls

The xmlrpclib module contains the following notice:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicit pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
Index

Symbols

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

.ini
.file
.pdbrc
.file
.pythonrc.py
.file

<protocol>_proxy (environment variable)

% formatting
% interpolation
== operator
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>add</td>
<td>(AddressList method)</td>
</tr>
<tr>
<td>add</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>and</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>bases</td>
<td>(class attribute)</td>
</tr>
<tr>
<td>built-in</td>
<td>(built-in module)</td>
</tr>
<tr>
<td>class</td>
<td>(instance attribute)</td>
</tr>
<tr>
<td>cmp</td>
<td>(instance method)</td>
</tr>
<tr>
<td>concat</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>contains</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>contains</td>
<td>(Mailbox method)</td>
</tr>
<tr>
<td>contains</td>
<td>(Message method)</td>
</tr>
<tr>
<td>copy</td>
<td>(copy protocol)</td>
</tr>
<tr>
<td>deepcopy</td>
<td>(copy protocol)</td>
</tr>
<tr>
<td>delitem</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>delitem</td>
<td>(Mailbox method)</td>
</tr>
<tr>
<td>delitem</td>
<td>(Message method)</td>
</tr>
<tr>
<td>delitem</td>
<td>(MH method)</td>
</tr>
<tr>
<td>delslice</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>dict</td>
<td>(instance attribute)</td>
</tr>
<tr>
<td>dict</td>
<td>(object attribute)</td>
</tr>
<tr>
<td>displayhook</td>
<td>(in module sys)</td>
</tr>
<tr>
<td>div</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>enter</td>
<td>(context manager method)</td>
</tr>
<tr>
<td>eq</td>
<td>(Charset method)</td>
</tr>
<tr>
<td>eq</td>
<td>(Header method)</td>
</tr>
<tr>
<td>eq</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>excepthook</td>
<td>(in module sys)</td>
</tr>
<tr>
<td>exit</td>
<td>(context manager method)</td>
</tr>
<tr>
<td>floordiv</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>future</td>
<td>(standard module)</td>
</tr>
<tr>
<td>ge</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>getinitargs</td>
<td>(copy protocol)</td>
</tr>
<tr>
<td>ixor</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>le</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>len</td>
<td>(AddressList method)</td>
</tr>
<tr>
<td>len</td>
<td>(Mailbox method)</td>
</tr>
<tr>
<td>len</td>
<td>(Message method)</td>
</tr>
<tr>
<td>lshift</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>lt</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>main</td>
<td>(built-in module)</td>
</tr>
<tr>
<td>members</td>
<td>(object attribute)</td>
</tr>
<tr>
<td>methods</td>
<td>(object attribute)</td>
</tr>
<tr>
<td>missing</td>
<td>(LocaleHTMLCalendar method)</td>
</tr>
<tr>
<td>mod</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>mul</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>name</td>
<td>(class attribute)</td>
</tr>
<tr>
<td>ne</td>
<td>(Header method), [Link]</td>
</tr>
<tr>
<td>neg</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>not</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>or</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>pos</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>pow</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>repeat</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>repr</td>
<td>(netrc method)</td>
</tr>
<tr>
<td>rshift</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>setitem</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>setitem</td>
<td>(Mailbox method)</td>
</tr>
<tr>
<td>setitem</td>
<td>(Maildir method)</td>
</tr>
<tr>
<td>setitem</td>
<td>(Message method)</td>
</tr>
<tr>
<td>setslice</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>setstate</td>
<td>(copy protocol)</td>
</tr>
<tr>
<td>stderr</td>
<td>(in module sys)</td>
</tr>
<tr>
<td>stdin</td>
<td>(in module sys)</td>
</tr>
<tr>
<td>stdout</td>
<td>(in module sys)</td>
</tr>
</tbody>
</table>
getitem () (in module operator)
ggetitem () (Mailbox method)
ggetitem () (Message method)
getnewargs () (copy protocol)
getslice () (in module operator)
gestate () (copy protocol)
gt () (in module operator)
iadd () (AddressList method)
iadd () (in module operator)
inand () (in module operator)
iconcat () (in module operator)
idiv () (in module operator)
ifloordiv () (in module operator)
ilshift () (in module operator)
imod () (in module operator)
imul () (in module operator)
index () (in module operator)
init () (DbfilenameShelf method)
init () (in module difflib)
instantiate constructor)
instantiate NullTranslations method
inv () (in module operator)
imul () (in module operator)
iopow () (module operator)
irrepeat () (in module operator)
irshift () (in module operator)
isub () (AddressList method)
isub () (in module operator)
iter () (container method)
iter () (iterator method)
iter () (Mailbox method)
itruediv () (in module operator)
str () (AddressList method)
str () (Charset method)
str () (date method)
str () (datetime method)
str () (Header method)
str () (Message method)
str () (time method)
sub () (AddressList method)
sub () (in module operator)
truediv () (in module operator)
unicode () (Header method)
xor () (in module operator)
anonymous (Structure attribute)
b_base_ (CData attribute)
b_needsfree_ (CData attribute)
CData (class in ctypes)
current frames() (in module sys)
exit (in module os)
fields (Structure attribute)
flush () (BaseHandler method)
FuncPtr (class in ctypes)
getframe() (in module sys)
handle (PyDLL attribute)
locale (built-in module)
nname (PyDLL attribute)
_objects_ (CData attribute)
pack (Structure attribute)
personal() (NullTranslations method)
parse() (NullTranslations method)
setroot() (ElementTree method)
SimpleCData (class in ctypes)
structure() (in module email.iterators)
urloper (in module urllib)
winreg (extension module)
write () (BaseHandler method)
A-LAW, [Link]
a-LAW
a2b_base64() (in module binascii)
a2b_hex() (in module binascii)
a2b_hqx() (in module binascii)
a2b_qp() (in module binascii)
a2b_uu() (in module binascii)
ABDAY_1 ..., ABDAY_7 (in module locale)
ABMON_1 ..., ABMON_12 (in module locale)
abort() (FTP method)
abort() (in module os)
above() (Textbox method)
abs() (built-in function)
abs() (Context method)
abs() (in module operator)
abspath() (in module os.path)
AbstractBasicAuthHandler (class in urllib2)
AbstractDigestAuthHandler (class in urllib2)
AbstractFormatter (class in formatter)
AbstractWriter (class in formatter)
ac_in_buffer_size (in module asyncore)
ac_out_buffer_size (in module asyncore)
accept() (dispatcher method)
accept() (socket method)
accept2dyear (in module time)
aiff() (aifc method)
AIFF-C, [Link]
al (built-in module)
AL (standard module), [Link]
alarm() (in module signal)
alaw2lin() (in module audioop)
algorithm
AES
alignment() (in module ctypes)
all() (built-in function)
all_errors (in module ftplib)
al_features (in module xml.sax.handler)
addLevelName() (in module logging)
addnstr() (window method)
AddPackagePath() (in module modulefinder)
address_family (SocketServer protocol)
address_string()
(BaseHTTPRequestHandler method)
addresslist (AddressList attribute)
AddressList (class in rfc822)
adressof() (in module ctypes)
addstr() (window method)
addSuccess() (TestResult method)
addTest() (TestSuite method)
addTests() (TestSuite method)
adjusted() (Decimal method)
adler32() (in module zlib)
ADPCM, Intel/DVI
adpcm2lin() (in module audioop)
AES
    algorithm
AF_INET (in module socket)
AF_INET6 (in module socket)
AF_UNIX (in module socket)
AI_* (in module socket)
aifc() (aifc method)
aifc (standard module)
AIFF, [Link]
b16decode() (in module base64)
b16encode() (in module base64)
b2a_base64() (in module binascii)
b2a_hex() (in module binascii)
b2a_hqx() (in module binascii)
b2a_qp() (in module binascii)
b2a_uu() (in module binascii)
b32decode() (in module base64)
b32encode() (in module base64)
b64decode() (in module base64)
b64encode() (in module base64)
Babyl (class in mailbox)
BabylMailbox (class in mailbox)
BabylMessage (class in mailbox)
backslashreplace_errors_errors() (in module codecs)
backward() (in module turtle)
backward_compatible (in module imageop)
BadStatusLine
Balloon (class in Tix)
base64
    encoding
base64 (standard module), [Link]
BaseCGIHandler (class in wsgiref.handlers)
BaseCookie (class in Cookie)
BaseException
BaseHandler (class in urllib2)
BaseHandler (class in wsgiref.handlers)
BaseHTTPRequestHandler (class in BaseHTTPServer)
bkgdset() (window method)
blocked_domains()
    (DefaultCookiePolicy method)
BLOCKSIZE (in module cd)
blocksize (in module sha)
body() (NNTP method)
body_encode() (Charset method)
body_encoding (in module email.charset)
body_line_iterator() (in module email.iterators)
BOM (in module codecs)
BOM_BE (in module codecs)
BOM_LE (in module codecs)
BOM_UTF16 (in module codecs)
BOM_UTF16_BE (in module codecs)
BOM_UTF16_LE (in module codecs)
BOM_UTF32 (in module codecs)
BOM_UTF32_BE (in module codecs)
BOM_UTF32_LE (in module codecs)
BOM_UTF8 (in module codecs)
bool() (built-in function)
Boolean
    operations, [Link]
type
    values
boolean() (in module xmlrpclib)
Boolean object
BooleanType (in module types)
border() (window method)
bottom() (Textbox method)
bottom_panel() (in module curses.panel)
BoundaryError (exception in
<table>
<thead>
<tr>
<th>Function/Class/Module</th>
<th>Location/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseHTTPServer</td>
<td>(standard module)</td>
</tr>
<tr>
<td>basename()</td>
<td>(in module os.path)</td>
</tr>
<tr>
<td>BaseResult</td>
<td>(class in urllibparse)</td>
</tr>
<tr>
<td>basestring()</td>
<td>(built-in function)</td>
</tr>
<tr>
<td>basicConfig()</td>
<td>(in module urllib)</td>
</tr>
<tr>
<td>BasicContext</td>
<td>(class in decimal)</td>
</tr>
<tr>
<td>Bastion()</td>
<td>(in module Bastion)</td>
</tr>
<tr>
<td>Bastion</td>
<td>(standard module)</td>
</tr>
<tr>
<td>BastionClass</td>
<td>(class in Bastion)</td>
</tr>
<tr>
<td>baudrate()</td>
<td>(in module curses)</td>
</tr>
<tr>
<td>bdb</td>
<td>(standard module)</td>
</tr>
<tr>
<td>beep()</td>
<td>(in module curses)</td>
</tr>
<tr>
<td>Beep()</td>
<td>(in module winsound)</td>
</tr>
<tr>
<td>begin_fill()</td>
<td>(in module turtle)</td>
</tr>
<tr>
<td>below()</td>
<td>(Textbox method)</td>
</tr>
<tr>
<td>Benchmarking</td>
<td></td>
</tr>
<tr>
<td>benchmarking</td>
<td></td>
</tr>
<tr>
<td>bestreadsize()</td>
<td>(CD player method)</td>
</tr>
<tr>
<td>betavariate()</td>
<td>(in module random)</td>
</tr>
<tr>
<td>bgn_group()</td>
<td>(form method)</td>
</tr>
<tr>
<td>bias()</td>
<td>(in module audioop)</td>
</tr>
<tr>
<td>bidirectional()</td>
<td>(in module unicodedata)</td>
</tr>
<tr>
<td>BigEndianStructure</td>
<td>(class in ctypes)</td>
</tr>
<tr>
<td>binary</td>
<td></td>
</tr>
<tr>
<td>data, packing</td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td>(class in msilib)</td>
</tr>
<tr>
<td>binary semaphores</td>
<td></td>
</tr>
<tr>
<td>binascii</td>
<td>(built-in module)</td>
</tr>
<tr>
<td>bind()</td>
<td>(dispatcher method)</td>
</tr>
<tr>
<td>bind()</td>
<td>(socket method)</td>
</tr>
<tr>
<td>bind (widgets)</td>
<td></td>
</tr>
<tr>
<td>bind_textdomain_codeset()</td>
<td>(in module gettext)</td>
</tr>
<tr>
<td>bindtextdomain()</td>
<td>(in module gettext)</td>
</tr>
<tr>
<td>binhex()</td>
<td>(in module binhex)</td>
</tr>
<tr>
<td>binhex</td>
<td>(standard module), [Link]</td>
</tr>
<tr>
<td>bisect()</td>
<td>(in module bisect)</td>
</tr>
<tr>
<td>bisect (standard module)</td>
<td></td>
</tr>
<tr>
<td>bisect_left()</td>
<td>(in module bisect)</td>
</tr>
<tr>
<td>box()</td>
<td>(window method)</td>
</tr>
<tr>
<td>break_long_words (TextWrapper attribute)</td>
<td></td>
</tr>
<tr>
<td>BROWSER (environment variable), [Link]</td>
<td></td>
</tr>
<tr>
<td>bsddb</td>
<td>(built-in module), [Link], [Link]</td>
</tr>
<tr>
<td>bsddb (extension module)</td>
<td></td>
</tr>
<tr>
<td>BsdDbShelf (class in shelve)</td>
<td></td>
</tr>
<tr>
<td>btopen()</td>
<td>(in module bsddb)</td>
</tr>
<tr>
<td>buffer()</td>
<td>(built-in function), [Link]</td>
</tr>
<tr>
<td>buffer() (non-essential built-in functions)</td>
<td></td>
</tr>
<tr>
<td>buffer object</td>
<td></td>
</tr>
<tr>
<td>buffer size I/O</td>
<td></td>
</tr>
<tr>
<td>buffer_info()</td>
<td>(array method)</td>
</tr>
<tr>
<td>buffer_size (xmlparser attribute)</td>
<td></td>
</tr>
<tr>
<td>buffer_text (xmlparser attribute)</td>
<td></td>
</tr>
<tr>
<td>buffer_used (xmlparser attribute)</td>
<td></td>
</tr>
<tr>
<td>BufferingHandler (class in logging)</td>
<td></td>
</tr>
<tr>
<td>BufferType (in module types)</td>
<td></td>
</tr>
<tr>
<td>buFSIZE (audio device method)</td>
<td></td>
</tr>
<tr>
<td>build_opener()</td>
<td>(in module urllib2)</td>
</tr>
<tr>
<td>built-in</td>
<td></td>
</tr>
<tr>
<td>constants</td>
<td></td>
</tr>
<tr>
<td>exceptions</td>
<td></td>
</tr>
<tr>
<td>functions</td>
<td></td>
</tr>
<tr>
<td>types, [Link]</td>
<td></td>
</tr>
<tr>
<td>builtin_module_names</td>
<td>(in module sys)</td>
</tr>
<tr>
<td>BuiltinFunctionType (in module types)</td>
<td></td>
</tr>
<tr>
<td>BuiltinMethodType (in module types)</td>
<td></td>
</tr>
<tr>
<td>ButtonBox                                           (class in Tix)</td>
<td></td>
</tr>
<tr>
<td>byref()</td>
<td>(in module ctypes)</td>
</tr>
<tr>
<td>byte-code</td>
<td></td>
</tr>
<tr>
<td>file, [Link], [Link]</td>
<td></td>
</tr>
<tr>
<td>byteorder (in module sys)</td>
<td></td>
</tr>
<tr>
<td>bytes (UUID attribute)</td>
<td></td>
</tr>
</tbody>
</table>
bisect_right() (in module bisect)
bit-string operations
bitmap() (Dialog method)
bkgd() (window method)

bytes_le (UUID attribute)
byteswap() (array method)
bz2 (built-in module)
BZ2Compressor (class in bz2)
BZ2Decompressor (class in bz2)
BZ2File (class in bz2)
C

language, [Link] structures
C_BUILTIN (in module imp)
c_byte (class in ctypes)
c_char (class in ctypes)
c_char_p (class in ctypes)
c_double (class in ctypes)
C_EXTENSION (in module imp)
c_float (class in ctypes)
c_int (class in ctypes)
c_int16 (class in ctypes)
c_int32 (class in ctypes)
c_int64 (class in ctypes)
c_int8 (class in ctypes)
c_long (class in ctypes)
c_longlong (class in ctypes)
c_short (class in ctypes)
c_size_t (class in ctypes)
c_ubyte (class in ctypes)
c_uint (class in ctypes)
c_uint16 (class in ctypes)
c_uint32 (class in ctypes)
c_uint64 (class in ctypes)
c_uint8 (class in ctypes)
c_ulong (class in ctypes)
c_ulonglong (class in ctypes)

ComboBox (class in Tix)
command (BaseHTTPRequestHandler attribute)
CommandCompiler (class in codeop)
commands (standard module)
comment (Cookie attribute)
COMMENT (in module tokenize)
Comment() (in module
xml.etree.ElementTree)
comment (ZipInfo attribute)
comment_url (Cookie attribute)
commenters (shlex attribute)
CommentHandler() (xmlparser method)
Commit() (Binary method)
common (dircmp attribute)
Common Gateway Interface
common_dirs (dircmp attribute)
common_files (dircmp attribute)
common_funny (dircmp attribute)
common_types (in module mimetypes),
commonprefix() (in module os.path)
communicate() (Popen method)
compare() (Context method)
compare() (Decimal method)
compare() (Differ method)
comparing
   objects
comparison
   operator
   COMPARISON_FLAGS (in module
doctest)
comparisons
   chaining
| c_ushort (class in ctypes)  | compile() (AST method)         |
| c_void_p (class in ctypes)  | compile() (built-in function), [Link], [Link], [Link], [Link] |
| c_wchar (class in ctypes)   | Compile (class in codeop)       |
| c_wchar_p (class in ctypes) | compile() (in module compiler)  |
| CAB (class in msilib)       | compile() (in module py_compile) |
| CacheFTPHandler (class in urllib2) | compile() (in module re) |
| calcsiz() (in module struct) | compile_command() (in module code) |
| Calendar (class in calendar) | compile_command() (in module codeop) |
| calendar() (in module calendar) | compile_dir() (in module compileall) |
| calendar (standard module)  | compile_path() (in module compileall) |
| call() (in module subprocess) | compileall (standard module)     |
| call() (Timer method)       | compileleast() (in module parser) |
| callable() (built-in function) | compileFile() (in module compiler) |
| CallableProxyType (in module weakref) | compiler (module) |
| can_change_color() (in module curses) | compiler.ast (module) |
| can_fetch() (RobotFileParser method) | compiler.visitor (module) |
| cancel() (scheduler method) | complete() (Completer method)    |
| cancel() (Timer method)     | complete_statement() (in module sqlite3) |
| CannotSendHeader            | completdedefault() (Cmd method) |
| CannotSendRequest           | complex() (built-in function), [Link] |
| capitalize() (in module string) | complex number object |
| capitalize() (string method) | ComplexType (in module types)    |
| capwords() (in module string) | compress() (BZ2Compressor method) |
| cast() (in module ctypes)   | compress() (Compress method)     |
| cat() (in module nis)       | compress() (in module bz2)       |
| catalog (in module cd)      | compress() (in module jpeg)      |
| category() (in module unicodedata) | compress() (in module zlib) |
| cbreak() (in module curses) | compress_size (ZipInfo attribute) |
| cd (built-in module)        | compress_type (ZipInfo attribute) |
| CDLL (class in ctypes)      | CompressionError                 |
| CDROM (in module cd)        | compressobj() (in module zlib)   |
| ceil() (in module math), [Link] | COMSPEC (environment variable), [Link] |
| center() (in module string) | concat() (in module operator)    |
| center() (string method)    |                              |
| CFUNCTYPE() (in module ctypes) |                              |
| CGI debugging exceptions    |                              |
protocol
security
tracebacks
cgi (standard module)
cgi_directories
(CGIHTTPRequestHandler attribute)
CGIHandler (class in wsgiref.handlers)
CGIHTTPRequestHandler (class in
CGIHTTPServer)
CGIHTTPServer (standard module),
[Link]
cgitb (standard module)
CGIXMLRPCRequestHandler (class in
SimpleXMLRPCServer)
chain() (in module itertools)
chaining
  comparisons
channels() (audio device method)
CHAR_MAX (in module locale)
character
CharacterDataHandler() (xmlparser
method)
characters() (ContentHandler method)
Charset (class in email.charset)
CHARSET (in module mimify)
charset() (NullTranslations method)
chdir() (in module os)
check() (IMAP4_stream method)
check() (in module tabnanny)
check_call() (in module subprocess)
check_forms() (in module file)
check_output() (OutputChecker
method)
checkbox() (Dialog method)
checkcache() (in module linecache)
CheckList (class in Tix)
checksum
  Cyclic Redundancy Check
  MD5
concatenation
operation
Condition (class in threading)
condition() (Control method)
Condition() (in module threading)
ConfigParser (class in ConfigParser)
ConfigParser (standard module)
configuration
file
debugger
file, path
file, user
confstr() (in module os)
confstr_names (in module os)
conjugate() (complex number method)
connect() (dispatcher method)
connect() (FTP method)
connect() (HTTPResponse method)
connect() (in module sqlite3)
connect() (SMTP method)
connect() (socket method)
connect() (socket method)
ConnectRegistry() (in module _winreg)
constants
  built-in
constructor() (in module copy_reg)
container
  iteration over
contains() (in module operator)
content type
  MIME
ContentHandler (class in
xml.sax.handler)
ContentTooShortError (exception in
urllib)
Context (class in decimal)
context management protocol
context manager
context_diff() (in module difflib)
<table>
<thead>
<tr>
<th>SHA</th>
<th>contextlib (standard module)</th>
</tr>
</thead>
<tbody>
<tr>
<td>childerr (Popen4 attribute)</td>
<td>contextmanager() (in module</td>
</tr>
<tr>
<td>childNodes (Node attribute)</td>
<td>contextlib)</td>
</tr>
<tr>
<td>chmod() (in module os)</td>
<td>Control (class in msilib)</td>
</tr>
<tr>
<td>choice() (in module os)</td>
<td>Control (class in Tix)</td>
</tr>
<tr>
<td>choose_boundary() (in module mime\tools)</td>
<td>control() (Dialog method)</td>
</tr>
<tr>
<td>chown() (in module os)</td>
<td>control (in module cd)</td>
</tr>
<tr>
<td>chr() (built-in function)</td>
<td>controlnames (in module curses,asci)</td>
</tr>
<tr>
<td>chroot() (in module os)</td>
<td>controls() (mixer device method)</td>
</tr>
<tr>
<td>Chunk (class in chunk)</td>
<td>ConversionError</td>
</tr>
<tr>
<td>chunk (standard module)</td>
<td>conversions</td>
</tr>
<tr>
<td>cipher</td>
<td>numeric</td>
</tr>
<tr>
<td>DES</td>
<td>convert() (Charset method)</td>
</tr>
<tr>
<td>circle() (in module turtle)</td>
<td>convert_charref() (SGMLParser</td>
</tr>
<tr>
<td>Clamped (class in decimal)</td>
<td>method)</td>
</tr>
<tr>
<td>Class browser</td>
<td>convert_codepoint() (SGMLParser</td>
</tr>
<tr>
<td>classmethod() (built-in function)</td>
<td>method)</td>
</tr>
<tr>
<td>classobj() (in module new)</td>
<td>convert_entityref() (SGMLParser</td>
</tr>
<tr>
<td>ClassType (in module types)</td>
<td>method)</td>
</tr>
<tr>
<td>clean() (Maildir method)</td>
<td>Cookie (class in cookielib)</td>
</tr>
<tr>
<td>clear() (CookieJar method)</td>
<td>Cookie (standard module)</td>
</tr>
<tr>
<td>clear() (dictionary method)</td>
<td>CookieError</td>
</tr>
<tr>
<td>clear() (Event method)</td>
<td>cookiejar (UnknownHandler attribute)</td>
</tr>
<tr>
<td>clear() (in module turtle)</td>
<td>cookielib (standard module)</td>
</tr>
<tr>
<td>clear() (LocaleHTMLCalendar method)</td>
<td>CookiePolicy (class in cookielib)</td>
</tr>
<tr>
<td>clear() (Mailbox method)</td>
<td>copy() (Compress method)</td>
</tr>
<tr>
<td>clear() (window method)</td>
<td>copy() (Context method)</td>
</tr>
<tr>
<td>clear_flags() (Context method)</td>
<td>copy() (Decompress method)</td>
</tr>
<tr>
<td>clear_history() (in module readline)</td>
<td>copy() (dictionary method)</td>
</tr>
<tr>
<td>clear_mem() (Pickler method)</td>
<td>copy() (hash method)</td>
</tr>
<tr>
<td>clear_session_cookies() (CookieJar method)</td>
<td>copy() (hmac method)</td>
</tr>
<tr>
<td>clearcache() (in module linecache)</td>
<td>copy() (IMAP4_stream method)</td>
</tr>
<tr>
<td>ClearData() (Binary method)</td>
<td>copy() (in copy)</td>
</tr>
<tr>
<td>clearok() (window method)</td>
<td>copy() (in module shutil)</td>
</tr>
<tr>
<td>client_address</td>
<td>copy() (md5 method)</td>
</tr>
<tr>
<td>(BaseHTTPRequestHandler attribute)</td>
<td>copy() (sha method)</td>
</tr>
<tr>
<td>clock() (in module time)</td>
<td>copy (standard module), [Link]</td>
</tr>
<tr>
<td></td>
<td>copy() (Template method)</td>
</tr>
</tbody>
</table>
clone() (Generator method)
clone() (Template method)
cloneNode() (Node method)
close() (aifc method), [Link]
close() (AU_read method)
close() (AU_write method)
close() (audio device method), [Link]
close() (BaseHandler method)
Close() (Binary method)
close() (BZ2File method)
close() (CD player method)
close() (Chunk method)
close() (DbfilenameShelf method), [Link]
Close() (Dialog method)
close() (dispatcher method)
close() (FeedParser method)
close() (file method)
close() (FileHandler method)
close() (FTP method)
close() (HTMLParser method)
close() (HTTPResponse method)
close() (in module fileinput)
close() (in module os)
close() (IncrementalParser method)
close() (Mailbox method)
close() (Maildir method)
close() (MemoryHandler method)
close() (MH method)
close() (mixer device method)
close() (NTEventLogHandler method)
close() (Profile method)
close() (SGMLParser method)
close() (socket method)
close() (SocketHandler method)
close() (StringIO method)
close() (SysLogHandler method)
close() (TarFile method)
copy2() (in module shutil)
copy_reg (standard module)
copybinary() (in module mimetypes)
copyfile() (in module shutil)
copyfileobj() (in module shutil)
copying files
copyliteral() (in module mimetypes)
copypassword() (Folder method)
copymode() (in module shutil)
copystat() (in module shutil)
copytree() (in module shutil)
cos() (in module cmath)
cos() (in module math)
cosh() (in module cmath)
cosh() (in module math)
count() (array method)
count() (in module itertools)
count() (in module string)
count() (list method)
count() (string method)
countOf() (in module operator)
countTestCases() (TestCase method)
countTestCases() (TestSuite method)
cPickle (built-in module), [Link]
cProfile (standard module)
CPU time
CRC (ZipInfo attribute)
crc32() (in module binascii)
crc32() (in module zlib)
crc_hqx() (in module binascii)
create() (IMAP4_stream method)
create_aggregate() (DbfilenameShelf method)
create_collation() (DbfilenameShelf method)
create_decimal() (Context method)
create_function() (DbfilenameShelf method)
close() (Telnet method)
close() (Timer method), [Link]
close() (TreeBuilder method)
close() (Wave_read method)
close() (Wave_write method)
close() (XMLTreeBuilder method)
close() (ZipFile method)
close_when_done() (async_chat method)
closed (audio device attribute)
closed (file attribute)
CloseKey() (in module _winreg)
closelog() (in module syslog)
closeport() (audio port method)
closing() (in module contextlib)
clrtobot() (window method)
clrtoeof() (window method)
cmath (built-in module)
Cmd (class in cmd)
cmd (standard module), [Link]
cmdloop() (Cmd method)
cmp() (built-in function), [Link]
cmp() (in module filecmp)
cmp_op (in module dis)
cmpfiles() (in module filecmp)
code (ExpatError attribute)
code() (in module new)
code (standard module)
code object, [Link]
Codecs
decode
encode
codecs (standard module)
coded_value (Morsel attribute)
codeop (standard module)
codepoint2name (in module htmlentitydefs)
CODESET (in module locale)
CodeType (in module types)
create_socket() (dispatcher method)
create_string_buffer() (in module ctypes)
create_system (ZipInfo attribute)
create_unicode_buffer() (in module ctypes)
create_version (ZipInfo attribute)
createAttribute() (Document method)
createAttributeNS() (Document method)
createComment() (Document method)
createDocument() (DOMImplementation method)
createDocumentType() (DOMImplementation method)
createElement() (Document method)
createElementNS() (Document method)
CreateKey() (in module _winreg)
createLock() (DbfilenameShelf method)
createparser() (in module cd)
createProcessingInstruction() (Document method)
CreateRecord() (in module msilib)
createTextNode() (Document method)
critical() (DbfilenameShelf method)
critical() (in module logging)
CRNCYSTR (in module locale)
crop() (in module imageop)
cross() (in module audioop)
crypt (built-in module), [Link]
crypt() (in module crypt)
crypt(3), [Link], [Link]
cryptography, [Link]
cStringIO (built-in module)
csv
csv (standard module)
ctermid() (in module os)
cftime() (date method)
coerce() (non-essential built-in functions)
collapse_rfc2231_value() (in module email.utils)
collect() (in module gc)
collect_incoming_data() (async_chat method)
collections (standard module)
color() (in module fl)
color() (in module turtle)
color_content() (in module curses)
color_pair() (in module curses)
colorsys (standard module)
COLUMNS (environment variable),
combine() (datetime method)
combining() (in module unicodedata)
ctime() (datetime method)
ctime() (in module time)
ctrl() (in module curses.ascii)
crypt() (standard module)
curdir (in module os)
currency() (in module locale)
CurrentByteIndex (xmlparser attribute)
CurrentColumnNumber (xmlparser attribute)
currentframe() (in module inspect)
CurrentLineNumber (xmlparser attribute)
currentThread() (in module threading)
curs_set() (in module curses)
curses (standard module)
curses.ascii (standard module)
curses.panel (standard module)
curses.textpad (standard module)
curses.wrapper (standard module)
cursor() (DbfilenameShelf method)
cursyncup() (window method)
cwd() (FTP method)
cycle() (in module itertools)
Cyclic Redundancy Check
D

D_FMT (in module locale)
D_T_FMT (in module locale)
data
  packing binary
tabular
data (Binary attribute)
data (Comment attribute)
data (IterableUserDict attribute)
data (MutableString attribute)
data (ProcessingInstruction attribute)
data (Text attribute)
data() (TreeBuilder method)
data (UserList attribute)
database
  Unicode
databases
  DatagramHandler (class in logging)
DATASIZE (in module cd)
date (class in datetime), [Link]
date() (datetime method)
date() (NNTP method)
date_time (ZipInfo attribute)
date_time_string()
(BaseHTTPRequestHandler method)
datetime (built-in module)
datetime (class in datetime), [Link]
day (date attribute)
day (datetime attribute)
DAY_1 ... DAY_7 (in module locale)
day_abbr (in module calendar)
day_name (in module calendar)
daylight (in module time)
Dialect (class in csv)
dialect (csv reader attribute)
dialect (csv writer attribute)
Dialog (class in msilib)
dict() (built-in function)
dictionary
type, operations on
dictionary object
DictionaryType (in module types)
DictMixin (class in UserDict)
DictProxyType (in module types)
DictReader (class in csv)
DictType (in module types)
DictWriter (class in csv)
diff_files (dircmp attribute)
Differ (class in difflib), [Link]
difflib (standard module)
digest() (hash method)
digest() (hmac method)
digest() (md5 method)
digest() (sha method)
digest_size (in module hashlib)
digest_size (in module md5)
digest_size (in module sha)
digit() (in module unicodedata)
digits (in module string)
dir() (built-in function)
dir() (FTP method)
dircache (standard module)
dircmp (class in filecmp)
directory
  changing
creating
deleting, [Link]
Daylight Saving Time
dbfilenameShelf (class in shelve)
dbhash (standard module), [Link]
dbm (built-in module), [Link], [Link],
[Link]
deactivate_form() (form method)
debug() (DbfilenameShelf method)
debug (IMAP4_stream attribute)
debug() (in module doctest)
debug() (in module logging)
debug (shlex attribute)
debug() (Template method)
debug() (TestCase method)
debug() (TestSuite method)
debug (ZipFile attribute)
debug=0 (TarFile attribute)
DEBUG_COLLECTABLE (in module gc)
DEBUG_INSTANCES (in module gc)
DEBUG_LEAK (in module gc)
DEBUG_OBJECTS (in module gc)
DEBUG_SAVEALL (in module gc)
debug_src() (in module doctest)
DEBUGSTATS (in module gc)
DEBUG_UNCOLLECTABLE (in module gc)
debugger, [Link]
configuration file
debugging
CGI
DebuggingServer (class in smtpd)
DebugRunner (class in doctest)
Decimal (class in decimal)
decimal() (in module unicodedata)
decimal (standard module)
DecimalException (class in decimal)
decode
    Codecs
decode() (Binary method)
site-packages
site-python
traversal
walking

Directory (class in msilib)
DirList (class in Tix)
dirname() (in module os.path)
DirSelectBox (class in Tix)
DirSelectDialog (class in Tix)
DirTree (class in Tix)
dis() (in module dis)
dis() (in module pickletools)
dis (standard module)
disable() (in module gc)
disable() (in module logging)
disassemble() (in module dis)
discard (Cookie attribute)
discard() (Mailbox method)
discard() (MH method)
discardBuffers() (async_chat method)
disco() (in module dis)
dispatch() (ASTVisitor method)
dispatcher (class in asyncore)
displayhook() (in module sys)
dist() (in module platform)
distb() (in module dis)
distutils (standard module)
dither2grey2() (in module imageop)
dither2mono() (in module imageop)
div() (in module operator)
divide() (Context method)
divide
    integer
    long
DivisionByZero (class in decimal)
divmod() (built-in function)
divmod() (Context method)
dl (extension module)
DllCanUnloadNow() (in module
<table>
<thead>
<tr>
<th>dereference (TarFile attribute)</th>
<th>dump() (in module xml.etree.ElementTree)</th>
</tr>
</thead>
<tbody>
<tr>
<td>derwin() (window method)</td>
<td>dump() (Pickler method)</td>
</tr>
<tr>
<td>DES</td>
<td>dump_address_pair() (in module rfc822)</td>
</tr>
<tr>
<td>cipher</td>
<td>dump_stats() (Stats method)</td>
</tr>
<tr>
<td>description() (NNTP method)</td>
<td>dumps() (in module marshal)</td>
</tr>
<tr>
<td>descriptions() (NNTP method)</td>
<td>dumps() (in module pickle)</td>
</tr>
<tr>
<td>descriptor, file</td>
<td>dumps() (in module xmlrpclib)</td>
</tr>
<tr>
<td>Detach() (Dialog method)</td>
<td>dup() (in module os)</td>
</tr>
<tr>
<td>deterministic profiling</td>
<td>dup() (posixfile method)</td>
</tr>
<tr>
<td>DEVICE (standard module)</td>
<td>dup2() (in module os)</td>
</tr>
<tr>
<td>devnull (in module os)</td>
<td>dup2() (posixfile method)</td>
</tr>
<tr>
<td>dgettext() (in module gettext)</td>
<td>DuplicateSectionError</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Environment Variables</td>
<td>Corresponding Modules</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>LANG, [Link], [Link], [Link], [Link]</td>
<td>LANG, [Link], [Link], [Link], [Link]</td>
</tr>
<tr>
<td>LANGUAGE, [Link]</td>
<td>LANGUAGE, [Link]</td>
</tr>
<tr>
<td>LC_ALL, [Link]</td>
<td>LC_ALL, [Link]</td>
</tr>
<tr>
<td>LC_MESSAGES, [Link]</td>
<td>LC_MESSAGES, [Link]</td>
</tr>
<tr>
<td>LINES, [Link]</td>
<td>LINES, [Link]</td>
</tr>
<tr>
<td>LNAME</td>
<td>LNAME</td>
</tr>
<tr>
<td>LOGNAME, [Link]</td>
<td>LOGNAME, [Link]</td>
</tr>
<tr>
<td>MIXERDEV</td>
<td>MIXERDEV</td>
</tr>
<tr>
<td>PATH, [Link], [Link], [Link], [Link]</td>
<td>PATH, [Link], [Link], [Link], [Link]</td>
</tr>
<tr>
<td>[Link], [Link], [Link], [Link]</td>
<td>[Link], [Link], [Link], [Link]</td>
</tr>
<tr>
<td>PYTHON_DOM</td>
<td>PYTHON_DOM</td>
</tr>
<tr>
<td>PYTHONDOCS</td>
<td>PYTHONDOCS</td>
</tr>
<tr>
<td>PYTHONPATH, [Link], [Link]</td>
<td>PYTHONPATH, [Link], [Link]</td>
</tr>
<tr>
<td>PYTHONSTARTUP, [Link], [Link]</td>
<td>PYTHONSTARTUP, [Link], [Link]</td>
</tr>
<tr>
<td>[Link]</td>
<td>[Link]</td>
</tr>
<tr>
<td>PYTHONY2K, [Link], [Link]</td>
<td>PYTHONY2K, [Link], [Link]</td>
</tr>
<tr>
<td>setting</td>
<td>setting</td>
</tr>
<tr>
<td>TEMP</td>
<td>TEMP</td>
</tr>
<tr>
<td>TIX_LIBRARY</td>
<td>TIX_LIBRARY</td>
</tr>
<tr>
<td>TMP, [Link]</td>
<td>TMP, [Link]</td>
</tr>
<tr>
<td>TMPDIR, [Link]</td>
<td>TMPDIR, [Link]</td>
</tr>
<tr>
<td>TZ, [Link], [Link], [Link], [Link]</td>
<td>TZ, [Link], [Link], [Link], [Link]</td>
</tr>
<tr>
<td>USER</td>
<td>USER</td>
</tr>
<tr>
<td>USERNAME</td>
<td>USERNAME</td>
</tr>
<tr>
<td>Wimp$ScrapDir</td>
<td>Wimp$ScrapDir</td>
</tr>
<tr>
<td>EnvironmentError</td>
<td>EnvironmentError</td>
</tr>
<tr>
<td>ENXIO, [module errno]</td>
<td>ENXIO, [module errno]</td>
</tr>
<tr>
<td>eof, [shlex attribute]</td>
<td>eof, [shlex attribute]</td>
</tr>
<tr>
<td>EOFError</td>
<td>EOFError</td>
</tr>
<tr>
<td>EOPNOTSUPP, [module errno]</td>
<td>EOPNOTSUPP, [module errno]</td>
</tr>
</tbody>
</table>
edit() (Textbox method)
EDOM (in module errno)
EDOTDOT (in module errno)
EDQUOT (in module errno)
EEXIST (in module errno)
EFAULT (in module errno)
EFBIG (in module errno)
ehlo() (SMTP method)
EHOSTDOWN (in module errno)
EHOSTUNREACH (in module errno)
EIDRM (in module errno)
EILSEQ (in module errno)
EINPROGRESS (in module errno)
EINTR (in module errno)
EINVAL (in module errno)
EIO (in module errno)
EISCONN (in module errno)
EISDIR (in module errno)
EISNAM (in module errno)
eject() (CD player method)
EL2HLT (in module errno)
EL2NSYNC (in module errno)
EL3HLT (in module errno)
EL3RST (in module errno)
ELNAME (in module errno)
Element() (in module xml.etree.ElementTree)
ElementDeclHandler() (xmlparser method)
ElementTree (class in xml.etree.ElementTree)
ELIBACC (in module errno)
ELIBBAD (in module errno)
ELIBEXEC (in module errno)
ELIBMAX (in module errno)
ELIBSCN (in module errno)
Ellinghouse, Lance
Ellipsis (built-in warning)
ELLIPSIS (in module doctest)
EllipsisType (in module types)
EOVERFLOW (in module errno)
EPERM (in module errno)
EPFNOSUPPORT (in module errno)
epilogue (in module email.message)
EPIPE (in module errno)
epoch
EPROTO (in module errno)
EPROTONOSUPPORT (in module errno)
EPROTOSOCK (in module errno)
EPROTOTYPE (in module errno)
eq() (in module operator)
ERA (in module locale)
ERA_D_FMT (in module locale)
ERA_D_T_FMT (in module locale)
ERA_YEAR (in module locale)
ERANGE (in module errno)
erase() (window method)
erasechar() (in module curses)
EREMCHG (in module errno)
EREMOTE (in module errno)
EREMOTEIO (in module errno)
ERESTART (in module errno)
EROFS (in module errno)
ERR (in module curses)
errcheck (_FuncPtr attribute)
errcode (ServerProxy attribute)
errmsg (ServerProxy attribute)
errno (built-in module)
errno (standard module)
Error (class in mailbox)
error() (DbfilenameShelf method)
<table>
<thead>
<tr>
<th>Function/Class/Method</th>
<th>Module/Submodule</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty() (scheduler method)</td>
<td>xml.dom</td>
</tr>
<tr>
<td>EMPTY_NAMESPACE (in module xml.dom)</td>
<td></td>
</tr>
<tr>
<td>emptyline() (Cmd method)</td>
<td></td>
</tr>
<tr>
<td>EMSIZE (in module errno)</td>
<td></td>
</tr>
<tr>
<td>EMULTIHOP (in module errno)</td>
<td></td>
</tr>
<tr>
<td>enable() (in module cgitb)</td>
<td></td>
</tr>
<tr>
<td>enable() (in module gc)</td>
<td></td>
</tr>
<tr>
<td>enable_callback_tracebacks() (in module sqlite3)</td>
<td></td>
</tr>
<tr>
<td>ENAME_TOO_LONG (in module errno)</td>
<td></td>
</tr>
<tr>
<td>EAVAIL (in module errno)</td>
<td></td>
</tr>
<tr>
<td>enclose() (window method)</td>
<td></td>
</tr>
<tr>
<td>encode</td>
<td></td>
</tr>
<tr>
<td>Codecs</td>
<td></td>
</tr>
<tr>
<td>encode() (Binary method)</td>
<td></td>
</tr>
<tr>
<td>encode() (Header method)</td>
<td></td>
</tr>
<tr>
<td>encode() (in module base64)</td>
<td></td>
</tr>
<tr>
<td>encode() (in module mimetools)</td>
<td></td>
</tr>
<tr>
<td>encode() (in module quopri)</td>
<td></td>
</tr>
<tr>
<td>encode() (in module uu)</td>
<td></td>
</tr>
<tr>
<td>encode() (IncrementalEncoder method)</td>
<td></td>
</tr>
<tr>
<td>encode() (ServerProxy method), [Link]</td>
<td></td>
</tr>
<tr>
<td>encode() (string method)</td>
<td></td>
</tr>
<tr>
<td>encode() (TextWrapper method)</td>
<td></td>
</tr>
<tr>
<td>encode_7or8bit() (in module email.encoders)</td>
<td></td>
</tr>
<tr>
<td>encode_base64() (in module email.encoders)</td>
<td></td>
</tr>
<tr>
<td>encode_noop() (in module email.encoders)</td>
<td></td>
</tr>
<tr>
<td>encode_quopri() (in module email.encoders)</td>
<td></td>
</tr>
<tr>
<td>encode/rfc2231() (in module email.utils)</td>
<td></td>
</tr>
<tr>
<td>encoded_header_len() (Charset method)</td>
<td></td>
</tr>
<tr>
<td>EncodedFile() (in module codecs)</td>
<td></td>
</tr>
<tr>
<td>encodePriority() (SysLogHandler</td>
<td></td>
</tr>
<tr>
<td>ESPIPE (in module errno)</td>
<td></td>
</tr>
<tr>
<td>ESRCH (in module errno)</td>
<td></td>
</tr>
<tr>
<td>ESRMNT (in module errno)</td>
<td></td>
</tr>
<tr>
<td>ESTALE (in module errno)</td>
<td></td>
</tr>
<tr>
<td>ESTRPIPE (in module errno)</td>
<td></td>
</tr>
<tr>
<td>ETIME (in module errno)</td>
<td></td>
</tr>
<tr>
<td>ETIMEDOUT (in module errno)</td>
<td></td>
</tr>
<tr>
<td>Etoo() (Context method)</td>
<td></td>
</tr>
<tr>
<td>EUSERS (in module errno)</td>
<td></td>
</tr>
<tr>
<td>EWOULDEnviar (in module errno)</td>
<td></td>
</tr>
<tr>
<td>EX_CANTCREAT (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_CONFIG (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_DATAERR (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_IOERR (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_NOHOST (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_NOINPUT (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_NOPERM (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_NOTFOUND (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_NOUSER (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_OK (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_OSERR (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_OSFILE (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_PROTOCOL (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_SOFTWARE (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_TEMPFAIL (in module os)</td>
<td></td>
</tr>
<tr>
<td>EX_UNAVAILABLE (in module os)</td>
<td></td>
</tr>
</tbody>
</table>
encodestring() (in module base64)
encodestring() (in module quopri)

encoding
    base64
    quoted-printable

encoding (file attribute)
encodings.idna (standard module)
encodings.utf-8-sig (standard module)
encodings_map (in module mimetypes), [Link]

end() (Template method)
end() (TreeBuilder method)
end_fill() (in module turtle)
end_group() (form method)
end_headers()
(BaseHTTPRequestHandler method)
end_marker() (MultiFile method)
end_paragraph() (formatter method)
EndCdataSectionHandler() (xmlparser method)
EndDoctypeDeclHandler() (xmlparser method)
endDocument() (ContentHandler method)
endElement() (ContentHandler method)
EndElementHandler() (xmlparser method)
endElementNS() (ContentHandler method)
endheaders() (HTTPResponse method)
EndNamespaceDeclHandler() (xmlparser method)
endpick() (in module gl)
endpos (MatchObject attribute)
endPrefixMapping() (ContentHandler method)
endselect() (in module gl)
endswith() (string method)

EX_USAGE (in module os)
Example (class in doctest)
example (DocTestFailure attribute)
example (UnexpectedException attribute)
examples (DocTest attribute)
exc_clear() (in module sys)
exc_info() (in module sys)
exc_info (UnexpectedException attribute)
exc_msg (Example attribute)
exc_traceback (in module sys)
exc_type (in module sys)
exc_value (in module sys)
excel (class in csv)
excel_tab (class in csv)
except statement
excepthook() (in module sys), [Link]
Exception
exception() (DbfilenameShelf method)
exception() (in module logging)
exceptions
    built-in
    in CGI scripts
exceptions (standard module)
EXDEV (in module errno)
exec statement
exec_prefix (in module sys)
execfile() (built-in function), [Link]
execl() (in module os)
execl() (in module os)
execlp() (in module os)
execlpe() (in module os)
executable (in module sys)
Execute() (Binary method)
execute() (DbfilenameShelf method), [Link]
executemany() (DbfilenameShelf method), [Link]
enumerate() (built-in function)
enumerate() (in module fm)
enumerate() (in module threading)
EnumKey() (in module _winreg)
EnumValue() (in module _winreg)
environ (in module os)
environ (in module posix)
environment variables

<protocol>_proxy
AUDIODEV
BROWSER, [Link]
COLUMNNS, [Link]
COMSPEC, [Link]
deleting
ftp_proxy
gopher_proxy
HOME, [Link], [Link]
HOMEDRIVE
HOMEPATH
http_proxy, [Link]
KDEDIR

extendleft() (LocaleHTMLCalendar method)
extensions_map
(SimpleHTTPRequestHandler attribute)
External Data Representation, [Link]
external_attr (ZipInfo attribute)
ExternalClashError (class in mailbox)
ExternalEntityParserCreate()
(xmlparser method)
ExternalEntityRefHandler() (xmlparser method)
extra (ZipInfo attribute)
extract() (TarFile method)
extract_cookies() (CookieJar method)
extract_stack() (in module traceback)
extract_tb() (in module traceback)
extract_version (ZipInfo attribute)
extractall() (TarFile method)
ExtractError
extractfile() (TarFile method)
extsep (in module os)
F_BAVAIL (in module statvfs)
F_BFREE (in module statvfs)
F_BLOCKS (in module statvfs)
F_BSIZE (in module statvfs)
F_FAVAIL (in module statvfs)
F_FFREE (in module statvfs)
F_FILES (in module statvfs)
F_FLAG (in module statvfs)
F_FRSIZE (in module statvfs)
F_NAMEMAX (in module statvfs)
F_OK (in module os)
fabs() (in module math)
fail() (TestCase method)
failIf() (TestCase method)
failIfAlmostEqual() (TextTestRunner method)
failIfEqual() (TextTestRunner method)
failUnless() (TextTestRunner method)
failUnlessAlmostEqual() (TextTestRunner method)
failUnlessEqual() (TextTestRunner method)
failUnlessRaises() (TextTestRunner method)
failureException (TestCase attribute)
failures (TestResult attribute)
False, [Link]
false
False (Built-in object)
False (built-in warning)
family (socket attribute)
FancyURLopener (class in urllib)
fatalError() (ErrorHandler method)
faultCode (ServerProxy attribute)
findtext() (ElementTree method)
finish() (SocketServer protocol)
finish_request() (SocketServer protocol)
first() (DbfilenameShelf method)
first() (dbhash method)
first() (fifo method)
firstChild (Node attribute)
firstkey() (in module gdbm)
firstweekday() (in module calendar)
fix() (in module fpformat)
fix_sentence_endings (TextWrapper attribute)
fl (built-in module)
FL (standard module)
flag_bits (ZipInfo attribute)
flags() (posixfile method)
flags (RegexObject attribute)
flash() (in module curses)
flatten() (Generator method)
flattening objects
floating point literals
floating point object
FloatingPointError, [Link]
FloatType (in module types)
flock() (in module fcntl)
floor() (in module math), [Link]
large files
mime.types
path configuration
temporary
user configuration
file() (built-in function), [Link]
file (class descriptor attribute)
file (function descriptor attribute)
file() (posixfile method)
file control
Unix
file descriptor
file name
temporary
file object
POSIX
file_open() (FileHandler method)
file_size (ZipInfo attribute)
filecmp (standard module)
fileConfig() (in module logging)
FileCookieJar (class in cookielib)
FileEntry (class in Tix)
FileHandler (class in logging)
FileHandler (class in urllib2)
FileInput (class in fileinput)
fileinput (standard module)
filelineno() (in module fileinput)
filename (Cookie attribute)
filename (DocTest attribute)
filename() (in module fileinput)
filename (ZipInfo attribute)
filename_only (in module tabnanny)
filenames
pathname expansion
wildcard expansion
fileno() (audio device method), [Link]
fileno() (file method)
fileno() (in module fileinput)
fileno() (mixer device method)
traceback
format_exception_only() (in module traceback)
format_list() (in module traceback)
format_stack() (in module traceback)
format_string() (in module locale)
format_tb() (in module traceback)
formataddr() (in module email.utils)
formatargspec() (in module inspect)
formatargvalues() (in module inspect)
formatdate() (in module email.utils)
FormatError (class in mailbox)
FormatError() (in module ctypes)
formatException() (Formatter method)
formatmonth() (HTMLCalendar method)
formatmonth() (TextCalendar method)
Formatter (class in logging)
formatter (HTMLParser attribute)
formatter (standard module), [Link]
formatTime() (Formatter method)
formatting_string (%)
formatwarning() (in module warnings)
formatyear() (HTMLCalendar method)
formatyear() (TextCalendar method)
formatyearpage() (HTMLCalendar method)
forward() (in module turtle)
found_terminator() (async_chat method)
fp (AddressList attribute)
fpformat (standard module)
fpexact (extension module)
frame (ScrolledText attribute)
FrameType (in module types)
freeze_form() (form method)
freeze_object() (FORMS object method)
fileno() (Profile method)
fileno() (socket method)
fileno() (SocketServer protocol)
fileno() (Telnet method)
fileopen() (in module posixfile)
FileSelectBox (class in Tix)
FileType (in module types)
FileWrapper (class in wsgiref.util)
fill() (in module textwrap)
fill() (in module turtle)
fill() (TextWrapper method)
filter() (built-in function)
Filter (class in logging)
filter() (DbfilenameShelf method), [Link]
filter() (Filter method)
filter() (in module curses)
filter() (in module fnmatch)
filterwarnings() (in module warnings)
find() (DocTestFinder method)
find() (ElementTree method)
find() (in module gettext)
find() (in module string)
find() (string method)
find() (Timer method)
find_first() (form method)
find_last() (form method)
find_longest_match()
(SequenceMatcher method)
find_module() (in module imp)
find_module() (NullImporter method)
find_module() (zipimporter method)
find_prefix_at_end() (in module asynchat)
find_user_password()
(HTTPPasswordMgr method)
findall() (ElementTree method)
findall() (in module re)
findall() (RegexObject method)
frexp() (in module math)
from_address() (CData method)
from_param() (CData method)
from_splittable() (Charset method)
frombuf() (TarInfo method)
fromchild (Popen4 attribute)
fromfd() (in module socket)
fromfile() (array method)
fromkeys() (dictionary method)
fromlist() (array method)
fromordinal() (date method)
fromordinal() (datetime method)
fromstring() (array method)
fromstring() (in module xml.etree.ElementTree)
fromtimestamp() (date method)
fromtimestamp() (datetime method)
fromunicode() (array method)
fromutc() (time method)
frozenset() (built-in function)
fstat() (in module os)
fstatvfs() (in module os)
fsync() (in module os)
FTP
ftplib (standard module)
protocol, [Link]
FTP (class in ftplib)
ftp_open() (FTPHandler method)
ftp_proxy (environment variable)
FTPHandler (class in urllib2)
ftplib (standard module)
ftpmirror.py
ftruncate() (in module os)
Full
full() (Queue method)
func (callable attribute)
func_code (function object attribute)
function() (in module new)
functions
findCaller() (DbfilenameShelf method)  built-in
findfactor() (in module audioop)
findfile() (in module test.test_support)
findfit() (in module audioop)
findfont() (in module fm)
finditer() (in module re)
finditer() (RegexObject method)
findmatch() (in module mailcap)
findmax() (in module audioop)

FunctionTestCase (class in unittest)
FunctionType (in module types)
functools (standard module)
funny_files (dircmp attribute)
FutureWarning
G.722
gaierror
gammavariate() (in module random)
garbage (in module gc)
gather() (Textbox method)
gauss() (in module random)
gc (extension module)
gdbm (built-in module), [Link], [Link]
ge() (in module operator)
gen_uuid() (in module msilib)
generate_tokens() (in module tokenize)
Generator (class in email.generator)
GeneratorExit
GeneratorType (in module types)
genops() (in module pickletools)
ge() (AddressList method)
ge() (dictionary method)
ge() (in module webbrowser)
ge() (Mailbox method)
ge() (Message method)
ge() (mixer device method)
ge() (Queue method)
ge() (SafeConfigParser method), [Link]
ge_all() (Headers method)
ge_all() (Message method)
ge_app() (WSGIServer method)
ge_begidx() (in module readline)
ge_body_encoding() (Charset method)
ge_boundary_encoding() (Charset method)
ge_buffer() (Message method)
ge_buffer() (Unpacker method)
ge_charset() (Message method)
ge_charset() (Message method)
getgroups() (in module os)
getheader() (AddressList method)
getheader() (HTTPHeaderResponse method)
getheaders() (HTTPHeaderResponse method)
gehostbyaddr() (in module socket), [Link]
gehostbyname() (in module socket)
gehostbyname_ex() (in module socket)
gehostname() (in module socket), [Link]
geincrementaldecoder() (in module codecs)
geincrementalencoder() (in module codecs)
geinfo() (audio device method)
geinfo() (ZipFile method)
geinnerframes() (in module inspect)
GetInputContext() (xmlparser method)
geint() (SafeConfigParser method)
geitem() (in module operator)
getiterator() (ElementTree method)
gekey() (window method)
geLastError() (in module ctypes)
geLength() (Attributes method)
geLevelName() (in module logging)
getline() (in module linecache)
get_close_matches() (in module difflib)
get_code() (zipimporter method)
get_completer() (in module readline)
get_completer_delims() (in module readline)
get_content_charset() (Message method)
get_content_maintype() (Message method)
get_content_subtype() (Message method)
get_content_type() (Message method)
get_count() (in module gc)
get_current_history_length() (in module readline)
get_data() (Request method)
get_data() (zipimporter method)
get_date() (MaildirMessage method)
get_debug() (in module gc)
get_default_domain() (in module nis)
get_default_type() (Message method)
get_dialect() (in module csv)
get_directory() (in module fl)
get_doctest() (DocTestParser method)
get_endidx() (in module readline)
get_environ() (WSGIRequestHandler method)
get_examples() (DocTestParser method)
get_file() (Babyl method)
get_file() (Mailbox method)
get_file() (Maildir method)
get_file() (mbox method)
get_file() (MH method)
get_file() (MMDF method)
get_filename() (in module fl)
get_filename() (Message method)
get_flags() (MaildirMessage method)
get_flags() (mboxMessage method)
getLineNumber() (Locator method)
getlist() (FieldStorage method)
getloadavg() (in module os)
getLocale() (in module locale)
getLogger() (in module logging)
getLoggerClass() (in module logging)
getlogin() (in module os)
getmaintype() (Message method)
getmark() (aifc method)
getmark() (AU_read method)
getmark() (Wave_read method)
getmarkers() (aifc method)
getmarkers() (AU_read method)
getmarkers() (Wave_read method)
getmaxyx() (window method)
getmcolor() (in module fl)
getmember() (TarFile method)
getmembers() (in module inspect)
getmembers() (TarFile method)
getMessage() (LogRecord method)
getMessage() (SAXException method)
getMessagefilename() (Folder method)
getMessageID() (NTEventLogHandler method)
getmodule() (in module inspect)
getmoduleinfo() (in module inspect)
getmodulename() (in module inspect)
getmouse() (in module curses)
getmro() (in module inspect)
getmtime() (in module os.path)
getName() (Chunk method)
getName() (Thread method)
getNameByQName() (AttributesNS method)
getNameinfo() (in module socket)
getNames() (Attributes method)
getnames() (TarFile method)
getchannels() (aifc method)
getchannels() (AU_read method)
get_flags() (MMDFMessage method)
get_folder() (Maildir method)
get_folder() (MH method)
get_from() (mboxMessage method)
get_from() (MMDFMessage method)
get_full_url() (Request method)
get_grouped_opcodes()
(SequenceMatcher method)
get_history_item() (in module readline)
get_history_length() (in module readline)
get_host() (Request method)
get_ident() (in module thread)
get_info() (MaildirMessage method)
get_labels() (Babyl method)
get_labels() (BabylMessage method)
get_line_buffer() (in module readline)
get_magic() (in module imp)
get_matching_blocks()
(SequenceMatcher method)
get_message() (Mailbox method)
get_method() (Request method)
get_mouse() (in module fl)
get_nonstandard_attr() (Cookie method)
get_nowait() (Queue method)
get_objects() (in module gc)
get_opcodes() (SequenceMatcher method)
georigin_rec_host() (Request method)
geosfhandle() (in module msvcrt)
geoutput_charset() (Charset method)
geparam() (Message method)
geparams() (aifc method)
geparams() (AU_read method)
geparams() (in module al)
geparams() (Wave_read method)
geparyx() (window method)
gepass() (in module getpass)
gepopt (standard module)
gepath() (MH method)
gepeernname() (socket method)
gepgid() (in module os)
gepgrp() (in module os)
gepid() (in module os)
gepplist() (Message method)
gepos() (HTMLParser method)
geppid() (in module os)
gepreferedencoding() (in module locale)
geprofile() (MH method)
GetProperty() (Binary method)
getProperty() (XMLReader method)
GetPropertyCount() (Binary method)
geprotobynname() (in module socket)
gePublicId() (InputSource method)
gePublicId() (Locator method)
gepwall() (in module pwd)
gepwnam() (in module pwd)
get_referents() (in module gc)
get_referrers() (in module gc)
get_request() (SocketServer protocol)
get_rgbmode() (in module fl)
get_scheme() (BaseHandler method)
get_selector() (Request method)
get_sequences() (MH method)
get_sequences() (MHMessage method)
get_socket() (Telnet method)
get_source() (zipimporter method)
get_starttag_text() (HTMLParser method)
get_starttag_text() (SGMLParser method)
get_stderr() (BaseHandler method)
get_stderr() (WSGIRequestHandler method)
get_stdin() (BaseHandler method)
get_string() (Mailbox method)
get_subdir() (MaildirMessage method)
get_suffixes() (in module imp)
get_terminator() (async_chat method)
get_threshold() (in module gc)
get_token() (shlex method)
get_type() (Request method)
get_unixfrom() (Message method)
get_visible() (BabylMessage method)
getacl() (IMAP4_stream method)
getaddr() (AddressList method)
getaddresses() (in module email.utils)
getaddrinfo() (in module socket)
getaddrlist() (AddressList method)
getallmatchingheaders() (AddressList method)
getannotation() (IMAP4_stream method)
getargspec() (in module inspect)
getargvalues() (in module inspect)
getatime() (in module os.path)
getpwuid() (in module pwd)
getQNameByName() (AttributesNS method)
getQNames() (AttributesNS method)
getqueue_size() (audio configuration method)
getquota() (IMAP4_stream method)
getquotaroot() (IMAP4_stream method)
getrandbits() (in module random)
getrawheader() (AddressList method)
getreader() (in module codecs)
getrecursionlimit() (in module sys)
getrefcount() (in module sys)
getresponse() (HTTPResponse method)
getrlimit() (in module resource)
getroot() (ElementTree method)
getusage() (in module resource)
getsampfmt() (audio configuration method)
getsample() (in module audioop)
getsampwidth() (aifc method)
getsampwidth() (AU_read method)
getsampwidth() (Wave_read method)
getsequences() (Folder method)
getsequencesfilename() (Folder method)
getservbyname() (in module socket)
getservbyport() (in module socket)
GetSetDescriptorType (in module types)
getsid() (in module os)
getsignal() (in module signal)
getsize() (Chunk method)
getsize() (in module os.path)
getsizes() (in module imgfile)
getslice() (in module operator)
getsockname() (socket method)
getssockopt() (socket method)
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Module/Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>getattr()</td>
<td>(built-in function)</td>
</tr>
<tr>
<td>getAttribute()</td>
<td>(Element method)</td>
</tr>
<tr>
<td>getAttributeNode()</td>
<td>(Element method)</td>
</tr>
<tr>
<td>getAttributeNodeNS()</td>
<td>(Element method)</td>
</tr>
<tr>
<td>getAttributeNS()</td>
<td>(Element method)</td>
</tr>
<tr>
<td>GetBase()</td>
<td>(xmlparser method)</td>
</tr>
<tr>
<td>getbegyx()</td>
<td>(window method)</td>
</tr>
<tr>
<td>getboolean()</td>
<td>(SafeConfigParser method)</td>
</tr>
<tr>
<td>getByteStream()</td>
<td>(InputSource method)</td>
</tr>
<tr>
<td>getch()</td>
<td>(in module msvcrt)</td>
</tr>
<tr>
<td>getch()</td>
<td>(window method)</td>
</tr>
<tr>
<td>getchannels()</td>
<td>(audio configuration method)</td>
</tr>
<tr>
<td>getCharacterStream()</td>
<td>(InputSource method)</td>
</tr>
<tr>
<td>getche()</td>
<td>(in module msvcrt)</td>
</tr>
<tr>
<td>getcheckinterval()</td>
<td>(in module sys)</td>
</tr>
<tr>
<td>getChildren()</td>
<td>(Node method)</td>
</tr>
<tr>
<td>getclassstree()</td>
<td>(in module inspect)</td>
</tr>
<tr>
<td>GetColumnInfo()</td>
<td>(Binary method)</td>
</tr>
<tr>
<td>getColumnNumber()</td>
<td>(Locator method)</td>
</tr>
<tr>
<td>getcomment()</td>
<td>(font handle method)</td>
</tr>
<tr>
<td>getcomments()</td>
<td>(in module inspect)</td>
</tr>
<tr>
<td>getcompname()</td>
<td>(aifc method)</td>
</tr>
<tr>
<td>getcompname()</td>
<td>(AU_read method)</td>
</tr>
<tr>
<td>getcompname()</td>
<td>(Wave_read method)</td>
</tr>
<tr>
<td>getcomptype()</td>
<td>(aifc method)</td>
</tr>
<tr>
<td>getcomptype()</td>
<td>(AU_read method)</td>
</tr>
<tr>
<td>getcomptype()</td>
<td>(Wave_read method)</td>
</tr>
<tr>
<td>getconfig()</td>
<td>(audio port method)</td>
</tr>
<tr>
<td>getContentHandler()</td>
<td>(XMLReader method)</td>
</tr>
<tr>
<td>getcontext()</td>
<td>(in module decimal)</td>
</tr>
<tr>
<td>getcontext()</td>
<td>(MH method)</td>
</tr>
<tr>
<td>getctime()</td>
<td>(in module os.path)</td>
</tr>
<tr>
<td>getsource()</td>
<td>(in module inspect)</td>
</tr>
<tr>
<td>getsourcefile()</td>
<td>(in module inspect)</td>
</tr>
<tr>
<td>getsourcelines()</td>
<td>(in module inspect)</td>
</tr>
<tr>
<td>getspall()</td>
<td>(in module spwd)</td>
</tr>
<tr>
<td>getsnam()</td>
<td>(in module spwd)</td>
</tr>
<tr>
<td>getstate()</td>
<td>(in module random)</td>
</tr>
<tr>
<td>getstatus()</td>
<td>(audio port method)</td>
</tr>
<tr>
<td>getstatus()</td>
<td>(CD player method)</td>
</tr>
<tr>
<td>getstatus()</td>
<td>(in module commands)</td>
</tr>
<tr>
<td>getstatusoutput()</td>
<td>(in module commands)</td>
</tr>
<tr>
<td>getstr()</td>
<td>(window method)</td>
</tr>
<tr>
<td>getstrwidth()</td>
<td>(font handle method)</td>
</tr>
<tr>
<td>getSubject()</td>
<td>(SMTPHandler method)</td>
</tr>
<tr>
<td>getsubtype()</td>
<td>(Message method)</td>
</tr>
<tr>
<td>GetSummaryInformation()</td>
<td>(Binary method)</td>
</tr>
<tr>
<td>getSystemId()</td>
<td>(InputSource method)</td>
</tr>
<tr>
<td>getSystemId()</td>
<td>(Locator method)</td>
</tr>
<tr>
<td>getsyx()</td>
<td>(in module curses)</td>
</tr>
<tr>
<td>gettarinfo()</td>
<td>(TarFile method)</td>
</tr>
<tr>
<td>gettempdir()</td>
<td>(in module tempfile)</td>
</tr>
<tr>
<td>gettempprefix()</td>
<td>(in module tempfile)</td>
</tr>
<tr>
<td>getTestCaseNames()</td>
<td>(TestLoader method)</td>
</tr>
<tr>
<td>gettext()</td>
<td>(GNUTranslations method)</td>
</tr>
<tr>
<td>gettext()</td>
<td>(in module gettext)</td>
</tr>
<tr>
<td>gettext()</td>
<td>(NullTranslations method)</td>
</tr>
<tr>
<td>gettext()</td>
<td>(standard module)</td>
</tr>
<tr>
<td>gettimeout()</td>
<td>(socket method)</td>
</tr>
<tr>
<td>gettrackinfo()</td>
<td>(CD player method)</td>
</tr>
<tr>
<td>getType()</td>
<td>(Attributes method)</td>
</tr>
<tr>
<td>gettype()</td>
<td>(Message method)</td>
</tr>
<tr>
<td>getuid()</td>
<td>(in module os)</td>
</tr>
<tr>
<td>geturl()</td>
<td>(ParseResult method)</td>
</tr>
<tr>
<td>getuser()</td>
<td>(in module getpass)</td>
</tr>
<tr>
<td>getValue()</td>
<td>(Attributes method)</td>
</tr>
<tr>
<td>getValue()</td>
<td>(StringIO method)</td>
</tr>
<tr>
<td>getValueByQName()</td>
<td>(AttributesNS method)</td>
</tr>
</tbody>
</table>
getcurrent() (Folder method)
getcwd() (in module os)
getcwdu() (in module os)
getdate() (AddressList method)
getdate_tz() (AddressList method)
getdecoder() (in module codecs)
getdefaulterror() (in module sys)
getdefaultlocale() (in module locale)
getdlopenflags() (in module socket)
getdoc() (in module inspect)
getDOMImplementation() (in module xml.dom)
getDTDHandler() (XMLReader method)
getEffectiveLevel() (DbfilenameShelf method)
getegid() (in module os)
getElementsByTag() (Element method)
getElementsByTagName() (Document method)
getElementsByTagName() (Element method)
getElementsByTagNameNS() (Document method)
getElementsByTagNameNS() (Element method)
getencoder() (in module codecs)
getEncoding() (InputSource method)
getencoding() (Message method)
getEntityResolver() (XMLReader method)
getrain() (in module os)
getEvent() (DOMEventStream method)
getEventType() (NTEventLogHandler method)
getweakrefcount() (in module weakref)
getweakrefs() (in module weakref)
getwelcome() (FTP method)
getwelcome() (NNTP method)
getwelcome() (POP3_SSL method)
getwidth() (audio configuration method)
getwin() (in module curses)
getwindowsversion() (in module sys)
getwriter() (in module codecs)
getyx() (window method)
gid (TarInfo attribute)
gl (built-in module)
GL (standard module)
glob() (Directory method)
glob() (in module glob)
glob (standard module) [Link]
globals (built-in function)
globs (DocTest attribute)
gmtime() (in module time)
gname (TarInfo attribute)
GNOME
gnu_getopt() (in module getopt)
Gopher protocol [Link], [Link]
gopher_open() (GopherHandler method)
gopher_proxy (environment variable)
GopherError
GopherHandler (class in urlib2)
gopherlib (standard module)
got (DocTestFailure attribute)
goto() (in module turtle)
Graphical User Interface
Greenwich Mean Time
grey22grey() (in module imageop)
grey2grey2() (in module imageop)
grey2grey4() (in module imageop)
method)
getException() (SAXException method)
getfd() (audio port method)
getFeature() (XMLReader method)
GetFieldCount() (Binary method)
getfile() (in module inspect)
getfilesystemencoding() (in module sys)
getfillable() (audio port method)
getfilled() (audio port method)
getfillpoint() (audio port method)
getfirst() (FieldStorage method)
getfirstmatchingheader() (AddressList method)
getfloat() (SafeConfigParser method)
getfloatmax() (audio configuration method)
getfmts() (audio device method)
getfontinfo() (font handle method)
getfontname() (font handle method)
getfqdn() (in module socket)
getframeinfo() (in module inspect)
getframerate() (aifc method)
getframerate() (AU_read method)
getframerate() (Wave_read method)
getfullname() (Folder method)
gettext() (in module os)
getgrall() (in module grp)
getgrgid() (in module grp)
getgrnam() (in module grp)
grey2mono() (in module imageop)
grey42grey() (in module imageop)
group() (MatchObject method)
group() (NNTP method)
groupby() (in module itertools)
groupdict() (MatchObject method)
groupindex (RegexObject attribute)
groups() (MatchObject method)
grp (built-in module)
gt() (in module operator)
guess_all_extensions() (in module mimetypes)
guess_extension() (in module mimetypes)
guess_extension() (MimeTypes method)
guess_scheme() (in module wsgiref.util)
guess_type() (in module mimetypes)
guess_type() (MimeTypes method)
GUI
gzip (standard module)
GzipFile (class in gzip)
halfdelay() (in module curses)
handle() (BaseHTTPRequestHandler method)
handle() (DbfilenameShelf method), [Link]
handle() (SocketServer protocol)
handle() (WSGIRequestHandler method)
handle_accept() (dispatcher method)
handle_charref() (HTMLParser method)
handle_charref() (SGMLParser method)
handle_close() (async_chat method)
handle_close() (dispatcher method)
handle_comment() (HTMLParser method)
handle_comment() (SGMLParser method)
handle_connect() (dispatcher method)
handle_data() (HTMLParser method)
handle_data() (SGMLParser method)
handle_decl() (HTMLParser method)
handle_decl() (SGMLParser method)
handle_endtag() (HTMLParser method)
handle_endtag() (SGMLParser method)
handle_entityref() (HTMLParser method)
handle_entityref() (SGMLParser method)
handle_error() (dispatcher method)
handle_error() (SocketServer protocol)
handle_expt() (dispatcher method)
handle_image() (HTMLParser method)

helo() (SMTP method)
help
  online
help() (built-in function)
help() (NNTP method)
hex
hex() (built-in function)
hex (UUID attribute)
hexadecimal
  literals
hexbin() (in module binhex)
hexdigest() (hash method)
hexdigest() (hmac method)
hexdigest() (md5 method)
hexdigest() (sha method)
hexdigits (in module string)
hexlify() (in module binascii)
hexversion (in module sys)
hidden() (Textbox method)
hide() (Textbox method)
hide_cookie2 (LWPCookieJar attribute)
hide_form() (form method)
hide_object() (FORMS object method)
HierarchyRequestErr
HIGHEST_PROTOCOL (in module pickle)
hline() (window method)
HList (class in Tix)
hs_to_rgb() (in module colorsys)
hmac (standard module)
HOME (environment variable), [Link], [Link]
HOMEDRIVE (environment variable)
HOMEPATH (environment variable)
has_section() (SafeConfigParser method)
hasattr() (built-in function)
hasAttribute() (Element method)
hasAttributeNS() (Element method)
hasAttributes() (Node method)
hasChildNodes() (Node method)
hasCompare (in module dis)
hasConst (in module dis)
hasFeature() (DOMImplementation method)
hasFree (in module dis)
hash() (built-in function)
hashlib (built-in module)
hashopen() (in module bsddb)
hashJabs (in module dis)
hashLocal (in module dis)
hasName (in module dis)
have_Unicode (in module test.test_support)
head() (NNTP method)
Header (class in email.header)
header_encode() (Charset method)
header_encoding (in module email.charset)
header_offset (ZipInfo attribute)
HeaderParseError (exception in email.errors)
headers
MIME, [Link]
headers (AddressList attribute)
headers (BaseHTTPRequestHandler attribute)
Headers (class in wsgiref.headers)
headers (ServerProxy attribute)
heading() (in module turtle)
heapify() (in module heapq)
http_error_407() (ProxyDigestAuthHandler method)
http_error_varnnn() (BaseHandler method)
http_error_auth_requed() (AbstractBasicAuthHandler method)
http_error_auth_requed() (AbstractDigestAuthHandler method)
http_error_default() (BaseHandler method)
http_open() (HTTPHandler method)
HTTP_PORT (in module httplib)
http_proxy (environment variable), [Link]
http_version (BaseHandler attribute)
HTTPBasicAuthHandler (class in urllib2)
HTTPConnection (class in httplib)
HTTPCookieProcessor (class in urllib2)
httpd
HTTPDefaultErrorHandler (class in urllib2)
HTTPDigestAuthHandler (class in urllib2)
HTTPError
HTTPException
HTTPHandler (class in logging)
HTTPHandler (class in urllib2)
httplib (standard module)
HTTPPasswordMgr (class in urllib2)
HTTPPasswordMgrWithDefaultRealm (class in urllib2)
HTTPRedirectHandler (class in urllib2)
HTTPResponse (class in httplib)
https_open() (HTTPSHandler method)
HTTPS_PORT (in module httplib)
HTTPSConnection (class in httplib)
HTTPSClient (class in urllib2)
heapmin() (in module msvcr7.0)
heappop() (in module heapq)
heappush() (in module heapq)
heapq (standard module)
heapreplace() (in module heapq)

BaseHTTPServer
HTTPSHandler (class in urllib2)
hypertext
hypot() (in module math)
I (in module re)  
I/O control
  buffering, [Link], [Link]  
POSIX
  tty
Unix
iadd() (in module operator)  
iand() (in module operator)  
ibufcount() (audio device method)  
iconcat() (in module operator)  
id() (built-in function)  
id() (TestCase method)  
idcok() (window method)  
ident (in module cd)  
identchars (Cmd attribute)  
idiv() (in module operator)  
Idle
idlok() (window method)  
IEEE-754
if statement  
ifilter() (in module itertools)  
ifilterfalse() (in module itertools)  
ifloordiv() (in module operator)  
iglob() (in module glob)  
ignoreWhitespace() (ContentHandler method)  
ignore_errors() (in module codecs)  
IGNORE_EXCEPTION_DETAIL (in module doctest)  
ignore_zeros (TarFile attribute)  
IGNORECASE (in module re)  
ihave() (NNTP method)  
ihooks (standard module)  
ilshift() (in module operator)  
intro (Cmd attribute)
IntType (in module types)  
InuseAttributeErr  
inv() (in module operator)  
InvalidAccessErr  
InvalidCharacterErr  
InvalidModificationErr  
InvalidOperation (class in decimal)  
InvalidStateErr  
InvalidURL  
invert() (in module operator)  
ioctl() (in module fcntl)  
IOError  
ior() (in module operator)  
IP_* (in module socket)  
ipow() (in module operator)  
IPPORT_* (in module socket)  
IPPROTO_* (in module socket)  
IPV6_* (in module socket)  
irepeat() (in module operator)  
IRIS Font Manager  
IRIX  
threads
irshift() (in module operator)  
is not operator  
is operator  
is () (in module operator)  
is_blocked() (DefaultCookiePolicy method)  
is_builtn() (in module imp)  
IS_CHARACTER_JUNK() (in module difflib)  
is_data() (MultiFile method)  
is_empty() (fifo method)
InstanceState (in module types)
iInstr() (window method)
InstrStream (shlex attribute)
int() (built-in function), [Link]
int (UUID attribute)
Int2AP() (in module imaplib)

integer
division
division, long
literals
literals, long
types, operations on

integer object
Integrated Development Environment
Intel/DVI ADPCM
interact() (in module code)
interact() (InteractiveConsole method)
interact() (Telnet method)
InteractiveConsole (class in code)
InteractiveInterpreter (class in code)
intern() (non-essential built-in functions)

internal_attr (ZipInfo attribute)
Internaldate2tuple() (in module imaplib)
internalSubset (DocumentType attribute)
Internet
Internet Config
interpolation, string (%)
InterpolationDepthError
InterpolationError
InterpolationMissingOptionError
InterpolationSyntaxError
interpreter prompts
interrupt() (DbfilenameShelf method)
interrupt_main() (in module thread)

isub() (in module operator)
isupper() (in module curses.ascii)
isupper() (string method)
isxdigit() (in module curses.ascii)
item() (NamedNodeMap method)
item() (NodeList method)
itemgetter() (in module operator)
items() (dictionary method)
items() (Mailbox method)
items() (Message method)
items() (SafeConfigParser method), [Link]

itemsize (array attribute)
iter() (built-in function)
IterableUserDict (class in UserDict)

iterator protocol
iterdecode() (in module codecs)
iterencode() (in module codecs)

iteritems() (dictionary method)
iteritems() (Mailbox method)

iterkeyrefs() (WeakKeyDictionary method)

iterkeys() (dictionary method)

iterkeys() (Mailbox method)

itermonthdates() (Calendar method)
itermonthdays() (Calendar method)

itermonthdays2() (Calendar method)

itervalues() (dictionary method)

itervaluerefs() (WeakValueDictionary method)

itertools (standard module)

itertools (standard module)

xml.etree.ElementTree

itertools (standard module)
Jansen, Jack
java_ver() (in module platform)
JFIF, [Link]
join() (in module os.path)
join() (in module string)
join() (Queue method)
join() (string method)
join() (Thread method)
joinfields() (in module string)
jpeg (built-in module)
js_output() (BaseCookie method)
js_output() (Morsel method)
jumpahead() (in module random)
kbhit() (in module msvcrt)
KDEDIR (environment variable)
key (Morsel attribute)
KeyboardInterrupt
KeyError
keyname() (in module curses)
keypad() (window method)
keyrefs() (WeakKeyDictionary method)
keys() (DbfilenameShelf method)
keys() (dictionary method)
keys() (Mailbox method)

keys() (Message method)
keyword (standard module)
keywords (dict attribute)
kill() (in module os)
killchar() (in module curses)
killpg() (in module os)
knee (module)
knownfiles (in module mimetypes)
Kuchling, Andrew
kwlist (in module keyword)
L (in module re)
LabelEntry (class in Tix)
LabelFrame (class in Tix)
LambdaType (in module types)
LANG (environment variable), [Link], [Link], [Link], [Link]
language
   C, [Link]
LANGUAGE (environment variable), [Link]
large files
LargeZipFile
last() (DbfilenameShelf method)
last() (dbhash method)
last (MultiFile attribute)
last() (NNTP method)
last_traceback (in module sys)
last_type (in module sys)
last_value (in module sys)
lastChild (Node attribute)
lastcmd (Cmd attribute)
lastgroup (MatchObject attribute)
lastindex (MatchObject attribute)
lastpart() (MimeWriter method)
LC_ALL (environment variable), [Link]
LC_ALL (in module locale)
LC_COLLATE (in module locale)
LC_CTYPE (in module locale)
LC_MESSAGES (environment variable), [Link]

ljust() (in module string)
ljust() (string method)
LK_LOCK (in module msvcr)
LK_NBLCK (in module msvcr)
LK_NBRLCK (in module msvcr)
LK_RLCK (in module msvcr)
LK_UNLCK (in module msvcr)
LNAME (environment variable)
lngettext() (GNUTranslations method)
lngettext() (in module gettext)
lngettext() (NullTranslations method)
load() (BaseCookie method)
load() (FileCookieJar method)
load() (in module hotshot.stats)
load() (in module marshal)
load() (in module pickle)
load() (Unpickler method)
load_compiled() (in module imp)
load_dynamic() (in module imp)
load_module() (in module imp)
load_module() (zipimporter method)
load_source() (in module imp)
LoadError
LoadLibrary() (LibraryLoader method)
loads() (in module marshal)
loads() (in module pickle)
loads() (in module xmlrpclib)
loadTestsFromModule() (TestLoader method)
loadTestsFromName() (TestLoader method)
loadTestsFromNames() (TestLoader method)
loadTestsFromTestCase() (TestLoader method)
<table>
<thead>
<tr>
<th>Function/Keyword</th>
<th>Description/Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_MESSAGES</td>
<td>(in module locale)</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>(in module locale)</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>(in module locale)</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>(in module locale)</td>
</tr>
<tr>
<td>lchown()</td>
<td>(in module os)</td>
</tr>
<tr>
<td>ldexp()</td>
<td>(in module math)</td>
</tr>
<tr>
<td>ldgettext()</td>
<td>(in module gettext)</td>
</tr>
<tr>
<td>ldngettext()</td>
<td>(in module gettext)</td>
</tr>
<tr>
<td>le()</td>
<td>(in module operator)</td>
</tr>
<tr>
<td>leapdays()</td>
<td>(in module calendar)</td>
</tr>
<tr>
<td>leaveok()</td>
<td>(window method)</td>
</tr>
<tr>
<td>left()</td>
<td>(in module turtle)</td>
</tr>
<tr>
<td>left_list</td>
<td>(dircmp attribute)</td>
</tr>
<tr>
<td>left_only</td>
<td>(dircmp attribute)</td>
</tr>
<tr>
<td>len()</td>
<td>(built-in function), [Link], [Link]</td>
</tr>
<tr>
<td>length</td>
<td>(NamedNodeMap attribute)</td>
</tr>
<tr>
<td>letters</td>
<td>(in module string)</td>
</tr>
<tr>
<td>level</td>
<td>(MultiFile attribute)</td>
</tr>
<tr>
<td>lexists()</td>
<td>(in module os.path)</td>
</tr>
<tr>
<td>lgettext()</td>
<td>(GNUTranslations method)</td>
</tr>
<tr>
<td>lgettext()</td>
<td>(in module gettext)</td>
</tr>
<tr>
<td>lgettext()</td>
<td>(NullTranslations method)</td>
</tr>
<tr>
<td>libc_ver()</td>
<td>(in module platform)</td>
</tr>
<tr>
<td>library</td>
<td>(in module dbm)</td>
</tr>
<tr>
<td>LibraryLoader</td>
<td>(class in ctypes)</td>
</tr>
<tr>
<td>light-weight</td>
<td>processes</td>
</tr>
<tr>
<td>lin2adpcm()</td>
<td>(in module audioop)</td>
</tr>
<tr>
<td>lin2alaw()</td>
<td>(in module audioop)</td>
</tr>
<tr>
<td>lin2lin()</td>
<td>(in module audioop)</td>
</tr>
<tr>
<td>lin2ulaw()</td>
<td>(in module audioop)</td>
</tr>
<tr>
<td>line()</td>
<td>(Dialog method)</td>
</tr>
<tr>
<td>line-buffered</td>
<td>I/O</td>
</tr>
<tr>
<td>line_num</td>
<td>(csv reader attribute)</td>
</tr>
<tr>
<td>linecache</td>
<td>(standard module)</td>
</tr>
<tr>
<td>lineno</td>
<td>(class descriptor attribute)</td>
</tr>
<tr>
<td>lineno</td>
<td>(DocTest attribute)</td>
</tr>
<tr>
<td>lineno</td>
<td>(Example attribute)</td>
</tr>
<tr>
<td>lineno</td>
<td>(ExpatError attribute)</td>
</tr>
<tr>
<td>localeconv()</td>
<td>(in module locale)</td>
</tr>
<tr>
<td>LOCALE</td>
<td>(in module re)</td>
</tr>
<tr>
<td>locale</td>
<td>(standard module)</td>
</tr>
<tr>
<td>localcontext()</td>
<td>(in module decimal)</td>
</tr>
<tr>
<td>LocaleHTMLCalendar</td>
<td>(class in calendar)</td>
</tr>
<tr>
<td>localeconv()</td>
<td>(in module locale)</td>
</tr>
<tr>
<td>localtime()</td>
<td>(in module time)</td>
</tr>
<tr>
<td>Lock()</td>
<td>(in module threading)</td>
</tr>
<tr>
<td>lock()</td>
<td>(Mailbox method)</td>
</tr>
<tr>
<td>lock()</td>
<td>(Maildir method)</td>
</tr>
<tr>
<td>lock()</td>
<td>(mbox method)</td>
</tr>
<tr>
<td>lock()</td>
<td>(MH method)</td>
</tr>
<tr>
<td>lock()</td>
<td>(mutex method)</td>
</tr>
<tr>
<td>lock()</td>
<td>(posixfile method)</td>
</tr>
<tr>
<td>lock_held()</td>
<td>(in module imp)</td>
</tr>
<tr>
<td>locked()</td>
<td>(lock method)</td>
</tr>
<tr>
<td>lockf()</td>
<td>(in module fcntl), [Link]</td>
</tr>
<tr>
<td>locking()</td>
<td>(in module msvcruntime)</td>
</tr>
<tr>
<td>LockType</td>
<td>(in module thread)</td>
</tr>
<tr>
<td>log()</td>
<td>(DbfilenameShelf method)</td>
</tr>
<tr>
<td>log()</td>
<td>(in module cmath)</td>
</tr>
<tr>
<td>log()</td>
<td>(in module logging)</td>
</tr>
<tr>
<td>log()</td>
<td>(in module math)</td>
</tr>
<tr>
<td>log10()</td>
<td>(in module cmath)</td>
</tr>
<tr>
<td>log10()</td>
<td>(in module math)</td>
</tr>
<tr>
<td>log_date_time_string()</td>
<td>(BaseHTTPRequestHandler method)</td>
</tr>
<tr>
<td>log_error()</td>
<td>(BaseHTTPRequestHandler method)</td>
</tr>
<tr>
<td>log_exception()</td>
<td>(BaseHandler method)</td>
</tr>
<tr>
<td>log_exception()</td>
<td>(BaseHandler method)</td>
</tr>
</tbody>
</table>
lineno (function descriptor attribute)

lineno() (in module fileinput)

lineno (shlex attribute)

LINES (environment variable), [Link]

linesep (in module os)

lineterminator (Dialect attribute)

link() (in module os)

linkname (TarInfo attribute)

list
type, operations on
list() (built-in function)
list() (IMAP4_stream method)
list() (NNTP method)
list() (POP3_SSL method)
list() (TarFile method)
list object, [Link]
list_dialects() (in module csv)
list_folders() (Maildir method)
list_folders() (MH method)
listallfolders() (MH method)
listdir() (in module dircache)
listdir() (in module os)
listen() (dispatcher method)
listen() (in module logging)
listen() (socket method)
listfolders() (MH method)
listmessages() (Folder method)
ListNoteBook (class in Tix)
listsubfolders() (MH method)
ListType (in module types)

literals
complex number
floating point
hexadecimal
integer
long integer
numeric
octal

log_message() (BaseHTTPRequestHandler method)

log_request() (BaseHTTPRequestHandler method)

logging

Errors

logging (standard module)

login() (FTP method)

login() (IMAP4_stream method)

login() (SMTP method)

login_cram_md5() (IMAP4_stream method)

LOGNAME (environment variable), [Link]

lognormvariate() (in module random)

logout() (IMAP4_stream method)

LogRecord (class in logging)

long

type, operations on
long() (built-in function), [Link], [Link]

long () (built-in function)

long integer object

longimagedata() (in module rgbimg)

longname() (in module curses)

longstimage() (in module rgbimg)

LongType (in module types)

lookup() (in module codecs)

lookup() (in module unicodedata)

lookup_error() (in module codecs)

LookupError

loop() (in module asyncore)

lower() (in module string)

lower() (string method)

lower() (in module string)

lseek() (in module os)

lshift() (in module operator)

lstat() (in module os)

lstrip() (in module string)

lstrip() (string method)
LittleEndianStructure (class in ctypes)  Isub() (IMAP4_stream method)
lt() (in module operator)
Lundh, Fredrik
LWPCookieJar (class in cookielib)
M (in module re)
mac_ver() (in module platform)
machine() (in module platform)
macros (netrc attribute)
Mailbox (class in mailbox)
mailbox (module)
mailbox (standard module)
mailcap (standard module)
Maildir (class in mailbox)
MaildirMessage (class in mailbox)
MailmanProxy (class in smtpd)
main() (in module py_compile)
main() (in module unittest)
major() (in module os)
make_cookies() (CookieJar method)
make_file() (in module difflib)
make_form() (in module fl)
make_header() (in module email.header)
make_msgid() (in module email.utils)
make_parser() (in module xml.sax)
make_server() (in module wsgiref.simple_server)
make_table() (in module difflib)
makedev() (in module os)
makedirs() (in module os)
makefile() (socket method)
makefolder() (MH method)
makeLogRecord() (in module logging)
makePickle() (SocketHandler method)
makeRecord() (DbfilenameShelf method)
makeSocket() (DatagramHandler method)

MIME (continued)
quoted-printable encoding
mime_decode_header() (in module mimify)
mime_encode_header() (in module mimify)
MIMEApplication (class in email.mime.text)
MIMEAudio (class in email.mime.text)
MIMEBase (class in email.mime.text)
MIMEImage (class in email.mime.text)
MIMEMessage (class in email.mime.text)
MIMEMultipart (class in email.mime.text)
MIMENonMultipart (class in email.mime.text)
MIMEText (class in email.mime.text)
mimetypes (standard module), [Link]
MimeTypes (class in mimetypes)
mimetypes (standard module)
MimeWriter (class in MimeWriter)
MimeWriter (standard module)
mimify() (in module mimify)
mimify (standard module)
min() (built-in function), [Link]
min() (Context method)
min (date attribute)
min (datetime attribute)
min() (Decimal method)
makeSocket() (SocketHandler method)
makeRepr (Repr attribute)
maketrans() (in module string)
map() (built-in function)
map_table_b2() (in module stringprep)
map_table_b3() (in module stringprep)
mapcolor() (in module fl)
mapping
types operations on mapping() (Control method)
mapping object
maps() (in module nis)
martial (built-in module)
marshalling
objects
masking
operations
match() (in module nis)
match() (in module re)
match() (RegexObject method)
math (built-in module).
max() (built-in function).
max() (Context method)
max (date attribute)
max (datetime attribute)
max() (Decimal method)
max() (in module audioop)
max (time attribute)
max (timedelta attribute)
MAX_INTERPATION_DEPTH (in module ConfigParser)
maxarray (Repr attribute)
maxdeque (Repr attribute)
maxdict (Repr attribute)
maxfrozenset (Repr attribute)
maxint (in module sys)
MAXLEN (in module mimify)
maxlevel (Repr attribute)
maxlist (Repr attribute)
maxlong (Repr attribute)
min (time attribute)
min (timedelta attribute)
minmax() (in module audioop)
minor() (in module audioop)
minus() (Context method)
minute (datetime attribute)
minute (time attribute)
MININTERPATION_DEPTH (in module ConfigParser)
minarray (Repr attribute)
mindeque (Repr attribute)
mindict (Repr attribute)
minfrozenset (Repr attribute)
minint (in module sys)
MINLEN (in module mimify)
minlevel (Repr attribute)
minlist (Repr attribute)
minlong (Repr attribute)
mkfile() (FTP method)
mkdir() (in module os)
mkdtemp() (in module tempfile)
mkfifo() (in module os)
mknod() (in module os)
mkstemp() (in module tempfile)
mktemp() (in module tempfile)
mktime() (in module time)
mktime() (in module email.utils)
mktime() (in module rfc822)
mmap (built-in module)
mmap() (in module mmap)
MMDF (class in mailbox)
mMdfMailbox (class in mailbox)
MMDFMessage (class in mailbox)
mod() (in module operator)
mode (audio device attribute)
mode (file attribute)
mode (TarInfo attribute)
modf() (in module math)
modified() (RobotFileParser method)
Modify() (Binary method)
module
search path, [Link], [Link]
(BaseHTTPRequestHandler attribute)
MessageError (exception in email.errors)
MessageParseError (exception in email.errors)
meta() (in module curses)
Meter (class in Tix)
method object
methods
  string
methods (class descriptor attribute)
MethodType (in module types)
MH (class in mailbox)
MH (class in mlib)
mhlib (standard module)
MHMailbox (class in mailbox)
MHMessage (class in mailbox)
microsecond (datetime attribute)
microsecond (time attribute)
MIME
  base64 encoding
  content type
  headers, [Link]
name (Attr attribute)
name (audio device attribute)
name (class descriptor attribute)
name (Cookie attribute)
name (DocTest attribute)
name (DocumentType attribute)
name (function descriptor attribute)
name (in module os)
name() (in module unicodedata)
name (TarInfo attribute)
name2codepoint (in module htmlentitydefs)
NamedTemporaryFile() (in module tempfile)
NameError
namelist() (ZipFile method)
nameprep() (in module encodings.idna)
namespace() (IMAP4_stream method)
NAMESPACE_DNS (in module uuid)
NAMESPACE_OID (in module uuid)
NAMESPACE_URL (in module uuid)
NAMESPACE_X500 (in module uuid)
NamespaceErr
namese URI (Node attribute)
NaN, [Link]
NannyNag
napms() (in module curses)
ndiff() (in module difflib)
ne() (in module operator)
neg() (in module operator)
nested() (in module contextlib)
nntplib (standard module)
NNTPPermanentError
NNTPProtocolError
NNTPReplyError
NNTPTemporaryError
nocbreak() (in module curses)
NoDataAllowedErr
Node (class in compiler.ast)
node() (in module platform)
node() (window method)
nodename (Node attribute)
nodeType (Node attribute)
nodeValue (Node attribute)
NODISC (in module cd)
noecho() (in module curses)
NOEXPR (in module locale)
nofill (HTMLParser attribute)
nok builtin names (RExec attribute)
noload() (Unpickler method)
NoModificationAllowedErr
nonblock() (audio device method)
None (Built-in object)
None (built-in warning)
NoneType (in module types)
nol() (in module curses)
nop() (IMAP4_stream method)
nop() (POP3_SSL method)
NoOptionError
noqiflush() (in module curses)
noraw() (in module curses)
normalize() (Context method)
normalize() (Decimal method)
normalize() (in module locale)
normalize() (in module unicodedata)
netrc (class in netrc)
netrc (standard module)
NetrcParseError
netscape (LWPCookieJar attribute)
Network News Transfer Protocol
new (built-in module)
new() (in module hmac)
new() (in module md5)
new() (in module sha)
new_alignment() (writer method)
new_font() (writer method)
new_margin() (writer method)
new_module() (in module imp)
new_panel() (in module curses.panel)
new_spacing() (writer method)
new_styles() (writer method)
newconfig() (in module al)
newgroups() (NNTP method)
newlines (file attribute)
newnews() (NNTP method)
newwin() (in module curses)
next() (csv reader method)
next() (DbfilenameShelf method)
next() (dbhash method)
next() (file method)
next() (FormatError method)
next() (iterator method)
next() (MultiFile method)
next() (NNTP method)
next() (TarFile method)
nextfile() (in module fileinput)
nextkey() (in module gdbm)
nextpart() (MimeWriter method)
nextSibling (Node attribute)
gettext() (GNUTranslations method)
gettext() (in module gettext)
gettext() (NullTranslations method)
NI * (in module socket)
normalize() (Node method)
NORMALIZE_WHITESPACE (in module doctest)
normalvariate() (in module random)
normcase() (in module os.path)
normpath() (in module os.path)
NoSectionError
NoSuchMailboxError (class in mailbox)
ot in operator, [Link]
not operator
not () (in module operator)
NotANumber
notationDecl() (DTDHandler method)
NotationDeclHandler() (xmlparser method)
notations (DocumentType attribute)
NotConnected
NoteBook (class in Tix)
NotEmptyErrorError (class in mailbox)
NotFoundErr
notify() (Condition method)
notifyAll() (Condition method)
otimeout() (window method)
NotImplemented (built-in warning)
NotImplementedError
NotImplementedType (in module types)
NotStandaloneHandler() (xmlparser method)
NotSupportedErr
noutrefresh() (window method)
now() (datetime method)
NSIG (in module signal)
nsmallest() (in module heapq)
NTEventLogHandler (class in logging)
ntohl() (in module socket)
ntohs() (in module socket)
ntransfercmd() (FTP method)
nice() (in module os)  NullFormatter (class in formatter)
nis (extension module)  NullImporter (class in imp)
NIST  NullWriter (class in formatter)
nl() (in module curses)  numeric
NL (in module tokenize)  conversions
nl_langinfo() (in module locale)  literals
nlargest() (in module heapq)  object
nlst() (FTP method)  types, operations on
numeric

NNTP

class in nntplib

NNTPDataError  numeric() (in module unicodedata)
NNTPErrortype
Numerical Python  numeric

nurbscurve() (in module gl)
nurbssurface() (in module gl)
narray() (in module gl)
<table>
<thead>
<tr>
<th>Object</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_APPEND</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_BINARY</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_CREAT</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_DSYNC</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_EXCL</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_EXLOCK</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_NOCTTY</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_NOINHERIT</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_NONBLOCK</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_RANDOM</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_RDONLY</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_RDWR</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_RSYNC</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_SEQUENTIAL</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_SHLOCK</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_SHORT_LIVED</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_SYNC</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_TEMPORARY</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_TEXT</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_TRUNC</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>O_WRONLY</td>
<td>open() (in module os)</td>
</tr>
<tr>
<td>object</td>
<td>open() (in module webbrowser), [Link]</td>
</tr>
<tr>
<td>Boolean</td>
<td>open() (OpenerDirector method)</td>
</tr>
<tr>
<td>buffer</td>
<td>open() (TarFile method)</td>
</tr>
<tr>
<td>code</td>
<td>open() (Template method)</td>
</tr>
<tr>
<td>complex number</td>
<td>open() (URLopener method)</td>
</tr>
<tr>
<td>dictionary</td>
<td>open_new() (in module webbrowser), [Link]</td>
</tr>
<tr>
<td>file</td>
<td>open_new_tab() (in module webbrowser), [Link]</td>
</tr>
<tr>
<td>floating point</td>
<td>open_osfhandle() (in module msvcrt)</td>
</tr>
<tr>
<td>integer</td>
<td>open_unknown() (URLopener method)</td>
</tr>
<tr>
<td>OpenGL</td>
<td>OpenDatabase() (in module msilib)</td>
</tr>
<tr>
<td>OpenKey()</td>
<td>opendir() (in module dircache)</td>
</tr>
<tr>
<td>OpenKeyEx()</td>
<td>OpenerDirector (class in urllib2)</td>
</tr>
<tr>
<td>openlog()</td>
<td>openfolder() (MH method)</td>
</tr>
<tr>
<td>openmessage()</td>
<td>openfp() (in module sunau)</td>
</tr>
<tr>
<td>openmixer()</td>
<td>openfp() (in module wave)</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>OpenGL</td>
</tr>
<tr>
<td>operation</td>
<td>OpenKey() (in module winreg)</td>
</tr>
<tr>
<td>operation</td>
<td>OpenKeyEx() (in module _winreg)</td>
</tr>
<tr>
<td>operation</td>
<td>openlog() (in module syslog)</td>
</tr>
<tr>
<td>operation</td>
<td>openmessage() (Message method)</td>
</tr>
<tr>
<td>operation</td>
<td>openmixer() (in module ossaudiodev)</td>
</tr>
<tr>
<td>operation</td>
<td>openport() (in module al)</td>
</tr>
<tr>
<td>operation</td>
<td>openpty() (in module os)</td>
</tr>
<tr>
<td>operation</td>
<td>openpty() (in module pty)</td>
</tr>
<tr>
<td>operation</td>
<td>OpenView() (Binary method)</td>
</tr>
</tbody>
</table>

Object types:
- Boolean
- buffer
- code, [Link]
- complex number
- dictionary
- file
- floating point
- integer

Operations:
- concatenation
- extended slice
- repetition
- slice
list, [Link] long integer
mapping method numeric, [Link] sequence set
socket string traceback, [Link] tuple
type Unicode
xrange, [Link] object() (built-in function)

objects
comparing flattening marshalling persistent
pickling serializing

obufcount() (audio device method), [Link]
obuffree() (audio device method)

oct() (built-in function)
ocital

octdigits (in module string)
offset (ExpatError attribute)
OK (in module curses)
ok_builtin_modules (RExec attribute)
ok_file_types (RExec attribute)
ok_path (RExec attribute)
ok_posix_names (RExec attribute)
ok_sys_names (RExec attribute)
OleDLL (class in ctypes)
onecmd() (Cmd method)
open() (built-in function)

subscript

operations
bit-string Boolean, [Link]
masking shifting

operations on
dictionary type integer types
list type mapping types
mutable sequence types
numeric types
sequence types, [Link]

operator
== and, [Link]
comparison in, [Link]
is is not
not
not in, [Link]
or, [Link]

operator (built-in module)
opmap (in module dis)
opname (in module dis)

OptionMenu (class in Tix)
optparse (standard module)
or operator, [Link]
or_() (in module operator)
ord() (built-in function)

ordered_attributes (xmlparser attribute)
origin_server (BaseHandler attribute)
oct (built-in function)
os (standard module), [Link], [Link]
open() (IMAP4_stream method)
open() (in module aifc)
open() (in module anydbm)
open() (in module cd)
open() (in module codecs)
open() (in module dbhash)
open() (in module dbm)
open() (in module dl)
open() (in module dumbdbm)
open() (in module gdbm)
open() (in module gzip)
open() (in module os)
open() (in module ossaudiodev)
open() (in module posixfile)
open() (in module shelve)
open() (in module sunau)
open() (in module sunaudiodev)
open() (in module tarfile)
open() (in module wave)

os.path (standard module)
os.environ (BaseHandler attribute)
OSError
ossaudiodev (built-in module)
OSSAudioError
output() (BaseCookie method)
output() (Morsel method)
output_charset (in module email.charset)
output_charset() (NullTranslations method)
output_codec (in module email.charset)
output_difference() (OutputChecker method)
OutputChecker (class in doctest)
OutputString() (Morsel method)
OutputType (in module cStringIO)
Overflow (class in decimal)
OverflowError
overlay() (window method)
Overmars, Mark
overwrite() (window method)
P

P_DETACH (in module os)
P_NOWAIT (in module os)
P_NOWAITO (in module os)
P_OVERLAY (in module os)
P_WAIT (in module os)
pack() (in module struct)
pack() (MH method)
pack_array() (Packer method)
pack_bytes() (Packer method)
pack_double() (Packer method)
pack_farray() (Packer method)
pack_float() (Packer method)
pack_fopaque() (Packer method)
pack_fstring() (Packer method)
pack_opaque() (Packer method)
pack_string() (Packer method)
package
Packer (class in xdrlib)
packing
   binary data
packing (widgets)
PAGER (environment variable)
pair_content() (in module curses)
pair_number() (in module curses)
PanedWindow (class in Tix)
pardir (in module os)
parent (BaseHandler attribute)
parentNode (Node attribute)

PopupMenu (class in Tix)
port (Cookie attribute)
port_specified (Cookie attribute)
PortableUnixMailbox (class in mailbox)
pos() (in module operator)
pos (MatchObject attribute)
position() (in module turtle)
POSIX
   file object
   I/O control
   threads
posix (built-in module)
posix (TarFile attribute)
posixfile (built-in module)
post() (audio device method)
post() (NNTP method)
post_mortem() (in module pdb)
postcmd() (Cmd method)
postloop() (Cmd method)
pow() (built-in function)
pow() (in module math)
pow() (in module operator)
power() (Context method)
pprint() (in module pprint)
pprint() (PrettyPrinter method)
pprint (standard module)
prcal() (in module calendar)
preamble (in module email.message)
precmd() (Cmd method)
prefix (Attr attribute)
prefix (in module sys)
prefix (Node attribute)
preloop() (Cmd method)
paretovariate() (in module random)
parse() (DocTestParser method)
parse() (ElementTree method)
parse() (module cgi)
parse() (module compiler)
parse() (module xml.dom.minidom)
parse() (module xml.dom.pulldom)
parse() (module xml.etree.ElementTree)
parse() (module xml.sax)
parse() (Parser method)
parse() (RobotFileParser method)
Parse() (xmlparser method)
parse() (XMLReader method)
parse_and_bind() (module readline)
PARSE_COLNAMES (module sqlite3)
PARSE_DECLTYPES (module sqlite3)
parse_header() (module cgi)
parse_multipart() (module cgi)
parse_qs() (module cgi)
parse_qsl() (module cgi)
paseaddr() (module email.utils)
paseaddr() (module rfc822)
parsedate() (module email.utils)
parsedate() (module rfc822)
parsedate_tz() (module email.utils)
parsedate_tz() (module rfc822)
pasrFile() (module compiler)
ParseFile() (xmlparser method)
ParseFlags() (module imaplib)
pasefrme() (CD parser method)
pase (module)
Parser (module email.parser)
ParserCreate() (module xml.parsers.expat)
ParserError
ParseResult (module urllib.parse)
preorder() (ASTVisitor method)
prepare_input_source() (module xml.sax.saxutils)
prepend() (Template method)
PrettyPrinter (class in pprint)
preventremoval() (CD player method)
previous() (DbfilenameShelf method)
previous() (dbhash method)
previousSibling (Node attribute)
print statement
print_callees() (Stats method)
print_callers() (Stats method)
print_directory() (module cgi)
print_envir() (module cgi)
print_envir_usage() (module cgi)
print_exc() (module traceback)
print_exc() (Timer method)
print_exception() (module traceback)
print_form() (module cgi)
print_last() (module traceback)
print_stack() (module traceback)
print_stats() (Stats method)
print_tb() (module traceback)
printable (module string)
printdir() (ZipFile method)
printf-style formatting
prmonth() (module calendar)
prmonth() (TextCalendar method)
process
    group, [Link]
id
    id of parent
    killing, [Link]
    signalling, [Link]
process_message() (SMTPServer method)
process_request() (SocketServer protocol)
processes, light-weight
parsesequence() (Folder method)
parsestr() (Parser method)
parseString() (in module xml.dom.minidom)
parseString() (in module xml.dom.pulldom)
parsing
    Python source code
    URL
ParsingError
    partial() (IMAP4_stream method)
partial() (in module functools)
partition() (string method)
pass_() (POP3_SSL method)
path
    configuration file
    module search, [Link], [Link]
    operations
    path (BaseHTTPRequestHandler attribute)
    path (Cookie attribute)
    PATH (environment variable), [Link], [Link], [Link], [Link], [Link], [Link],
    [Link], [Link]
    path (in module os)
    path (in module sys)
    Path browser
    path_return_ok() (CookiePolicy method)
    pathconf() (in module os)
    pathconf_names (in module os)
    pathname2url() (in module urllib)
    pathsep (in module os)
    pattern (RegexObject attribute)
    pause() (in module signal)
    PAUSED (in module cd)
    Pdb (class in pdb)
    pdb (standard module)

processingInstruction() (ContentHandler method)
ProcessingInstruction() (in module xml.etree.ElementTree)
ProcessingInstructionHandler() (xmlparser method)
processor() (in module platform)
processor time
Profile (class in hotshot)
profile (standard module)
profile_function, [Link]
profiler
    profiling, deterministic
    prompt (Cmd attribute)
    prompt_user_passwd()
(FancyURLopener method)
prompts, interpreter
    propagate (in module logging)
    property() (built-in function)
    property_declaration_handler (in module xml.sax.handler)
    property_dom_node (in module xml.sax.handler)
    property_lexical_handler (in module xml.sax.handler)
    property_xml_string (in module xml.sax.handler)
    proto (socket attribute)

protocol
    CGI
    context management
    FTP, [Link]
    Gopher, [Link], [Link]
    HTTP, [Link], [Link], [Link], [Link]
    IMAP4
    IMAP4_SSL
    IMAP4_stream
    iterator
Pen (class in turtle)
PendingDeprecationWarning
Performance
Persist() (Binary method)
persistence
persistent
    objects
pformat() (in module pprint)
pformat() (PrettyPrinter method)
pi (in module cmath)
pi (in module math)
pick() (in module gl)
pickle() (in module copy_reg)
pickle (standard module), [Link], [Link], [Link], [Link]
Pickler (class in pickle)
pickletools (standard module)
pickling
    objects
PicklingError
pid (Popen attribute)
pid (Popen4 attribute)
PIL (the Python Imaging Library)
pipe() (in module os)
pipes (standard module)
PKG_DIRECTORY (in module imp)
pkgutil (standard module)
platform() (in module platform)
platform (in module sys)
platform (standard module)
play() (CD player method)
playabs() (CD player method)
PLAYING (in module cd)
PlaySound() (in module winsound)
playtrack() (CD player method)
playtrackabs() (CD player method)
plock() (in module os)
plus() (Context method)

NNTP
POP3
SMTP
Telnet

protocol_version
(BaseHTTPRequestHandler attribute)
PROTOCOL_VERSION
(IMAP4_stream attribute)
proxy() (in module weakref)
proxyauth() (IMAP4_stream method)
ProxyBasicAuthHandler (class in urllib2)
ProxyDigestAuthHandler (class in urllib2)
ProxyHandler (class in urllib2)
ProxyType (in module weakref)
ProxyTypes (in module weakref)
prstr() (in module fm)
pyear() (TextCalendar method)
ps1 (in module sys)
ps2 (in module sys)
pstats (standard module)
pthreads
ptime (in module cd)
pty (standard module), [Link]
publicId (DocumentType attribute)
PullDOM (class in xml.dom.pulldom)
punctuation (in module string)
PureProxy (class in smtpd)
push() (async_chat method)
push() (fifo method)
push() (InteractiveConsole method)
push() (MultiFile method)
push_alignment() (formatter method)
push_font() (formatter method)
push_margin() (formatter method)
push_source() (shlex method)
push_style() (formatter method)
push_token() (shlex method)
pm() (in module pdb)
pnum (in module cd)
POINTER() (in module ctypes)
pointer() (in module ctypes)
poll() (in module select)
poll() (Popen method)
poll() (Popen4 method)
poll() (Structure method)
pop() (array method)
pop() (dictionary method)
pop() (fifo method)
pop() (list method)
pop() (LocaleHTMLCalendar method)
pop() (Mailbox method)
pop() (MultiFile method)
POP3
  protocol
POP3 (class in poplib)
POP3_SSL (class in poplib)
pop_alignment() (formatter method)
pop_font() (formatter method)
pop_margin() (formatter method)
pop_style() (formatter method)
Popen (class in subprocess)
popen() (in module os)
popen() (in module platform)
popen2() (in module os)
popen2() (in module popen2)
popen2 (standard module)
Popen3 (class in popen2)
popen3() (in module os)
popen3() (in module popen2)
Popen4 (class in popen2)
popen4() (in module os)
popen4() (in module popen2)
popitem() (dictionary method)
popitem() (Mailbox method)
popleft() (LocaleHTMLCalendar method)
push_with_producer() (async_chat method)
pushbutton() (Dialog method)
put() (Queue method)
put_nowait() (Queue method)
putch() (in module msvcr1)
putenv() (in module os)
putheader() (HTTPResponse method)
putp() (in module curses)
putrequest() (HTTPResponse method)
putsequences() (Folder method)
putwin() (window method)
pwd (built-in module)
pwlcnode() (in module gl)
pyCompileError
PyDLL (class in ctypes)
Pydoc (standard module)
Pyexpat (built-in module)
PYFUNCTYPE() (in module ctypes)
PyOpenGL
Python Editor
Python Imaging Library
python_build() (in module platform)
python_compiler() (in module platform)
PYTHON_DOM (environment variable)
python_version() (in module platform)
python_version_tuple() (in module platform)
PYTHONDOCS (environment variable)
method) poplib (standard module)

PYTHONPATH (environment variable), [Link], [Link]
PYTHONSTARTUP (environment variable), [Link], [Link]
PYTHONY2K (environment variable), [Link], [Link]
PyZipFile (class in zipfile)
qdevice() (in module fl)
qenter() (in module fl)
quiflush() (in module curses)
QName (class in xml.etree.ElementTree)
qread() (in module fl)
qreset() (in module fl)
qsize() (Queue method)
qtest() (in module fl)
quantize() (Context method)
quantize() (Decimal method)
QueryInfoKey() (in module _winreg)
queryparams() (in module al)
QueryValue() (in module _winreg)
QueryValueEx() (in module _winreg)
Queue (class inQueue)
Queue (standard module)
quick_ratio() (SequenceMatcher method)
quit() (FTP method)
quit() (NNTP method)
quit() (POP3_SSL method)
quit() (SMTP method)
quopri (standard module)
quote() (in module email.utils)
quote() (in module RFC822)
quote() (in module urllib)
QUOTE_ALL (in module csv)
QUOTE_MINIMAL (in module csv)
QUOTE_NONE (in module csv)
QUOTE_NONNUMERIC (in module csv)
quote_plus() (in module urllib)
quoteattr() (in module xml.sax.saxutils)
quotechar (Dialect attribute)
quoting (Dialect attribute)

encoding
quotes (shlex attribute)
r_eval() (RExec method)
r_exec() (RExec method)
r_execfile() (RExec method)
r_import() (RExec method)
R_OK (in module os)
r_open() (RExec method)
r_reload() (RExec method)
r_unload() (RExec method)
radians() (in module math)
radians() (in module turtle)
RadioButtonGroup (class in msilib)
radiogroup() (Dialog method)
RADIXCHAR (in module locale)
raise statement
randint() (in module random)
random() (in module random)
random (standard module)
randrange() (in module random)
rangle() (built-in function)
ratecv() (in module audioop)
ratio() (SequenceMatcher method)
raw() (in module curses)
raw_input() (built-in function), [Link]
raw_input() (InteractiveConsole method)
RawConfigParser (class in ConfigParser)
RawPen (class in turtle)
re (MatchObject attribute)
re (standard module), [Link], [Link], [Link]
read() (array method)
read() (audio device method), [Link]
read() (BZ2File method)
replace_whitespace (TextWrapper attribute)
replaceChild() (Node method)
ReplacePackage() (in module modulefinder)
report() (dircmp method)
report() (ModuleFinder method)
REPORT_CDIFF (in module doctest)
report_failure() (DocTestRunner method)
report_full_closure() (dircmp method)
REPORT_NDIFF (in module doctest)
REPORT_ONLY_FIRST_FAILURE (in module doctest)
report_partial_closure() (dircmp method)
report_start() (DocTestRunner method)
report_success() (DocTestRunner method)
REPORT_UDIFF (in module doctest)
report_unbalanced() (SGMLParser method)
report_unexpected_exception() (DocTestRunner method)
REPORTING_FLAGS (in module doctest)
repr() (built-in function)
Repr (class in repr)
repr() (in module repr)
repr() (Repr method)
repr (standard module)
read() (Chunk method)
read() (file method)
read() (HTTPRequestResponse method)
read() (IMAP4_stream method)
read() (in module imgfile)
read() (in module os)
read() (MimeTypes method)
read() (MultiFile method)
read() (Popen method)
read() (RobotFileParser method)
read() (SafeConfigParser method)
read() (StreamReader method)
read() (Timer method)
read() (ZipFile method)
read_all() (Telnet method)
read_byte() (Timer method)
read_eager() (Telnet method)
read_history_file() (in module readline)
read_init_file() (in module readline)
read_lazy() (Telnet method)
read_mime_types() (in module mimetypes)
read_sb_data() (Telnet method)
read_some() (Telnet method)
read_token() (shlex method)
read_until() (Telnet method)
read_very_eager() (Telnet method)
read_very_lazy() (Telnet method)
readable() (async_chat method)
readable() (dispatcher method)
readadd() (CD player method)
reader() (in module csv)
ReadError
readfp() (MimeTypes method)
readfp() (SafeConfigParser method)
readframes() (aifc method)
readframes() (AU_read method)
readframes() (Wave_read method)
readline (built-in module)
repr1() (Repr method)
Request (class in urllib2)
request() (HTTPRequestResponse method)
request_queue_size (SocketServer protocol)
request_uri() (in module wsgiref.util)
request_version
(BaseHTTPRequestHandler attribute)
RequestHandlerClass (SocketServer protocol)
requires() (in module test.test_support)
reserved (ZipInfo attribute)
RESERVED_FUTURE (in module uuid)
RESERVED_MICROSOFT (in module uuid)
RESERVED_NCS (in module uuid)
reset() (audio device method)
reset() (DOMEventStream method)
reset() (HTMLParser method)
reset() (in module dircache)
reset() (in module turtle)
reset() (IncrementalDecoder method)
reset() (IncrementalEncoder method)
reset() (IncrementalParser method)
reset() (Packer method)
reset() (SGMLParser method)
reset() (StreamReader method)
reset() (StreamWriter method)
reset() (Template method)
reset() (Unpacker method)
reset_prog_mode() (in module curses)
reset_shell_mode() (in module curses)
resetbuffer() (InteractiveConsole method)
resetlocale() (in module locale)
resetparser() (CD parser method)
resetwarnings() (in module warnings)
resize() (in module ctypes)
readline() (BZ2File method)
readline() (file method)
readline() (IMAP4_stream method)
readline() (MultiFile method)
readline() (StreamReader method)
readline() (Timer method)
readlines() (BZ2File method)
readlines() (file method)
readlines() (MultiFile method)
readlines() (StreamReader method)
readlink() (in module os)
readmodule() (in module pycclbr)
readmodule_ex() (in module pycclbr)
readsamps() (audio port method)
readscaled() (in module imgfile)
READY (in module cd)
Real Media File Format
real_quick_ratio() (SequenceMatcher method)
realpath() (in module os.path)
reason (in module httplib)
reccontrols() (mixer device method)
recent() (IMAP4_stream method)
rectangle() (in module curses.textpad)
recv() (dispatcher method)
recv() (socket method)
recvfrom() (socket method)
redirect_request() (HTTPRedirectHandler method)
redisplay() (in module readline)
redraw_form() (form method)
redraw_object() (FORMS object method)
redrawln() (window method)
redrawwin() (window method)
reduce() (built-in function)
ref (class in weakref)
ReferenceError, [Link]
ReferenceType (in module weakref)
resize() (Timer method)
resolution (date attribute)
resolution (datetime attribute)
resolution (time attribute)
resolution (timedelta attribute)
resolveEntity() (EntityResolver method)
resource (built-in module)
ResourceDenied
response() (IMAP4_stream method)
ResponseNotReady
responses (BaseHTTPRequestHandler attribute)
responses (in module httpplib)
restore() (in module difflib)
restype (_FuncPtr attribute)
retr() (POP3_SSL method)
retrbinary() (FTP method)
retrieve() (URLopener method)
retrlines() (FTP method)
return_ok() (CookiePolicy method)
returncode (Popen attribute)
returns_unicode (xmlparser attribute)
reverse() (array method)
reverse() (in module audioop)
reverse() (list method)
reverse_order() (Stats method)
reversed() (built-in function)
revert() (FileCookieJar method)
rewind() (aifc method)
rewind() (AU_read method)
rewind() (Wave_read method)
rewindbody() (AddressList method)
RExec (class in rexec)
rexec (standard module), [Link]
rfc2109 (Cookie attribute)
rfc2109_as_netscape (LWPCookieJar attribute)
rfc2965 (LWPCookieJar attribute)
refilemessages() (Folder method)
refill_buffer() (async_chat method)
refresh() (window method)
register() (in module atexit)
register() (in module codecs)
register() (in module webbrowser)
register() (Structure method)
register_adapter() (in module sqlite3)
register_converter() (in module sqlite3)
register_dialect() (in module sqlitedb)
register_error() (in module codecs)
register_adapter() (in module sqlite3)
register_converter() (in module sqlite3)
register_dialect() (in module sqlitedb)
register_function()
(SimpleXMLRPCRequestHandler
method)
register_function()
(SimpleXMLRPCServer method)
register_instance()
(SimpleXMLRPCRequestHandler
method)
register_instance()
(SimpleXMLRPCServer method)
register_introspection_functions()
(SimpleXMLRPCRequestHandler
method), [Link]
register_multicall_functions()
(SimpleXMLRPCRequestHandler
method), [Link]
register_optionflag() (in module
doctest)
registerDOMImplementation() (in
module xml.dom)
RegLoadKey() (in module _winreg)
relative
  URL
release() (Condition method)
release() (DbfilenameShelf method)
release() (in module platform)
release() (lock method)
release() (Semaphore method)
rfc822 (standard module), [Link]
RFC_4122 (in module uuid)
rfile (BaseHTTPRequestHandler
attribute)
rfind() (in module string)
rfind() (string method)
rgb_to_hls() (in module colorsys)
rgb_to_hsv() (in module colorsys)
rgb_to_yiq() (in module colorsys)
rgbimg (built-in module)
right() (in module turtle)
right_list (dircmp attribute)
right_only (dircmp attribute)
rindex() (in module string)
rindex() (string method)
rjust() (in module string)
rjust() (string method)
rlcompleter (standard module)
rldecode_hqx() (in module binascii)
rldecode_hqx() (in module binascii)
RLIMIT_AS (in module resource)
RLIMIT_CORE (in module resource)
RLIMIT_CPU (in module resource)
RLIMIT_DATA (in module resource)
RLIMIT_FSIZE (in module resource)
RLIMIT_MEMLOCK (in module
resource)
RLIMIT_NOFILE (in module
resource)
RLIMIT_NPROC (in module resource)
RLIMIT_OFILE (in module resource)
RLIMIT_RSS (in module resource)
RLIMIT_STACK (in module resource)
RLIMIT_VMEM (in module resource)
RLock() (in module threading)
rmd() (FTP method)
rmdir() (in module os)
RMFF
rms() (in module audioop)
release() (Timer method), [Link]
release_lock() (in module imp)
reload() (built-in function), [Link], [Link], [Link]
remainder(). (Context method)
remainder_near(). (Context method)
remainder_near(). (Decimal method)
remove(). (array method)
remove() (in module os)
remove() (list method)
remove() (LocaleHTMLCalendar method)
remove() (Mailbox method)
remove() (MH method)
remove_flag() (MaildirMessage method)
remove_flag() (mboxMessage method)
remove_flag() (MMDFMessage method)
remove_folder() (Maildir method)
remove_folder() (MH method)
remove_history_item() (in module readline)
remove_label() (BabylMessage method)
remove_option() (SafeConfigParser method)
remove_pyc() (Directory method)
remove_section() (SafeConfigParser method)
remove_sequence() (MHMessage method)
removeAttribute() (Element method)
removeAttributeNode() (Element method)
removeAttributeNS() (Element method)
removecallback() (CD parser method)
removeChild() (Node method)
rmtree() (in module shutil)
rnopen() (in module bsddb)
RobotFileParser (class in robotparser)
robotparser (standard module)
robots.txt
rotate() (LocaleHTMLCalendar method)
RotatingFileHandler (class in logging)
round() (built-in function)
Rounded (class in decimal)
row_factory (DbfilenameShelf attribute)
rowcount (DbfilenameShelf attribute)
partition() (string method)
rpc_paths (SimpleXMLRPCServer attribute)
rpop() (POP3_SSL method)
rset() (POP3_SSL method)
rshift() (in module operator)/rsplit() (in module string)
rsplit() (string method)
rstrip() (in module string)
rstrip() (string method)
RTLD_LAZY (in module dl)
RTLD_NOW (in module dl)
ruler (Cmd attribute)
run() (BaseHandler method)
run() (DocTestRunner method)
run() (in module cProfile)
run() (in module pdb)
run() (Profile method)
run() (scheduler method)
run() (TestCase method)
run() (TestSuite method)
run() (Thread method)
run() (Trace method)
Run script
run_docstring_examples() (in module doctest)
S (in module re)
s_eval() (RExec method)
s_exec() (RExec method)
s_execfile() (RExec method)
S_IFMT() (in module stat)
S_IMODE() (in module stat)
s_import() (RExec method)
S_ISBLK() (in module stat)
S_ISCHR() (in module stat)
S_ISDIR() (in module stat)
S_ISFIFO() (in module stat)
S_ISLNK() (in module stat)
S_ISREG() (in module stat)
S_ISSOCK() (in module stat)
s_reload() (RExec method)
s_unload() (RExec method)
safe_substitute() (Template method)
SafeConfigParser (class in ConfigParser)
saferepr() (in module pprint)
same_files (dircmp attribute)
same_quantum() (Context method)
same_quantum() (Decimal method)
samefile() (in module os.path)
sameopenfile() (in module os.path)
samestat() (in module os.path)
sample() (in module random)
save() (FileCookieJar method)
save_bgn() (HTMLParser method)
save_end() (HTMLParser method)
SaveKey() (in module _winreg)
SAX2DOM (class in xml.dom.pulldom)
SAXException (exception in xml.sax)
seek() (MultiFile method)
seek() (Timer method)
SEEK_CUR (in module os)
SEEK_CUR (in module posixfile)
SEEK_END (in module os)
SEEK_END (in module posixfile)
SEEK_SET (in module os)
SEEK_SET (in module posixfile)
seekblock() (CD player method)
seektrack() (CD player method)
select (built-in module)
Select (class in Tix)
select() (IMAP4_stream method)
select() (in module gl)
select() (in module select)
Semaphore (class in threading)
Semaphore() (in module threading)
semaphores, binary
send() (DatagramHandler method)
send() (dispatcher method)
send() (HTTPResponse method)
send() (IMAP4_stream method)
send() (socket method)
send() (SocketHandler method)
send_error()
(BaseHTTPRequestHandler method)
send_flowing_data() (writer method)
send_header()
(BaseHTTPRequestHandler method)
send_hor_rule() (writer method)
send_label_data() (writer method)
send_line_break() (writer method)
send_literal_data() (writer method)
send_paragraph() (writer method)
send_query() (in module gopherlib)
send_response()
(BaseHTTPRequestHandler method)
send_selector() (in module gopherlib)
sendall() (socket method)

[Link]
SliceType (in module types)
SmartCookie (class in Cookie)
SMTP
protocol
SMTP (class in smtplib)
SMTPConnectError
smtpd (standard module)
SMTPDataError
SMTPException
SMTPHandler (class in logging)
SMTPHeloError
smtplib (standard module)
SMTPRecipientsRefused
SMTPResponseException
SMTPSenderRefused
SMTPServer (class in smtplib)
SMTPServerDisconnected
SND_ALIAS (in module winsound)
SND_ASYNC (in module winsound)
SND_FILENAME (in module winsound)
SND_LOOP (in module winsound)
SND_MEMORY (in module winsound)
SND_NODEFAULT (in module winsound)
SND_NOSTOP (in module winsound)
SND_NOWAIT (in module winsound)
SND_PURGE (in module winsound)
sndhdr (standard module)
niff() (Sniffer method)
Sniffer (class in csv)
SO_* (in module socket)
SOCK_DGRAM (in module socket)
SOCK_RAW (in module socket)
SOCK_RDM (in module socket)
SOCK_SEQPACKET (in module socket)
SOCK_STREAM (in module socket)
sendcmd() (FTP method)
sendfile() (BaseHandler method)
sendmail() (SMTP method)
sendto() (socket method)
sep (in module os)
seq

sequence

sequence (in module os)
sequence object
sequence2ast() (in module parser)
sequenceIncludes() (in module operator)
SequenceMatcher (class in difflib), [Link]
SerialCookie (class in Cookie)

serializing

objects

serve_forever() (SocketServer protocol)
server

WWW, [Link]
server() (Popen method)
server_activate() (SocketServer protocol)
server_address (SocketServer protocol)
server_bind() (SocketServer protocol)
server_software (BaseHandler attribute)
server_version (BaseHTTPRequestHandler attribute)
server_version (SimpleHTTPRequestHandler attribute)
ServerProxy (class in xmlrpclib)
set() (built-in function)
Set (class in sets)
set() (Event method)
set() (mixer device method)
set() (Morsel method)
set() (SafeConfigParser method),
set allowed_domains()
(DefaultCookiePolicy method)
set_app() (WSGIServer method)
set_authorizer() (DbfilenameShelf method)
set_blocked_domains()
(DefaultCookiePolicy method)
set_boundary() (Message method)
set call_back() (FORMS object method)
set_charset() (Message method)
set_completer()
(in module readline)
set_completer_delims()
(in module readline)
set_conversion_mode()
(in module ctypes)
set_cookie()
(CookieJar method)
set_cookie_if_ok()
(CookieJar method)
set_current() (Feature method)
set_date() (MaildirMessage method)
set_debug()
(in module gc)
set_debuglevel() (FTP method)
set_debuglevel() (HTTPResponse method)
set_debuglevel() (NNTP method)
set_debuglevel() (POP3_SSL method)
set_debuglevel() (SMTP method)
set_debuglevel() (Telnet method)
set_default_type() (Message method)
set_event_call_back() (in module fl)
set_flags()
(MailldirMessage method)
set_flags()
(mboxMessage method)
set_flags() (MMDFMessage method)
set_form_position() (form method)
set() (RegexObject method)
set() (string method)
splittest() (in module os.path)
splitfields() (in module string)
splitlines() (string method)
SplitResult (class in urlparse)
spliteunc() (in module os.path)
sprintf-style formatting
spwd (built-in module)
sqlite3 (built-in module)
sgtr() (Context method)
sgtr() (Decimal method)
sgtr() (in module cmath)
sgtr() (in module math)
ssl() (IMAP4_stream method)
ssl() (in module socket)
ST_ATIME (in module stat)
ST_CTIME (in module stat)
ST_DEV (in module stat)
ST_GID (in module stat)
ST_INO (in module stat)
ST_MODE (in module stat)
ST_MTIME (in module stat)
ST_NLINK (in module stat)
ST_SIZE (in module stat)
ST_UID (in module stat)
stack() (in module inspect)
stack() (in module thread)
stack() (in module threading)
stackable
stack() (in module inspect)
stack_size() (in module thread)
stack_size() (in module threading)
stackable
streams
standard_b64decode() (in module base64)
standard_b64encode() (in module base64)
StandardError
set_spacing() (formatter method)
set_startup_hook() (in module readline)
set_subdir() (MaildirMessage method)
set_terminator() (async_chat method)
set_threshold() (in module gc)
set_trace() (in module pdb)
set_type() (Message method)
set_unittest_reportflags() (in module doctest)
set_unixfrom() (Message method)
set_url() (RobotFileParser method)
set_userptr() (Textbox method)
set_visible() (Baby1Message method)
setacl() (IMAP4_stream method)
setannotation() (IMAP4_stream method)
setattr() (built-in function)
setAttribute() (Element method)
setAttributeNode() (Element method)
setAttributeNodeNS() (Element method)
setAttributeNS() (Element method)
SetBase() (xmlparser method)
setblocking() (socket method)
setByteStream() (InputSource method)
setcbreak() (in module tty)
setchannels() (audio configuration method)
setCharacterStream() (InputSource method)
setcheckinterval() (in module sys)
setcomptype() (aiffc method)
setcomptype() (AU_write method)
setcomptype() (Wave_write method)
setconfig() (audio port method)
setContentHandler() (XMLReader method)
setcontext() (in module decimal)
setcontext() (MH method)
assert
del, [Link]
extcept
evac
if
import, [Link]
print
raise
try
while
staticmethod() (built-in function)
Stats (class in cProfile)
status() (IMAP4_stream method)
status (in module httpplib)
statvfs() (in module os)
statvfs (standard module), [Link]
StdButtonBox (class in Tix)
stderr (in module sys)
stderr (Popen attribute)
stdin (in module sys)
stdin (Popen attribute)
stdout (in module sys)
stdout (Popen attribute)
Stein, Greg
stereocontrols() (mixer device method)
STILL (in module cd)
stop() (CD player method)
stop() (Profile method)
stop() (TestResult method)
StopIteration
stopListening() (in module logging)
stopTest() (TestResult method)
storbinary() (FTP method)
store() (IMAP4_stream method)
storlines() (FTP method)
str() (built-in function)
str() (in module locale)
strcoll() (in module locale)
StreamError
setcurrent() (Folder method)
setDaemon() (Thread method)
setdefault() (dictionary method)
setdefaultencoding() (in module sys)
setdefaulttimeout() (in module socket)
setdlopenflags() (in module sys)
setDocumentLocator() (ContentHandler method)
setDTDHandler() (XMLReader method)
setegid() (in module os)
setEncoding() (InputSource method)
setEntityResolver() (XMLReader method)
setErrorHandler() (XMLReader method)
seteuid() (in module os)
setFeature() (XMLReader method)
setfillpoint() (audio port method)
setfirstweekday() (in module calendar)
setfloatmax() (audio configuration method)
setfmt() (audio device method)
setfont() (font handle method)
setFormatter() (DbfilenameShelf method)
setframerate() (aiffc method)
setframerate() (AU_write method)
setframerate() (Wave_write method)
setgid() (in module os)
setgroups() (in module os)
setheading() (in module turtle)
setinfo() (audio device method)
SetInteger() (Binary method)
setitem() (in module operator)
setlast() (Folder method)
setLevel() (DbfilenameShelf method), [Link]
setliteral() (SGMLParser method)

StreamHandler (class in logging)
StreamReader (class in codecs)
StreamReaderWriter (class in codecs)
StreamRecoder (class in codecs)
streams
stackable
StreamWriter (class in codecs)
strerror() (in module os)
strftime() (date method)
strftime() (datetime method)
strftime() (in module time)
strftime() (time method)
strict_domain (LWPCookieJar attribute)
strict_errors() (in module codecs)
strict_ns_domain (LWPCookieJar attribute)
strict_ns_set_initial_dollar (LWPCookieJar attribute)
strict_ns_set_path (LWPCookieJar attribute)
strict_ns_unverifiable (LWPCookieJar attribute)
strict_rfc2965_unverifiable (LWPCookieJar attribute)
string
documentation
formatting
interpolation
methods
string (MatchObject attribute)
string (standard module), [Link], [Link], [Link]
string object
string_at() (in module ctypes)
StringIO (class in StringIO)
StringIO (standard module)
stringprep (standard module)
StringType (in module types)
setreuid() (in module os)
setrlimit() (in module resource)
sets (standard module)
setsampfmt() (audio configuration method)
setsampwidth() (aiffc method)
setsampwidth() (AU_write method)
setsampwidth() (Wave_write method)
setscrreg() (window method)
setsid() (in module os)
setslice() (in module operator)
setsockopt() (socket method)
setstate() (in module random)
SetStream() (Binary method)
SetString() (Binary method)
setSystemId() (InputSource method)
setsyx() (in module curses)
setTarget() (MemoryHandler method)
settimeout() (CacheFTPHandler method)
settimeout() (socket method)
settrace() (in module sys)
settscdump() (in module sys)
suid() (in module os)
system() (in module os)
system() (in module platform)
symlink() (in module os)
symlink() (DbfilenameShelf method)
sym() (Timer method)
sym_name (in module symbol)
symbol (standard module)
symbol table
sync() (audio device method)
sync() (DbfilenameShelf method),
sync() (Dbhash method)
sync() (module gdbm)
sync() (Shelf method)
syncdown() (window method)
syncok() (window method)
syncup() (window method)
SyntaxErr
SyntaxError
SyntaxWarning
sys (built-in module)
sys_version
(BaseHTTPRequestHandler attribute)
sysconf() (module os)
sysconf_names (module os)
syslog (built-in module)
syslog() (module syslog)
SysLogHandler (class in logging)
system() (module os)
system() (module platform)
system.listMethods() (ServerProxy
sgmlib (standard module), [Link]
SGMLParseError
SGMLParser (class in sgmllib)
SGMLParser (in module sgmllib)
sha (built-in module)
Shelf (class in shelve)
shelve (standard module), [Link]
shift_path_info() (in module wsgiref.util)
shifting
  operations
shlex (class in shlex)
shlex (standard module)
shortDescription() (TestCase method)
shouldFlush() (BufferingHandler method)

system.methodHelp() (ServerProxy method)
system.methodSignature() (ServerProxy method)
system_alias() (in module platform)
SystemError
SystemExit
systemId (DocumentType attribute)
SystemRandom (class in random)
T_FMT (in module locale)
T_FMT_AMPM (in module locale)
tabnanny (standard module)
tabular
    data
tagName (Element attribute)
takewhile() (in module itertools)
tan() (in module cmath)
tan() (in module math)
tanh() (in module cmath)
tanh() (in module math)
TAR_GZIPPED (in module tarfile)
TAR_PLAIN (in module tarfile)
TarError
TarFile (class in tarfile), [Link]
tarfile (standard module)
TarFileCompat (class in tarfile)
target (ProcessingInstruction attribute)
TarInfo (class in tarfile)
task_done() (Queue method)
tb_lineno() (in module traceback)
tcdrain() (in module termios)
tcflow() (in module termios)
tcflush() (in module termios)
tcgetattr() (in module termios)
tcgetpgrp() (in module os)
Tcl() (in module Tkinter)
TCP_* (in module socket)
tcsendbreak() (in module termios)
tcsetattr() (in module termios)
tix_getbitmap() (tixCommand method)
tix_getimage() (tixCommand method)
TIX_LIBRARY (environment variable)
tix_option_get() (tixCommand method)
tix_resetoptions() (tixCommand method)
tixCommand (class in Tix)
Tk
Tk (class in Tkinter)
Tk Option Data Types
Tkinter
Tkinter (standard module)
TList (class in Tix)
TMP (environment variable), [Link]
TMP_MAX (in module os)
TMPDIR (environment variable), [Link]
tmpfile() (in module os)
tmpnam() (in module os)
to_eng_string() (Context method)
to_eng_string() (Decimal method)
to_integral() (Context method)
to_integral() (Decimal method)
to_sci_string() (Context method)
to_splittable() (Charset method)
ToASCII() (in module encodings.idna)
tobuf() (TarInfo method)
tochild (Popen4 attribute)
today() (date method)
today() (datetime method)
tofile() (array method)
togglepause() (CD player method)
tok_name (in module token)
token (shlex attribute)
tcsetpgrp() (in module os)
tearDown() (TestCase method)
tee() (in module itertools)
tell() (aifc method), [Link]
tell() (AU_read method)
tell() (AU_write method)
tell() (BZ2File method)
tell() (Chunk method)
tell() (file method)
tell() (MultiFile method)
tell() (Timer method)
tell() (Wave_read method)
tell() (Wave_write method)
Telnet (class in telnetlib)
telnetlib (standard module)
TEMP (environment variable)
tempdir (in module tempfile)
template (standard module)
Template (class in pipes)
Template (class in string)
template (in module tempfile)
template (string attribute)
templnam() (in module os)
temporary
    file
    file name
TemporaryFile() (in module tempfile)
termattrr() (in module curses)
termios (built-in module)
termname() (in module curses)
test (DocTestFailure attribute)
test() (in module cgi)
test() (mutex method)
test (standard module)
test (UnexpectedException attribute)
test.test_support (standard module)
testandset() (mutex method)
TestCase (class in unittest)
TestCase (class in unittest)
TestFailed
token (standard module)
tokeneater() (in module tabnanny)
tokenize() (in module tokenize)
tokenize (standard module)
tolist() (array method)
tolist() (AST method)
tomono() (in module audioop)
toordinal() (date method)
toordinal() (datetime method)
top() (POP3_SSL method)
top() (Textbox method)
top_panel() (in module curses.panel)
toprettyxml() (Node method)
tostereo() (in module audioop)
tostring() (array method)
tostring() (in module xml.etree.ElementTree)
total_changes (DbfilenameShelf attribute)
totuple() (AST method)
touchline() (window method)
touchwin() (window method)
tounicode() (array method)
ToUnicode() (in module encodings.idna)
tovideo() (in module imageop)
towards() (in module turtle)
toxml() (Node method)
tparm() (in module curses)
Trace (class in trace)
trace() (in module inspect)
trace (standard module)
trace_function, [Link]
traceback (standard module)
traceback object, [Link]
traceback_limit (BaseHandler attribute)
tracebacklimit (in module sys)
tracebacks
    in CGI scripts
testfile() (in module doctest)
TESTFN (in module test.test_support)
TestLoader (class in unittest)
testMethodPrefix (TestLoader attribute)
testmod() (in module doctest)
TestResult (class in unittest)
tests (in module imghdr)
TestSkipped
testsSource () (in module doctest)
testsRun (TestResult attribute)
TestSuite (class in unittest)
testzip() (ZipFile method)
text() (Dialog method)
text (in module msilib)
text_factory (DbffilenameShelf attribute)
Textbox (class in curses.textpad)
TextCalendar (class in calendar)
textdomain() (in module gettext)
TextTestRunner (class in unittest)
textwrap (standard module)
TextWrapper (class in textwrap)
THOUSEP (in module locale)
thread (built-in module)
Thread (class in threading), [Link]
thread() (IMAP4_stream method)
threading (standard module)
threads
    IRIX
    POSIX
tie() (in module fl)
tigetflag() (in module curses)
tigetnum() (in module curses)
tigetstr() (in module curses)
time (built-in module)
time (class in datetime), [Link]
time() (datetime method)
time() (in module time)
Time2Internaldate() (in module
TracebackType (in module types)
tracer() (in module turtle)
 transferrcmd() (FTP method)
 translate() (in module string)
 translate() (string method)
 translation() (in module gettext)
Tree (class in Tix)
TreeBuilder (class in
testsRun (TestResult attribute)
TreeSuite (class in unittest)
testsSource () (in module doctest)
testsRun (TestResult attribute)
TestSuite (class in unittest)
testzip() (ZipFile method)
text() (Dialog method)
text (in module msilib)
text_factory (DbffilenameShelf attribute)
Textbox (class in curses.textpad)
TextCalendar (class in calendar)
textdomain() (in module gettext)
TextTestRunner (class in unittest)
textwrap (standard module)
TextWrapper (class in textwrap)
THOUSEP (in module locale)
thread (built-in module)
Thread (class in threading), [Link]
thread() (IMAP4_stream method)
threading (standard module)
threads
    IRIX
    POSIX
tie() (in module fl)
tigetflag() (in module curses)
tigetnum() (in module curses)
tigetstr() (in module curses)
time (built-in module)
time (class in datetime), [Link]
time() (datetime method)
time() (in module time)
Time2Internaldate() (in module
type
    Boolean
    operations on dictionary
    operations on list
type() (built-in function), [Link],
[Link], [Link]
imaplib

timedelta (class in datetime), [Link]
TimedRotatingFileHandler (class in logging)
timegm() (in module calendar)
timeit (standard module)
timeit() (Timer method)
timeout
timeout() (window method)
Timer (class in threading), [Link]
Timer (class in timeit)
times() (in module os)
timetuple() (date method)
timetuple() (datetime method)
timetz() (datetime method)
timezone (in module time)
title() (in module turtle)
title() (string method)
Tix
Tix (class in Tix)
Tix (standard module)
tix_addbitmapdir() (tixCommand method)
tix_cget() (tixCommand method)
tix_configure() (tixCommand method)
tix_filedialog() (tixCommand method)
type (socket attribute)
type (TarInfo attribute)
type object
typeahead() (in module curses)
typecode (array attribute)
typed_subpart_iterator() (in module email.iterators)
TypeError
types

built-in, [Link]
mutable sequence
operations on integer
operations on mapping
operations on mutable sequence
operations on numeric
operations on sequence, [Link]
types (standard module), [Link]
types_map (in module mimetypes), [Link]
TypeType (in module types)
TZ (environment variable), [Link], [Link], [Link], [Link]
tzinfo (class in datetime)
tzinfo (datetime attribute)
tzinfo (time attribute)
tzname() (datetime method)
tzname (in module time)
tzname() (time method), [Link]
tzset() (in module time)
U (in module re)
u-LAW, [Link], [Link], [Link]
ucd_3_2_0 (in module unicodedata)
ugettext() (GNUTranslations method)
ugettext() (NullTranslations method)
uid() (IMAP4_stream method)
uid (TarInfo attribute)
uidl() (POP3_SSL method)
ulaw2lin() (in module audioop)
umask() (in module os)
uname() (in module os)
uname() (in module platform)
uname (TarInfo attribute)
UnboundLocalError
UnboundMethodType (in module types)
unbuffered I/O
UNC paths
and os.makedirs()
unconsumed_tail (dircmp attribute)
unctrl() (in module curses)
unctrl() (in module curses.ascii)
Underflow (class in decimal)
undoc_header (Cmd attribute)
unescape() (in module xml.sax.saxutils)
UnexpectedException (exception in doctest)
unfreeze_form() (form method)
unfreeze_object() (FORMS object method)
ungetch() (in module curses)
ungetch() (in module msvcrt)
ungetmouse() (in module curses)
ungettext() (GNUTranslations method)
unqdevice() (in module fl)
unquote() (in module email.utils)
unquote() (in module rfc822)
unquote() (in module urllib)
unquote_plus() (in module urllib)
unregister() (Structure method)
unregister_dialect() (in module csv)
unsetenv() (in module os)
unsubscribe() (IMAP4_stream method)
untokenize() (in module tokenize)
untouchwin() (window method)
unused_data (dircmp attribute)
up() (in module turtle)
update() (dictionary method)
update() (hash method)
update() (hmac method)
update() (Mailbox method)
update() (Maildir method)
update() (md5 method)
update() (sha method)
update_panels() (in module curses.panel)
update_visible() (BabylMessage method)
update_wrapper() (in module functools)
upper() (in module string)
upper() (string method)
uppercase (in module string)
ungettext() (NullTranslations method)
unhexlify() (in module binascii)
unichr() (built-in function)
Unicode, [Link]
database
unicode() (built-in function)
UNICODE (in module re)
Unicode object
unicodedata (standard module)
UnicodeDecodeError
UnicodeEncodeError
UnicodeError
UnicodeTranslateError
UnicodeType (in module types)
UnicodeWarning
unidata_version (in module unicodedata)
unified_diff() (in module difflib)
uniform() (in module random)
UnimplementedFilemode
Union (class in ctypes)
unittest (standard module)
Unix
   file control
   I/O control
unixfrom (AddressList attribute)
UnixMailbox (class in mailbox)
unknown_charref() (SGMLParser method)
unknown_endtag() (SGMLParser method)
unknown_entityref() (SGMLParser method)
unknown_open() (BaseHandler method)
unknown_open() (HTTPErrorProcessor method)
unknown_open() (UnknownHandler method)
urandom() (in module os)
URL, [Link], [Link], [Link], [Link]
parsing
relative
url (ServerProxy attribute)
url2pathname() (in module urllib)
urlcleanup() (in module urllib)
urlobrq() (in module urlparse)
urlencode() (in module urllib)
URLError
urljoin() (in module urllib)
urllib (standard module), [Link]
urllib2 (standard module)
urlopen() (in module urllib)
urlopen() (in module urllib2)
URLopener (class in urllib)
urlparse() (in module urlparse)
urlparse (standard module), [Link]
urlretrive() (in module urllib)
urllsafe_b64decode() (in module base64)
urllsafe_b64encode() (in module base64)
urllsplit() (in module urlparse)
urllunparse() (in module urlparse)
urllunparse() (in module urlparse)
urn (UUID attribute)
use_default_colors() (in module curses)
use_env() (in module curses)
use_rawinput (Cmd attribute)
UseForeignDTD() (xmlparser method)
user
collection file
effective id
id
id, setting
USER (environment variable)
user() (POP3_SSL method)
user (standard module)
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>validator()</td>
<td>(in module wsgiref.validate)</td>
</tr>
<tr>
<td>value</td>
<td>truth</td>
</tr>
<tr>
<td>value</td>
<td>(_SimpleCData attribute)</td>
</tr>
<tr>
<td>value</td>
<td>(Cookie attribute)</td>
</tr>
<tr>
<td>value</td>
<td>(Morsel attribute)</td>
</tr>
<tr>
<td>value_decode()</td>
<td>(BaseCookie method)</td>
</tr>
<tr>
<td>value_encode()</td>
<td>(BaseCookie method)</td>
</tr>
<tr>
<td>ValueError</td>
<td></td>
</tr>
<tr>
<td>valuerefs()</td>
<td>(WeakValueDictionary method)</td>
</tr>
<tr>
<td>values</td>
<td>Boolean</td>
</tr>
<tr>
<td>values()</td>
<td>(dictionary method)</td>
</tr>
<tr>
<td>values()</td>
<td>(Mailbox method)</td>
</tr>
<tr>
<td>values()</td>
<td>(Message method)</td>
</tr>
<tr>
<td>variant()</td>
<td>(UUID attribute)</td>
</tr>
<tr>
<td>varray()</td>
<td>(in module gl)</td>
</tr>
<tr>
<td>vars()</td>
<td>(built-in function)</td>
</tr>
<tr>
<td>vbar</td>
<td>(ScrolledText attribute)</td>
</tr>
<tr>
<td>VERBOSE</td>
<td>(in module re)</td>
</tr>
<tr>
<td>verbose</td>
<td>(in module tabnanny)</td>
</tr>
<tr>
<td>verbose</td>
<td>(in module test.test_support)</td>
</tr>
<tr>
<td>version</td>
<td></td>
</tr>
<tr>
<td>version()</td>
<td>(in module platform)</td>
</tr>
<tr>
<td>version()</td>
<td>(in module sys)</td>
</tr>
<tr>
<td>version()</td>
<td>(URLopener attribute)</td>
</tr>
<tr>
<td>version()</td>
<td>(UUID attribute)</td>
</tr>
<tr>
<td>version()</td>
<td>(URLopener attribute)</td>
</tr>
<tr>
<td>version()</td>
<td>(UUID attribute)</td>
</tr>
<tr>
<td>version_info()</td>
<td>(in module sys)</td>
</tr>
<tr>
<td>version_string()</td>
<td></td>
</tr>
<tr>
<td>vline()</td>
<td>(window method)</td>
</tr>
<tr>
<td>vnarray()</td>
<td>(in module gl)</td>
</tr>
<tr>
<td>voidcmd()</td>
<td>(FTP method)</td>
</tr>
<tr>
<td>volume</td>
<td>(ZipInfo attribute)</td>
</tr>
<tr>
<td>vonmisesvariate()</td>
<td>(in module random)</td>
</tr>
<tr>
<td>Function/Calls</td>
<td>Module/Class</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
</tr>
<tr>
<td>W_OK</td>
<td>(in module os)</td>
</tr>
<tr>
<td>wait()</td>
<td>(Condition method)</td>
</tr>
<tr>
<td>wait()</td>
<td>(Event method)</td>
</tr>
<tr>
<td>wait()</td>
<td>(in module os)</td>
</tr>
<tr>
<td>wait()</td>
<td>(Popen method)</td>
</tr>
<tr>
<td>wait()</td>
<td>(Popen4 method)</td>
</tr>
<tr>
<td>wait3()</td>
<td>(in module os)</td>
</tr>
<tr>
<td>wait4()</td>
<td>(in module os)</td>
</tr>
<tr>
<td>waitpid()</td>
<td>(in module os)</td>
</tr>
<tr>
<td>walk()</td>
<td>(in module compiler)</td>
</tr>
<tr>
<td>walk()</td>
<td>(in module compiler.visitor)</td>
</tr>
<tr>
<td>walk()</td>
<td>(in module os)</td>
</tr>
<tr>
<td>walk()</td>
<td>(in module os.path)</td>
</tr>
<tr>
<td>walk()</td>
<td>(Message method)</td>
</tr>
<tr>
<td>want()</td>
<td>(Example attribute)</td>
</tr>
<tr>
<td>warn()</td>
<td>(in module warnings)</td>
</tr>
<tr>
<td>warn()</td>
<td>(message string method)</td>
</tr>
<tr>
<td>Warning</td>
<td>(DbfilenameShelf method)</td>
</tr>
<tr>
<td>warning()</td>
<td>(ErrorHandler method)</td>
</tr>
<tr>
<td>warning()</td>
<td>(in module logging)</td>
</tr>
<tr>
<td>warnings</td>
<td>(standard module)</td>
</tr>
<tr>
<td>warnoptions</td>
<td>(in module sys)</td>
</tr>
<tr>
<td>wasSuccessful()</td>
<td>(TestResult method)</td>
</tr>
<tr>
<td>wave</td>
<td>(standard module)</td>
</tr>
<tr>
<td>WCONTINUED</td>
<td>(in module os)</td>
</tr>
<tr>
<td>WCOREDUMP()</td>
<td>(in module os)</td>
</tr>
<tr>
<td>WeakKeyDictionary</td>
<td>(class in weakref)</td>
</tr>
<tr>
<td>weakref</td>
<td>(extension module)</td>
</tr>
<tr>
<td>WeakValueDictionary</td>
<td>(class in weakref)</td>
</tr>
</tbody>
</table>

- WinSock
- winsound (built-in module)
- winver (in module sys)
- WNOHANG (in module os)
- wordchars (shlex attribute)
- World Wide Web, [Link], [Link], [Link]
- wrap() (in module textwrap)
- wrap() (TextWrapper method)
- wrappert (in module curses.wrapper)
- wraps() (in module funtools)
- writable() (async_chat method)
- writable() (dispatcher method)
- write() (array method)
- write() (audio device method), [Link]
- write() (BZ2File method)
- write() (ElementTree method)
- write() (file method)
- write() (Generator method)
- write() (in module imgfile)
- write() (in module os)
- write() (in module turtle)
- write() (InteractiveConsole method)
- write() (Popen method)
- write() (SafeConfigParser method)
- write() (StreamReader method)
- write() (Telnet method)
- write() (Timer method)
- write() (ZipFile method)
- write_byte() (Timer method)
- write_history_file() (in module readline)
- writeall() (audio device method)
- writeframes() (aifc method)
<table>
<thead>
<tr>
<th>function</th>
<th>module</th>
<th>function</th>
<th>module</th>
</tr>
</thead>
<tbody>
<tr>
<td>webbrowser</td>
<td>standard module</td>
<td>writeframes</td>
<td>AU_write method</td>
</tr>
<tr>
<td>weekday</td>
<td>date method</td>
<td>writeframes</td>
<td>Wave_write method</td>
</tr>
<tr>
<td>weekday</td>
<td>datetime method</td>
<td>writeframesraw</td>
<td>aifc method</td>
</tr>
<tr>
<td>weekday</td>
<td>in module calendar</td>
<td>writeframesraw</td>
<td>AU_write method</td>
</tr>
<tr>
<td>weekheader</td>
<td>in module calendar</td>
<td>writeframesraw</td>
<td>Wave_write method</td>
</tr>
<tr>
<td>weibullvariate</td>
<td>in module random</td>
<td>writelines</td>
<td>BZ2File method</td>
</tr>
<tr>
<td>WEXITSTATUS</td>
<td>in module os</td>
<td>writelines</td>
<td>file method</td>
</tr>
<tr>
<td>wfile</td>
<td>BaseHTTPRequestHandler attribute</td>
<td>writelines</td>
<td>StreamWriter method</td>
</tr>
<tr>
<td>what</td>
<td>in module imghdr</td>
<td>writepy</td>
<td>PyZipFile method</td>
</tr>
<tr>
<td>what</td>
<td>in module sndhdr</td>
<td>writer</td>
<td>formatter attribute</td>
</tr>
<tr>
<td>whathdr</td>
<td>in module sndhdr</td>
<td>writer</td>
<td>in module csv</td>
</tr>
<tr>
<td>whichdb</td>
<td>in module whichdb</td>
<td>writerow</td>
<td>csv writer method</td>
</tr>
<tr>
<td>whichdb</td>
<td>standard module</td>
<td>writerows</td>
<td>csv writer method</td>
</tr>
<tr>
<td>while statement</td>
<td></td>
<td>writesamps</td>
<td>audio port method</td>
</tr>
<tr>
<td>whitespace</td>
<td>in module string</td>
<td>writestr</td>
<td>ZipFile method</td>
</tr>
<tr>
<td>whitespace</td>
<td>shlex attribute</td>
<td>writexml</td>
<td>Node method</td>
</tr>
<tr>
<td>whitespace_split</td>
<td>shlex attribute</td>
<td>WrongDocumentErr</td>
<td></td>
</tr>
<tr>
<td>whseed</td>
<td>in module random</td>
<td>wsgi_file_wrapper</td>
<td>BaseHandler attribute</td>
</tr>
<tr>
<td>WichmannHill</td>
<td>class in random</td>
<td>wsgi_multiprocess</td>
<td>BaseHandler attribute</td>
</tr>
<tr>
<td>width</td>
<td>in module turtle</td>
<td>wsgi_multithread</td>
<td>BaseHandler attribute</td>
</tr>
<tr>
<td>width</td>
<td>TextWrapper attribute</td>
<td>wsgi_run_once</td>
<td>BaseHandler attribute</td>
</tr>
<tr>
<td>WIFCONTINUED</td>
<td>in module os</td>
<td>wsgiref</td>
<td>module</td>
</tr>
<tr>
<td>WIFEXITED</td>
<td>in module os</td>
<td>wsgiref.handlers</td>
<td>module</td>
</tr>
<tr>
<td>WIFSIGNALED</td>
<td>in module os</td>
<td>wsgiref.headers</td>
<td>module</td>
</tr>
<tr>
<td>WIFSTOPPED</td>
<td>in module os</td>
<td>wsgiref.simple_server</td>
<td>module</td>
</tr>
<tr>
<td>Wimp$ScrapDir</td>
<td>environment variable</td>
<td>wsgiref.util</td>
<td>module</td>
</tr>
<tr>
<td>win32_ver</td>
<td>in module platform</td>
<td>wsgiref.validate</td>
<td>module</td>
</tr>
<tr>
<td>WinDLL</td>
<td>class in ctypes</td>
<td>WSGIRequestHandler</td>
<td>class in wsgiref.simple_server</td>
</tr>
<tr>
<td>window</td>
<td>Textbox method</td>
<td>WSGIServer</td>
<td>class in wsgiref.simple_server</td>
</tr>
<tr>
<td>window manager</td>
<td>widgets</td>
<td>WSTOPSIG</td>
<td>in module os</td>
</tr>
<tr>
<td>window_height</td>
<td>in module turtle</td>
<td>wstring_at</td>
<td>in module ctypes</td>
</tr>
<tr>
<td>window_width</td>
<td>in module turtle</td>
<td>WTERMSIG</td>
<td>in module os</td>
</tr>
<tr>
<td>Windows ini file</td>
<td></td>
<td>WUNTRACED</td>
<td>in module os</td>
</tr>
<tr>
<td>WindowsError</td>
<td></td>
<td>WWW, [Link], [Link], [Link]</td>
<td></td>
</tr>
</tbody>
</table>
server, [Link]
X (in module re)
X_OK (in module os)
xatom() (IMAP4_stream method)
XDR, [Link]
xdrlib (standard module)
xgtitle() (NNTP method)
xhrdr() (NNTP method)
XHTML
XML_NAMESPACE (in module xml.dom)
XML() (in module xml.etree.ElementTree)
xml.dom (standard module)
xml.dom.minidom (standard module)
xml.dom.pulldom (standard module)
xml.etree.ElementTree (standard module)
xmiparsers.expat (standard module)
xmiparser (standard module)
xmiparser.handler (standard module)
xmiparser.saxutils (standard module)
xmiparser.xmlreader (standard module)
XMLcharrefreplace_errors_errors() (in module codecs)
XmlDeclHandler() (xmlparser method)
XMLFilterBase (class in xml.sax.saxutils)
XMLGenerator (class in xml.sax.saxutils)
XMLID() (in module xml.etree.ElementTree)
XMLNS_NAMESPACE (in module xml.dom)
XMLParserType (in module xml.parsers.expat)
XMLReader (class in xml.sax.xmlreader)
xmlrpclib (standard module)
XMLTreeBuilder (class in xml.etree.ElementTree)
xor() (in module operator)
xover() (NNTP method)
xpath() (NNTP method)
xrange() (built-in function), [Link], [Link]
xrange object, [Link]
XRangeType (in module types)
xreadlines() (BZ2File method)
xreadlines() (file method)
Y

Y2K
year (date attribute)
year (datetime attribute)
Year 2000
Year 2038

yeardatescalendar() (Calendar method)
yeardays2calendar() (Calendar method)
yeardayscalendar() (Calendar method)
YESEXPR (in module locale)
yiq_to_rgb() (in module colorsys)
ZeroDivisionError
zfll() (in module string)
zfill() (string method)
zip() (built-in function)
ZIP_DEFLATED (in module zipfile)
ZIP_STORED (in module zipfile)
ZipFile (class in zipfile), [Link]
zipfile (standard module)
zipimport (standard module)
zipimporter (class in zipimport), [Link]
ZipImportError
ZipInfo (class in zipfile)
zlib (built-in module)
Front Matter

Contents

1. Introduction
   1.1 Alternate Implementations
   1.2 Notation

2. Lexical analysis
   2.1 Line structure
      2.1.1 Logical lines
      2.1.2 Physical lines
      2.1.3 Comments
      2.1.4 Encoding declarations
      2.1.5 Explicit line joining
      2.1.6 Implicit line joining
      2.1.7 Blank lines
      2.1.8 Indentation
      2.1.9 Whitespace between tokens
   2.2 Other tokens
   2.3 Identifiers and keywords
      2.3.1 Keywords
      2.3.2 Reserved classes of identifiers
   2.4 Literals
      2.4.1 String literals
      2.4.2 String literal concatenation
5.4 The power operator
5.5 Unary arithmetic operations
5.6 Binary arithmetic operations
5.7 Shifting operations
5.8 Binary bit-wise operations
5.9 Comparisons
5.10 Boolean operations
5.11 Lambdas
5.12 Expression lists
5.13 Evaluation order
5.14 Summary

6. Simple statements
6.1 Expression statements
6.2 Assert statements
6.3 Assignment statements
   6.3.1 Augmented assignment statements
6.4 The pass statement
6.5 The del statement
6.6 The print statement
6.7 The return statement
6.8 The yield statement
6.9 The raise statement
6.10 The break statement
6.11 The continue statement
6.12 The import statement
   6.12.1 Future statements
6.13 The global statement
6.14 The exec statement

7. Compound statements
7.1 The if statement
7.2 The while statement
7.3 The for statement
7.4 The try statement
7.5 The with statement
7.6 Function definitions
7.7 Class definitions

8. Top-level components
8.1 Complete Python programs
8.2 File input
8.3 Interactive input
8.4 Expression input

A. History and License
A.1 History of the software
A.2 Terms and conditions for accessing or otherwise using Python
A.3 Licenses and Acknowledgements for Incorporated Software
  A.3.1 Mersenne Twister
  A.3.2 Sockets
  A.3.3 Floating point exception control
  A.3.4 MD5 message digest algorithm
  A.3.5 Asynchronous socket services
  A.3.6 Cookie management
  A.3.7 Profiling
  A.3.8 Execution tracing
  A.3.9 UUencode and UUdecode functions
  A.3.10 XML Remote Procedure Calls

Index
About this document...
1. Introduction

This reference manual describes the Python programming language. It is not intended as a tutorial.

While I am trying to be as precise as possible, I chose to use English rather than formal specifications for everything except syntax and lexical analysis. This should make the document more understandable to the average reader, but will leave room for ambiguities. Consequently, if you were coming from Mars and tried to re-implement Python from this document alone, you might have to guess things and in fact you would probably end up implementing quite a different language. On the other hand, if you are using Python and wonder what the precise rules about a particular area of the language are, you should definitely be able to find them here. If you would like to see a more formal definition of the language, maybe you could volunteer your time -- or invent a cloning machine :-).

It is dangerous to add too many implementation details to a language reference document -- the implementation may change, and other implementations of the same language may work differently. On the other hand, there is currently only one Python implementation in widespread use (although alternate implementations exist), and its particular quirks are sometimes worth being mentioned, especially where the implementation imposes additional limitations. Therefore, you'll find short ``implementation notes" sprinkled throughout the text.

Every Python implementation comes with a number of built-in and standard modules. These are not documented here, but in the separate Python Library Reference document. A few built-in modules are mentioned when they interact in a significant way with the language definition.

Subsections

- 1.1 Alternate Implementations
- 1.2 Notation
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1.1 Alternate Implementations

Though there is one Python implementation which is by far the most popular, there are some alternate implementations which are of particular interest to different audiences.

Known implementations include:

**CPython**
This is the original and most-maintained implementation of Python, written in C. New language features generally appear here first.

**Jython**
Python implemented in Java. This implementation can be used as a scripting language for Java applications, or can be used to create applications using the Java class libraries. It is also often used to create tests for Java libraries. More information can be found at the Jython website.

**Python for .NET**
This implementation actually uses the CPython implementation, but is a managed .NET application and makes .NET libraries available. This was created by Brian Lloyd. For more information, see the Python for .NET home page.

**IronPython**
An alternate Python for .NET. Unlike Python.NET, this is a complete Python implementation that generates IL, and compiles Python code directly to .NET assemblies. It was created by Jim Hugunin, the original creator of Jython. For more information, see the IronPython website.

**PyPy**
An implementation of Python written in Python; even the bytecode interpreter is written in Python. This is executed using CPython as the underlying interpreter. One of the goals of the project is to encourage experimentation with the language itself by making it easier to modify the interpreter (since it is written in Python). Additional information is available on the PyPy project's home page.
Each of these implementations varies in some way from the language as documented in this manual, or introduces specific information beyond what's covered in the standard Python documentation. Please refer to the implementation-specific documentation to determine what else you need to know about the specific implementation you're using.
1.2 Notation

The descriptions of lexical analysis and syntax use a modified BNF grammar notation. This uses the following style of definition:

```plaintext
name ::= lc_letter (lc_letter | "_")*
lc_letter ::= "a"..."z"
```

The first line says that a `name` is an `lc_letter` followed by a sequence of zero or more `lc_letter`s and underscores. An `lc_letter` in turn is any of the single characters "a" through "z". (This rule is actually adhered to for the names defined in lexical and grammar rules in this document.)

Each rule begins with a name (which is the name defined by the rule) and `::=`. A vertical bar (`|`) is used to separate alternatives; it is the least binding operator in this notation. A star (`*`) means zero or more repetitions of the preceding item; likewise, a plus (`+`) means one or more repetitions, and a phrase enclosed in square brackets (`[ ]`) means zero or one occurrences (in other words, the enclosed phrase is optional). The * and + operators bind as tightly as possible; parentheses are used for grouping. Literal strings are enclosed in quotes. White space is only meaningful to separate tokens. Rules are normally contained on a single line; rules with many alternatives may be formatted alternatively with each line after the first beginning with a vertical bar.

In lexical definitions (as the example above), two more conventions are used: Two literal characters separated by three dots mean a choice of any single character in the given (inclusive) range of ASCII characters. A phrase between angular brackets (`<...>`) gives an informal description of the symbol defined; e.g., this could be used to describe the notion of `control character` if needed.

Even though the notation used is almost the same, there is a big difference between the meaning of lexical and syntactic definitions: a lexical definition operates on the individual characters of the input source, while a syntax definition operates on the stream of tokens generated by the lexical analysis. All uses of BNF in the next chapter (``Lexical Analysis``) are lexical definitions;
uses in subsequent chapters are syntactic definitions.
2. Lexical analysis

A Python program is read by a parser. Input to the parser is a stream of tokens, generated by the lexical analyzer. This chapter describes how the lexical analyzer breaks a file into tokens.

Python uses the 7-bit ASCII character set for program text. New in version 2.3: An encoding declaration can be used to indicate that string literals and comments use an encoding different from ASCII. For compatibility with older versions, Python only warns if it finds 8-bit characters; those warnings should be corrected by either declaring an explicit encoding, or using escape sequences if those bytes are binary data, instead of characters.

The run-time character set depends on the I/O devices connected to the program but is generally a superset of ASCII.

Future compatibility note: It may be tempting to assume that the character set for 8-bit characters is ISO Latin-1 (an ASCII superset that covers most western languages that use the Latin alphabet), but it is possible that in the future Unicode text editors will become common. These generally use the UTF-8 encoding, which is also an ASCII superset, but with very different use for the characters with ordinals 128-255. While there is no consensus on this subject yet, it is unwise to assume either Latin-1 or UTF-8, even though the current implementation appears to favor Latin-1. This applies both to the source character set and the run-time character set.

Subsections

- 2.1 Line structure
  - 2.1.1 Logical lines
  - 2.1.2 Physical lines
  - 2.1.3 Comments
  - 2.1.4 Encoding declarations
  - 2.1.5 Explicit line joining
  - 2.1.6 Implicit line joining
2.1.7 Blank lines
2.1.8 Indentation
2.1.9 Whitespace between tokens

2.2 Other tokens

2.3 Identifiers and keywords
  2.3.1 Keywords
  2.3.2 Reserved classes of identifiers

2.4 Literals
  2.4.1 String literals
  2.4.2 String literal concatenation
  2.4.3 Numeric literals
  2.4.4 Integer and long integer literals
  2.4.5 Floating point literals
  2.4.6 Imaginary literals

2.5 Operators

2.6 Delimiters
2.1 Line structure

A Python program is divided into a number of *logical lines*.
2.1.1 Logical lines

The end of a logical line is represented by the token NEWLINE. Statements cannot cross logical line boundaries except where NEWLINE is allowed by the syntax (e.g., between statements in compound statements). A logical line is constructed from one or more physical lines by following the explicit or implicit line joining rules.
2.1.2 Physical lines

A physical line is a sequence of characters terminated by an end-of-line sequence. In source files, any of the standard platform line termination sequences can be used - the UNIX form using ASCII LF (linefeed), the Windows form using the ASCII sequence CR LF (return followed by linefeed), or the Macintosh form using the ASCII CR (return) character. All of these forms can be used equally, regardless of platform.

When embedding Python, source code strings should be passed to Python APIs using the standard C conventions for newline characters (the \n character, representing ASCII LF, is the line terminator).
2.1.3 Comments

A comment starts with a hash character (#) that is not part of a string literal, and ends at the end of the physical line. A comment signifies the end of the logical line unless the implicit line joining rules are invoked. Comments are ignored by the syntax; they are not tokens.
2.1.4 Encoding declarations

If a comment in the first or second line of the Python script matches the regular expression `coding[=:]\s*([-\w.]+)`, this comment is processed as an encoding declaration; the first group of this expression names the encoding of the source code file. The recommended forms of this expression are

```
# -*- coding: <encoding-name> -*-
```

which is recognized also by GNU Emacs, and

```
# vim:fileencoding=<encoding-name>
```

which is recognized by Bram Moolenaar's VIM. In addition, if the first bytes of the file are the UTF-8 byte-order mark (`\xef\xbb\xbf`), the declared file encoding is UTF-8 (this is supported, among others, by Microsoft's notepad).

If an encoding is declared, the encoding name must be recognized by Python. The encoding is used for all lexical analysis, in particular to find the end of a string, and to interpret the contents of Unicode literals. String literals are converted to Unicode for syntactical analysis, then converted back to their original encoding before interpretation starts. The encoding declaration must appear on a line of its own.
2.1.5 Explicit line joining

Two or more physical lines may be joined into logical lines using backslash characters (\), as follows: when a physical line ends in a backslash that is not part of a string literal or comment, it is joined with the following forming a single logical line, deleting the backslash and the following end-of-line character. For example:

```python
if 1900 < year < 2100 and 1 <= month <= 12 \
and 1 <= day <= 31 and 0 <= hour < 24 \
and 0 <= minute < 60 and 0 <= second < 60:    # Looks like a valid
    return 1
```

A line ending in a backslash cannot carry a comment. A backslash does not continue a comment. A backslash does not continue a token except for string literals (i.e., tokens other than string literals cannot be split across physical lines using a backslash). A backslash is illegal elsewhere on a line outside a string literal.
2.1.6 Implicit line joining

Expressions in parentheses, square brackets or curly braces can be split over more than one physical line without using backslashes. For example:

```python
code
month_names = ['Januari', 'Februari', 'Maart',]  # These are the
       [ 'April', 'Mei', 'Juni',]  # Dutch names
       [ 'Juli', 'Augustus', 'September',]  # for the months
       [ 'Oktober', 'November', 'December']  # of the year
```

Implicitly continued lines can carry comments. The indentation of the continuation lines is not important. Blank continuation lines are allowed. There is no NEWLINE token between implicit continuation lines. Implicitly continued lines can also occur within triple-quoted strings (see below); in that case they cannot carry comments.
2.1.7 Blank lines

A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored (i.e., no NEWLINE token is generated). During interactive input of statements, handling of a blank line may differ depending on the implementation of the read-eval-print loop. In the standard implementation, an entirely blank logical line (i.e. one containing not even whitespace or a comment) terminates a multi-line statement.
2.1.8 Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

First, tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the same rule as used by UNIX). The total number of spaces preceding the first non-blank character then determines the line's indentation. Indentation cannot be split over multiple physical lines using backslashes; the whitespace up to the first backslash determines the indentation.

**Cross-platform compatibility note:** because of the nature of text editors on non-UNIX platforms, it is unwise to use a mixture of spaces and tabs for the indentation in a single source file. It should also be noted that different platforms may explicitly limit the maximum indentation level.

A formfeed character may be present at the start of the line; it will be ignored for the indentation calculations above. Formfeed characters occurring elsewhere in the leading whitespace have an undefined effect (for instance, they may reset the space count to zero).

The indentation levels of consecutive lines are used to generate INDENT and DEDENT tokens, using a stack, as follows.

Before the first line of the file is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one INDENT token is generated. If it is smaller, it **must** be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated. At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.

Here is an example of a correctly (though confusingly) indented piece of Python
code:

```python
def perm(l):
    # Compute the list of all permutations of l
    if len(l) <= 1:
        return [l]
    r = []
    for i in range(len(l)):
        s = l[:i] + l[i+1:]
        p = perm(s)
        for x in p:
            r.append(l[i:i+1] + x)
    return r
```

The following example shows various indentation errors:

```python
def perm(l):
    for i in range(len(l)):
        s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])
        for x in p:
            r.append(l[i:i+1] + x)
    return r
```

(Actually, the first three errors are detected by the parser; only the last error is found by the lexical analyzer -- the indentation of `return r` does not match a level popped off the stack.)
2.1.9 Whitespace between tokens

Except at the beginning of a logical line or in string literals, the whitespace characters space, tab and formfeed can be used interchangeably to separate tokens. Whitespace is needed between two tokens only if their concatenation could otherwise be interpreted as a different token (e.g., ab is one token, but a b is two tokens).
2.2 Other tokens

Besides NEWLINE, INDENT and DEDENT, the following categories of tokens exist: identifiers, keywords, literals, operators, and delimiters. Whitespace characters (other than line terminators, discussed earlier) are not tokens, but serve to delimit tokens. Where ambiguity exists, a token comprises the longest possible string that forms a legal token, when read from left to right.
2.3 Identifiers and keywords

Identifiers (also referred to as names) are described by the following lexical definitions:

```
   identifier ::= (letter|"_") (letter  |  digit
                 |  "_")*

   letter ::= lowercase  |  uppercase

   lowercase ::= "a"..."z"

   uppercase ::= "A"..."Z"

   digit ::= "0"..."9"
```

Identifiers are unlimited in length. Case is significant.
2.3.2 Reserved classes of
2.3.1 Keywords

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

```
and  del  from  not  while
as   elif  global  or  with
assert  else  if  pass  yield
break  except  import  print
class  exec  in  raise
continue  finally  is  return
def  for  lambda  try
```

Changed in version 2.4: `None` became a constant and is now recognized by the compiler as a name for the built-in object `None`. Although it is not a keyword, you cannot assign a different object to it.

Changed in version 2.5: Both `as` and `with` are only recognized when the `with_statement` future feature has been enabled. It will always be enabled in Python 2.6. See section 7.5 for details. Note that using `as` and `with` as identifiers will always issue a warning, even when the `with_statement` future directive is not in effect.

---

*Release 2.5, documentation updated on 19th September, 2006.*

*See [About this document...](#) for information on suggesting changes.*
2.3.2 Reserved classes of identifiers

Certain classes of identifiers (besides keywords) have special meanings. These classes are identified by the patterns of leading and trailing underscore characters:

_ *

Not imported by "from module import *". The special identifier "_" is used in the interactive interpreter to store the result of the last evaluation; it is stored in the __builtin__ module. When not in interactive mode, "_" has no special meaning and is not defined. See section 6.12, ``The import statement."

Note: The name "_" is often used in conjunction with internationalization; refer to the documentation for the gettext module for more information on this convention.

__*_

System-defined names. These names are defined by the interpreter and its implementation (including the standard library); applications should not expect to define additional names using this convention. The set of names of this class defined by Python may be extended in future versions. See section 3.4, ``Special method names."

__*

Class-private names. Names in this category, when used within the context of a class definition, are re-written to use a mangled form to help avoid name clashes between "private" attributes of base and derived classes. See section 5.2.1, ``Identifiers (Names)."
2.4 Literals

Literals are notations for constant values of some built-in types.

Subsections

- 2.4.1 String literals
- 2.4.2 String literal concatenation
- 2.4.3 Numeric literals
- 2.4.4 Integer and long integer literals
- 2.4.5 Floating point literals
- 2.4.6 Imaginary literals
## 2.4.1 String literals

String literals are described by the following lexical definitions:

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>stringliteral</td>
<td>[ [stringprefix] (shortstring</td>
</tr>
<tr>
<td>stringprefix</td>
<td>&quot;r&quot;</td>
</tr>
<tr>
<td>shortstring</td>
<td>&quot;&quot;&quot;&quot; shortstringitem* &quot;&quot;&quot;&quot;</td>
</tr>
<tr>
<td>longstring</td>
<td>&quot;&quot;&quot;&quot; longstringitem* &quot;&quot;&quot;&quot;</td>
</tr>
<tr>
<td>shortstringitem</td>
<td>shortstringchar</td>
</tr>
<tr>
<td>longstringitem</td>
<td>longstringchar</td>
</tr>
<tr>
<td>shortstringchar</td>
<td>&lt;any source character except &quot;&quot; or newline or the quote&gt;</td>
</tr>
<tr>
<td>longstringchar</td>
<td>&lt;any source character except &quot;&quot;&gt;</td>
</tr>
<tr>
<td>escapeseq</td>
<td>&quot;&quot; &lt;any ASCII character&gt;</td>
</tr>
</tbody>
</table>

One syntactic restriction not indicated by these productions is that whitespace is not allowed between the stringprefix and the rest of the string literal. The source character set is defined by the encoding declaration; it is ASCII if no encoding declaration is given in the source file; see section 2.1.4.
In plain English: String literals can be enclosed in matching single quotes (' ) or double quotes (" ). They can also be enclosed in matching groups of three single or double quotes (these are generally referred to as *triple-quoted strings*). The backslash (\) character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character. String literals may optionally be prefixed with a letter "r" or "R"; such strings are called *raw strings* and use different rules for interpreting backslash escape sequences. A prefix of "u" or "U" makes the string a Unicode string. Unicode strings use the Unicode character set as defined by the Unicode Consortium and ISO 10646. Some additional escape sequences, described below, are available in Unicode strings. The two prefix characters may be combined; in this case, "u" must appear before "r".

In triple-quoted strings, unescaped newlines and quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the string. (A "`quote" is the character used to open the string, i.e. either ' or ".)

Unless an "r" or "R" prefix is present, escape sequences in strings are interpreted according to rules similar to those used by Standard C. The recognized escape sequences are:

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>\newline</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>\  \</td>
<td>Backslash ()</td>
<td></td>
</tr>
<tr>
<td>\  '</td>
<td>Single quote (' )</td>
<td></td>
</tr>
<tr>
<td>\  &quot;</td>
<td>Double quote (&quot; )</td>
<td></td>
</tr>
<tr>
<td>\a</td>
<td>ASCII Bell (BEL)</td>
<td></td>
</tr>
<tr>
<td>\b</td>
<td>ASCII Backspace (BS)</td>
<td></td>
</tr>
<tr>
<td>\f</td>
<td>ASCII Formfeed (FF)</td>
<td></td>
</tr>
<tr>
<td>\n</td>
<td>ASCII Linefeed (LF)</td>
<td></td>
</tr>
<tr>
<td>\N{name}</td>
<td>Character named name in the Unicode database (Unicode only)</td>
<td></td>
</tr>
<tr>
<td>\r</td>
<td>ASCII Carriage Return (CR)</td>
<td></td>
</tr>
<tr>
<td>\t</td>
<td>ASCII Horizontal Tab (TAB)</td>
<td></td>
</tr>
<tr>
<td>\uxxxx</td>
<td>Character with 16-bit hex value xxxx (Unicode only)</td>
<td>(1)</td>
</tr>
<tr>
<td>\Uxxxxxxxx</td>
<td>Character with 32-bit hex value xxxxxxxxx (Unicode only)</td>
<td>(2)</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>\v</td>
<td>ASCII Vertical Tab (VT)</td>
<td></td>
</tr>
<tr>
<td>\ooo</td>
<td>Character with octal value ooo</td>
<td>(3,5)</td>
</tr>
<tr>
<td>\xhh</td>
<td>Character with hex value hh</td>
<td>(4,5)</td>
</tr>
</tbody>
</table>

Notes:

(1) Individual code units which form parts of a surrogate pair can be encoded using this escape sequence.

(2) Any Unicode character can be encoded this way, but characters outside the Basic Multilingual Plane (BMP) will be encoded using a surrogate pair if Python is compiled to use 16-bit code units (the default). Individual code units which form parts of a surrogate pair can be encoded using this escape sequence.

(3) As in Standard C, up to three octal digits are accepted.

(4) Unlike in Standard C, at most two hex digits are accepted.

(5) In a string literal, hexadecimal and octal escapes denote the byte with the given value; it is not necessary that the byte encodes a character in the source character set. In a Unicode literal, these escapes denote a Unicode character with the given value.

Unlike Standard C, all unrecognized escape sequences are left in the string unchanged, i.e., the backslash is left in the string. (This behavior is useful when debugging: if an escape sequence is mistyped, the resulting output is more easily recognized as broken.) It is also important to note that the escape sequences marked as ``(Unicode only)`` in the table above fall into the category of unrecognized escapes for non-Unicode string literals.

When an "r" or "R" prefix is present, a character following a backslash is included in the string without change, and all backslashes are left in the string. For example, the string literal r"\n" consists of two characters: a backslash and a lowercase "n". String quotes can be escaped with a backslash, but the
backslash remains in the string; for example, \"\" is a valid string literal consisting of two characters: a backslash and a double quote; \"\ is not a valid string literal (even a raw string cannot end in an odd number of backslashes). Specifically, a raw string cannot end in a single backslash (since the backslash would escape the following quote character). Note also that a single backslash followed by a newline is interpreted as those two characters as part of the string, not as a line continuation.

When an "r" or "R" prefix is used in conjunction with a "u" or "U" prefix, then the \uXXXX and \UXXXXXXXX escape sequences are processed while all other backslashes are left in the string. For example, the string literal ur"\u0062\n" consists of three Unicode characters: `LATIN SMALL LETTER B', `REVERSE SOLIDUS', and `LATIN SMALL LETTER N'. Backslashes can be escaped with a preceding backslash; however, both remain in the string. As a result, \uXXXX escape sequences are only recognized when there are an odd number of backslashes.

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
2.4.2 String literal concatenation

Multiple adjacent string literals (delimited by whitespace), possibly using different quoting conventions, are allowed, and their meaning is the same as their concatenation. Thus, "hello' world' is equivalent to "helloworld". This feature can be used to reduce the number of backslashes needed, to split long strings conveniently across long lines, or even to add comments to parts of strings, for example:

```
re.compile("[A-Za-z_]" # letter or underscore
          "[A-Za-z0-9_]"* # letter, digit or underscore
       )
```

Note that this feature is defined at the syntactical level, but implemented at compile time. The `+' operator must be used to concatenate string expressions at run time. Also note that literal concatenation can use different quoting styles for each component (even mixing raw strings and triple quoted strings).
2.4.3 Numeric literals

There are four types of numeric literals: plain integers, long integers, floating point numbers, and imaginary numbers. There are no complex literals (complex numbers can be formed by adding a real number and an imaginary number).

Note that numeric literals do not include a sign; a phrase like -1 is actually an expression composed of the unary operator `\-\' and the literal 1.
2.4.4 Integer and long integer literals

Integer and long integer literals are described by the following lexical definitions:

```
longinteger ::= integer ("l" | "L")
integer ::= decimalinteger | octinteger | hexinteger
decimalinteger ::= nonzerodigit digit* | "0"
octinteger ::= "0" octdigit+
hexinteger ::= "0" ("x" | "X") hexdigit+
nonzerodigit ::= "1"..."9"
octdigit ::= "0"..."7"
hexdigit ::= digit | "a"..."f" | "A"..."F"
```

Although both lower case "l" and upper case "L" are allowed as suffix for long integers, it is strongly recommended to always use "L", since the letter "l" looks too much like the digit "1".

Plain integer literals that are above the largest representable plain integer (e.g., 2147483647 when using 32-bit arithmetic) are accepted as if they were long integers instead. There is no limit for long integer literals apart from what can be stored in available memory.

Some examples of plain integer literals (first row) and long integer literals (second and third rows):

```
  7     2147483647                      0177
3L    79228162514264337593543950336L  0377L  0x1000000000L
  79228162514264337593543950336       0xdeadbeef
```
In versions of Python prior to 2.4, octal and hexadecimal literals in the range just above the largest representable plain integer but below the largest unsigned 32-bit number (on a machine using 32-bit arithmetic), 4294967296, were taken as the negative plain integer obtained by subtracting 4294967296 from their unsigned value.
## 2.4.5 Floating point literals

Floating point literals are described by the following lexical definitions:

```
floatnumber ::= pointfloat | exponentfloat
pointfloat ::= [intpart] fraction | intpart "." 
exponentfloat ::= (intpart | pointfloat) exponent
intpart ::= digit+
fraction ::= "." digit+
exponent ::= ("e" | "E") ["+" | "-"] digit+
```

Note that the integer and exponent parts of floating point numbers can look like octal integers, but are interpreted using radix 10. For example, "077e010" is legal, and denotes the same number as "77e10". The allowed range of floating point literals is implementation-dependent. Some examples of floating point literals:

- 3.14
- 10.
- .001
- 1e100
- 3.14e-10
- 0e0

Note that numeric literals do not include a sign; a phrase like -1 is actually an expression composed of the unary operator - and the literal 1.
2.4.6 Imaginary literals

Imaginary literals are described by the following lexical definitions:

```
imagnumber ::= (floatnumber | intpart) ("j" | "J")
```

An imaginary literal yields a complex number with a real part of 0.0. Complex numbers are represented as a pair of floating point numbers and have the same restrictions on their range. To create a complex number with a nonzero real part, add a floating point number to it, e.g., (3+4j). Some examples of imaginary literals:

```
3.14j  10.j  10j  .001j  1e100j  3.14e-10j
```

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
2.5 Operators

The following tokens are operators:

```
+  -  *  **  /  //  %
<<  >>  &  |  ^  ~
<  >  <=  >=  ==  !=  <>
```

The comparison operators <> and != are alternate spellings of the same operator. != is the preferred spelling; <> is obsolescent.
2.6 Delimiters

The following tokens serve as delimiters in the grammar:

```
(    )  [    ]  {    }   @
,    :    .    `    =    ;
+=   -=   *=   /=   //=   %=   &+=   |=   ^=   >>=   <<=   **=
```

The period can also occur in floating-point and imaginary literals. A sequence of three periods has a special meaning as an ellipsis in slices. The second half of the list, the augmented assignment operators, serve lexically as delimiters, but also perform an operation.

The following printing ASCII characters have special meaning as part of other tokens or are otherwise significant to the lexical analyzer:

```
'   "   #    \n```

The following printing ASCII characters are not used in Python. Their occurrence outside string literals and comments is an unconditional error:

```
$    ?
```
3. Data model

Subsections

- 3.1 Objects, values and types
- 3.2 The standard type hierarchy
- 3.3 New-style and classic classes
- 3.4 Special method names
  - 3.4.1 Basic customization
  - 3.4.2 Customizing attribute access
    - 3.4.2.1 More attribute access for new-style classes
    - 3.4.2.2 Implementing Descriptors
    - 3.4.2.3 Invoking Descriptors
    - 3.4.2.4 __slots__
  - 3.4.3 Customizing class creation
  - 3.4.4 Emulating callable objects
  - 3.4.5 Emulating container types
  - 3.4.6 Additional methods for emulation of sequence types
  - 3.4.7 Emulating numeric types
  - 3.4.8 Coercion rules
  - 3.4.9 With Statement Context Managers
3.1 Objects, values and types

*Objects* are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a ``stored program computer," code is also represented by objects.)

Every object has an identity, a type and a value. An object's *identity* never changes once it has been created; you may think of it as the object's address in memory. The `is' operator compares the identity of two objects; the `id()' function returns an integer representing its identity (currently implemented as its address). An object's *type* is also unchangeable. An object's type determines the operations that the object supports (e.g., ``does it have a length?'') and also defines the possible values for objects of that type. The `type()' function returns an object's type (which is an object itself). The value of some objects can change. Objects whose value can change are said to be *mutable*; objects whose value is unchangeable once they are created are called *immutable*. (The value of an immutable container object that contains a reference to a mutable object can change when the latter's value is changed; however the container is still considered immutable, because the collection of objects it contains cannot be changed. So, immutability is not strictly the same as having an unchangeable value, it is more subtle.) An object's mutability is determined by its type; for instance, numbers, strings and tuples are immutable, while dictionaries and lists are mutable.

Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected. An implementation is allowed to postpone garbage collection or omit it altogether -- it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable. (Implementation note: the current implementation uses a reference-counting scheme with (optional) delayed detection of cyclically linked garbage, which collects most objects as soon as they become unreachable, but is not guaranteed to collect garbage containing circular references. See the Python Library Reference for information on controlling the collection of cyclic garbage.)

Note that the use of the implementation's tracing or debugging facilities may
keep objects alive that would normally be collectable. Also note that catching an exception with a `try...except' statement may keep objects alive.

Some objects contain references to ``external'' resources such as open files or windows. It is understood that these resources are freed when the object is garbage-collected, but since garbage collection is not guaranteed to happen, such objects also provide an explicit way to release the external resource, usually a `close()' method. Programs are strongly recommended to explicitly close such objects. The `try...finally' statement provides a convenient way to do this.

Some objects contain references to other objects; these are called containers. Examples of containers are tuples, lists and dictionaries. The references are part of a container's value. In most cases, when we talk about the value of a container, we imply the values, not the identities of the contained objects; however, when we talk about the mutability of a container, only the identities of the immediately contained objects are implied. So, if an immutable container (like a tuple) contains a reference to a mutable object, its value changes if that mutable object is changed.

Types affect almost all aspects of object behavior. Even the importance of object identity is affected in some sense: for immutable types, operations that compute new values may actually return a reference to any existing object with the same type and value, while for mutable objects this is not allowed. E.g., after "a = 1; b = 1", a and b may or may not refer to the same object with the value one, depending on the implementation, but after "c = []; d = []", c and d are guaranteed to refer to two different, unique, newly created empty lists. (Note that "c = d = []" assigns the same object to both c and d.)

Footnotes

... unchangeable.

Since Python 2.2, a gradual merging of types and classes has been started that makes this and a few other assertions made in this manual not 100% accurate and complete: for example, it is now possible in some cases to change an object's type, under certain controlled conditions. Until this
manual undergoes extensive revision, it must now be taken as authoritative only regarding `classic classes', that are still the default, for compatibility purposes, in Python 2.2 and 2.3. For more information, see http://www.python.org/doc/newstyle.html.
3.2 The standard type hierarchy

Below is a list of the types that are built into Python. Extension modules (written in C, Java, or other languages, depending on the implementation) can define additional types. Future versions of Python may add types to the type hierarchy (e.g., rational numbers, efficiently stored arrays of integers, etc.).

Some of the type descriptions below contain a paragraph listing `special attributes.' These are attributes that provide access to the implementation and are not intended for general use. Their definition may change in the future.

None
This type has a single value. There is a single object with this value. This object is accessed through the built-in name None. It is used to signify the absence of a value in many situations, e.g., it is returned from functions that don't explicitly return anything. Its truth value is false.

NotImplemented
This type has a single value. There is a single object with this value. This object is accessed through the built-in name NotImplemented. Numeric methods and rich comparison methods may return this value if they do not implement the operation for the operands provided. (The interpreter will then try the reflected operation, or some other fallback, depending on the operator.) Its truth value is true.

Ellipsis
This type has a single value. There is a single object with this value. This object is accessed through the built-in name Ellipsis. It is used to indicate the presence of the "..." syntax in a slice. Its truth value is true.

Numbers
These are created by numeric literals and returned as results by arithmetic operators and arithmetic built-in functions. Numeric objects are immutable; once created their value never changes. Python numbers are of course strongly related to mathematical numbers, but subject to the limitations of numerical representation in computers.
Python distinguishes between integers, floating point numbers, and complex numbers:

**Integers**
These represent elements from the mathematical set of integers (positive and negative).

There are three types of integers:

**Plain integers**
These represent numbers in the range \(-2147483648\) through \(2147483647\). (The range may be larger on machines with a larger natural word size, but not smaller.) When the result of an operation would fall outside this range, the result is normally returned as a long integer (in some cases, the exception `OverflowError` is raised instead). For the purpose of shift and mask operations, integers are assumed to have a binary, 2’s complement notation using 32 or more bits, and hiding no bits from the user (i.e., all \(4294967296\) different bit patterns correspond to different values).

**Long integers**
These represent numbers in an unlimited range, subject to available (virtual) memory only. For the purpose of shift and mask operations, a binary representation is assumed, and negative numbers are represented in a variant of 2’s complement which gives the illusion of an infinite string of sign bits extending to the left.

**Booleans**
These represent the truth values False and True. The two objects representing the values False and True are the only Boolean objects. The Boolean type is a subtype of plain integers, and Boolean values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that when converted to a string, the strings "False" or "True" are returned, respectively.

The rules for integer representation are intended to give the most
meaningful interpretation of shift and mask operations involving negative integers and the least surprises when switching between the plain and long integer domains. Any operation except left shift, if it yields a result in the plain integer domain without causing overflow, will yield the same result in the long integer domain or when using mixed operands.

**Floating point numbers**
These represent machine-level double precision floating point numbers. You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow. Python does not support single-precision floating point numbers; the savings in processor and memory usage that are usually the reason for using these is dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating point numbers.

**Complex numbers**
These represent complex numbers as a pair of machine-level double precision floating point numbers. The same caveats apply as for floating point numbers. The real and imaginary parts of a complex number \( z \) can be retrieved through the read-only attributes \( z.real \) and \( z.imag \).

**Sequences**
These represent finite ordered sets indexed by non-negative numbers. The built-in function `len()` returns the number of items of a sequence. When the length of a sequence is \( n \), the index set contains the numbers 0, 1, ..., \( n-1 \). Item \( i \) of sequence \( a \) is selected by \( a[i] \).

Sequences also support slicing: \( a[i:j] \) selects all items with index \( k \) such that \( i <= k < j \). When used as an expression, a slice is a sequence of the same type. This implies that the index set is renumbered so that it starts at 0.

Some sequences also support ```extended slicing``" with a third ```step``" parameter: \( a[i:j:k] \) selects all items of \( a \) with index \( x \) where \( x = i + n*k \), \( n >= 0 \) and \( i <= x < j \).
Sequences are distinguished according to their mutability:

**Immutable sequences**
An object of an immutable sequence type cannot change once it is created. (If the object contains references to other objects, these other objects may be mutable and may be changed; however, the collection of objects directly referenced by an immutable object cannot change.)

The following types are immutable sequences:

**Strings**
The items of a string are characters. There is no separate character type; a character is represented by a string of one item. Characters represent (at least) 8-bit bytes. The built-in functions `chr()` and `ord()` convert between characters and nonnegative integers representing the byte values. Bytes with the values 0-127 usually represent the corresponding ASCII values, but the interpretation of values is up to the program. The string data type is also used to represent arrays of bytes, e.g., to hold data read from a file.

(On systems whose native character set is not ASCII, strings may use EBCDIC in their internal representation, provided the functions `chr()` and `ord()` implement a mapping between ASCII and EBCDIC, and string comparison preserves the ASCII order. Or perhaps someone can propose a better rule?)

**Unicode**
The items of a Unicode object are Unicode code units. A Unicode code unit is represented by a Unicode object of one item and can hold either a 16-bit or 32-bit value representing a Unicode ordinal (the maximum value for the ordinal is given in `sys.maxunicode`, and depends on how Python is configured at compile time). Surrogate pairs may be present in the Unicode object, and will be reported as two separate items. The built-in functions `unichr()` and `ord()` convert between code units and nonnegative integers representing the Unicode ordinals as defined in the Unicode Standard 3.0. Conversion from and to other encodings are possible through the Unicode method `encode()` and the built-in function `unicode()`.)
**Tuples**

The items of a tuple are arbitrary Python objects. Tuples of two or more items are formed by comma-separated lists of expressions. A tuple of one item (a `singleton`) can be formed by affixing a comma to an expression (an expression by itself does not create a tuple, since parentheses must be usable for grouping of expressions). An empty tuple can be formed by an empty pair of parentheses.

**Mutable sequences**

Mutable sequences can be changed after they are created. The subscription and slicing notations can be used as the target of assignment and `del` (delete) statements.

There is currently a single intrinsic mutable sequence type:

**Lists**

The items of a list are arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets. (Note that there are no special cases needed to form lists of length 0 or 1.)

The extension module `array` provides an additional example of a mutable sequence type.

**Mappings**

These represent finite sets of objects indexed by arbitrary index sets. The subscript notation `a[k]` selects the item indexed by `k` from the mapping `a`; this can be used in expressions and as the target of assignments or `del` statements. The built-in function `len()` returns the number of items in a mapping.

There is currently a single intrinsic mapping type:

**Dictionaries**

These represent finite sets of objects indexed by nearly arbitrary values. The only types of values not acceptable as keys are values containing lists or dictionaries or other mutable types that are compared by value rather than by object identity, the reason being that
the efficient implementation of dictionaries requires a key's hash value
to remain constant. Numeric types used for keys obey the normal rules
for numeric comparison: if two numbers compare equal (e.g., 1 and
1.0) then they can be used interchangeably to index the same
dictionary entry.

Dictionaries are mutable; they can be created by the \{ \ldots \} notation
(see section 5.2.6, "Dictionary Displays").

The extension modules dbm, gdbm, and bsddb provide additional
examples of mapping types.

**Callable types**
These are the types to which the function call operation (see section 5.3.4,
``Calls"") can be applied:

**User-defined functions**
A user-defined function object is created by a function definition (see
section 7.6, "Function definitions"). It should be called with an
argument list containing the same number of items as the function's
formal parameter list.

Special attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
<th>Writable</th>
</tr>
</thead>
<tbody>
<tr>
<td>func_doc</td>
<td>The function's documentation string, or None if unavailable</td>
<td></td>
</tr>
<tr>
<td><strong>doc</strong></td>
<td>Another way of spelling func_doc</td>
<td></td>
</tr>
<tr>
<td>func_name</td>
<td>The function's name</td>
<td></td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>Another way of spelling func_name</td>
<td></td>
</tr>
<tr>
<td><strong>module</strong></td>
<td>The name of the module the function was defined in, or None if unavailable.</td>
<td></td>
</tr>
<tr>
<td>func_defaults</td>
<td>A tuple containing default argument values for those arguments that have defaults, or None if no arguments</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
<td>Access</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td><code>func_code</code></td>
<td>The code object representing the compiled function body.</td>
<td>Writable</td>
</tr>
<tr>
<td><code>func_globals</code></td>
<td>A reference to the dictionary that holds the function's global variables -- the global namespace of the module in which the function was defined.</td>
<td>Read-only</td>
</tr>
<tr>
<td><code>func_dict</code></td>
<td>The namespace supporting arbitrary function attributes.</td>
<td>Writable</td>
</tr>
<tr>
<td><code>func_closure</code></td>
<td>None or a tuple of cells that contain bindings for the function's free variables.</td>
<td>Read-only</td>
</tr>
</tbody>
</table>

Most of the attributes labelled ``Writable`` check the type of the assigned value.

Changed in version 2.4: `func_name` is now writable.

Function objects also support getting and setting arbitrary attributes, which can be used, for example, to attach metadata to functions. Regular attribute dot-notation is used to get and set such attributes. *Note that the current implementation only supports function attributes on user-defined functions. Function attributes on built-in functions may be supported in the future.*

Additional information about a function's definition can be retrieved from its code object; see the description of internal types below.

**User-defined methods**

A user-defined method object combines a class, a class instance (or `None`) and any callable object (normally a user-defined function).

Special read-only attributes: `im_self` is the class instance object, `im_func` is the function object; `im_class` is the class of `im_self` for bound methods or the class that asked for the method for unbound methods; `__doc__` is the method's documentation (same as `im_func.__doc__`); `__name__` is the method name (same as
im_func.__name__); __module__ is the name of the module the method was defined in, or None if unavailable. Changed in version 2.2: im_self used to refer to the class that defined the method.

Methods also support accessing (but not setting) the arbitrary function attributes on the underlying function object.

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if that attribute is a user-defined function object, an unbound user-defined method object, or a class method object. When the attribute is a user-defined method object, a new method object is only created if the class from which it is being retrieved is the same as, or a derived class of, the class stored in the original method object; otherwise, the original method object is used as it is.

When a user-defined method object is created by retrieving a user-defined function object from a class, its im_self attribute is None and the method object is said to be unbound. When one is created by retrieving a user-defined function object from a class via one of its instances, its im_self attribute is the instance, and the method object is said to be bound. In either case, the new method's im_class attribute is the class from which the retrieval takes place, and its im_func attribute is the original function object.

When a user-defined method object is created by retrieving another method object from a class or instance, the behaviour is the same as for a function object, except that the im_func attribute of the new instance is not the original method object but its im_func attribute.

When a user-defined method object is created by retrieving a class method object from a class or instance, its im_self attribute is the class itself (the same as the im_class attribute), and its im_func attribute is the function object underlying the class method.

When an unbound user-defined method object is called, the underlying function (im_func) is called, with the restriction that the first argument must be an instance of the proper class (im_class) or of a
When a bound user-defined method object is called, the underlying function (im_func) is called, inserting the class instance (im_self) in front of the argument list. For instance, when C is a class which contains a definition for a function f(), and x is an instance of C, calling x.f(1) is equivalent to calling C.f(x, 1).

When a user-defined method object is derived from a class method object, the "class instance" stored in im_self will actually be the class itself, so that calling either x.f(1) or C.f(1) is equivalent to calling f(C, 1) where f is the underlying function.

Note that the transformation from function object to (unbound or bound) method object happens each time the attribute is retrieved from the class or instance. In some cases, a fruitful optimization is to assign the attribute to a local variable and call that local variable. Also notice that this transformation only happens for user-defined functions; other callable objects (and all non-callable objects) are retrieved without transformation. It is also important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this only happens when the function is an attribute of the class.

**Generator functions**

A function or method which uses the `yield` statement (see section 6.8, "The `yield` statement") is called a *generator function*. Such a function, when called, always returns an iterator object which can be used to execute the body of the function: calling the iterator's `next()` method will cause the function to execute until it provides a value using the `yield` statement. When the function executes a `return` statement or falls off the end, a `StopIteration` exception is raised and the iterator will have reached the end of the set of values to be returned.

**Built-in functions**

A built-in function object is a wrapper around a C function. Examples of built-in functions are `len()` and `math.sin()` (math is a standard built-in module). The number and type of the arguments are
determined by the C function. Special read-only attributes: __doc__ is the function's documentation string, or None if unavailable; __name__ is the function's name; __self__ is set to None (but see the next item); __module__ is the name of the module the function was defined in or None if unavailable.

Built-in methods
This is really a different disguise of a built-in function, this time containing an object passed to the C function as an implicit extra argument. An example of a built-in method is alist.append(), assuming alist is a list object. In this case, the special read-only attribute __self__ is set to the object denoted by list.

Class Types
Class types, or "new-style classes," are callable. These objects normally act as factories for new instances of themselves, but variations are possible for class types that override __new__( ). The arguments of the call are passed to __new__( ) and, in the typical case, to __init__( ) to initialize the new instance.

Classic Classes
Class objects are described below. When a class object is called, a new class instance (also described below) is created and returned. This implies a call to the class's __init__( ) method if it has one. Any arguments are passed on to the __init__( ) method. If there is no __init__( ) method, the class must be called without arguments.

Class instances
Class instances are described below. Class instances are callable only when the class has a __call__( ) method; x(arguments) is a shorthand for x.__call__(arguments).

Modules
Modules are imported by the import statement (see section 6.12, "The import statement"). A module object has a namespace implemented by a dictionary object (this is the dictionary referenced by the func_globals attribute of functions defined in the module). Attribute references are translated to lookups in this dictionary, e.g., m.x is equivalent to
__dict__['x']. A module object does not contain the code object used to initialize the module (since it isn't needed once the initialization is done).

Attribute assignment updates the module's namespace dictionary, e.g., "m.x = 1" is equivalent to "m.__dict__['x'] = 1".

Special read-only attribute: __dict__ is the module's namespace as a dictionary object.

Predefined (writable) attributes: __name__ is the module's name; __doc__ is the module's documentation string, or None if unavailable; __file__ is the pathname of the file from which the module was loaded, if it was loaded from a file. The __file__ attribute is not present for C modules that are statically linked into the interpreter; for extension modules loaded dynamically from a shared library, it is the pathname of the shared library file.

Classes

Class objects are created by class definitions (see section 7.7, ``Class definitions''). A class has a namespace implemented by a dictionary object. Class attribute references are translated to lookups in this dictionary, e.g., "C.x" is translated to "C.__dict__['x']". When the attribute name is not found there, the attribute search continues in the base classes. The search is depth-first, left-to-right in the order of occurrence in the base class list.

When a class attribute reference (for class C, say) would yield a user-defined function object or an unbound user-defined method object whose associated class is either C or one of its base classes, it is transformed into an unbound user-defined method object whose im_class attribute is C. When it would yield a class method object, it is transformed into a bound user-defined method object whose im_class and im_self attributes are both C. When it would yield a static method object, it is transformed into the object wrapped by the static method object. See section 3.4.2 for another way in which attributes retrieved from a class may differ from those actually contained in its __dict__.

Class attribute assignments update the class's dictionary, never the
dictionary of a base class.

A class object can be called (see above) to yield a class instance (see below).

Special attributes: `__name__` is the class name; `__module__` is the module name in which the class was defined; `__dict__` is the dictionary containing the class's namespace; `__bases__` is a tuple (possibly empty or a singleton) containing the base classes, in the order of their occurrence in the base class list; `__doc__` is the class's documentation string, or None if undefined.

**Class instances**

A class instance is created by calling a class object (see above). A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes. If a class attribute is found that is a user-defined function object or an unbound user-defined method object whose associated class is the class (call it `C`) of the instance for which the attribute reference was initiated or one of its bases, it is transformed into a bound user-defined method object whose `im_class` attribute is `C` and whose `im_self` attribute is the instance. Static method and class method objects are also transformed, as if they had been retrieved from class `C`; see above under ``Classes``. See section 3.4.2 for another way in which attributes of a class retrieved via its instances may differ from the objects actually stored in the class's `__dict__`. If no class attribute is found, and the object's class has a `__getattr__()` method, that is called to satisfy the lookup.

Attribute assignments and deletions update the instance's dictionary, never a class's dictionary. If the class has a `__setattr__()` or `__delattr__()` method, this is called instead of updating the instance dictionary directly.

Class instances can pretend to be numbers, sequences, or mappings if they have methods with certain special names. See section 3.4, ``Special method names.
Special attributes: \_\_dict\_\_ is the attribute dictionary; \_\_class\_\_ is the instance's class.

Files
A file object represents an open file. File objects are created by the open() built-in function, and also by os.popen(), os.fdopen(), and the makefile() method of socket objects (and perhaps by other functions or methods provided by extension modules). The objects sys.stdin, sys.stdout and sys.stderr are initialized to file objects corresponding to the interpreter's standard input, output and error streams. See the Python Library Reference for complete documentation of file objects.

Internal types
A few types used internally by the interpreter are exposed to the user. Their definitions may change with future versions of the interpreter, but they are mentioned here for completeness.

Code objects
Code objects represent byte-compiled executable Python code, or bytecode. The difference between a code object and a function object is that the function object contains an explicit reference to the function's globals (the module in which it was defined), while a code object contains no context; also the default argument values are stored in the function object, not in the code object (because they represent values calculated at run-time). Unlike function objects, code objects are immutable and contain no references (directly or indirectly) to mutable objects.

Special read-only attributes: co\_name gives the function name; co\_argcount is the number of positional arguments (including arguments with default values); co\_nlocals is the number of local variables used by the function (including arguments); co\_varnames is a tuple containing the names of the local variables (starting with the argument names); co\_cellvars is a tuple containing the names of local variables that are referenced by nested functions; co\_freevars is a tuple containing the names of free variables; co\_code is a string representing the sequence of bytecode instructions; co\_consts is a tuple containing the literals used by the
bytecode; **co_names** is a tuple containing the names used by the bytecode; **co_filename** is the filename from which the code was compiled; **co_firstlineno** is the first line number of the function; **co_lnotab** is a string encoding the mapping from byte code offsets to line numbers (for details see the source code of the interpreter); **co_stacksize** is the required stack size (including local variables); **co_flags** is an integer encoding a number of flags for the interpreter.

The following flag bits are defined for **co_flags**: bit \(0x04\) is set if the function uses the "*arguments" syntax to accept an arbitrary number of positional arguments; bit \(0x08\) is set if the function uses the "**keywords" syntax to accept arbitrary keyword arguments; bit \(0x20\) is set if the function is a generator.

Future feature declarations ("from __future__ import division") also use bits in **co_flags** to indicate whether a code object was compiled with a particular feature enabled: bit \(0x2000\) is set if the function was compiled with future division enabled; bits \(0x10\) and \(0x1000\) were used in earlier versions of Python.

Other bits in **co_flags** are reserved for internal use.

If a code object represents a function, the first item in **co_consts** is the documentation string of the function, or **None** if undefined.

**Frame objects**
Frame objects represent execution frames. They may occur in traceback objects (see below).

Special read-only attributes: **f_back** is to the previous stack frame (towards the caller), or **None** if this is the bottom stack frame; **f_code** is the code object being executed in this frame; **f_locals** is the dictionary used to look up local variables; **f_globals** is used for global variables; **f_builtins** is used for built-in (intrinsic) names; **f_restricted** is a flag indicating whether the function is executing in restricted execution mode; **f_lasti** gives the precise instruction (this is an index into the bytecode string of the code
object).

Special writable attributes: `f_trace`, if not `None`, is a function called at the start of each source code line (this is used by the debugger); `f_exc_type`, `f_exc_value`, `f_exc_traceback` represent the last exception raised in the parent frame provided another exception was ever raised in the current frame (in all other cases they are `None`); `f_lineno` is the current line number of the frame -- writing to this from within a trace function jumps to the given line (only for the bottom-most frame). A debugger can implement a Jump command (aka Set Next Statement) by writing to `f_lineno`.

**Traceback objects**

Traceback objects represent a stack trace of an exception. A traceback object is created when an exception occurs. When the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section 7.4, "The try statement.") It is accessible as `sys.exc_traceback`, and also as the third item of the tuple returned by `sys.exc_info()`. The latter is the preferred interface, since it works correctly when the program is using multiple threads. When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream; if the interpreter is interactive, it is also made available to the user as `sys.last_traceback`.

Special read-only attributes: `tb_next` is the next level in the stack trace (towards the frame where the exception occurred), or `None` if there is no next level; `tb_frame` points to the execution frame of the current level; `tb_lineno` gives the line number where the exception occurred; `tb_lasti` indicates the precise instruction. The line number and last instruction in the traceback may differ from the line number of its frame object if the exception occurred in a `try` statement with no matching except clause or with a finally clause.

**Slice objects**

Slice objects are used to represent slices when *extended slice syntax* is used. This is a slice using two colons, or multiple slices or ellipses
separated by commas, e.g., \texttt{a[i:j:step]}, \texttt{a[i:j, k:l]}, or \texttt{a[... , i:j]}. They are also created by the built-in \texttt{slice()} function.

Special read-only attributes: \texttt{start} is the lower bound; \texttt{stop} is the upper bound; \texttt{step} is the step value; each is \texttt{None} if omitted. These attributes can have any type.

Slice objects support one method:

\texttt{indices( self, length)}

This method takes a single integer argument \texttt{length} and computes information about the extended slice that the slice object would describe if applied to a sequence of \texttt{length} items. It returns a tuple of three integers; respectively these are the \texttt{start} and \texttt{stop} indices and the \texttt{step} or stride length of the slice. Missing or out-of-bounds indices are handled in a manner consistent with regular slices. New in version 2.3.

\textbf{Static method objects}

Static method objects provide a way of defeating the transformation of function objects to method objects described above. A static method object is a wrapper around any other object, usually a user-defined method object. When a static method object is retrieved from a class or a class instance, the object actually returned is the wrapped object, which is not subject to any further transformation. Static method objects are not themselves callable, although the objects they wrap usually are. Static method objects are created by the built-in \texttt{staticmethod()} constructor.

\textbf{Class method objects}

A class method object, like a static method object, is a wrapper around another object that alters the way in which that object is retrieved from classes and class instances. The behaviour of class method objects upon such retrieval is described above, under \texttt{``User-defined methods''}. Class method objects are created by the built-in \texttt{classmethod()} constructor.
Release 2.5, documentation updated on 19th September, 2006.  
See About this document... for information on suggesting changes.
3.3 New-style and classic classes

Classes and instances come in two flavors: old-style or classic, and new-style.

Up to Python 2.1, old-style classes were the only flavour available to the user. The concept of (old-style) class is unrelated to the concept of type: if \( x \) is an instance of an old-style class, then \( x.__class__ \) designates the class of \( x \), but \( \text{type}(x) \) is always \(<\text{type} \ '\text{instance}'>\). This reflects the fact that all old-style instances, independently of their class, are implemented with a single built-in type, called \text{instance}.

New-style classes were introduced in Python 2.2 to unify classes and types. A new-style class neither more nor less than a user-defined type. If \( x \) is an instance of a new-style class, then \( \text{type}(x) \) is the same as \( x.__class__ \).

The major motivation for introducing new-style classes is to provide a unified object model with a full meta-model. It also has a number of immediate benefits, like the ability to subclass most built-in types, or the introduction of "descriptors", which enable computed properties.

For compatibility reasons, classes are still old-style by default. New-style classes are created by specifying another new-style class (i.e. a type) as a parent class, or the "top-level type" \text{object} if no other parent is needed. The behaviour of new-style classes differs from that of old-style classes in a number of important details in addition to what \text{type} returns. Some of these changes are fundamental to the new object model, like the way special methods are invoked. Others are "fixes" that could not be implemented before for compatibility concerns, like the method resolution order in case of multiple inheritance.

This manual is not up-to-date with respect to new-style classes. For now, please see \url{http://www.python.org/doc/newstyle.html} for more information.

The plan is to eventually drop old-style classes, leaving only the semantics of new-style classes. This change will probably only be feasible in Python 3.0.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
3.4 Special method names

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python's approach to operator overloading, allowing classes to define their own behavior with respect to language operators. For instance, if a class defines a method named \texttt{__getitem__}(), and \texttt{x} is an instance of this class, then \texttt{x[i]} is equivalent\footnote{3.2} to \texttt{x.__getitem__(i)}. Except where mentioned, attempts to execute an operation raise an exception when no appropriate method is defined.

When implementing a class that emulates any built-in type, it is important that the emulation only be implemented to the degree that it makes sense for the object being modelled. For example, some sequences may work well with retrieval of individual elements, but extracting a slice may not make sense. (One example of this is the \texttt{NodeList} interface in the W3C's Document Object Model.)

Footnotes

... equivalent\footnote{3.2}  
This, and other statements, are only roughly true for instances of new-style classes.

Subsections

- 3.4.1 Basic customization
- 3.4.2 Customizing attribute access
  - 3.4.2.1 More attribute access for new-style classes
  - 3.4.2.2 Implementing Descriptors
  - 3.4.2.3 Invoking Descriptors
3.4.2.4 __slots__

- 3.4.3 Customizing class creation
- 3.4.4 Emulating callable objects
- 3.4.5 Emulating container types
- 3.4.6 Additional methods for emulation of sequence types
- 3.4.7 Emulating numeric types
- 3.4.8 Coercion rules
- 3.4.9 With Statement Context Managers

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
3.4.1 Basic customization

__new__( cls[, ...])

Called to create a new instance of class cls. __new__() is a static method (special-cased so you need not declare it as such) that takes the class of which an instance was requested as its first argument. The remaining arguments are those passed to the object constructor expression (the call to the class). The return value of __new__() should be the new object instance (usually an instance of cls).

Typical implementations create a new instance of the class by invoking the superclass's __new__() method using "super(currentclass, cls).__new__(cls[, ...])" with appropriate arguments and then modifying the newly-created instance as necessary before returning it.

If __new__() returns an instance of cls, then the new instance's __init__() method will be invoked like "__init__(self[, ...])", where self is the new instance and the remaining arguments are the same as were passed to __new__().

If __new__() does not return an instance of cls, then the new instance's __init__() method will not be invoked.

__new__() is intended mainly to allow subclasses of immutable types (like int, str, or tuple) to customize instance creation.

__init__( self[, ...])

Called when the instance is created. The arguments are those passed to the class constructor expression. If a base class has an __init__() method, the derived class's __init__() method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example: "BaseClass.__init__(self, [args...])". As a special constraint on constructors, no value may be returned; doing so will cause a TypeError to be raised at runtime.

__del__( self)
Called when the instance is about to be destroyed. This is also called a
destructor. If a base class has a __del__() method, the derived class's
__del__() method, if any, must explicitly call it to ensure proper
deletion of the base class part of the instance. Note that it is possible
(though not recommended!) for the __del__() method to postpone
destruction of the instance by creating a new reference to it. It may then be
called at a later time when this new reference is deleted. It is not guaranteed
that __del__() methods are called for objects that still exist when the
interpreter exits.

Note: "del x" doesn't directly call x.__del__() -- the
former decrements the reference count for x by one, and the
latter is only called when x's reference count reaches zero.
Some common situations that may prevent the reference
count of an object from going to zero include: circular
references between objects (e.g., a doubly-linked list or a
tree data structure with parent and child pointers); a
reference to the object on the stack frame of a function that
catched an exception (the traceback stored in
sys.exc_traceback keeps the stack frame alive); or a
reference to the object on the stack frame that raised an
unhandled exception in interactive mode (the traceback
stored in sys.last_traceback keeps the stack frame
alive). The first situation can only be remedied by explicitly
breaking the cycles; the latter two situations can be resolved
by storing None in sys.exc_traceback or
sys.last_traceback. Circular references which are
garbage are detected when the option cycle detector is
enabled (it's on by default), but can only be cleaned up if
there are no Python-level __del__() methods involved.
Refer to the documentation for the gc module for more
information about how __del__() methods are handled
by the cycle detector, particularly the description of the
garbage value.
**Warning:** Due to the precarious circumstances under which `__del__()` methods are invoked, exceptions that occur during their execution are ignored, and a warning is printed to `sys.stderr` instead. Also, when `__del__()` is invoked in response to a module being deleted (e.g., when execution of the program is done), other globals referenced by the `__del__()` method may already have been deleted. For this reason, `__del__() `methods should do the absolute minimum needed to maintain external invariants.

Starting with version 1.5, Python guarantees that globals whose name begins with a single underscore are deleted from their module before other globals are deleted; if no other references to such globals exist, this may help in assuring that imported modules are still available at the time when the `__del__() `method is called.

---

**`__repr__(self)`**

Called by the `repr()` built-in function and by string conversions (reverse quotes) to compute the "official" string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment). If this is not possible, a string of the form "<...some useful description...>" should be returned. The return value must be a string object. If a class defines `__repr__() `but not `__str__()`, then `__repr__() `is also used when an "informal" string representation of instances of that class is required.

This is typically used for debugging, so it is important that the representation is information-rich and unambiguous.

---

**`__str__(self)`**

Called by the `str()` built-in function and by the `print` statement to compute the "informal" string representation of an object. This differs from `__repr__() `in that it does not have to be a valid Python expression: a more convenient or concise representation may be used instead. The return value must be a string object.
__lt__(self, other)
__le__(self, other)
__eq__(self, other)
__ne__(self, other)
__gt__(self, other)
__ge__(self, other)

New in version 2.1. These are the so-called ``rich comparison'' methods, and are called for comparison operators in preference to __cmp__() below. The correspondence between operator symbols and method names is as follows: x<y calls x.__lt__(y), x<=y calls x.__le__(y), x==y calls x.__eq__(y), x!=y and x<>y call x.__ne__(y), x>y calls x.__gt__(y), and x>=y calls x.__ge__(y). These methods can return any value, but if the comparison operator is used in a Boolean context, the return value should be interpretable as a Boolean value, else a TypeError will be raised. By convention, False is used for false and True for true.

There are no implied relationships among the comparison operators. The truth of x==y does not imply that x!=y is false. Accordingly, when defining __eq__( ), one should also define __ne__( ) so that the operators will behave as expected.

There are no reflected (swapped-argument) versions of these methods (to be used when the left argument does not support the operation but the right argument does); rather, __lt__( ) and __gt__( ) are each other's reflection, __le__( ) and __ge__( ) are each other's reflection, and __eq__( ) and __ne__( ) are their own reflection.

Arguments to rich comparison methods are never coerced. A rich comparison method may return NotImplemented if it does not implement the operation for a given pair of arguments.

__cmp__(self, other)

Called by comparison operations if rich comparison (see above) is not defined. Should return a negative integer if self < other, zero if self == other, a positive integer if self > other. If no __cmp__( ), __eq__( ) or __ne__( ) operation is defined, class instances are
compared by object identity (``address''). See also the description of
\_\_hash\_\_( ) for some important notes on creating objects which support
custom comparison operations and are usable as dictionary keys. (Note: the
restriction that exceptions are not propagated by \_\_cmp\_\_( ) has been
removed since Python 1.5.)

\_\_rcmp\_\_( self, other)
Changed in version 2.1: No longer supported.

\_\_hash\_\_( self)
Called for the key object for dictionary operations, and by the built-in
function hash( ). Should return a 32-bit integer usable as a hash value for
dictionary operations. The only required property is that objects which
compare equal have the same hash value; it is advised to somehow mix
together (e.g., using exclusive or) the hash values for the components of the
object that also play a part in comparison of objects. If a class does not
define a \_\_cmp\_\_( ) method it should not define a \_\_hash\_\_( )
operation either; if it defines \_\_cmp\_\_( ) or \_\_eq\_\_( ) but not
\_\_hash\_\_( ), its instances will not be usable as dictionary keys. If a class
defines mutable objects and implements a \_\_cmp\_\_( ) or \_\_eq\_\_( )
method, it should not implement \_\_hash\_\_( ), since the dictionary
implementation requires that a key's hash value is immutable (if the object's
hash value changes, it will be in the wrong hash bucket).

Changed in version 2.5: \_\_hash\_\_( ) may now also return a long integer
object; the 32-bit integer is then derived from the hash of that object.

\_\_nonzero\_\_( self)
Called to implement truth value testing, and the built-in operation bool( );
should return False or True, or their integer equivalents 0 or 1. When
this method is not defined, \_\_len\_\_( ) is called, if it is defined (see
below). If a class defines neither \_\_len\_\_( ) nor \_\_nonzero\_\_( ), all
its instances are considered true.

\_\_unicode\_\_( self)
Called to implement unicode( ) builtin; should return a Unicode object.
When this method is not defined, string conversion is attempted, and the
result of string conversion is converted to Unicode using the system default encoding.
3.4.2 Customizing attribute access

The following methods can be defined to customize the meaning of attribute access (use of, assignment to, or deletion of x.name) for class instances.

__getattr__(self, name)
Called when an attribute lookup has not found the attribute in the usual places (i.e. it is not an instance attribute nor is it found in the class tree for self). name is the attribute name. This method should return the (computed) attribute value or raise an AttributeError exception.

Note that if the attribute is found through the normal mechanism, __getattr__() is not called. (This is an intentional asymmetry between __getattr__() and __setattr__().) This is done both for efficiency reasons and because otherwise __setattr__() would have no way to access other attributes of the instance. Note that at least for instance variables, you can fake total control by not inserting any values in the instance attribute dictionary (but instead inserting them in another object). See the __getattribute__() method below for a way to actually get total control in new-style classes.

__setattr__(self, name, value)
Called when an attribute assignment is attempted. This is called instead of the normal mechanism (i.e. store the value in the instance dictionary). name is the attribute name, value is the value to be assigned to it.

If __setattr__() wants to assign to an instance attribute, it should not simply execute "self.name = value" -- this would cause a recursive call to itself. Instead, it should insert the value in the dictionary of instance attributes, e.g., "self.__dict__[name] = value". For new-style classes, rather than accessing the instance dictionary, it should call the base class method with the same name, for example, "object.__setattr__(self, name, value)".

__delattr__(self, name)
Like __setattr__() but for attribute deletion instead of assignment.
This should only be implemented if "\texttt{del \ obj. name}" is meaningful for the object.

---

Subsections

- 3.4.2.1 More attribute access for new-style classes
- 3.4.2.2 Implementing Descriptors
- 3.4.2.3 Invoking Descriptors
- 3.4.2.4 \texttt{__slots__}

---

Release 2.5, documentation updated on 19th September, 2006.
See \texttt{About this document...} for information on suggesting changes.
3.4.3 Customizing class creation

By default, new-style classes are constructed using `type()`. A class definition is read into a separate namespace and the value of class name is bound to the result of `type(name, bases, dict).

When the class definition is read, if `__metaclass__` is defined then the callable assigned to it will be called instead of `type()`. The allows classes or functions to be written which monitor or alter the class creation process:

- Modifying the class dictionary prior to the class being created.
- Returning an instance of another class - essentially performing the role of a factory function.

`__metaclass__`

This variable can be any callable accepting arguments for `name`, `bases`, and `dict`. Upon class creation, the callable is used instead of the built-in `type()`. New in version 2.2.

The appropriate metaclass is determined by the following precedence rules:

- If `dict['__metaclass__']` exists, it is used.
- Otherwise, if there is at least one base class, its metaclass is used (this looks for a `__class__` attribute first and if not found, uses its type).
- Otherwise, if a global variable named `__metaclass__` exists, it is used.
- Otherwise, the old-style, classic metaclass (types.ClassType) is used.

The potential uses for metaclasses are boundless. Some ideas that have been explored including logging, interface checking, automatic delegation, automatic property creation, proxies, frameworks, and automatic resource locking/synchronization.
Emulating callable objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
3.4.4 Emulating callable objects

__call__( self[, args...])

Called when the instance is "called" as a function; if this method is defined, 
x(arg1, arg2, ...) is a shorthand for x.__call__(arg1, 
arg2, ...).
3.4.5 Emulating container types

The following methods can be defined to implement container objects. Containers usually are sequences (such as lists or tuples) or mappings (like dictionaries), but can represent other containers as well. The first set of methods is used either to emulate a sequence or to emulate a mapping; the difference is that for a sequence, the allowable keys should be the integers \( k \) for which \( 0 \leq k < N \) where \( N \) is the length of the sequence, or slice objects, which define a range of items. (For backwards compatibility, the method `__getslice__()` (see below) can also be defined to handle simple, but not extended slices.) It is also recommended that mappings provide the methods `keys()`, `values()`, `items()`, `has_key()`, `get()`, `clear()`, `setdefault()`, `iterkeys()`, `itervalues()`, `iteritems()`, `pop()`, `popitem()`, `copy()`, and `update()` behaving similar to those for Python's standard dictionary objects. The `UserDict` module provides a `DictMixin` class to help create those methods from a base set of `__getitem__()`, `__setitem__()`, `__delitem__()`, and `keys()`. Mutable sequences should provide methods `append()`, `count()`, `index()`, `extend()`, `insert()`, `pop()`, `remove()`, `reverse()` and `sort()`, like Python standard list objects. Finally, sequence types should implement addition (meaning concatenation) and multiplication (meaning repetition) by defining the methods `__add__()`, `__radd__()`, `__iadd__()`, `__mul__()`, `__rmul__()`, and `__imul__()` described below; they should not define `__coerce__()` or other numerical operators. It is recommended that both mappings and sequences implement the `__contains__()` method to allow efficient use of the `in` operator; for mappings, `in` should be equivalent of `has_key()`; for sequences, it should search through the values. It is further recommended that both mappings and sequences implement the `__iter__()` method to allow efficient iteration through the container; for mappings, `__iter__()` should be the same as `iterkeys()`; for sequences, it should iterate through the values.

`__len__(self)`

Called to implement the built-in function `len()`. Should return the length of the object, an integer \( \geq 0 \). Also, an object that doesn't define a `__nonzero__()` method and whose `__len__()` method returns zero is
considered to be false in a Boolean context.

__getitem__(self, key)
Called to implement evaluation of self[key]. For sequence types, the accepted keys should be integers and slice objects. Note that the special interpretation of negative indexes (if the class wishes to emulate a sequence type) is up to the __getitem__() method. If key is of an inappropriate type, TypeError may be raised; if of a value outside the set of indexes for the sequence (after any special interpretation of negative values), IndexError should be raised. For mapping types, if key is missing (not in the container), KeyError should be raised. Note: for loops expect that an IndexError will be raised for illegal indexes to allow proper detection of the end of the sequence.

__setitem__(self, key, value)
Called to implement assignment to self[key]. Same note as for __getitem__(). This should only be implemented for mappings if the objects support changes to the values for keys, or if new keys can be added, or for sequences if elements can be replaced. The same exceptions should be raised for improper key values as for the __getitem__() method.

__delitem__(self, key)
Called to implement deletion of self[key]. Same note as for __getitem__()(). This should only be implemented for mappings if the objects support removal of keys, or for sequences if elements can be removed from the sequence. The same exceptions should be raised for improper key values as for the __getitem__() method.

__iter__(self)
This method is called when an iterator is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container, and should also be made available as the method iterkeys().

Iterator objects also need to implement this method; they are required to return themselves. For more information on iterator objects, see "Iterator Types" in the Python Library Reference.
The membership test operators (in and not in) are normally implemented as an iteration through a sequence. However, container objects can supply the following special method with a more efficient implementation, which also does not require the object be a sequence.

__contains__( self, item)

Called to implement membership test operators. Should return true if item is in self, false otherwise. For mapping objects, this should consider the keys of the mapping rather than the values or the key-item pairs.
3.4.6 Additional methods for emulation of sequence types

The following optional methods can be defined to further emulate sequence objects. Immutable sequences methods should at most only define __getslice__(); mutable sequences might define all three methods.

__getslice__(self, i, j)

*Deprecated since release 2.0.* Support slice objects as parameters to the __getitem__() method.

Called to implement evaluation of self[i:j]. The returned object should be of the same type as self. Note that missing i or j in the slice expression are replaced by zero or sys.maxint, respectively. If negative indexes are used in the slice, the length of the sequence is added to that index. If the instance does not implement the __len__() method, an AttributeError is raised. No guarantee is made that indexes adjusted this way are not still negative. Indexes which are greater than the length of the sequence are not modified. If no __getslice__() is found, a slice object is created instead, and passed to __getitem__() instead.

__setslice__(self, i, j, sequence)

Called to implement assignment to self[i:j]. Same notes for i and j as for __getslice__().

This method is deprecated. If no __setslice__() is found, or for extended slicing of the form self[i:j:k], a slice object is created, and passed to __setitem__(), instead of __setslice__() being called.

__delslice__(self, i, j)

Called to implement deletion of self[i:j]. Same notes for i and j as for __getslice__(). This method is deprecated. If no __delslice__() is found, or for extended slicing of the form self[i:j:k], a slice object is created, and passed to __delitem__(), instead of __delslice__() being called.
Notice that these methods are only invoked when a single slice with a single colon is used, and the slice method is available. For slice operations involving extended slice notation, or in absence of the slice methods, __getitem__(), __setitem__() or __delitem__() is called with a slice object as argument.

The following example demonstrate how to make your program or module compatible with earlier versions of Python (assuming that methods __getitem__(), __setitem__() and __delitem__() support slice objects as arguments):

```python
class MyClass:
    ...
    def __getitem__(self, index):
        ...
    def __setitem__(self, index, value):
        ...
    def __delitem__(self, index):
        ...

    if sys.version_info < (2, 0):
        # They won't be defined if version is at least 2.0 final
        def __getslice__(self, i, j):
            return self[max(0, i):max(0, j):]
        def __setslice__(self, i, j, seq):
            self[max(0, i):max(0, j):] = seq
        def __delslice__(self, i, j):
            del self[max(0, i):max(0, j):]
    ...
```

Note the calls to max(); these are necessary because of the handling of negative indices before the ___*slice__() methods are called. When negative indexes are used, the ___*item__() methods receive them as provided, but the ___*slice__() methods get a ```cooked`` form of the index values. For each negative index value, the length of the sequence is added to the index before calling the method (which may still result in a negative index); this is the customary handling of negative indexes by the built-in sequence types, and the ___*item__() methods are expected to do this as well. However, since they should already be doing that, negative indexes cannot be passed in; they must be constrained to the bounds of the sequence before being passed to the ___*item__() methods. Calling max(0, i) conveniently returns the proper value.
3.4.7 Emulating numeric types

The following methods can be defined to emulate numeric objects. Methods corresponding to operations that are not supported by the particular kind of number implemented (e.g., bitwise operations for non-integral numbers) should be left undefined.

```
__add__(self, other)
__sub__(self, other)
__mul__(self, other)
__floordiv__(self, other)
__mod__(self, other)
__divmod__(self, other)
__pow__(self, other[, modulo])
__lshift__(self, other)
__rshift__(self, other)
__and__(self, other)
__xor__(self, other)
__or__(self, other)
```

These methods are called to implement the binary arithmetic operations (+, -, *, //, %, divmod(), pow(), **, <<, >>, &, ^, |). For instance, to evaluate the expression x+y, where x is an instance of a class that has an __add__() method, x.__add__(y) is called. The __divmod__() method should be the equivalent to using __floordiv__() and __mod__(); it should not be related to __truediv__() (described below). Note that __pow__() should be defined to accept an optional third argument if the ternary version of the built-in pow() function is to be supported.

If one of those methods does not support the operation with the supplied arguments, it should return NotImplemented.

```
__div__(self, other)
```
__truediv__( self, other)
The division operator (/) is implemented by these methods. The
__truediv__( ) method is used when __future__.division is in
effect, otherwise __div__( ) is used. If only one of these two methods is
defined, the object will not support division in the alternate context;
TypeError will be raised instead.

__radd__( self, other)
__rsub__( self, other)
__rmul__( self, other)
__rdiv__( self, other)
__rtruediv__( self, other)
__rfloordiv__( self, other)
__rmod__( self, other)
__rdivmod__( self, other)
__rpow__( self, other)
__rlshift__( self, other)
__rrshift__( self, other)
__rand__( self, other)
__rxor__( self, other)
__ror__( self, other)

These methods are called to implement the binary arithmetic operations (+, -,-, *, /, %, divmod(), pow(), **, <<, >>, & , ^, | ) with reflected
(swapped) operands. These functions are only called if the left operand does
not support the corresponding operation and the operands are of different
types. For instance, to evaluate the expression x - y, where y is an instance
of a class that has an __rsub__( ) method, y . __rsub__( x ) is called if
x . __sub__( y ) returns NotImplemented.

Note that ternary pow( ) will not try calling __rpow__( ) (the coercion
rules would become too complicated).

Note: If the right operand's type is a subclass of the left operand's type and
that subclass provides the reflected method for the operation, this method
will be called before the left operand's non-reflected method. This behavior allows subclasses to override their ancestors' operations.

```python
__iadd__(self, other)
__isub__(self, other)
__imul__(self, other)
__idiv__(self, other)
__itruediv__(self, other)
__ifloordiv__(self, other)
__imod__(self, other)
__ipow__(self, other[, modulo])
__ilshift__(self, other)
__irshift__(self, other)
__iand__(self, other)
__ixor__(self, other)
__ior__(self, other)
```

These methods are called to implement the augmented arithmetic operations (+=, -=, *=, /=, %=, **=, <<=, >=, &=, ^=, |=). These methods should attempt to do the operation in-place (modifying `self`) and return the result (which could be, but does not have to be, `self`). If a specific method is not defined, the augmented operation falls back to the normal methods. For instance, to evaluate the expression `x+=y`, where `x` is an instance of a class that has an `__iadd__()` method, `x.__iadd__(y)` is called. If `x` is an instance of a class that does not define a `__iadd__()` method, `x.__add__(y)` and `y.__radd__(x)` are considered, as with the evaluation of `x+y`.

```python
__neg__(self)
__pos__(self)
__abs__(self)
__invert__(self)
```

Called to implement the unary arithmetic operations (-, +, abs() and ~).
__complex__(self)
__int__(self)
__long__(self)
__float__(self)

Called to implement the built-in functions complex(), int(), long(), and float(). Should return a value of the appropriate type.

__oct__(self)
__hex__(self)

Called to implement the built-in functions oct() and hex(). Should return a string value.

__index__(self)

Called to implement operator.index(). Also called whenever Python needs an integer object (such as in slicing). Must return an integer (int or long). New in version 2.5.

__coerce__(self, other)

Called to implement "mixed-mode" numeric arithmetic. Should either return a 2-tuple containing self and other converted to a common numeric type, or None if conversion is impossible. When the common type would be the type of other, it is sufficient to return None, since the interpreter will also ask the other object to attempt a coercion (but sometimes, if the implementation of the other type cannot be changed, it is useful to do the conversion to the other type here). A return value of NotImplemented is equivalent to returning None.

Footnotes

... types.\textsuperscript{33}

For operands of the same type, it is assumed that if the non-reflected method (such as \_\_\_add\_\_( ) fails the operation is not supported, which is why the reflected method is not called.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
3.4.8 Coercion rules

This section used to document the rules for coercion. As the language has evolved, the coercion rules have become hard to document precisely; documenting what one version of one particular implementation does is undesirable. Instead, here are some informal guidelines regarding coercion. In Python 3.0, coercion will not be supported.

- If the left operand of a % operator is a string or Unicode object, no coercion takes place and the string formatting operation is invoked instead.

- It is no longer recommended to define a coercion operation. Mixed-mode operations on types that don't define coercion pass the original arguments to the operation.

- New-style classes (those derived from object) never invoke the __coerce__() method in response to a binary operator; the only time __coerce__() is invoked is when the built-in function coerce() is called.

- For most intents and purposes, an operator that returns NotImplemented is treated the same as one that is not implemented at all.

- Below, __op__() and __rop__() are used to signify the generic method names corresponding to an operator; __iop__() is used for the corresponding in-place operator. For example, for the operator `+`, __add__() and __radd__() are used for the left and right variant of the binary operator, and __iadd__() for the in-place variant.

- For objects x and y, first x.__op__(y) is tried. If this is not implemented or returns NotImplemented, y.__rop__(x) is tried. If this is also not implemented or returns NotImplemented, a TypeError exception is raised. But see the following exception:

- Exception to the previous item: if the left operand is an instance of a built-in type or a new-style class, and the right operand is an instance of a proper subclass of that type or class and overrides the base's __rop__() method,
the right operand's __rop__( ) method is tried before the left operand's __op__( ) method.

This is done so that a subclass can completely override binary operators. Otherwise, the left operand's __op__( ) method would always accept the right operand: when an instance of a given class is expected, an instance of a subclass of that class is always acceptable.

- When either operand type defines a coercion, this coercion is called before that type's __op__( ) or __rop__( ) method is called, but no sooner. If the coercion returns an object of a different type for the operand whose coercion is invoked, part of the process is redone using the new object.

- When an in-place operator (like `+=`) is used, if the left operand implements __iop__( ), it is invoked without any coercion. When the operation falls back to __op__( ) and/or __rop__( ), the normal coercion rules apply.

- In x+y, if x is a sequence that implements sequence concatenation, sequence concatenation is invoked.

- In x*y, if one operator is a sequence that implements sequence repetition, and the other is an integer (int or long), sequence repetition is invoked.

- Rich comparisons (implemented by methods __eq__( ) and so on) never use coercion. Three-way comparison (implemented by __cmp__( )) does use coercion under the same conditions as other binary operations use it.

- In the current implementation, the built-in numeric types int, long and float do not use coercion; the type complex however does use it. The difference can become apparent when subclassing these types. Over time, the type complex may be fixed to avoid coercion. All these types implement a __coerce__( ) method, for use by the built-in coerce() function.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.


3.4.9 With Statement Context Managers

New in version 2.5.

A context manager is an object that defines the runtime context to be established when executing a `with` statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the `with` statement (described in section 7.5), but can also be used by directly invoking their methods.

Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc.

For more information on context managers, see ``Context Types'' in the Python Library Reference.

``

```enter__(self)

Enter the runtime context related to this object. The `with` statement will bind this method's return value to the target(s) specified in the `as` clause of the statement, if any.

```exit__(self, exc_type, exc_value, traceback)

Exit the runtime context related to this object. The parameters describe the exception that caused the context to be exited. If the context was exited without an exception, all three arguments will be `None`.

If an exception is supplied, and the method wishes to suppress the exception (i.e., prevent it from being propagated), it should return a true value. Otherwise, the exception will be processed normally upon exit from this method.

Note that `__exit__` methods should not reraise the passed-in exception; this is the caller's responsibility.

See Also:
PEP 0343, The "with" statement

The specification, background, and examples for the Python with statement.
4. Execution model

Subsections

- [4.1 Naming and binding](#)
  - [4.1.1 Interaction with dynamic features](#)
- [4.2 Exceptions](#)
4. Execution model

4.1.1 Interaction with dynamic
4.1 Naming and binding

Names refer to objects. Names are introduced by name binding operations. Each occurrence of a name in the program text refers to the binding of that name established in the innermost function block containing the use.

A block is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition. Each command typed interactively is a block. A script file (a file given as standard input to the interpreter or specified on the interpreter command line the first argument) is a code block. A script command (a command specified on the interpreter command line with the `-c` option) is a code block. The file read by the built-in function `execfile()` is a code block. The string argument passed to the built-in function `eval()` and to the `exec` statement is a code block. The expression read and evaluated by the built-in function `input()` is a code block.

A code block is executed in an execution frame. A frame contains some administrative information (used for debugging) and determines where and how execution continues after the code block's execution has completed.

A scope defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name. The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods.

When a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block's environment.

If a name is bound in a block, it is a local variable of that block. If a name is bound at the module level, it is a global variable. (The variables of the module code block are local and global.) If a variable is used in a code block but not defined there, it is a free variable.

When a name is not found at all, a NameError exception is raised. If the name
refers to a local variable that has not been bound, a `UnboundLocalError` exception is raised. `UnboundLocalError` is a subclass of `NameError`.

The following constructs bind names: formal parameters to functions, `import` statements, class and function definitions (these bind the class or function name in the defining block), and targets that are identifiers if occurring in an assignment, `for` loop header, or in the second position of an `except` clause header. The `import` statement of the form ```from ...import *``` binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

A target occurring in a `del` statement is also considered bound for this purpose (though the actual semantics are to unbind the name). It is illegal to unbind a name that is referenced by an enclosing scope; the compiler will report a `SyntaxError`.

Each assignment or `import` statement occurs within a block defined by a class or function definition or at the module level (the top-level code block).

If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. This can lead to errors when a name is used within a block before it is bound. This rule is subtle. Python lacks declarations and allows name binding operations to occur anywhere within a code block. The local variables of a code block can be determined by scanning the entire text of the block for name binding operations.

If the `global` statement occurs within a block, all uses of the name specified in the statement refer to the binding of that name in the top-level namespace. Names are resolved in the top-level namespace by searching the global namespace, i.e. the namespace of the module containing the code block, and the `builtin` namespace, the namespace of the module `__builtin__`. The global namespace is searched first. If the name is not found there, the `builtin` namespace is searched. The `global` statement must precede all uses of the name.

The built-in namespace associated with the execution of a code block is actually found by looking up the name `__builtins__` in its global namespace; this should be a dictionary or a module (in the latter case the module's dictionary is used). By default, when in the `__main__` module, `__builtins__` is the built-in module `__builtin__` (note: no `\`s); when in any other module,
__builtins__ is an alias for the dictionary of the __builtin__ module itself. __builtins__ can be set to a user-created dictionary to create a weak form of restricted execution.

**Note:** Users should not touch __builtins__; it is strictly an implementation detail. Users wanting to override values in the built-in namespace should import the __builtin__ (no `s`) module and modify its attributes appropriately.

The namespace for a module is automatically created the first time a module is imported. The main module for a script is always called __main__.

The global statement has the same scope as a name binding operation in the same block. If the nearest enclosing scope for a free variable contains a global statement, the free variable is treated as a global.

A class definition is an executable statement that may use and define names. These references follow the normal rules for name resolution. The namespace of the class definition becomes the attribute dictionary of the class. Names defined at the class scope are not visible in methods.

---

**Subsections**

- [4.1.1 Interaction with dynamic features](#)

---

*Release 2.5, documentation updated on 19th September, 2006.*

*See [About this document...](#) for information on suggesting changes.*
4.1.1 Interaction with dynamic features

There are several cases where Python statements are illegal when used in conjunction with nested scopes that contain free variables.

If a variable is referenced in an enclosing scope, it is illegal to delete the name. An error will be reported at compile time.

If the wild card form of import -- "import *" -- is used in a function and the function contains or is a nested block with free variables, the compiler will raise a SyntaxError.

If exec is used in a function and the function contains or is a nested block with free variables, the compiler will raise a SyntaxError unless the exec explicitly specifies the local namespace for the exec. (In other words, "exec obj" would be illegal, but "exec obj in ns" would be legal.)

The eval(), execfile(), and input() functions and the exec statement do not have access to the full environment for resolving names. Names may be resolved in the local and global namespaces of the caller. Free variables are not resolved in the nearest enclosing namespace, but in the global namespace.4.1 The exec statement and the eval() and execfile() functions have optional arguments to override the global and local namespace. If only one namespace is specified, it is used for both.

Footnotes

... namespace.4.1

This limitation occurs because the code that is executed by these operations is not available at the time the module is compiled.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.2 Exceptions

Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions. An exception is raised at the point where the error is detected; it may be handled by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred.

The Python interpreter raises an exception when it detects a run-time error (such as division by zero). A Python program can also explicitly raise an exception with the raise statement. Exception handlers are specified with the try ... except statement. The try ... finally statement specifies cleanup code which does not handle the exception, but is executed whether an exception occurred or not in the preceding code.

Python uses the "termination" model of error handling: an exception handler can find out what happened and continue execution at an outer level, but it cannot repair the cause of the error and retry the failing operation (except by re-entering the offending piece of code from the top).

When an exception is not handled at all, the interpreter terminates execution of the program, or returns to its interactive main loop. In either case, it prints a stack backtrace, except when the exception is SystemExit.

Exceptions are identified by class instances. The except clause is selected depending on the class of the instance: it must reference the class of the instance or a base class thereof. The instance can be received by the handler and can carry additional information about the exceptional condition.

Exceptions can also be identified by strings, in which case the except clause is selected by object identity. An arbitrary value can be raised along with the identifying string which can be passed to the handler.

**Deprecated since release 2.5.** String exceptions should not be used in new code. They will not be supported in a future version of Python. Old code should be rewritten to use class exceptions instead.
Warning: Messages to exceptions are not part of the Python API. Their contents may change from one version of Python to the next without warning and should not be relied on by code which will run under multiple versions of the interpreter.

See also the description of the `try` statement in section 7.4 and `raise` statement in section 6.9.
5. Expressions

This chapter explains the meaning of the elements of expressions in Python.

Syntax Notes: In this and the following chapters, extended BNF notation will be used to describe syntax, not lexical analysis. When (one alternative of) a syntax rule has the form

```
name ::= othername
```

and no semantics are given, the semantics of this form of name are the same as for othername.

---

Subsections

- 5.1 Arithmetic conversions
- 5.2 Atoms
  - 5.2.1 Identifiers (Names)
  - 5.2.2 Literals
  - 5.2.3 Parenthesized forms
  - 5.2.4 List displays
  - 5.2.5 Generator expressions
  - 5.2.6 Dictionary displays
  - 5.2.7 String conversions
- 5.3 Primaries
  - 5.3.1 Attribute references
  - 5.3.2 Subscriptions
  - 5.3.3 Slicings
  - 5.3.4 Calls
- 5.4 The power operator
- 5.5 Unary arithmetic operations
- 5.6 Binary arithmetic operations
- 5.7 Shifting operations
- 5.8 Binary bit-wise operations
5.9 Comparisons
5.10 Boolean operations
5.11 Lambdas
5.12 Expression lists
5.13 Evaluation order
5.14 Summary

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.1 Arithmetic conversions

When a description of an arithmetic operator below uses the phrase `the numeric arguments are converted to a common type," the arguments are coerced using the coercion rules listed at 3.4.8. If both arguments are standard numeric types, the following coercions are applied:

- If either argument is a complex number, the other is converted to complex;
- otherwise, if either argument is a floating point number, the other is converted to floating point;
- otherwise, if either argument is a long integer, the other is converted to long integer;
- otherwise, both must be plain integers and no conversion is necessary.

Some additional rules apply for certain operators (e.g., a string left argument to the `%` operator). Extensions can define their own coercions.
5.2 Atoms

Atoms are the most basic elements of expressions. The simplest atoms are identifiers or literals. Forms enclosed in reverse quotes or in parentheses, brackets or braces are also categorized syntactically as atoms. The syntax for atoms is:

```
atom ::= identifier  |  literal  |  enclosure

enclosure ::= parenth_form  |  list_display
             |  generator_expression  |  dict_display
             |  string_conversion
```

Subsections

- 5.2.1 Identifiers (Names)
- 5.2.2 Literals
- 5.2.3 Parenthesized forms
- 5.2.4 List displays
- 5.2.5 Generator expressions
- 5.2.6 Dictionary displays
- 5.2.7 String conversions
5.2.1 Identifiers (Names)

An identifier occurring as an atom is a name. See section 4.1 for documentation of naming and binding.

When the name is bound to an object, evaluation of the atom yields that object. When a name is not bound, an attempt to evaluate it raises a NameError exception.

**Private name mangling:** When an identifier that textually occurs in a class definition begins with two or more underscore characters and does not end in two or more underscores, it is considered a *private name* of that class. Private names are transformed to a longer form before code is generated for them. The transformation inserts the class name in front of the name, with leading underscores removed, and a single underscore inserted in front of the class name. For example, the identifier `__spam` occurring in a class named `Ham` will be transformed to `_Ham__spam`. This transformation is independent of the syntactical context in which the identifier is used. If the transformed name is extremely long (longer than 255 characters), implementation defined truncation may happen. If the class name consists only of underscores, no transformation is done.
5.2.2 Literals

Python supports string literals and various numeric literals:

\[
\text{literal ::= stringliteral | integer | longinteger | floatnumber | imagnumber}
\]

Evaluation of a literal yields an object of the given type (string, integer, long integer, floating point number, complex number) with the given value. The value may be approximated in the case of floating point and imaginary (complex) literals. See section 2.4 for details.

All literals correspond to immutable data types, and hence the object's identity is less important than its value. Multiple evaluations of literals with the same value (either the same occurrence in the program text or a different occurrence) may obtain the same object or a different object with the same value.
5.2.3 Parenthesized forms

A parenthesized form is an optional expression list enclosed in parentheses:

\[
\text{parenth\_form} ::= "(" [expression\_list] ")"
\]

A parenthesized expression list yields whatever that expression list yields: if the list contains at least one comma, it yields a tuple; otherwise, it yields the single expression that makes up the expression list.

An empty pair of parentheses yields an empty tuple object. Since tuples are immutable, the rules for literals apply (i.e., two occurrences of the empty tuple may or may not yield the same object).

Note that tuples are not formed by the parentheses, but rather by use of the comma operator. The exception is the empty tuple, for which parentheses are required -- allowing unparenthesized `nothing" in expressions would cause ambiguities and allow common typos to pass uncaught.
5.2.4 List displays

A list display is a possibly empty series of expressions enclosed in square brackets:

```
test ::= or_test | lambda_form

testlist ::= test ( "," test )* [ "," ]

list_display ::= "[" [listmaker] "]"

listmaker ::= expression ( list_for | ( "," expression )* ["","] )

list_iter ::= list_for | list_if

list_for ::= "for" expression_list "in" testlist [list_iter]

list_if ::= "if" test [list_iter]
```

A list display yields a new list object. Its contents are specified by providing either a list of expressions or a list comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and placed into the list object in that order. When a list comprehension is supplied, it consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. In this case, the elements of the new list are those that would be produced by considering each of the `for` or `if` clauses a block, nesting from left to right, and evaluating the expression to produce a list element each time the innermost block is reached.

Footnotes

... reached 5.1

In Python 2.3, a list comprehension "leaks" the control variables of each "for" it contains into the containing scope. However, this behavior is
deprecated, and relying on it will not work once this bug is fixed in a future release
5.2.5 Generator expressions

A generator expression is a compact generator notation in parentheses:

```
generator_expression ::= "(" test
genexpr_for ")"
genexpr_for ::= "for"
expression_list
"in" test
[genexpr_iter]
genexpr_iter ::= genexpr_for | genexpr_if
genexpr_if ::= "if" test
[genexpr_iter]
```

A generator expression yields a new generator object. It consists of a single expression followed by at least one for clause and zero or more for or if clauses. The iterating values of the new generator are those that would be produced by considering each of the for or if clauses a block, nesting from left to right, and evaluating the expression to yield a value that is reached the innermost block for each iteration.

Variables used in the generator expression are evaluated lazily when the next() method is called for generator object (in the same fashion as normal generators). However, the leftmost for clause is immediately evaluated so that error produced by it can be seen before any other possible error in the code that handles the generator expression. Subsequent for clauses cannot be evaluated immediately since they may depend on the previous for loop. For example: "(x*y for x in range(10) for y in bar(x))".

The parentheses can be omitted on calls with only one argument. See section 5.3.4 for the detail.
5.2.6 Dictionary displays

A dictionary display is a possibly empty series of key/datum pairs enclosed in curly braces:

```
dict_display ::= "{" [key_datum_list] "}"  
key_datum_list ::= key_datum ("," key_datum)* [","]  
key_datum ::= expression ":" expression  
```

A dictionary display yields a new dictionary object.

The key/datum pairs are evaluated from left to right to define the entries of the dictionary: each key object is used as a key into the dictionary to store the corresponding datum.

Restrictions on the types of the key values are listed earlier in section 3.2. (To summarize, the key type should be hashable, which excludes all mutable objects.) Clashes between duplicate keys are not detected; the last datum (textually rightmost in the display) stored for a given key value prevails.
5.2.7 String conversions

A string conversion is an expression list enclosed in reverse (a.k.a. backward) quotes:

```
string_conversion ::= "\"" expression_list "\""
```

Download entire grammar as text.

A string conversion evaluates the contained expression list and converts the resulting object into a string according to rules specific to its type.

If the object is a string, a number, `None`, or a tuple, list or dictionary containing only objects whose type is one of these, the resulting string is a valid Python expression which can be passed to the built-in function `eval()` to yield an expression with the same value (or an approximation, if floating point numbers are involved).

(In particular, converting a string adds quotes around it and converts ``funny'' characters to escape sequences that are safe to print.)

Recursive objects (for example, lists or dictionaries that contain a reference to themselves, directly or indirectly) use "..." to indicate a recursive reference, and the result cannot be passed to `eval()` to get an equal value (`SyntaxError` will be raised instead).

The built-in function `repr()` performs exactly the same conversion in its argument as enclosing it in parentheses and reverse quotes does. The built-in function `str()` performs a similar but more user-friendly conversion.
5.3 Primaries

Primaries represent the most tightly bound operations of the language. Their syntax is:

```
primary ::= atom | attributeref | subscription | slicing | call
```

Download entire grammar as text.

Subsections

- 5.3.1 Attribute references
- 5.3.2 Subscriptions
- 5.3.3 Slicings
- 5.3.4 Calls

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.3.1 Attribute references

An attribute reference is a primary followed by a period and a name:

```
attributeref := primary "." identifier
```

The primary must evaluate to an object of a type that supports attribute references, e.g., a module, list, or an instance. This object is then asked to produce the attribute whose name is the identifier. If this attribute is not available, the exception `AttributeError` is raised. Otherwise, the type and value of the object produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

---

*Previous:* 5.3 Primaries  *Up:* 5.3 Primaries  *Next:* 5.3.2 Subscriptions

*Release 2.5, documentation updated on 19th September, 2006.*

*See* About this document... *for information on suggesting changes.*
5.3.2 Subscriptions

A subscription selects an item of a sequence (string, tuple or list) or mapping (dictionary) object:

```
subscription ::= primary "[" expression_list "]"
```

The primary must evaluate to an object of a sequence or mapping type.

If the primary is a mapping, the expression list must evaluate to an object whose value is one of the keys of the mapping, and the subscription selects the value in the mapping that corresponds to that key. (The expression list is a tuple except if it has exactly one item.)

If the primary is a sequence, the expression (list) must evaluate to a plain integer. If this value is negative, the length of the sequence is added to it (so that, e.g., `x[-1]` selects the last item of `x.`) The resulting value must be a nonnegative integer less than the number of items in the sequence, and the subscription selects the item whose index is that value (counting from zero).

A string's items are characters. A character is not a separate data type but a string of exactly one character.
5.3.3 Slicings

A slicing selects a range of items in a sequence object (e.g., a string, tuple or list). Slicings may be used as expressions or as targets in assignment or `del` statements. The syntax for a slicing:

```
slicing ::= simple_slicing | extended_slicing

simple_slicing ::= primary "[" short_slice "]"

extended_slicing ::= primary "[" slice_list "]"

slice_list ::= slice_item ("," slice_item)* [",",]

slice_item ::= expression | proper_slice | ellipsis

proper_slice ::= short_slice | long_slice

short_slice ::= [lower_bound] ":" [upper_bound]

long_slice ::= short_slice ":" [stride]

lower_bound ::= expression

upper_bound ::= expression

stride ::= expression

ellipsis ::= "..."
```

There is ambiguity in the formal syntax here: anything that looks like an expression list also looks like a slice list, so any subscription can be interpreted as a slicing. Rather than further complicating the syntax, this is disambiguated by defining that in this case the interpretation as a subscription takes priority over the interpretation as a slicing (this is the case if the slice list contains no
proper slice nor ellipses). Similarly, when the slice list has exactly one short slice and no trailing comma, the interpretation as a simple slicing takes priority over that as an extended slicing.

The semantics for a simple slicing are as follows. The primary must evaluate to a sequence object. The lower and upper bound expressions, if present, must evaluate to plain integers; defaults are zero and the \texttt{sys.maxint}, respectively. If either bound is negative, the sequence's length is added to it. The slicing now selects all items with index \( k \) such that \( i \leq k < j \) where \( i \) and \( j \) are the specified lower and upper bounds. This may be an empty sequence. It is not an error if \( i \) or \( j \) lie outside the range of valid indexes (such items don't exist so they aren't selected).

The semantics for an extended slicing are as follows. The primary must evaluate to a mapping object, and it is indexed with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of an ellipsis slice item is the built-in \texttt{Ellipsis} object. The conversion of a proper slice is a slice object (see section 3.2) whose \texttt{start}, \texttt{stop} and \texttt{step} attributes are the values of the expressions given as lower bound, upper bound and stride, respectively, substituting \texttt{None} for missing expressions.
5.3.4 Calls

A call calls a callable object (e.g., a function) with a possibly empty series of arguments:

```
primary "([argument_list [","] | testgenexpr_for] ")"
call ::= primary "(" [argument_list [",",]] ")"
argument_list ::= positional_arguments [","
                  keyword_arguments [","
                                    "*" expression] [","
                                    "**" expression]
                        | keyword_arguments [","
                                        "*" expression]
                        | "*" expression [","
                                 "**" expression]
                        | "**" expression
positional_arguments ::= expression (","
                        expression)*
keyword_arguments ::= keyword_item (","
                        keyword_item)*
keyword_item ::= identifier "=" expression
```

A trailing comma may be present after the positional and keyword arguments but does not affect the semantics.

The primary must evaluate to a callable object (user-defined functions, built-in functions, methods of built-in objects, class objects, methods of class instances,
and certain class instances themselves are callable; extensions may define additional callable object types). All argument expressions are evaluated before the call is attempted. Please refer to section 7.6 for the syntax of formal parameter lists.

If keyword arguments are present, they are first converted to positional arguments, as follows. First, a list of unfilled slots is created for the formal parameters. If there are N positional arguments, they are placed in the first N slots. Next, for each keyword argument, the identifier is used to determine the corresponding slot (if the identifier is the same as the first formal parameter name, the first slot is used, and so on). If the slot is already filled, a TypeError exception is raised. Otherwise, the value of the argument is placed in the slot, filling it (even if the expression is None, it fills the slot). When all arguments have been processed, the slots that are still unfilled are filled with the corresponding default value from the function definition. (Default values are calculated, once, when the function is defined; thus, a mutable object such as a list or dictionary used as default value will be shared by all calls that don't specify an argument value for the corresponding slot; this should usually be avoided.) If there are any unfilled slots for which no default value is specified, a TypeError exception is raised. Otherwise, the list of filled slots is used as the argument list for the call.

If there are more positional arguments than there are formal parameter slots, a TypeError exception is raised, unless a formal parameter using the syntax "*identifier" is present; in this case, that formal parameter receives a tuple containing the excess positional arguments (or an empty tuple if there were no excess positional arguments).

If any keyword argument does not correspond to a formal parameter name, a TypeError exception is raised, unless a formal parameter using the syntax "**identifier" is present; in this case, that formal parameter receives a dictionary containing the excess keyword arguments (using the keywords as keys and the argument values as corresponding values), or a (new) empty dictionary if there were no excess keyword arguments.

If the syntax "*expression" appears in the function call, "expression" must evaluate to a sequence. Elements from this sequence are treated as if they were additional positional arguments; if there are positional arguments \(x_1,\ldots,x_N\), and "expression" evaluates to a sequence \(y_1,\ldots,y_M\), this is equivalent to a
call with M+N positional arguments $x_1,...,x_N,y_1,...,y_M$.

A consequence of this is that although the "*expression" syntax appears after any keyword arguments, it is processed before the keyword arguments (and the "**expression" argument, if any - see below). So:

```python
>>> def f(a, b):
...   print a, b
... >>> f(b=1, *(2,))
  2 1
>>> f(a=1, *(2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
  1 2
```

It is unusual for both keyword arguments and the "*expression" syntax to be used in the same call, so in practice this confusion does not arise.

If the syntax "**expression" appears in the function call, "expression" must evaluate to a (subclass of) dictionary, the contents of which are treated as additional keyword arguments. In the case of a keyword appearing in both "expression" and as an explicit keyword argument, a TypeError exception is raised.

Formal parameters using the syntax "*identifier" or "**identifier" cannot be used as positional argument slots or as keyword argument names. Formal parameters using the syntax "(sublist)" cannot be used as keyword argument names; the outermost sublist corresponds to a single unnamed argument slot, and the argument value is assigned to the sublist using the usual tuple assignment rules after all other parameter processing is done.

A call always returns some value, possibly None, unless it raises an exception. How this value is computed depends on the type of the callable object.

If it is--

**a user-defined function:**

The code block for the function is executed, passing it the argument list. The first thing the code block will do is bind the formal parameters to the
arguments; this is described in section 7.6. When the code block executes a return statement, this specifies the return value of the function call.

a built-in function or method:
The result is up to the interpreter; see the Python Library Reference for the descriptions of built-in functions and methods.

a class object:
A new instance of that class is returned.

a class instance method:
The corresponding user-defined function is called, with an argument list that is one longer than the argument list of the call: the instance becomes the first argument.

a class instance:
The class must define a __call__() method; the effect is then the same as if that method was called.
5.4 The power operator

The power operator binds more tightly than unary operators on its left; it binds less tightly than unary operators on its right. The syntax is:

```
power ::= primary ["**" u_expr]
```

Thus, in an unparenthesized sequence of power and unary operators, the operators are evaluated from right to left (this does not constrain the evaluation order for the operands).

The power operator has the same semantics as the built-in `pow()` function, when called with two arguments: it yields its left argument raised to the power of its right argument. The numeric arguments are first converted to a common type. The result type is that of the arguments after coercion.

With mixed operand types, the coercion rules for binary arithmetic operators apply. For int and long int operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns `100`, but `10**-2` returns `0.01`. (This last feature was added in Python 2.2. In Python 2.1 and before, if both arguments were of integer types and the second argument was negative, an exception was raised).

Raising `0.0` to a negative power results in a `ZeroDivisionError`. Raising a negative number to a fractional power results in a `ValueError`.
5.5 Unary arithmetic operations

All unary arithmetic (and bit-wise) operations have the same priority:

\[
u_{-\text{expr}} ::= \text{power} | \text{"-"} u_{\text{expr}} | \text{"+"} u_{\text{expr}} | \text{"~"} u_{\text{expr}}
\]

Download entire grammar as text.

The unary - (minus) operator yields the negation of its numeric argument.

The unary + (plus) operator yields its numeric argument unchanged.

The unary ~ (invert) operator yields the bit-wise inversion of its plain or long integer argument. The bit-wise inversion of \(x\) is defined as \(- (x+1)\). It only applies to integral numbers.

In all three cases, if the argument does not have the proper type, a TypeError exception is raised.
5.6 Binary arithmetic operations

The binary arithmetic operations have the conventional priority levels. Note that some of these operations also apply to certain non-numeric types. Apart from the power operator, there are only two levels, one for multiplicative operators and one for additive operators:

```
m_expr ::= u_expr | m_expr "*" u_expr | m_expr "//" u_expr | m_expr "/" u_expr | m_expr "%" u_expr

a_expr ::= m_expr | a_expr "+" m_expr | a_expr "-" m_expr
```

The * (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be an integer (plain or long) and the other must be a sequence. In the former case, the numbers are converted to a common type and then multiplied together. In the latter case, sequence repetition is performed; a negative repetition factor yields an empty sequence.

The / (division) and // (floor division) operators yield the quotient of their arguments. The numeric arguments are first converted to a common type. Plain or long integer division yields an integer of the same type; the result is that of mathematical division with the `floor` function applied to the result. Division by zero raises the `ZeroDivisionError` exception.

The % (modulo) operator yields the remainder from the division of the first argument by the second. The numeric arguments are first converted to a common type. A zero right argument raises the `ZeroDivisionError` exception. The arguments may be floating point numbers, e.g., `3.14%0.7` equals `0.34` (since `3.14` equals `4*0.7 + 0.34`). The modulo operator always yields a result with the same sign as its second operand (or zero); the absolute value of the result is strictly smaller than the absolute value of the second operand.
The integer division and modulo operators are connected by the following identity: \( x = (x/y) \times y + (x\%y) \). Integer division and modulo are also connected with the built-in function `divmod()`:

\[
\text{divmod}(x, y) = (x/y, x\%y)
\]

These identities don't hold for floating point numbers; there similar identities hold approximately where \( x/y \) is replaced by \( \text{floor}(x/y) \) or \( \text{floor}(x/y) - 1 \).

In addition to performing the modulo operation on numbers, the % operator is also overloaded by string and unicode objects to perform string formatting (also known as interpolation). The syntax for string formatting is described in the Python Library Reference, section ``Sequence Types''.

**Deprecated since release 2.3.** The floor division operator, the modulo operator, and the `divmod()` function are no longer defined for complex numbers. Instead, convert to a floating point number using the `abs()` function if appropriate.

The + (addition) operator yields the sum of its arguments. The arguments must either both be numbers or both sequences of the same type. In the former case, the numbers are converted to a common type and then added together. In the latter case, the sequences are concatenated.

The - (subtraction) operator yields the difference of its arguments. The numeric arguments are first converted to a common type.

---

**Footnotes**

... operand

While \( \text{abs}(x\%y) < \text{abs}(y) \) is true mathematically, for floats it may not be true numerically due to roundoff. For example, and assuming a platform on which a Python float is an IEEE 754 double-precision number, in order that \(-1e-100 \% 1e100\) have the same sign as \(1e100\), the computed result is \(-1e-100 + 1e100\), which is numerically exactly equal to \(1e100\). Function `fmod()` in the `math` module returns a result whose sign matches the sign of the first argument instead, and so returns \(-1e-100\) in
this case. Which approach is more appropriate depends on the application. 

If $x$ is very close to an exact integer multiple of $y$, it's possible for $\text{floor}(x/y)$ to be one larger than $(x - x\%y)/y$ due to rounding. In such cases, Python returns the latter result, in order to preserve that $\text{divmod}(x, y)[0] * y + x \% y$ be very close to $x$. 

Release 2.5, documentation updated on 19th September, 2006. 
See About this document... for information on suggesting changes.
5.7 Shifting operations

The shifting operations have lower priority than the arithmetic operations:

```
shift_expr ::= a_expr | shift_expr ( "<<" | ">>" ) a_expr
```

These operators accept plain or long integers as arguments. The arguments are converted to a common type. They shift the first argument to the left or right by the number of bits given by the second argument.

A right shift by \( n \) bits is defined as division by \( \text{pow}(2, n) \). A left shift by \( n \) bits is defined as multiplication with \( \text{pow}(2, n) \); for plain integers there is no overflow check so in that case the operation drops bits and flips the sign if the result is not less than \( \text{pow}(2, 31) \) in absolute value. Negative shift counts raise a `ValueError` exception.
5.8 Binary bit-wise operations

Each of the three bitwise operations has a different priority level:

```
and_expr ::= shift_expr | and_expr "&" shift_expr
xor_expr ::= and_expr | xor_expr "^" and_expr
or_expr ::= xor_expr | or_expr "|" xor_expr
```

The & operator yields the bitwise AND of its arguments, which must be plain or long integers. The arguments are converted to a common type.

The ^ operator yields the bitwise XOR (exclusive OR) of its arguments, which must be plain or long integers. The arguments are converted to a common type.

The | operator yields the bitwise (inclusive) OR of its arguments, which must be plain or long integers. The arguments are converted to a common type.
5.9 Comparisons

Unlike C, all comparison operations in Python have the same priority, which is lower than that of any arithmetic, shifting or bitwise operation. Also unlike C, expressions like \( a < b < c \) have the interpretation that is conventional in mathematics:

```
comparison ::= or_expr ( comp_operator
                        or_expr )*

comp_operator ::= 
                
                "<" | ">" | ">=\" | ">=" | 
                "<=\" | "><\" | ">!=\" |
                "is" ["not"] | ["not"]
                "in"
```

Comparisons yield boolean values: \texttt{True} or \texttt{False}.

Comparisons can be chained arbitrarily, e.g., \( x < y <= z \) is equivalent to \( x < y \) and \( y <= z \), except that \( y \) is evaluated only once (but in both cases \( z \) is not evaluated at all when \( x < y \) is found to be false).

Formally, if \( a, b, c, ..., y, z \) are expressions and \( opa, opb, ..., opy \) are comparison operators, then \( a opa b opb c ... y opy z \) is equivalent to \( a opa b \) and \( b opb c \) and ... \( y opy z \), except that each expression is evaluated at most once.

Note that \( a opa b opb c \) doesn't imply any kind of comparison between \( a \) and \( c \), so that, e.g., \( x < y > z \) is perfectly legal (though perhaps not pretty).

The forms \( <> \) and \( != \) are equivalent; for consistency with C, \( != \) is preferred; where \( != \) is mentioned below \( <> \) is also accepted. The \( <> \) spelling is considered obsolescent.

The operators \( <, >, ==, >=, <=, \) and \( != \) compare the values of two objects. The objects need not have the same type. If both are numbers, they are converted to a common type. Otherwise, objects of different types \textit{always} compare unequal, and are ordered consistently but arbitrarily. You can control comparison behavior
of objects of non-builtin types by defining a __cmp__ method or rich comparison methods like __gt__, described in section 3.4.

(This unusual definition of comparison was used to simplify the definition of operations like sorting and the in and not in operators. In the future, the comparison rules for objects of different types are likely to change.)

Comparison of objects of the same type depends on the type:

- Numbers are compared arithmetically.
- Strings are compared lexicographically using the numeric equivalents (the result of the built-in function ord()) of their characters. Unicode and 8-bit strings are fully interoperable in this behavior.
- Tuples and lists are compared lexicographically using comparison of corresponding elements. This means that to compare equal, each element must compare equal and the two sequences must be of the same type and have the same length.

If not equal, the sequences are ordered the same as their first differing elements. For example, cmp([1, 2, x], [1, 2, y]) returns the same as cmp(x, y). If the corresponding element does not exist, the shorter sequence is ordered first (for example, [1, 2] < [1, 2, 3]).

- Mappings (dictionaries) compare equal if and only if their sorted (key, value) lists compare equal. Outcomes other than equality are resolved consistently, but are not otherwise defined.

- Most other objects of builtin types compare unequal unless they are the same object; the choice whether one object is considered smaller or larger than another one is made arbitrarily but consistently within one execution of a program.

The operators in and not in test for set membership. x in s evaluates to true if x is a member of the set s, and false otherwise. x not in s returns the negation of x in s. The set membership test has traditionally been bound to sequences; an object is a member of a set if the set is a sequence and contains an element equal to that object. However, it is possible for an object to support
membership tests without being a sequence. In particular, dictionaries support membership testing as a nicer way of spelling `key in dict`; other mapping types may follow suit.

For the list and tuple types, `x in y` is true if and only if there exists an index `i` such that `x == y[i]` is true.

For the Unicode and string types, `x in y` is true if and only if `x` is a substring of `y`. An equivalent test is `y.find(x) != -1`. Note, `x` and `y` need not be the same type; consequently, `'ab' in 'abc'` will return `True`. Empty strings are always considered to be a substring of any other string, so `'' in ''abc''` will return `True`. Changed in version 2.3: Previously, `x` was required to be a string of length 1.

For user-defined classes which define the `__contains__()` method, `x in y` is true if and only if `y.__contains__(x)` is true.

For user-defined classes which do not define `__contains__()` and do define `__getitem__()`, `x in y` is true if and only if there is a non-negative integer index `i` such that `x == y[i]`, and all lower integer indices do not raise `IndexError` exception. (If any other exception is raised, it is as if `in` raised that exception).

The operator `not in` is defined to have the inverse true value of `in`.

The operators `is` and `is not` test for object identity: `x is y` is true if and only if `x` and `y` are the same object. `x is not y` yields the inverse truth value.

---

### Footnotes

... equal.\(^5.4\)

The implementation computes this efficiently, without constructing lists or sorting.

... defined.\(^5.5\)

Earlier versions of Python used lexicographic comparison of the sorted
(key, value) lists, but this was very expensive for the common case of comparing for equality. An even earlier version of Python compared dictionaries by identity only, but this caused surprises because people expected to be able to test a dictionary for emptiness by comparing it to `{}`.
5.10 Boolean operations

Boolean operations have the lowest priority of all Python operations:

```
expression ::= or_test [if or_test else test] | lambda_form
or_test ::= and_test | or_test "or" and_test
and_test ::= not_test | and_test "and" not_test
not_test ::= comparison | "not" not_test
```

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: `False`, `None`, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true.

The operator `not` yields `True` if its argument is false, `False` otherwise.

The expression `x if C else y` first evaluates `C (not x)`; if `C` is true, `x` is evaluated and its value is returned; otherwise, `y` is evaluated and its value is returned. New in version 2.5.

The expression `x and y` first evaluates `x`; if `x` is false, its value is returned; otherwise, `y` is evaluated and the resulting value is returned.

The expression `x or y` first evaluates `x`; if `x` is true, its value is returned; otherwise, `y` is evaluated and the resulting value is returned.

(Note that neither `and` nor `or` restrict the value and type they return to `False` and `True`, but rather return the last evaluated argument. This is sometimes useful, e.g., if `s` is a string that should be replaced by a default value if it is empty, the expression `s or 'foo'` yields the desired value. Because `not` has to invent a value anyway, it does not bother to return a value of the same type as
its argument, so e.g., not 'foo' yields False, not ''.)
5.11 Lambdas

\[
\text{lambda_form ::= "lambda" [parameter_list]: expression}
\]

Lambda forms (lambda expressions) have the same syntactic position as expressions. They are a shorthand to create anonymous functions; the expression \text{lambda arguments: expression} yields a function object. The unnamed object behaves like a function object defined with

\[
def \text{name(arguments): return expression}
\]

See section 7.6 for the syntax of parameter lists. Note that functions created with lambda forms cannot contain statements.
5.12 Expression lists

expression_list ::= expression ("," expression )* ["","]

An expression list containing at least one comma yields a tuple. The length of the tuple is the number of expressions in the list. The expressions are evaluated from left to right.

The trailing comma is required only to create a single tuple (a.k.a. a singleton); it is optional in all other cases. A single expression without a trailing comma doesn't create a tuple, but rather yields the value of that expression. (To create an empty tuple, use an empty pair of parentheses: ( ).)
5.13 Evaluation order

Python evaluates expressions from left to right. Notice that while evaluating an assignment, the right-hand side is evaluated before the left-hand side.

In the following lines, expressions will be evaluated in the arithmetic order of their suffixes:

```python
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
func(expr1, expr2, *expr3, **expr4)
expr3, expr4 = expr1, expr2
```
5.14 Summary

The following table summarizes the operator precedences in Python, from lowest precedence (least binding) to highest precedence (most binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for comparisons, including tests, which all have the same precedence and chain from left to right -- see section 5.9 - and exponentiation, which groups from right to left).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lambda</td>
<td>Lambda expression</td>
</tr>
<tr>
<td>or</td>
<td>Boolean OR</td>
</tr>
<tr>
<td>and</td>
<td>Boolean AND</td>
</tr>
<tr>
<td>not x</td>
<td>Boolean NOT</td>
</tr>
<tr>
<td>in, not in</td>
<td>Membership tests</td>
</tr>
<tr>
<td>is, is not</td>
<td>Identity tests</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=, &lt;&gt;, !=, ==</td>
<td>Comparisons</td>
</tr>
<tr>
<td></td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;</td>
<td>Shifts</td>
</tr>
<tr>
<td>+, -</td>
<td>Addition and subtraction</td>
</tr>
<tr>
<td>*, /, %</td>
<td>Multiplication, division, remainder</td>
</tr>
<tr>
<td>+x, -x</td>
<td>Positive, negative</td>
</tr>
<tr>
<td>~x</td>
<td>Bitwise not</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>x.attribute</td>
<td>Attribute reference</td>
</tr>
<tr>
<td>x[index]</td>
<td>Subscription</td>
</tr>
<tr>
<td>x[index:index]</td>
<td>Slicing</td>
</tr>
<tr>
<td>f(arguments...)</td>
<td>Function call</td>
</tr>
<tr>
<td>(expressions...)</td>
<td>Binding or tuple display</td>
</tr>
<tr>
<td>[expressions...]</td>
<td>List display</td>
</tr>
</tbody>
</table>
{key: datum ...}  Dictionary display

`expressions ...`  String conversion
6. Simple statements

Simple statements are comprised within a single logical line. Several simple statements may occur on a single line separated by semicolons. The syntax for simple statements is:

```
simple_stmt ::= expression_stmt
               | assert_stmt
               | assignment_stmt
               | augmented_assignment_stmt
               | pass_stmt
               | del_stmt
               | print_stmt
               | return_stmt
               | yield_stmt
               | raise_stmt
               | break_stmt
               | continue_stmt
               | import_stmt
               | global_stmt
               | exec_stmt
```

Subsections

- 6.1 Expression statements
- 6.2 Assert statements
- 6.3 Assignment statements
  - 6.3.1 Augmented assignment statements
- 6.4 The **pass** statement
6.5 The `del` statement
6.6 The `print` statement
6.7 The `return` statement
6.8 The `yield` statement
6.9 The `raise` statement
6.10 The `break` statement
6.11 The `continue` statement
6.12 The `import` statement
   - 6.12.1 Future statements
6.13 The `global` statement
6.14 The `exec` statement
6.1 Expression statements

Expression statements are used (mostly interactively) to compute and write a value, or (usually) to call a procedure (a function that returns no meaningful result; in Python, procedures return the value None). Other uses of expression statements are allowed and occasionally useful. The syntax for an expression statement is:

```
expression_stmt ::= expression_list
```

An expression statement evaluates the expression list (which may be a single expression).

In interactive mode, if the value is not None, it is converted to a string using the built-in `repr()` function and the resulting string is written to standard output (see section 6.6) on a line by itself. (Expression statements yielding None are not written, so that procedure calls do not cause any output.)
6.2 Assert statements

Assert statements are a convenient way to insert debugging assertions into a program:

```python
assert_stmt ::= "assert" expression ["," expression]
```

The simple form, "assert expression", is equivalent to

```python
if __debug__:
    if not expression: raise AssertionError
```

The extended form, "assert expression1, expression2", is equivalent to

```python
if __debug__:
    if not expression1: raise AssertionError, expression2
```

These equivalences assume that __debug__ and AssertionError refer to the built-in variables with those names. In the current implementation, the built-in variable __debug__ is True under normal circumstances, False when optimization is requested (command line option -O). The current code generator emits no code for an assert statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

Assignments to __debug__ are illegal. The value for the built-in variable is determined when the interpreter starts.
Augmented assignment statements
6.3 Assignment statements

Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects:

\[
\text{assignment_stmt} ::= (\text{target_list } "=")+ \text{expression_list}
\]
\[
\text{target_list} ::= \text{target } ("," \text{target})* [","]
\]
\[
\text{target} ::= \text{identifier} \\
| "(" \text{target_list } ")"
| "[" \text{target_list } "]"
| \text{attributeref}
| \text{subscription}
| \text{slicing}
\]

(See section 5.3 for the syntax definitions for the last three symbols.)

An assignment statement evaluates the expression list (remember that this can be a single expression or a comma-separated list, the latter yielding a tuple) and assigns the single resulting object to each of the target lists, from left to right.

Assignment is defined recursively depending on the form of the target (list). When a target is part of a mutable object (an attribute reference, subscription or slicing), the mutable object must ultimately perform the assignment and decide about its validity, and may raise an exception if the assignment is unacceptable. The rules observed by various types and the exceptions raised are given with the definition of the object types (see section 3.2).

Assignment of an object to a target list is recursively defined as follows.

- If the target list is a single target: The object is assigned to that target.
- If the target list is a comma-separated list of targets: The object must be a
sequence with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets. (This rule is relaxed as of Python 1.5; in earlier versions, the object had to be a tuple. Since strings are sequences, an assignment like "a, b = "xy"" is now legal as long as the string has the right length.)

Assignment of an object to a single target is recursively defined as follows.

- If the target is an identifier (name):
  - If the name does not occur in a `global` statement in the current code block: the name is bound to the object in the current local namespace.
  - Otherwise: the name is bound to the object in the current global namespace.

The name is rebound if it was already bound. This may cause the reference count for the object previously bound to the name to reach zero, causing the object to be deallocated and its destructor (if it has one) to be called.

- If the target is a target list enclosed in parentheses or in square brackets:
The object must be a sequence with the same number of items as there are targets in the target list, and its items are assigned, from left to right, to the corresponding targets.

- If the target is an attribute reference: The primary expression in the reference is evaluated. It should yield an object with assignable attributes; if this is not the case, `TypeError` is raised. That object is then asked to assign the assigned object to the given attribute; if it cannot perform the assignment, it raises an exception (usually but not necessarily `AttributeError`).

- If the target is a subscription: The primary expression in the reference is evaluated. It should yield either a mutable sequence object (such as a list) or a mapping object (such as a dictionary). Next, the subscript expression is evaluated.

  If the primary is a mutable sequence object (such as a list), the subscript must yield a plain integer. If it is negative, the sequence's length is added to it. The resulting value must be a nonnegative integer less than the
sequence's length, and the sequence is asked to assign the assigned object to its item with that index. If the index is out of range, \texttt{IndexError} is raised (assignment to a subscripted sequence cannot add new items to a list).

If the primary is a mapping object (such as a dictionary), the subscript must have a type compatible with the mapping's key type, and the mapping is then asked to create a key/datum pair which maps the subscript to the assigned object. This can either replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

- If the target is a slicing: The primary expression in the reference is evaluated. It should yield a mutable sequence object (such as a list). The assigned object should be a sequence object of the same type. Next, the lower and upper bound expressions are evaluated, insofar they are present; defaults are zero and the sequence's length. The bounds should evaluate to (small) integers. If either bound is negative, the sequence's length is added to it. The resulting bounds are clipped to lie between zero and the sequence's length, inclusive. Finally, the sequence object is asked to replace the slice with the items of the assigned sequence. The length of the slice may be different from the length of the assigned sequence, thus changing the length of the target sequence, if the object allows it.

(In the current implementation, the syntax for targets is taken to be the same as for expressions, and invalid syntax is rejected during the code generation phase, causing less detailed error messages.)

\textbf{WARNING:} Although the definition of assignment implies that overlaps between the left-hand side and the right-hand side are `safe' (for example "$a, \ b = \ b, \ a" swaps two variables), overlaps \textit{within} the collection of assigned-to variables are not safe! For instance, the following program prints "$[0, 2]":

\begin{verbatim}
x = [0, 1]
i = 0
i, x[i] = 1, 2
print x
\end{verbatim}
### 6.3.1 Augmented assignment statements

Augmented assignment is the combination, in a single statement, of a binary operation and an assignment statement:

```plaintext
augmented_assignment_stmt ::= target augop expression_list
```

```plaintext
augop ::= "+=" | "-=" | "*=" | "/=" | "%=" | "+=" | "**=" | ">>=" | "<<=" | "&=" | "^=" | "|="
```

(See section [5.3](#) for the syntax definitions for the last three symbols.)

An augmented assignment evaluates the target (which, unlike normal assignment statements, cannot be an unpacking) and the expression list, performs the binary operation specific to the type of assignment on the two operands, and assigns the result to the original target. The target is only evaluated once.

An augmented assignment expression like `x += 1` can be rewritten as `x = x + 1` to achieve a similar, but not exactly equal effect. In the augmented version, `x` is only evaluated once. Also, when possible, the actual operation is performed *in-place*, meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.

With the exception of assigning to tuples and multiple targets in a single statement, the assignment done by augmented assignment statements is handled the same way as normal assignments. Similarly, with the exception of the possible *in-place* behavior, the binary operation performed by augmented assignment is the same as the normal binary operations.

For targets which are attribute references, the initial value is retrieved with a `getattr()` and the result is assigned with a `setattr()`. Notice that the two methods do not necessarily refer to the same variable. When `getattr()` refers
to a class variable, `setattr()` still writes to an instance variable. For example:

```python
class A:
    x = 3    # class variable
a = A()
a.x += 1  # writes a.x as 4 leaving A.x as 3
```
6.4 The pass statement

```
pass_stmt ::= "pass"
```

pass is a null operation -- when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # a class with no methods (yet)
```
6.4 The pass statement

6. Simple statements

6.6 The print statement
6.5 The del statement

```
del_stmt ::= "del" target_list
```

Deletion is recursively defined very similar to the way assignment is defined. Rather than spelling it out in full details, here are some hints.

Deletion of a target list recursively deletes each target, from left to right.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a `global` statement in the same code block. If the name is unbound, a `NameError` exception will be raised.

It is illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.

Deletion of attribute references, subscriptions and slicings is passed to the primary object involved; deletion of a slicing is in general equivalent to assignment of an empty slice of the right type (but even this is determined by the sliced object).
6.6 The print statement

```
print_stmt ::= "print" ( [expression ("," expression)* [","]]
              | ">>" expression [(""," expression)+ [","]]
```

`print` evaluates each expression in turn and writes the resulting object to standard output (see below). If an object is not a string, it is first converted to a string using the rules for string conversions. The (resulting or original) string is then written. A space is written before each object is (converted and) written, unless the output system believes it is positioned at the beginning of a line. This is the case (1) when no characters have yet been written to standard output, (2) when the last character written to standard output is "\n", or (3) when the last write operation on standard output was not a `print` statement. (In some cases it may be functional to write an empty string to standard output for this reason.)

**Note:** Objects which act like file objects but which are not the built-in file objects often do not properly emulate this aspect of the file object's behavior, so it is best not to rely on this.

A "\n" character is written at the end, unless the `print` statement ends with a comma. This is the only action if the statement contains just the keyword `print`.

Standard output is defined as the file object named `stdout` in the built-in module `sys`. If no such object exists, or if it does not have a `write()` method, a `RuntimeError` exception is raised.

`print` also has an extended form, defined by the second portion of the syntax described above. This form is sometimes referred to as ``print chevron." In this form, the first expression after the `>>` must evaluate to a ``file-like" object, specifically an object that has a `write()` method as described above. With this extended form, the subsequent expressions are printed to this file object. If the first expression evaluates to `None`, then `sys.stdout` is used as the file for
output.
6.7 The return statement

```
return_stmt ::= "return" [expression_list]
```

The `return` statement may only occur syntactically nested in a function definition, not within a nested class definition.

If an expression list is present, it is evaluated, else `None` is substituted.

The `return` leaves the current function call with the expression list (or `None`) as return value.

When `return` passes control out of a `try` statement with a `finally` clause, that `finally` clause is executed before really leaving the function.

In a generator function, the `return` statement is not allowed to include an `expression_list`. In that context, a bare `return` indicates that the generator is done and will cause `StopIteration` to be raised.
6.8 The yield statement

yield_stmt ::= "yield" expression_list

The yield statement is only used when defining a generator function, and is only used in the body of the generator function. Using a yield statement in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

When a generator function is called, it returns an iterator known as a generator iterator, or more commonly, a generator. The body of the generator function is executed by calling the generator's `next()` method repeatedly until it raises an exception.

When a yield statement is executed, the state of the generator is frozen and the value of expression_list is returned to `next()`'s caller. By ``frozen'' we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, and the internal evaluation stack: enough information is saved so that the next time `next()` is invoked, the function can proceed exactly as if the yield statement were just another external call.

As of Python version 2.5, the yield statement is now allowed in the try clause of a try ... finally construct. If the generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), the generator-iterator's `close()` method will be called, allowing any pending finally clauses to execute.

**Note:** In Python 2.2, the yield statement is only allowed when the generators feature has been enabled. It will always be enabled in Python 2.3. This `__future__` import statement can be used to enable the feature:

```python
from __future__ import generators
```
See Also:

**PEP 0255, Simple Generators**
The proposal for adding generators and the `yield` statement to Python.

**PEP 0342, Coroutines via Enhanced Generators**
The proposal that, among other generator enhancements, proposed allowing `yield` to appear inside a `try ... finally` block.
6.9 The raise statement

```python
raise_stmt ::= "raise" [expression ["," expression ["," expression]]]
```

If no expressions are present, `raise` re-raises the last exception that was active in the current scope. If no exception is active in the current scope, a `TypeError` exception is raised indicating that this is an error (if running under IDLE, a `Queue.Empty` exception is raised instead).

Otherwise, `raise` evaluates the expressions to get three objects, using `None` as the value of omitted expressions. The first two objects are used to determine the `type` and `value` of the exception.

If the first object is an instance, the type of the exception is the class of the instance, the instance itself is the value, and the second object must be `None`.

If the first object is a class, it becomes the type of the exception. The second object is used to determine the exception value: If it is an instance of the class, the instance becomes the exception value. If the second object is a tuple, it is used as the argument list for the class constructor; if it is `None`, an empty argument list is used, and any other object is treated as a single argument to the constructor. The instance so created by calling the constructor is used as the exception value.

If a third object is present and not `None`, it must be a traceback object (see section 3.2), and it is substituted instead of the current location as the place where the exception occurred. If the third object is present and not a traceback object or `None`, a `TypeError` exception is raised. The three-expression form of `raise` is useful to re-raise an exception transparently in an except clause, but `raise` with no expressions should be preferred if the exception to be re-raised was the most recently active exception in the current scope.

Additional information on exceptions can be found in section 4.2, and information about handling exceptions is in section 7.4.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
6.10 The break statement

`break_stmt ::= "break"
``

`break` may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition within that loop.

It terminates the nearest enclosing loop, skipping the optional `else` clause if the loop has one.

If a `for` loop is terminated by `break`, the loop control target keeps its current value.

When `break` passes control out of a `try` statement with a `finally` clause, that `finally` clause is executed before really leaving the loop.
6.11 The continue statement

\[ \text{continue_stmt ::= "continue"} \]

```
continue may only occur syntactically nested in a for or while loop, but
not nested in a function or class definition or finally statement within that
loop. It continues with the next cycle of the nearest enclosing loop.
```

Footnotes

... loop. It may occur within an except or else clause. The restriction on
occuring in the try clause is implementor's laziness and will eventually be
lifted.
6.12 The import statement

Import statements are executed in two steps: (1) find a module, and initialize it if necessary; (2) define a name or names in the local namespace (of the scope where the import statement occurs). The first form (without from) repeats these steps for each identifier in the list. The form with from performs step (1) once, and then performs step (2) repeatedly.

In this context, to `initialize` a built-in or extension module means to call an initialization function that the module must provide for the purpose (in the reference implementation, the function's name is obtained by prepending string `init` to the module's name); to `initialize` a Python-coded module means to execute the module's body.

The system maintains a table of modules that have been or are being initialized, indexed by module name. This table is accessible as sys.modules. When a module name is found in this table, step (1) is finished. If not, a search for a module definition is started. When a module is found, it is loaded. Details of the module searching and loading process are implementation and platform specific. It generally involves searching for a `built-in` module with the given name and then searching a list of locations given as sys.path.
If a built-in module is found, its built-in initialization code is executed and step (1) is finished. If no matching file is found, ImportError is raised. If a file is found, it is parsed, yielding an executable code block. If a syntax error occurs, SyntaxError is raised. Otherwise, an empty module of the given name is created and inserted in the module table, and then the code block is executed in the context of this module. Exceptions during this execution terminate step (1).

When step (1) finishes without raising an exception, step (2) can begin.

The first form of import statement binds the module name in the local namespace to the module object, and then goes on to import the next identifier, if any. If the module name is followed by as, the name following as is used as the local name for the module.

The from form does not bind the module name: it goes through the list of identifiers, looks each one of them up in the module found in step (1), and binds the name in the local namespace to the object thus found. As with the first form of import, an alternate local name can be supplied by specifying "as localname". If a name is not found, ImportError is raised. If the list of identifiers is replaced by a star ("*"), all public names defined in the module are bound in the local namespace of the import statement..

The public names defined by a module are determined by checking the module's namespace for a variable named __all__; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in __all__ are all considered public and are required to exist. If __all__ is not defined, the set of public names includes all names found in the module's namespace which do not begin with an underscore character ("_"). __all__ should contain the entire public API. It is intended to avoid accidentally exporting items that are not part of the API (such as library modules which were imported and used within the module).

The from form with "*" may only occur in a module scope. If the wild card form of import -- "import *" -- is used in a function and the function contains or is a nested block with free variables, the compiler will raise a SyntaxError.

Hierarchical module names: when the module names contains one or more dots, the module search path is carried out differently. The sequence of
identifiers up to the last dot is used to find a ``package''; the final identifier is then searched inside the package. A package is generally a subdirectory of a directory on sys.path that has a file __init__.py. [XXX Can't be bothered to spell this out right now; see the URL http://www.python.org/doc/essays/packages.html for more details, also about how the module search works from inside a package.]

The built-in function __import__( ) is provided to support applications that determine which modules need to be loaded dynamically; refer to Built-in Functions in the Python Library Reference for additional information.

Subsections

- 6.12.1 Future statements

Python Reference Manual


Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
6.12.1 Future statements

A future statement is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python. The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

```
future_statement ::=  "from" "__future__" "import" feature ["as" name] ("," feature ["as" name])* 
| "from" "__future__" "import" "(" feature ["as" name] ("," feature ["as" name])* [",",] ")"
feature ::= identifier
name ::= identifier
```

A future statement must appear near the top of the module. The only lines that can appear before a future statement are:

- the module docstring (if any),
- comments,
- blank lines, and
- other future statements.

The features recognized by Python 2.3 are "generators", "division" and "nested_scopes". "generators" and "nested_scopes" are redundant in 2.3 because they are always enabled.

A future statement is recognized and treated specially at compile time: Changes to the semantics of core constructs are often implemented by generating different code. It may even be the case that a new feature introduces new incompatible
syntax (such as a new reserved word), in which case the compiler may need to parse the module differently. Such decisions cannot be pushed off until runtime.

For any given release, the compiler knows which feature names have been defined, and raises a compile-time error if a future statement contains a feature not known to it.

The direct runtime semantics are the same as for any import statement: there is a standard module `__future__`, described later, and it will be imported in the usual way at the time the future statement is executed.

The interesting runtime semantics depend on the specific feature enabled by the future statement.

Note that there is nothing special about the statement:

```python
import __future__ [as name]
```

That is not a future statement; it's an ordinary import statement with no special semantics or syntax restrictions.

Code compiled by an `exec` statement or calls to the built-in functions `compile()` and `execfile()` that occur in a module M containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can, starting with Python 2.2 be controlled by optional arguments to `compile()` -- see the documentation of that function in the *Python Library Reference* for details.

A future statement typed at an interactive interpreter prompt will take effect for the rest of the interpreter session. If an interpreter is started with the `-i` option, is passed a script name to execute, and the script includes a future statement, it will be in effect in the interactive session started after the script is executed.
6.13 The global statement

```
global_stmt ::= "global" identifier ("," identifier)*
```

The `global` statement is a declaration which holds for the entire current code block. It means that the listed identifiers are to be interpreted as globals. It would be impossible to assign to a global variable without `global`, although free variables may refer to globals without being declared global.

Names listed in a `global` statement must not be used in the same code block textually preceding that `global` statement.

Names listed in a `global` statement must not be defined as formal parameters or in a `for` loop control target, `class` definition, function definition, or `import` statement.

(The current implementation does not enforce the latter two restrictions, but programs should not abuse this freedom, as future implementations may enforce them or silently change the meaning of the program.)

**Programmer's note:** the `global` is a directive to the parser. It applies only to code parsed at the same time as the `global` statement. In particular, a `global` statement contained in an `exec` statement does not affect the code block containing the `exec` statement, and code contained in an `exec` statement is unaffected by `global` statements in the code containing the `exec` statement. The same applies to the `eval()`, `execfile()` and `compile()` functions.
This statement supports dynamic execution of Python code. The first expression should evaluate to either a string, an open file object, or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs). If it is an open file, the file is parsed until EOF and executed. If it is a code object, it is simply executed. In all cases, the code that's executed is expected to be valid as file input (see section 8.2, ``File input''). Be aware that the return and yield statements may not be used outside of function definitions even within the context of code passed to the exec statement.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only the first expression after in is specified, it should be a dictionary, which will be used for both the global and the local variables. If two expressions are given, they are used for the global and local variables, respectively. If provided, locals can be any mapping object. Changed in version 2.4: formerly locals was required to be a dictionary.

As a side effect, an implementation may insert additional keys into the dictionaries given besides those corresponding to variable names set by the executed code. For example, the current implementation may add a reference to the dictionary of the built-in module __builtin__ under the key __builtins__(!).

**Programmer's hints:** dynamic evaluation of expressions is supported by the built-in function eval(). The built-in functions globals() and locals() return the current global and local dictionary, respectively, which may be useful to pass around for use by exec.
7. Compound statements

Compound statements contain (groups of) other statements; they affect or control the execution of those other statements in some way. In general, compound statements span multiple lines, although in simple incarnations a whole compound statement may be contained in one line.

The `if`, `while` and `for` statements implement traditional control flow constructs. `try` specifies exception handlers and/or cleanup code for a group of statements. Function and class definitions are also syntactically compound statements.

Compound statements consist of one or more `clauses.' A clause consists of a header and a `suite.' The clause headers of a particular compound statement are all at the same indentation level. Each clause header begins with a uniquely identifying keyword and ends with a colon. A suite is a group of statements controlled by a clause. A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header's colon, or it can be one or more indented statements on subsequent lines. Only the latter form of suite can contain nested compound statements; the following is illegal, mostly because it wouldn't be clear to which `if` clause a following `else` clause would belong:

```
if test1: if test2: print x
```

Also note that the semicolon binds tighter than the colon in this context, so that in the following example, either all or none of the `print` statements are executed:

```
if x < y < z: print x; print y; print z
```

Summarizing:

```
compound_stmt ::= if_stmt
    | while_stmt
    | for_stmt
    | try_stmt
```
Note that statements always end in a `NEWLINE` possibly followed by a `DEDENT`. Also note that optional continuation clauses always begin with a keyword that cannot start a statement, thus there are no ambiguities (the `dangling else` problem is solved in Python by requiring nested `if` statements to be indented).

The formatting of the grammar rules in the following sections places each clause on a separate line for clarity.

**Subsections**

- 7.1 The **if** statement
- 7.2 The **while** statement
- 7.3 The **for** statement
- 7.4 The **try** statement
- 7.5 The **with** statement
- 7.6 Function definitions
- 7.7 Class definitions

---

Previous: 6.14 The **exec** statement  Up: Python Reference Manual  Next: 7.1 The **if** statement
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
The while statement
7.1 The if statement

The if statement is used for conditional execution:

```
if_stmt ::= "if" expression ":" suite
            ( "elif" expression ":" suite )*
            ["else" ":" suite]
```

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section 5.10 for the definition of true and false); then that suite is executed (and no other part of the if statement is executed or evaluated). If all expressions are false, the suite of the else clause, if present, is executed.
7.2 The while statement

The `while` statement is used for repeated execution as long as an expression is true:

```
while_stmt ::= "while" expression "::=
            suite
            ["else" ::=" suite]
```

This repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the `else` clause, if present, is executed and the loop terminates.

A **break** statement executed in the first suite terminates the loop without executing the `else` clause's suite. A **continue** statement executed in the first suite skips the rest of the suite and goes back to testing the expression.
7.3 The for statement

The for statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

```
for_stmt ::=  "for" target_list "in"
expression_list ":" suite
["else" ":" suite]
```

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the expression_list. The suite is then executed once for each item provided by the iterator, in the order of ascending indices. Each item in turn is assigned to the target list using the standard rules for assignments, and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty), the suite in the else clause, if present, is executed, and the loop terminates.

A break statement executed in the first suite terminates the loop without executing the else clause's suite. A continue statement executed in the first suite skips the rest of the suite and continues with the next item, or with the else clause if there was no next item.

The suite may assign to the variable(s) in the target list; this does not affect the next item assigned to it.

The target list is not deleted when the loop is finished, but if the sequence is empty, it will not have been assigned to at all by the loop. Hint: the built-in function range() returns a sequence of integers suitable to emulate the effect of Pascal's for i := a to b do; e.g., range(3) returns the list [0, 1, 2].

**Warning:** There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, i.e. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop
terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop. This can lead to nasty bugs that can be avoided by making a temporary copy using a slice of the whole sequence, e.g.,

```python
for x in a[:]
    if x < 0: a.remove(x)
```
7.4 The `try` statement

The `try` statement specifies exception handlers and/or cleanup code for a group of statements:

```
try_stmt ::= try1_stmt | try2_stmt
try1_stmt ::= "try" ":" suite
             ("except" [expression ["," target]] ":" suite)+
             ["else" ":" suite]
             ["finally" ":" suite]
try2_stmt ::= "try" ":" suite
             "finally" ":" suite
```

Download entire grammar as text.

Changed in version 2.5: In previous versions of Python, `try...except...finally` did not work. `try...except` had to be nested in `try...finally`.

The `except` clause(s) specify one or more exception handlers. When no exception occurs in the `try` clause, no exception handler is executed. When an exception occurs in the `try` suite, a search for an exception handler is started. This search inspects the `except` clauses in turn until one is found that matches the exception. An expression-less `except` clause, if present, must be last; it matches any exception. For an `except` clause with an expression, that expression is evaluated, and the clause matches the exception if the resulting object is `"compatible"` with the exception. An object is compatible with an exception if it is the class or a base class of the exception object, a tuple containing an item compatible with the exception, or, in the (deprecated) case of string exceptions, is the raised string itself (note that the object identities must match, i.e. it must be the same string object, not just a string with the same value).

If no `except` clause matches the exception, the search for an exception handler continues in the surrounding code and on the invocation stack.
If the evaluation of an expression in the header of an except clause raises an exception, the original search for a handler is canceled and a search starts for the new exception in the surrounding code and on the call stack (it is treated as if the entire try statement raised the exception).

When a matching except clause is found, the exception is assigned to the target specified in that except clause, if present, and the except clause's suite is executed. All except clauses must have an executable block. When the end of this block is reached, execution continues normally after the entire try statement. (This means that if two nested handlers exist for the same exception, and the exception occurs in the try clause of the inner handler, the outer handler will not handle the exception.)

Before an except clause's suite is executed, details about the exception are assigned to three variables in the sys module: `sys.exc_type` receives the object identifying the exception; `sys.exc_value` receives the exception's parameter; `sys.exc_traceback` receives a traceback object (see section 3.2) identifying the point in the program where the exception occurred. These details are also available through the `sys.exc_info()` function, which returns a tuple `(exc_type, exc_value, exc_traceback)`. Use of the corresponding variables is deprecated in favor of this function, since their use is unsafe in a threaded program. As of Python 1.5, the variables are restored to their previous values (before the call) when returning from a function that handled an exception.

The optional else clause is executed if and when control flows off the end of the try clause. Exceptions in the else clause are not handled by the preceding except clauses.

If finally is present, it specifies a `cleanup' handler. The try clause is executed, including any except and else clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The finally clause is executed. If there is a saved exception, it is re-raised at the end of the finally clause. If the finally clause raises another exception or executes a return or break statement, the saved exception is lost. The exception information is not available to the program during execution of the finally clause.

When a return, break or continue statement is executed in the try suite
of a `try...finally` statement, the `finally` clause is also executed `on the way out.' A `continue` statement is illegal in the `finally` clause. (The reason is a problem with the current implementation -- this restriction may be lifted in the future).

Additional information on exceptions can be found in section 4.2, and information on using the `raise` statement to generate exceptions may be found in section 6.9.

---

**Footnotes**

... stack.7.1

The exception is propogated to the invocation stack only if there is no `finally` clause that negates the exception.

... clause.7.2

Currently, control ``flows off the end'' except in the case of an exception or the execution of a `return`, `continue`, or `break` statement.

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.5 The with statement

New in version 2.5.

The with statement is used to wrap the execution of a block with methods defined by a context manager (see section 3.4.9). This allows common try...except...finally usage patterns to be encapsulated for convenient reuse.

```
with_stmt ::= "with" expression ["as" target] ":" suite
```

The execution of the with statement proceeds as follows:

1. The context expression is evaluated to obtain a context manager.

2. The context manager's __enter__() method is invoked.

3. If a target was included in the with statement, the return value from __enter__() is assigned to it.

   **Note:** The with statement guarantees that if the __enter__() method returns without an error, then __exit__() will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 5 below.

4. The suite is executed.

5. The context manager's __exit__() method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to __exit__(). Otherwise, three None arguments are supplied.

   If the suite was exited due to an exception, and the return value from the __exit__() method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with
the statement following the `with` statement.

If the suite was exited for any reason other than an exception, the return value from `__exit__()` is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

**Note:** In Python 2.5, the `with` statement is only allowed when the `with_statement` feature has been enabled. It will always be enabled in Python 2.6. This `__future__` import statement can be used to enable the feature:

```
from __future__ import with_statement
```

**See Also:**

PEP 0343, *The "with" statement*

The specification, background, and examples for the Python `with` statement.

---

Release 2.5, documentation updated on 19th September, 2006.
See *About this document...* for information on suggesting changes.
A function definition defines a user-defined function object (see section \[section 3.2\]):

A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global namespace as the global namespace to be used when the function is called.
The function definition does not execute the function body; this gets executed only when the function is called.

A function definition may be wrapped by one or more decorator expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition. The result must be a callable, which is invoked with the function object as the only argument. The returned value is bound to the function name instead of the function object. Multiple decorators are applied in nested fashion. For example, the following code:

```python
@f1(arg)
@f2
def func(): pass
```

is equivalent to:

```python
def func(): pass
func = f1(arg)(f2(func))
```

When one or more top-level parameters have the form `parameter = expression`, the function is said to have "default parameter values." For a parameter with a default value, the corresponding argument may be omitted from a call, in which case the parameter's default value is substituted. If a parameter has a default value, all following parameters must also have a default value -- this is a syntactic restriction that is not expressed by the grammar.

**Default parameter values are evaluated when the function definition is executed.** This means that the expression is evaluated once, when the function is defined, and that that same "pre-computed" value is used for each call. This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g.:

```python
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section 5.3.4. A function
call always assigns values to all parameters mentioned in the parameter list, either from position arguments, from keyword arguments, or from default values. If the form `"*identifier"` is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form `"**identifier"` is present, it is initialized to a new dictionary receiving any excess keyword arguments, defaulting to a new empty dictionary.

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda forms, described in section 5.11. Note that the lambda form is merely a shorthand for a simplified function definition; a function defined in a `"def"` statement can be passed around or assigned to another name just like a function defined by a lambda form. The `"def"` form is actually more powerful since it allows the execution of multiple statements.

**Programmer's note:** Functions are first-class objects. A `"def"` form executed inside a function definition defines a local function that can be returned or passed around. Free variables used in the nested function can access the local variables of the function containing the def. See section 4.1 for details.
7.7 Class definitions

A class definition defines a class object (see section 3.2):

```
classdef ::= "class" classname
           [inheritance] ":" suite
inheritance ::= "(" [expression_list] ")"
classname ::= identifier
```

A class definition is an executable statement. It first evaluates the inheritance list, if present. Each item in the inheritance list should evaluate to a class object or class type which allows subclassing. The class's suite is then executed in a new execution frame (see section 4.1), using a newly created local namespace and the original global namespace. (Usually, the suite contains only function definitions.) When the class's suite finishes execution, its execution frame is discarded but its local namespace is saved. A class object is then created using the inheritance list for the base classes and the saved local namespace for the attribute dictionary. The class name is bound to this class object in the original local namespace.

**Programmer's note:** Variables defined in the class definition are class variables; they are shared by all instances. To define instance variables, they must be given a value in the `__init__()` method or in another method. Both class and instance variables are accessible through the notation `\self\name`, and an instance variable hides a class variable with the same name when accessed in this way. Class variables with immutable values can be used as defaults for instance variables. For new-style classes, descriptors can be used to create instance variables with different implementation details.
See About this document... for information on suggesting changes.
8. Top-level components

The Python interpreter can get its input from a number of sources: from a script passed to it as standard input or as program argument, typed in interactively, from a module source file, etc. This chapter gives the syntax used in these cases.

Subsections

- 8.1 Complete Python programs
- 8.2 File input
- 8.3 Interactive input
- 8.4 Expression input

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
8.1 Complete Python programs

While a language specification need not prescribe how the language interpreter is invoked, it is useful to have a notion of a complete Python program. A complete Python program is executed in a minimally initialized environment: all built-in and standard modules are available, but none have been initialized, except for sys (various system services), __builtins__ (built-in functions, exceptions and None) and __main__. The latter is used to provide the local and global namespace for execution of the complete program.

The syntax for a complete Python program is that for file input, described in the next section.

The interpreter may also be invoked in interactive mode; in this case, it does not read and execute a complete program but reads and executes one statement (possibly compound) at a time. The initial environment is identical to that of a complete program; each statement is executed in the namespace of __main__.

Under Unix, a complete program can be passed to the interpreter in three forms: with the -c string command line option, as a file passed as the first command line argument, or as standard input. If the file or standard input is a tty device, the interpreter enters interactive mode; otherwise, it executes the file as a complete program.
Previous: 8.1 Complete Python programs  Up: 8. Top-level components  Next: 8.3 Interactive input
8.2 File input

All input read from non-interactive files has the same form:

\[
\text{file_input} ::= (\text{NEWLINE} \mid \text{statement})^*
\]

This syntax is used in the following situations:

- when parsing a complete Python program (from a file or from a string);
- when parsing a module;
- when parsing a string passed to the \texttt{exec} statement;

Release 2.5, documentation updated on 19th September, 2006.
See \texttt{About this document...} for information on suggesting changes.
8.3 Interactive input

Input in interactive mode is parsed using the following grammar:

```
interactive_input ::= [stmt_list] NEWLINE | compound_stmt NEWLINE
```

Download entire grammar as text.

Note that a (top-level) compound statement must be followed by a blank line in interactive mode; this is needed to help the parser detect the end of the input.
8.4 Expression input

There are two forms of expression input. Both ignore leading whitespace. The string argument to `eval()` must have the following form:

```
expr ::= expression_list NEWLINE*
```

The input line read by `input()` must have the following form:

```
expr ::= expression_list NEWLINE
```

Note: to read `raw' input line without interpretation, you can use the built-in function `raw_input()' or the `readline()' method of file objects.
A. History and License

Subsections

- A.1 History of the software
- A.2 Terms and conditions for accessing or otherwise using Python
- A.3 Licenses and Acknowledgements for Incorporated Software
  - A.3.1 Mersenne Twister
  - A.3.2 Sockets
  - A.3.3 Floating point exception control
  - A.3.4 MD5 message digest algorithm
  - A.3.5 Asynchronous socket services
  - A.3.6 Cookie management
  - A.3.7 Profiling
  - A.3.8 Execution tracing
  - A.3.9 UUencode and UUdecode functions
  - A.3.10 XML Remote Procedure Calls

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
A.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

<table>
<thead>
<tr>
<th>Release</th>
<th>Derived from</th>
<th>Year</th>
<th>Owner</th>
<th>GPL compatible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.0 thru 1.2</td>
<td>n/a</td>
<td>1991-1995</td>
<td>CWI</td>
<td>yes</td>
</tr>
<tr>
<td>1.3 thru 1.5.2</td>
<td>1.2</td>
<td>1995-1999</td>
<td>CNRI</td>
<td>yes</td>
</tr>
<tr>
<td>1.6</td>
<td>1.5.2</td>
<td>2000</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.0</td>
<td>1.6</td>
<td>2000</td>
<td>BeOpen.com</td>
<td>no</td>
</tr>
<tr>
<td>1.6.1</td>
<td>1.6</td>
<td>2001</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>no</td>
</tr>
<tr>
<td>2.0.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.1</td>
<td>2.1+2.0.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2</td>
<td>2.1.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.2</td>
<td>2.1.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>2.1.3</td>
<td>2.1.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.1</td>
<td>2.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.2</td>
<td>2.2.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.3</td>
<td>2.2.2</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3</td>
<td>2.2.2</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.1</td>
<td>2.3</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.2</td>
<td>2.3.1</td>
<td>2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.3</td>
<td>2.3.2</td>
<td>2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.4</td>
<td>2.3.3</td>
<td>2004</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.5</td>
<td>2.3.4</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4</td>
<td>2.3</td>
<td>2004</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.1</td>
<td>2.4</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.2</td>
<td>2.4.1</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.3</td>
<td>2.4.2</td>
<td>2006</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.5</td>
<td>2.4</td>
<td>2006</td>
<td>PSF</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Note:** GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.
A.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.5

1. This LICENSE AGREEMENT is between the Python Software Foundation (``PSF''), and the Individual or Organization (``Licensee'') accessing and otherwise using Python 2.5 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.5 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., ``Copyright © 2001-2006 Python Software Foundation; All Rights Reserved'' are retained in Python 2.5 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.5 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.5.

4. PSF is making Python 2.5 available to Licensee on an ``AS IS'' basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.5 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS
OF PYTHON 2.5 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.5, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.5, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com (``BeOpen''), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (``Licensee'') accessing and otherwise using this software in source or binary form and its associated documentation (``the Software'').

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an ``AS IS'' basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT
LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

**CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1**

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (``CNRI''), and the Individual or Organization (``Licensee'') accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to
reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): ``Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual
property law of the United States, including without limitation the federal
copyright law, and, to the extent such U.S. federal law does not apply, by
the law of the Commonwealth of Virginia, excluding Virginia's conflict of
law provisions. Notwithstanding the foregoing, with regard to derivative
works based on Python 1.6.1 that incorporate non-separable material that
was previously distributed under the GNU General Public License (GPL),
the law of the Commonwealth of Virginia shall govern this License
Agreement only as to issues arising under or with respect to Paragraphs 4,
5, and 7 of this License Agreement. Nothing in this License Agreement
shall be deemed to create any relationship of agency, partnership, or joint
venture between CNRI and Licensee. This License Agreement does not
grant permission to use CNRI trademarks or trade name in a trademark
sense to endorse or promote products or services of Licensee, or any third
party.

8. By clicking on the "ACCEPT" button where indicated, or by copying,
installing or otherwise using Python 1.6.1, Licensee agrees to be bound by
the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising
or publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL
WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR
ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
A.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

Subsections

- A.3.1 Mersenne Twister
- A.3.2 Sockets
- A.3.3 Floating point exception control
- A.3.4 MD5 message digest algorithm
- A.3.5 Asynchronous socket services
- A.3.6 Cookie management
- A.3.7 Profiling
- A.3.8 Execution tracing
- A.3.9 UUencode and UUdecode functions
- A.3.10 XML Remote Procedure Calls
A.3.1 Mersenne Twister

The `_random` module includes code based on a download from http://www.math.keio.ac.jp/~matumoto/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26. Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed) or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura, All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS AND CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.
http://www.math.keio.ac.jp/matumoto/emt.html
email: matumoto@math.keio.ac.jp
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
A.3.2 Sockets

The socket module uses the functions, `getaddrinfo`, and `getnameinfo`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/about/index.html.

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND GAI ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR GAI ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION HOWEVER CAUSED AND ON GAI ANY THEORY OF LIABILITY, WHETHER IN CONTRA LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN GAI OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY SUCH DAMAGE.
A.3.3 Floating point exception control

The source for the fpectl1 module includes the following notice:

```
/                              Copyright (c) 1996.
|                              The Regents of the University of California.
|                              All rights reserved.
|                              Permission to use, copy, modify, and distribute this software
|                              any purpose without fee is hereby granted, provided that this
|                              tire notice is included in all copies of any software which i
|                              includes a copy or modification of this software and in
|                              copies of the supporting documentation for such software.
|
| This work was produced at the University of California, Lawr
| Livermore National Laboratory under contract no. W-7405-EN
| between the U.S. Department of Energy and The Regents of
| University of California for the operation of UC LLNL.
|
| DISCLAIMER
|
| This software was prepared as an account of work sponsored b
| agency of the United States Government. Neither the United St
| Government nor the University of California nor any of their
| ployees, makes any warranty, express or implied, or assumes
| liability or responsibility for the accuracy, completeness
| usefulness of any information, apparatus, product, or pro
| disclosed, or represents that its use would not infr
| privately-owned rights. Reference herein to any specific com-
| cial products, process, or service by trade name, tradem
| manufacturer, or otherwise, does not necessarily constitute
| imply its endorsement, recommendation, or favoring by the Un
| States Government or the University of California. The views
| opinions of authors expressed herein do not necessarily stat
| reflect those of the United States Government or the Univer
| of California, and shall not be used for advertising or pro
\ endorsement purposes.
```

Python Reference Manual

Previous: A.3.2 Sockets  Up: A.3 Licenses and Acknowledgements  Next: A.3.4 MD5 message digest

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
A.3.4 MD5 message digest algorithm

The source code for the md5 module contains the following notice:

Copyright (C) 1999, 2002 Aladdin Enterprises. All rights reserved

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and not misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose text is available at
http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite (section A.5) but excluding the rest of Appendix A. It does not include any code or documentation that is identified in the RFC as being copyrighted.

The original and principal author of md5.h is L. Peter Deutsch <ghost@aladdin.com>. Other authors are noted in the change history that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed references to Ghostscript; clarified derivation from RFC 132
1999-11-04 lpd Edited comments slightly for automatic TOC extracti
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than added conditionalization for C++ compilation from Martin
Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
A.3.5 Asynchronous socket services

The asynchat and asyncore modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
A.3.6 Cookie management

The Cookie module contains the following notice:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Timothy O'Malley not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
A.3.7 Profiling

The profile and pstats modules contain the following notice:

Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose (subject to the restriction in the following sentence) without fee is hereby granted provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of InfoSeek not be used advertising or publicity pertaining to distribution of the software without specific, written prior permission. This permission is explicitly restricted to the copying and modification of the software to remain in Python, compiled Python, or other languages (such as C wherein the modified or derived code is exclusively imported into a Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
A.3.8 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rig
err... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights res
```

Permission to use, copy, modify, and distribute this Python software
and its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all co
files and that both that copyright notice and this permission notice ap
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertai
distribution of the software without specific, written prior permis

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
A.3.9 UUencode and UUdecode functions

The uu module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
All Rights Reserved
Permission to use, copy, modify, and distribute this software and documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and both that copyright notice and this permission notice appear in supporting documentation, and that the name of Lance Ellinghouse not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion between ascii and binary. This results in a 1000-fold speedup. Th version is still 5 times faster, though.
- Arguments more compliant with python standard
A.3.10 XML Remote Procedure Calls

The xmlrpclib module contains the following notice:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publiccit pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
Index

| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
abs  () (numeric object method)
add  () (numeric object method)
add  () (sequence object method)
all  (optional module attribute)
and  () (numeric object method)
bases  (class attribute)
builtin  (built-in module), [Link]
builtins
call  () (object method), [Link]
class  (instance attribute)
cmp  () (object method), [Link]
coerce  () (numeric object method), [Link]
complex  () (numeric object method)
contains  () (container object method)
contains  () (mapping object method)
contains  () (sequence object method)
dict  (class attribute)
dict  (function attribute)
dict  (instance attribute), [Link]
dict  (module attribute)
div  () (numeric object method)

ior  () (numeric object method)
ipow  () (numeric object method)
irshift  () (numeric object method)
isub  () (numeric object method)
iter  () (container object method)
iter  () (sequence object method)
itruediv  () (numeric object method)
ixor  () (numeric object method)
le  () (object method)
len  () (container object method)
len  () (mapping object method)
long  () (numeric object method)
lshift  () (numeric object method)
lt  () (object method)
main  (built-in module), [Link], [Link]
metaclass
mod  () (numeric object method)
module  (class attribute)
module  (function attribute)
module  (method attribute)
mul  () (numeric object method)
mul  () (sequence object method)
name  (class attribute)
name  (function attribute)
name  (method attribute)
name  (module attribute)
ne  () (object method)
neg  () (numeric object method)
new  () (object method)
nonzero  () (object method), [Link]
__int__() (numeric object method)
__invert__() (numeric object method)
abs() (built-in function)
addition
and
bit-wise
and operator
anonymous
function
append() (sequence object method)
argument
function
arithmetic
conversion
operation, binary
operation, unary
array (standard module)
ASCII, [Link], [Link], [Link], [Link], [Link]
assert statement
AssertionError exception
assertions
debugging
assignment
attribute, [Link]
augmented
class attribute
assignment (continued)
class instance attribute
slicing
statement, [Link]
subscription
target list
atom
attribute
assignment, [Link]
assignment, class
assignment, class instance
class
class instance
deletion
generic special
reference
special
AttributeError exception
augmented
assignment
back-quotes, [Link]
backslash character
backward
quotes, [Link]
binary
arithmetic operation
bit-wise operation
binding
global name
name, [Link], [Link], [Link], [Link], [Link]
bit-wise
and
operation, binary
operation, unary
or
xor
blank line
block
block (continued)
code
BNF, [Link]
Boolean
operation
Boolean object
break statement, [Link], [Link], [Link], [Link]
bsddb (standard module)
built-in
method
module
built-in function
call
built-in function object, [Link]
built-in method
call
built-in method object, [Link]
byte
bytecode
C
language, [Link], [Link], [Link]
call
built-in function
built-in method
class instance
class object, [Link], [Link], [Link]
function, [Link], [Link]
instance, [Link]
method
procedure
user-defined function
callable object, [Link]
chaining
    comparisons
character, [Link], [Link]
character set
chr() (built-in function), [Link]
class, [Link], [Link]
attribute
attribute assignment
constructor
definition, [Link]
instance
name
class instance
    attribute
    attribute assignment
call
class instance object, [Link], [Link], [Link]
class object, [Link], [Link], [Link]
co_filename (code object attribute)
co_firstlineno (code object attribute)
co_flags (code object attribute)
co_freevars (code object attribute)
co_innotab (code object attribute)
co_name (code object attribute)
co_names (code object attribute)
co_nlocals (code object attribute)
co_stacksize (code object attribute)
co_varnames (code object attribute)
co_code
        block
code block
code object
comma
trailing, [Link]
command line
comment
comparison
string
comparisons
chaining
compile() (built-in function)
complex
        literal
        number
complex() (built-in function)
complex object
compound
        statement
comprehensions
        list, [Link]
constant
constructor
call, [Link], [Link], [Link]
class statement
class clause
clear() (mapping object method)
cmp() (built-in function)
co_argcount (code object attribute)
co_cellvars (code object attribute)
co_code (code object attribute)
co_consts (code object attribute)

class container, [Link]
context manager
continue statement, [Link], [Link],
[Link], [Link]
conversion
arithmetic
string, [Link], [Link]
copy() (mapping object method)
count() (sequence object method)
<table>
<thead>
<tr>
<th>dangling</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>else</td>
<td>deletion</td>
</tr>
<tr>
<td>data</td>
<td>attribute</td>
</tr>
<tr>
<td>type</td>
<td>target</td>
</tr>
<tr>
<td>type, immutable</td>
<td>target list</td>
</tr>
<tr>
<td>datum</td>
<td>delimiters</td>
</tr>
<tr>
<td>dbm (standard module)</td>
<td>destructor [Link]</td>
</tr>
<tr>
<td>debugging</td>
<td>dictionary</td>
</tr>
<tr>
<td>assertions</td>
<td>display</td>
</tr>
<tr>
<td>decimal literal</td>
<td>dictionary object [Link], [Link], [Link], [Link]</td>
</tr>
<tr>
<td>DEDENT token [Link]</td>
<td>display</td>
</tr>
<tr>
<td>def statement</td>
<td>dictionary</td>
</tr>
<tr>
<td>default</td>
<td>list</td>
</tr>
<tr>
<td>parameter value</td>
<td>tuple</td>
</tr>
<tr>
<td>definition</td>
<td>division</td>
</tr>
<tr>
<td>class [Link]</td>
<td>divmod() (built-in function) [Link]</td>
</tr>
<tr>
<td>function [Link]</td>
<td>documentation string</td>
</tr>
<tr>
<td>del statement [Link], [Link]</td>
<td></td>
</tr>
</tbody>
</table>
elif keyword
Ellipsis object
else dangling keyword, [Link], [Link], [Link], [Link]
empty list tuple, [Link]
encodings environment error handling
errors escape sequence eval() (built-in function), [Link], [Link]
evaluation order exc_info (in module sys)
exc_traceback (in module sys), [Link]
exc_type (in module sys) exc_value (in module sys)
except keyword exception, [Link]
AssertionError AttributeError handler
ImportError, [Link] NameError raising

exception (continued)
RuntimeError StopIteration SyntaxError TypeError ValueError ZeroDivisionError
exception handler exclusive or
exec statement, [Link] execfile() (built-in function)
execution frame, [Link] restricted stack
execution model expression generator lambda list, [Link], [Link]
statement extend() (sequence object method)
extended slicing extended print statement extended slicing
extension filename module
f_back (frame attribute)
frame
f_builtins (frame attribute)
frame_object
f_code (frame attribute)
free
f_exc_traceback (frame attribute)
variable, [Link]
f_exc_type (frame attribute)
from
f_exc_value (frame attribute)
keyword, [Link]
f_globals (frame attribute)
from statement, [Link]
f_lasti (frame attribute)
func_closure (function attribute)
f_lineno (frame attribute)
func_code (function attribute)
f_locals (frame attribute)
func_defaults (function attribute)
f_restricted (frame attribute)
func_dict (function attribute)
f_trace (frame attribute)
func_doc (function attribute)
False
func_globals (function attribute)
function
file object, [Link]
anonymous
filename
call, [Link], [Link]
extension
call, user-defined
call,
from keyword, [Link]
from statement, [Link]
for statement, [Link], [Link]
form
generator
lambda, [Link]
name
user-defined
function object, [Link], [Link], [Link], [Link]
future
statement
garbage collection
gdbm (standard module)
generator
expression
function, [Link]
iterator, [Link]
generator expression object
generator object, [Link]
generic
special attribute
get() (mapping object method)
global
name binding
namespace
global statement, [Link], [Link]
globals() (built-in function)
grammar
grouping
handle an exception
handler
exception
has_key() (mapping object method)
hash() (built-in function)
hash character

hex() (built-in function)
hexadecimal literal
hierarchical
module names
hierarchy
type
id() (built-in function)
identifier, [Link]
identity
test
identity of an object
if statement
im_class (method attribute), [Link]
im_func (method attribute), [Link], [Link], [Link]
im_self (method attribute), [Link], [Link]
imaginary literal
immutable
data type
object, [Link]
immutable object, [Link]
immutable sequence object
import statement, [Link]
ImportError exception, [Link]
in
keyword
in operator
inclusive
or
INDENT token
indentation
index() (sequence object method)
index operation
indices() (slice method)
inheritance
initialization
module
input

input (continued)
raw
input() (built-in function)
insert() (sequence object method)
instance
call, [Link]
class
instance object, [Link], [Link], [Link]
int() (built-in function)
integer
representation
integer literal
integer object
interactive mode
internal type
interpreter
inversion
invocation
is not operator
is operator
item
sequence
string
item selection
items() (mapping object method)
iteritems() (mapping object method)
iterkeys() (mapping object method)
itervalues() (mapping object method)
Java language
K

key
date
key/datum pair
keys() (mapping object method)
keyword
elif

else, [Link], [Link], [Link], [Link]
except
finally, [Link], [Link], [Link]
from, [Link]
in

--------------------------
Lambda

- expression form, [Link]

Language

- C, [Link], [Link], [Link]
- Java
- Pascal

Last traceback (in module sys)
Leading whitespace
Len() (built-in function), [Link], [Link]
Lexical analysis
Lexical definitions
Line continuation
Line joining, [Link]
Line structure

List

- Assignment, target
- Comprehensions, [Link]

List (continued)

- Deletion target
- Display
- Empty
- Expression, [Link], [Link]
- Target, [Link]
- List object, [Link], [Link], [Link], [Link], [Link]
- Literal, [Link]
- Locals() (built-in function)
- Logical line
- Long() (built-in function)
- Long integer literal
- Long integer object

Loop

- Over mutable sequence
- Statement, [Link], [Link], [Link]

Loop control

- Target
makefile() (socket method)
mangling
name
mapping object, [Link], [Link], [Link]
membership
test
method
built-in
call
user-defined
method object, [Link], [Link]
minus
module
built-in
extension
importing
initialization

module (continued)
name
names, hierarchical
namespace
search path
user-defined
module object, [Link]
modules (in module sys)
modulo
multiplication
mutable object, [Link], [Link], [Link]
mutable sequence
loop over
mutable sequence object
<table>
<thead>
<tr>
<th>term</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>[Link], [Link]</td>
</tr>
<tr>
<td>binding</td>
<td>[Link], [Link], [Link], [Link], [Link], [Link]</td>
</tr>
<tr>
<td>global</td>
<td>class function mangling module rebinding unbinding NameError exception names</td>
</tr>
<tr>
<td>hierarchical module</td>
<td></td>
</tr>
<tr>
<td>private</td>
<td>namespace global module</td>
</tr>
<tr>
<td>negation</td>
<td>newline</td>
</tr>
<tr>
<td>suppression</td>
<td>NEWLINE token, [Link]</td>
</tr>
<tr>
<td>None object</td>
<td>not in operator</td>
</tr>
<tr>
<td>not operator</td>
<td>notation</td>
</tr>
<tr>
<td>NotImplementedException</td>
<td>null</td>
</tr>
<tr>
<td>null</td>
<td>operation</td>
</tr>
<tr>
<td>number</td>
<td>complex</td>
</tr>
<tr>
<td>floating point</td>
<td>numeric literal</td>
</tr>
<tr>
<td>numeric object</td>
<td>[Link]</td>
</tr>
</tbody>
</table>
object
Boolean
built-in function, [Link]
built-in method, [Link]
callable, [Link]
class, [Link], [Link], [Link]
class instance, [Link], [Link], [Link]
code
complex
dictionary, [Link], [Link], [Link], [Link], [Link]
Ellipsis
file, [Link]
floating point
frame
function, [Link], [Link], [Link], [Link]
generator, [Link]
generator expression
immutable, [Link], [Link]
immutable sequence
instance, [Link], [Link], [Link]
inger
list, [Link], [Link], [Link], [Link], [Link]
long integer
mapping, [Link], [Link], [Link]
method, [Link], [Link]
module, [Link]
mutable, [Link], [Link]
immutable sequence
None, [Link]
NotImplemented

object (continued)
traceback, [Link], [Link]
tuple, [Link], [Link], [Link]
unicode
user-defined function, [Link], [Link]
user-defined method
oct() (built-in function)
octal literal
open() (built-in function)
operation
binary arithmetic
binary bit-wise
Boolean
null
shifting
unary arithmetic
unary bit-wise
operator
and
in
is
is not
not
not in
or
overloading
precedence
operators
or
bit-wise
exclusive
inclusive
or operator
numeric, [Link]
plain integer
recursive
sequence, [Link], [Link], [Link], [Link], [Link], [Link]
slice
string, [Link], [Link]
ord() (built-in function), [Link], [Link]
order
evaluation
output, [Link]
standard, [Link]
OverflowError (built-in exception)
overloading
operator
packages
parameter
    value, default
parenthesized form
parser
Pascal
    language
pass statement
path
    module search
physical line, [Link], [Link]
plain integer literal
plain integer object
plus

pop() (mapping object method)
pop() (sequence object method)
popen() (in module os)
popitem() (mapping object method)
pow() (built-in function), [Link],
[Link], [Link]
precedence
    operator
primary
print statement, [Link]
private
    names
procedure
    call
program
quotes backward, [Link]

quotes (continued)

reverse, [Link]
scope, [Link]
search
path, module
sequence
- item
sequence object, [Link], [Link], [Link], [Link], [Link], [Link]
setdefault() (mapping object method)
shifting
- operation
simple
- statement
singleton
tuple
slice
slice() (built-in function)
slice object
slicing, [Link], [Link]
assignment
extended
sort() (sequence object method)
source character set
space
special
- attribute
- attribute, generic
stack
- execution
- trace
standard
- output, [Link]
Standard C
standard input
start (slice object attribute), [Link]
statement (continued)
import, [Link]
loop, [Link], [Link], [Link]
pass
print, [Link]
raise
return, [Link], [Link]
simple
try, [Link]
while, [Link], [Link]
with, [Link]
yield
statement grouping
stderr (in module sys)
stdin (in module sys)
stdio
stdout (in module sys), [Link]
step (slice object attribute), [Link]
stop (slice object attribute), [Link]
StopIteration exception
str() (built-in function), [Link]
string
comparison
conversion, [Link], [Link]
item
Unicode
string literal
string object, [Link], [Link]
subscription, [Link], [Link], [Link]
assignment
<table>
<thead>
<tr>
<th>statement</th>
<th>subtraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert</td>
<td>suite</td>
</tr>
<tr>
<td>assignment, [Link]</td>
<td>suppression</td>
</tr>
<tr>
<td>assignment, augmented</td>
<td></td>
</tr>
<tr>
<td>break, [Link], [Link], [Link], [Link]</td>
<td></td>
</tr>
<tr>
<td>class</td>
<td>newline</td>
</tr>
<tr>
<td>compound</td>
<td>syntax, [Link]</td>
</tr>
<tr>
<td>continue, [Link], [Link], [Link], [Link]</td>
<td></td>
</tr>
<tr>
<td>def</td>
<td>SyntaxError exception</td>
</tr>
<tr>
<td>del, [Link], [Link]</td>
<td></td>
</tr>
<tr>
<td>exec, [Link]</td>
<td>sys (built-in module), [Link], [Link], [Link]</td>
</tr>
<tr>
<td>expression</td>
<td>[Link]</td>
</tr>
<tr>
<td>for, [Link], [Link]</td>
<td></td>
</tr>
<tr>
<td>from, [Link]</td>
<td>sys.exc_info</td>
</tr>
<tr>
<td>future</td>
<td>sys.exc_traceback</td>
</tr>
<tr>
<td>global, [Link], [Link]</td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>sys.last_traceback</td>
</tr>
<tr>
<td></td>
<td>sys.modules</td>
</tr>
<tr>
<td></td>
<td>sys.stderr</td>
</tr>
<tr>
<td></td>
<td>sys.stdin</td>
</tr>
<tr>
<td></td>
<td>sys.stdout</td>
</tr>
</tbody>
</table>
tab, target, deletion list, [Link] list assignment list, deletion loop control

\texttt{tb\_frame} (traceback attribute) \texttt{tb\_lasti} (traceback attribute) \texttt{tb\_lineno} (traceback attribute) \texttt{tb\_next} (traceback attribute) termination model

test

\texttt{identity} \texttt{membership}

token

trace

\texttt{stack} traceback object, [Link], [Link]

trailing comma, [Link] triple-quoted string

\texttt{True} try statement, [Link] tuple

display empty, [Link] singleton

tuple object, [Link], [Link], [Link] type data

hierarchy immutable data

type() (built-in function) type of an object

TypeError exception

types, internal
unary
  arithmetic operation
  bit-wise operation
unbinding
  name
  UnboundLocalError
  unichr() (built-in function)
  Unicode
  unicode() (built-in function), [Link]
  Unicode Consortium
  unicode object
  UNIX

unreachable object
unrecognized escape sequence
update() (mapping object method)
user-defined
  function
  function call
  method
  module
  user-defined function object, [Link], [Link]
  user-defined method object
V

value
default parameter
value of an object
ValueError exception

values
writing, [Link]
values() (mapping object method)
variable
free, [Link]
while statement, [Link], [Link]
whitespace

with statement, [Link]
writing
g Values, [Link]
xor

bit-wise
yield statement
ZeroDivisionError exception
Macintosh Library Modules

Guido van Rossum

Python Software Foundation Email: docs@python.org

Fred L. Drake, Jr., editor

Release 2.5
19th September, 2006

- Front Matter
- Contents
- 1. Using Python on a Macintosh
  - 1.1 Getting and Installing MacPython
    - 1.1.1 How to run a Python script
    - 1.1.2 Running scripts with a GUI
    - 1.1.3 configuration
  - 1.2 The IDE
    - 1.2.1 Using the ``Python Interactive'' window
    - 1.2.2 Writing a Python Script
    - 1.2.3 Executing a script from within the IDE
    - 1.2.4 ``Save as'' versus ``Save as Applet''
  - 1.3 The Package Manager
- 2. MacPython Modules
  - 2.1 macpath -- MacOS path manipulation functions
  - 2.2 macfs -- Various file system services
    - 2.2.1 FSSpec Objects
    - 2.2.2 Alias Objects
    - 2.2.3 FInfo Objects
  - 2.3 ic -- Access to Internet Config
    - 2.3.1 IC Objects
  - 2.4 MacOS -- Access to Mac OS interpreter features
  - 2.5 macostools -- Convenience routines for file manipulation
2.6 findertools -- The finder's Apple Events interface
2.7 EasyDialogs -- Basic Macintosh dialogs
   2.7.1 ProgressBar Objects
2.8 Framework -- Interactive application framework
   2.8.1 Application Objects
   2.8.2 Window Objects
   2.8.3 ControlsWindow Object
   2.8.4 ScrolledWindow Object
   2.8.5 DialogWindow Objects
2.9 autoGIL -- Global Interpreter Lock handling in event loops

3. MacPython OSA Modules
   3.1 gensuitemodule -- Generate OSA stub packages
   3.2 aetools -- OSA client support
   3.3 aepack -- Conversion between Python variables and AppleEvent data containers
   3.4 aetypes -- AppleEvent objects
   3.5 MiniAEFrame -- Open Scripting Architecture server support
      3.5.1 AEServer Objects

4. MacOS Toolbox Modules
   4.1 Carbon.AE -- Apple Events
   4.2 Carbon.AH -- Apple Help
   4.3 Carbon.App -- Appearance Manager
   4.4 Carbon.CF -- Core Foundation
   4.5 Carbon.CG -- Core Graphics
   4.6 Carbon.CarbonEvt -- Carbon Event Manager
   4.7 Carbon.Cm -- Component Manager
   4.8 Carbon.Ctl -- Control Manager
   4.9 Carbon.Dlg -- Dialog Manager
   4.10 Carbon.Evt -- Event Manager
   4.11 Carbon.Fm -- Font Manager
   4.12 Carbon.Folder -- Folder Manager
   4.13 Carbon.Help -- Help Manager
   4.14 Carbon.List -- List Manager
   4.15 Carbon.Menu -- Menu Manager
   4.16 Carbon.Mlte -- MultiLingual Text Editor
   4.17 Carbon.Qd -- QuickDraw
   4.18 Carbon.Qdoffs -- QuickDraw Offscreen
4.19 Carbon.Qt -- QuickTime
4.20 Carbon.Res -- Resource Manager and Handles
4.21 Carbon.Scrap -- Scrap Manager
4.22 Carbon.Snd -- Sound Manager
4.23 Carbon.TE -- TextEdit
4.24 Carbon.Win -- Window Manager
4.25 ColorPicker -- Color selection dialog

- 5. Undocumented Modules
  - 5.1 applesingle -- AppleSingle decoder
  - 5.2 buildtools -- Helper module for BuildApplet and Friends
  - 5.3 cfmfile -- Code Fragment Resource module
  - 5.4 icopen -- Internet Config replacement for open()
  - 5.5 macerrors -- Mac OS Errors
  - 5.6 macresource -- Locate script resources
  - 5.7 Nav -- NavServices calls
  - 5.8 PixMapWrapper -- Wrapper for PixMap objects
  - 5.9 videoreader -- Read QuickTime movies
  - 5.10 W -- Widgets built on FrameWork

- A. History and License
  - A.1 History of the software
  - A.2 Terms and conditions for accessing or otherwise using Python
  - A.3 Licenses and Acknowledgements for Incorporated Software
    - A.3.1 Mersenne Twister
    - A.3.2 Sockets
    - A.3.3 Floating point exception control
    - A.3.4 MD5 message digest algorithm
    - A.3.5 Asynchronous socket services
    - A.3.6 Cookie management
    - A.3.7 Profiling
    - A.3.8 Execution tracing
    - A.3.9 UUencode and UUdecode functions
    - A.3.10 XML Remote Procedure Calls

- Module Index
- Index
- About this document...
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1. Using Python on a Macintosh

Python on a Macintosh running Mac OS X is in principle very similar to Python on any other UNIX platform, but there are a number of additional features such as the IDE and the Package Manager that are worth pointing out.

Python on Mac OS 9 or earlier can be quite different from Python on UNIX or Windows, but is beyond the scope of this manual, as that platform is no longer supported, starting with Python 2.4. See http://www.cwi.nl/~jack/macpython for installers for the latest 2.3 release for Mac OS 9 and related documentation.

Subsections

- **1.1 Getting and Installing MacPython**
  - 1.1.1 How to run a Python script
  - 1.1.2 Running scripts with a GUI
  - 1.1.3 configuration
- **1.2 The IDE**
  - 1.2.1 Using the `Python Interactive` window
  - 1.2.2 Writing a Python Script
  - 1.2.3 Executing a script from within the IDE
  - 1.2.4 ``Save as'' versus ``Save as Applet"
- **1.3 The Package Manager**
1.1 Getting and Installing MacPython

Mac OS X 10.3 comes with Python 2.3 pre-installed by Apple. This installation does not come with the IDE and other additions, however, so to get these you need to install the MacPython for Panther additions from the MacPython website, http://www.cwi.nl/~jack/macpython.

For MacPython 2.4, or for any MacPython on earlier releases of Mac OS X, you need to install a full distribution from the same website.

What you get after installing is a number of things:

- A MacPython-2.3 folder in your Applications folder. In here you find the PythonIDE Integrated Development Environment; PythonLauncher, which handles double-clicking Python scripts from the Finder; and the Package Manager.

- A fairly standard UNIX commandline Python interpreter in /usr/local/bin/python, but without the usual /usr/local/lib/python.

- A framework /Library/Frameworks/Python.framework, where all the action really is, but which you usually do not have to be aware of.

To uninstall MacPython you can simply remove these three things.

If you use the ``additions'' installer to install on top of an existing Apple-Python you will not get the framework and the commandline interpreter, as they have been installed by Apple already, in /System/Library/Frameworks/Python.framework and /usr/bin/python, respectively. You should in principle never modify or delete these, as they are Apple-controlled and may be used by Apple- or third-party software.

PythonIDE contains an Apple Help Viewer book called "MacPython Help" which you can access through its help menu. If you are completely new to Python you should start reading the IDE introduction in that document.

If you are familiar with Python on other UNIX platforms you should read the section on running Python scripts from the UNIX shell.
Subsections

- 1.1.1 How to run a Python script
- 1.1.2 Running scripts with a GUI
- 1.1.3 configuration

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1.1.1 How to run a Python script

Your best way to get started with Python on Mac OS X is through the PythonIDE integrated development environment, see section 1.2 and use the Help menu when the IDE is running.

If you want to run Python scripts from the Terminal window command line or from the Finder you first need an editor to create your script. Mac OS X comes with a number of standard UNIX command line editors, vim and emacs among them. If you want a more Mac-like editor BBEdit or TextWrangler from Bare Bones Software (see http://www.barebones.com/products/bbedit/index.shtml) are good choices. AppleWorks or any other word processor that can save files in ASCII is also a possibility, including TextEdit which is included with OS X.

To run your script from the Terminal window you must make sure that /usr/local/bin is in your shell search path.

To run your script from the Finder you have two options:

- Drag it to PythonLauncher
- Select PythonLauncher as the default application to open your script (or any .py script) through the finder Info window and double-click it.

PythonLauncher has various preferences to control how your script is launched. Option-dragging allows you to change these for one invocation, or use its Preferences menu to change things globally.
1.1.2 Running scripts with a GUI

There is one Mac OS X quirk that you need to be aware of: programs that talk to
the Aqua window manager (in other words, anything that has a GUI) need to be
run in a special way. Use `pythonw` instead of `python` to start such scripts.
1.1.3 configuration

MacPython honours all standard UNIX environment variables such as PYTHONPATH, but setting these variables for programs started from the Finder is non-standard as the Finder does not read your .profile or .cshrc at startup. You need to create a file ~/.MacOSX/environment.plist. See Apple's Technical Document QA1067 for details.

Installing additional Python packages is most easily done through the Package Manager, see the MacPython Help Book for details.
1.2 The IDE

The **Python IDE** (Integrated Development Environment) is a separate application that acts as a text editor for your Python code, a class browser, a graphical debugger, and more.

The online Python Help contains a quick walkthrough of the IDE that shows the major features and how to use them.

---

**Subsections**

- 1.2.1 Using the ``Python Interactive'' window
- 1.2.2 Writing a Python Script
- 1.2.3 Executing a script from within the IDE
- 1.2.4 ``Save as" versus ``Save as Applet"

---

*Macintosh Library Modules*

Previous: 1.1.3 configuration  Up: 1. Using Python on  Next: 1.2.1 Using the ``Python"

*Release 2.5, documentation updated on 19th September, 2006.*

*See [About this document...](About this document...) for information on suggesting changes.*
1.2.1 Using the ``Python Interactive'' window

Use this window like you would use a normal UNIX command line interpreter.
1.2.2 Writing a Python Script

In addition to using the Python IDE interactively, you can also type out a complete Python program, saving it incrementally, and execute it or smaller selections of it.

You can create a new script, open a previously saved script, and save your currently open script by selecting the appropriate item in the "File" menu. Dropping a Python script onto the Python IDE will open it for editing.

When the Python IDE saves a script, it uses the creator code settings which are available by clicking on the small black triangle on the top right of the document window, and selecting "save options". The default is to save the file with the Python IDE as the creator, this means that you can open the file for editing by simply double-clicking on its icon. You might want to change this behaviour so that it will be opened by the PythonLauncher, and run. To do this simply choose "PythonLauncher" from the "save options". Note that these options are associated with the file not the application.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1.2.3 Executing a script from within the IDE

You can run the script in the frontmost window of the Python IDE by hitting the run all button. You should be aware, however that if you use the Python convention "if __name__ == "__main__":" the script will not be `__main__` by default. To get that behaviour you must select the `Run as __main__` option from the small black triangle on the top right of the document window. Note that this option is associated with the file not the application. It will stay active after a save, however; to shut this feature off simply select it again.
1.2.4 ``Save as'' versus ``Save as Applet''

When you are done writing your Python script you have the option of saving it as an ``applet'' (by selecting ``Save as applet'' from the ``File'' menu). This has a significant advantage in that you can drop files or folders onto it, to pass them to the applet the way command-line users would type them onto the command-line to pass them as arguments to the script. However, you should make sure to save the applet as a separate file, do not overwrite the script you are writing, because you will not be able to edit it again.

Accessing the items passed to the applet via ``drag-and-drop'' is done using the standard `sys.argv` mechanism. See the general documentation for more.

Note that saving a script as an applet will not make it runnable on a system without a Python installation.
1.3 The Package Manager

Historically MacPython came with a number of useful extension packages included, because most Macintosh users do not have access to a development environment and C compiler. For Mac OS X that bundling is no longer done, but a new mechanism has been made available to allow easy access to extension packages.

The Python Package Manager helps you installing additional packages that enhance Python. It determines the exact MacOS version and Python version you have and uses that information to download a database that has packages that are tested and tried on that combination. In other words: if something is in your Package Manager window but does not work you are free to blame the database maintainer.

PackageManager then checks which of the packages you have installed and which ones are not. This should also work when you have installed packages outside of PackageManager. You can select packages and install them, and PackageManager will work out the requirements and install these too.

Often PackageManager will list a package in two flavors: binary and source. Binary should always work, source will only work if you have installed the Apple Developer Tools. PackageManager will warn you about this, and also about other external dependencies.

PackageManager is available as a separate application and also as a function of the IDE, through the File->Package Manager menu entry.
2. MacPython Modules

The following modules are only available on the Macintosh, and are documented here:

- **macpath**: MacOS path manipulation functions.
- **macfs**: Support for FSSpec, the Alias Manager, **finder** aliases, and the Standard File package.
- **ic**: Access to Internet Config.
- **MacOS**: Access to Mac OS-specific interpreter features.
- **macostools**: Convenience routines for file manipulation.
- **findertools**: Wrappers around the **finder**'s Apple Events interface.
- **EasyDialogs**: Basic Macintosh dialogs.
- **FrameWork**: Interactive application framework.
- **autoGIL**: Global Interpreter Lock handling in event loops.

---

*Release 2.5, documentation updated on 19th September, 2006.*

See [About this document...](#) for information on suggesting changes.
2.1 macpath -- MacOS path manipulation functions

This module is the Mac OS 9 (and earlier) implementation of the `os.path` module. It can be used to manipulate old-style Macintosh pathnames on Mac OS X (or any other platform). Refer to the [Python Library Reference](#) for documentation of `os.path`.

The following functions are available in this module: `normcase()`, `normpath()`, `isabs()`, `join()`, `split()`, `isdir()`, `isfile()`, `walk()`, `exists()`. For other functions available in `os.path` dummy counterparts are available.
2.2 macfs -- Various file system services

Availability: Macintosh.

**Deprecated since release 2.3.** The macfs module should be considered obsolete. For FSSpec, FSRef and Alias handling use the Carbon.File or Carbon.Folder module. For file dialogs use the EasyDialogs module. Also, this module is known to not work correctly with UFS partitions.

This module provides access to Macintosh FSSpec handling, the Alias Manager, finder aliases and the Standard File package.

Whenever a function or method expects a file argument, this argument can be one of three things: (1) a full or partial Macintosh pathname, (2) an FSSpec object or (3) a 3-tuple \((\text{wdRefNum}, \text{parID}, \text{name})\) as described in *Inside Macintosh: Files*. An FSSpec can point to a non-existing file, as long as the folder containing the file exists. Under MacPython the same is true for a pathname, but not under UNIX-Python because of the way pathnames and FSRefs works. See Apple's documentation for details.

A description of aliases and the Standard File package can also be found there.

**FSSpec**(*file*)

Create an FSSpec object for the specified file.

**RawFSSpec**(*data*)

Create an FSSpec object given the raw data for the C structure for the FSSpec as a string. This is mainly useful if you have obtained an FSSpec structure over a network.

**RawAlias**(*data*)

Create an Alias object given the raw data for the C structure for the alias as a string. This is mainly useful if you have obtained an FSSpec structure over a network.
**FInfo()**

Create a zero-filled FInfo object.

**ResolveAliasFile(file)**

Resolve an alias file. Returns a 3-tuple \((\text{fsspec}, \text{isfolder}, \text{aliased})\) where \(\text{fsspec}\) is the resulting FSSpec object, \(\text{isfolder}\) is true if \(\text{fsspec}\) points to a folder and \(\text{aliased}\) is true if the file was an alias in the first place (otherwise the FSSpec object for the file itself is returned).

**StandardGetFile([type, ...])**

Present the user with a standard ```open input file``` dialog. Optionally, you can pass up to four 4-character file types to limit the files the user can choose from. The function returns an FSSpec object and a flag indicating that the user completed the dialog without cancelling.

**PromptGetFile(prompt[, type, ...])**

Similar to **StandardGetFile()** but allows you to specify a prompt which will be displayed at the top of the dialog.

**StandardPutFile(prompt[, default])**

Present the user with a standard ```open output file``` dialog. \(\text{prompt}\) is the prompt string, and the optional \(\text{default}\) argument initializes the output file name. The function returns an FSSpec object and a flag indicating that the user completed the dialog without cancelling.

**GetDirectory([prompt])**

Present the user with a non-standard ```select a directory``` dialog. You have to first open the directory before clicking on the ```select current directory``` button. \(\text{prompt}\) is the prompt string which will be displayed at the top of the dialog. Return an FSSpec object and a success-indicator.

**SetFolder([fsspec])**

Set the folder that is initially presented to the user when one of the file selection dialogs is presented. \(\text{fsspec}\) should point to a file in the folder, not the folder itself (the file need not exist, though). If no argument is passed the folder will be set to the current directory, i.e. what \(\text{os.getcwd()}\)
returns.

Note that starting with System 7.5 the user can change Standard File behaviour with the "general controls" control panel, thereby making this call inoperative.

**FindFolder( where, which, create)**

Locates one of the "special" folders that Mac OS knows about, such as the trash or the Preferences folder. *where* is the disk to search, *which* is the 4-character string specifying which folder to locate. Setting *create* causes the folder to be created if it does not exist. Returns a *(vrefnum, dirid)* tuple.

The constants for *where* and *which* can be obtained from the standard module *Carbon.Folders*.

**NewAliasMinimalFromFullPath( pathname)**

Return a minimal *alias* object that points to the given file, which must be specified as a full pathname. This is the only way to create an *Alias* pointing to a non-existing file.

**FindApplication( creator)**

Locate the application with 4-character creator code *creator*. The function returns an *FSSpec* object pointing to the application.
2.2.1 FSSpec Objects

data
The raw data from the FSSpec object, suitable for passing to other applications, for instance.

as_pathname()
Return the full pathname of the file described by the FSSpec object.

as_tuple()
Return the \((wdRefNum, parID, name)\) tuple of the file described by the FSSpec object.

NewAlias([file])
Create an Alias object pointing to the file described by this FSSpec. If the optional \(file\) parameter is present the alias will be relative to that file, otherwise it will be absolute.

NewAliasMinimal()
Create a minimal alias pointing to this file.

GetCreatorType()
Return the 4-character creator and type of the file.

SetCreatorType(creator, type)
Set the 4-character creator and type of the file.

GetFInfo()
Return a FInfo object describing the finder info for the file.

SetFInfo(finfo)
Set the finder info for the file to the values given as \(finfo\) (an FInfo object).

GetDates()
Return a tuple with three floating point values representing the creation date, modification date and backup date of the file.

**SetDates** (crdate, moddate, backupdate)

Set the creation, modification and backup date of the file. The values are in the standard floating point format used for times throughout Python.
2.2.2 Alias Objects

**data**

The raw data for the Alias record, suitable for storing in a resource or transmitting to other programs.

**Resolve([file])**

Resolve the alias. If the alias was created as a relative alias you should pass the file relative to which it is. Return the FSSpec for the file pointed to and a flag indicating whether the Alias object itself was modified during the search process. If the file does not exist but the path leading up to it does exist a valid fsspec is returned.

**GetInfo( num)**

An interface to the C routine GetAliasInfo().

**Update( file[ , file2])**

Update the alias to point to the file given. If file2 is present a relative alias will be created.

Note that it is currently not possible to directly manipulate a resource as an Alias object. Hence, after calling Update() or after Resolve() indicates that the alias has changed the Python program is responsible for getting the data value from the Alias object and modifying the resource.
2.2.3 FInfo Objects

See Inside Macintosh: Files for a complete description of what the various fields mean.

**Creator**
The 4-character creator code of the file.

**Type**
The 4-character type code of the file.

**Flags**
The finder flags for the file as 16-bit integer. The bit values in Flags are defined in standard module MACFS.

**Location**
A Point giving the position of the file's icon in its folder.

**Fldr**
The folder the file is in (as an integer).
2.3 ic -- Access to Internet Config

Availability: Macintosh.

This module provides access to various internet-related preferences set through System Preferences or the Finder.

There is a low-level companion module icglue which provides the basic Internet Config access functionality. This low-level module is not documented, but the docstrings of the routines document the parameters and the routine names are the same as for the Pascal or C API to Internet Config, so the standard IC programmers' documentation can be used if this module is needed.

The ic module defines the error exception and symbolic names for all error codes Internet Config can produce; see the source for details.

**exception error**

Exception raised on errors in the ic module.

The ic module defines the following class and function:

**class IC([signature[, ic]])**

Create an Internet Config object. The signature is a 4-character creator code of the current application (default 'Pyth') which may influence some of ICs settings. The optional ic argument is a low-level icglue.icinstance created beforehand, this may be useful if you want to get preferences from a different config file, etc.

**launchurl(url[, hint])**

**parseurl(data[, start[, end[, hint]]])**

**mapfile(file)**

**maptypecreator(type, creator[, filename])**

**settypecreator(file)**

These functions are ``shortcuts'' to the methods of the same name, described below.
2.3.1 IC Objects

IC objects have a mapping interface, hence to obtain the mail address you simply get `ic['MailAddress']`. Assignment also works, and changes the option in the configuration file.

The module knows about various datatypes, and converts the internal IC representation to a "logical" Python data structure. Running the `ic` module standalone will run a test program that lists all keys and values in your IC database, this will have to serve as documentation.

If the module does not know how to represent the data it returns an instance of the `ICOpaqueData` type, with the raw data in its `data` attribute. Objects of this type are also acceptable values for assignment.

Besides the dictionary interface, IC objects have the following methods:

**launchurl**( `url[, hint]` )

Parse the given URL, launch the correct application and pass it the URL. The optional `hint` can be a scheme name such as `'mailto:'`, in which case incomplete URLs are completed with this scheme. If `hint` is not provided, incomplete URLs are invalid.

**parseurl**( `data[, start[, end[, hint]]]` )

Find an URL somewhere in `data` and return start position, end position and the URL. The optional `start` and `end` can be used to limit the search, so for instance if a user clicks in a long text field you can pass the whole text field and the click-position in `start` and this routine will return the whole URL in which the user clicked. As above, `hint` is an optional scheme used to complete incomplete URLs.

**mapfile**( `file` )

Return the mapping entry for the given `file`, which can be passed as either a filename or an `FSSpec()` result, and which need not exist.

The mapping entry is returned as a tuple `(version, type, creator,`
postcreator, flags, extension, appname, postappname, mimetype, entryname), where version is the entry version number, type is the 4-character filetype, creator is the 4-character creator type, postcreator is the 4-character creator code of an optional application to post-process the file after downloading, flags are various bits specifying whether to transfer in binary or ascii and such, extension is the filename extension for this file type, appname is the printable name of the application to which this file belongs, postappname is the name of the postprocessing application, mimetype is the MIME type of this file and entryname is the name of this entry.

maptypecreator(type, creator[, filename])

Return the mapping entry for files with given 4-character type and creator codes. The optional filename may be specified to further help finding the correct entry (if the creator code is '????', for instance).

The mapping entry is returned in the same format as for mapfile.

settypecreator(file)

Given an existing file, specified either as a filename or as an FSSpec() result, set its creator and type correctly based on its extension. The finder is told about the change, so the finder icon will be updated quickly.
2.4 MacOS -- Access to Mac OS interpreter features

Availability: Macintosh.

This module provides access to MacOS specific functionality in the Python interpreter, such as how the interpreter eventloop functions and the like. Use with care.

Note the capitalization of the module name; this is a historical artifact.

**runtimemodel**
Always 'mačo', from Python 2.4 on. In earlier versions of Python the value could also be 'ppc' for the classic Mac OS 8 runtime model or 'carbon' for the Mac OS 9 runtime model.

**linkmodel**
The way the interpreter has been linked. As extension modules may be incompatible between linking models, packages could use this information to give more decent error messages. The value is one of 'static' for a statically linked Python, 'framework' for Python in a Mac OS X framework, 'shared' for Python in a standard UNIX shared library. Older Pythons could also have the value 'cfm' for Mac OS 9-compatible Python.

**exception Error**
This exception is raised on MacOS generated errors, either from functions in this module or from other mac-specific modules like the toolbox interfaces. The arguments are the integer error code (the OSErr value) and a textual description of the error code. Symbolic names for all known error codes are defined in the standard module macerrors.

**GetErrorString**(*errno*)
Return the textual description of MacOS error code *errno*. 
**DebugStr**( message [, object])

On Mac OS X the string is simply printed to stderr (on older Mac OS systems more elaborate functionality was available), but it provides a convenient location to attach a breakpoint in a low-level debugger like **gdb**.

**SysBeep**( )

Ring the bell.

**GetTicks**( )

Get the number of clock ticks (1/60th of a second) since system boot.

**GetCreatorAndType**( file)

Return the file creator and file type as two four-character strings. The *file* parameter can be a pathname or an **FSSpec** or **FSRef** object.

**SetCreatorAndType**( file, creator, type)

Set the file creator and file type. The *file* parameter can be a pathname or an **FSSpec** or **FSRef** object. *creator* and *type* must be four character strings.

**openrf**( name [, mode])

Open the resource fork of a file. Arguments are the same as for the built-in function **open**( ). The object returned has file-like semantics, but it is not a Python file object, so there may be subtle differences.

**WMAvailable**( )

Checks whether the current process has access to the window manager. The method will return **False** if the window manager is not available, for instance when running on Mac OS X Server or when logged in via ssh, or when the current interpreter is not running from a fullblown application bundle. A script runs from an application bundle either when it has been started with **pythonw** instead of **python** or when running as an applet.
See About this document... for information on suggesting changes.
2.5 macostools -- Convenience routines for file manipulation

Availability: Macintosh.

This module contains some convenience routines for file-manipulation on the Macintosh. All file parameters can be specified as pathnames, FSRef or FSSpec objects. This module expects a filesystem which supports forked files, so it should not be used on UFS partitions.

The macostools module defines the following functions:

**copy** (src, dst[, createpath[, copytimes]])
Copy file src to dst. If createpath is non-zero the folders leading to dst are created if necessary. The method copies data and resource fork and some finder information (creator, type, flags) and optionally the creation, modification and backup times (default is to copy them). Custom icons, comments and icon position are not copied.

**copytree** (src, dst)
Recursively copy a file tree from src to dst, creating folders as needed. src and dst should be specified as pathnames.

**mkalias** (src, dst)
Create a finder alias dst pointing to src.

**touched** (dst)
Tell the finder that some bits of finder-information such as creator or type for file dst has changed. The file can be specified by pathname or fsspec. This call should tell the finder to redraw the files icon.

**BUFSIZ**
The buffer size for copy, default 1 megabyte.

Note that the process of creating finder aliases is not specified in the Apple
documentation. Hence, aliases created with `mkalias()` could conceivably have incompatible behaviour in some cases.
2.6 findertools -- The finder's Apple Events interface

Availability: Macintosh.

This module contains routines that give Python programs access to some functionality provided by the finder. They are implemented as wrappers around the AppleEvent interface to the finder.

All file and folder parameters can be specified either as full pathnames, or as FSRef or FSSpec objects.

The findertools module defines the following functions:

`launch(file)`
Tell the finder to launch file. What launching means depends on the file: applications are started, folders are opened and documents are opened in the correct application.

`Print(file)`
Tell the finder to print a file. The behaviour is identical to selecting the file and using the print command in the finder's file menu.

`copy(file, destdir)`
Tell the finder to copy a file or folder file to folder destdir. The function returns an Alias object pointing to the new file.

`move(file, destdir)`
Tell the finder to move a file or folder file to folder destdir. The function returns an Alias object pointing to the new file.

`sleep()`
Tell the finder to put the Macintosh to sleep, if your machine supports it.

`restart()`
Tell the finder to perform an orderly restart of the machine.

```
shutdown()
```

Tell the finder to perform an orderly shutdown of the machine.

*Release 2.5, documentation updated on 19th September, 2006.*

*See [About this document...](#) for information on suggesting changes.*
2.7 EasyDialogs -- Basic Macintosh dialogs

Availability: Macintosh.

The EasyDialogs module contains some simple dialogs for the Macintosh. All routines take an optional resource ID parameter id with which one can override the DLOG resource used for the dialog, provided that the dialog items correspond (both type and item number) to those in the default DLOG resource. See source code for details.

The EasyDialogs module defines the following functions:

**Message**(str[, id[, ok]])
Displays a modal dialog with the message text str, which should be at most 255 characters long. The button text defaults to ``OK``, but is set to the string argument ok if the latter is supplied. Control is returned when the user clicks the ``OK`` button.

**AskString**(prompt[, default[, id[, ok[, cancel]]]])
Asks the user to input a string value via a modal dialog. prompt is the prompt message, and the optional default supplies the initial value for the string (otherwise "" is used). The text of the ``OK`` and ``Cancel`` buttons can be changed with the ok and cancel arguments. All strings can be at most 255 bytes long. AskString() returns the string entered or None in case the user cancelled.

**AskPassword**(prompt[, default[, id[, ok[, cancel]]]])
Asks the user to input a string value via a modal dialog. Like AskString(), but with the text shown as bullets. The arguments have the same meaning as for AskString().

**AskYesNoCancel**(question[, default[, yes[, no[, cancel[, id]]]]])
Presents a dialog with prompt question and three buttons labelled ``Yes``,
``No'', and ``Cancel''. Returns 1 for ``Yes'', 0 for ``No'' and -1 for 
``Cancel''. The value of default (or 0 if default is not supplied) is returned 
when the RETURN key is pressed. The text of the buttons can be changed 
with the yes, no, and cancel arguments; to prevent a button from appearing, 
supply "" for the corresponding argument.

**ProgressBar([title[, maxval[, label[, id]]]])**

Displays a modeless progress-bar dialog. This is the constructor for the 
**ProgressBar** class described below. title is the text string displayed 
(default ``Working...''), maxval is the value at which progress is complete 
(default 0, indicating that an indeterminate amount of work remains to be 
done), and label is the text that is displayed above the progress bar itself.

**GetArgv([optionlist[ commandlist[, addoldfile[, addnewfile[, addfolder[, id]]]]]])**

Displays a dialog which aids the user in constructing a command-line 
argument list. Returns the list in sys.argv format, suitable for passing as 
an argument to getopt.getopt(). addoldfile, addnewfile, and 
addfolder are boolean arguments. When nonzero, they enable the user to 
insert into the command line paths to an existing file, a (possibly) not-yet-
extistent file, and a folder, respectively. (Note: Option arguments must 
appear in the command line before file and folder arguments in order to be 
recognized by getopt.getopt().) Arguments containing spaces can be 
specified by enclosing them within single or double quotes. A 
SystemExit exception is raised if the user presses the ``Cancel'' button.

optionlist is a list that determines a popup menu from which the allowed 
options are selected. Its items can take one of two forms: optstr or (optstr, 
descr). When present, descr is a short descriptive string that is displayed in 
the dialog while this option is selected in the popup menu. The 
correspondence between optstrs and command-line arguments is:

<table>
<thead>
<tr>
<th>optstr format</th>
<th>Command-line format</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-x (short option)</td>
</tr>
<tr>
<td>x: or x=</td>
<td>-x (short option with value)</td>
</tr>
<tr>
<td>xyz</td>
<td>--xyz (long option)</td>
</tr>
<tr>
<td>xyz: or xyz=</td>
<td>--xyz (long option with value)</td>
</tr>
</tbody>
</table>
commandlist is a list of items of the form cmdstr or (cmdstr, descr), where descr is as above. The cmdstrs will appear in a popup menu. When chosen, the text of cmdstr will be appended to the command line as is, except that a trailing " : " or " = " (if present) will be trimmed off.

New in version 2.0.


Post a dialog asking the user for a file to open, and return the file selected or None if the user cancelled. message is a text message to display, typeList is a list of 4-char filetypes allowable, defaultLocation is the pathname, FSSpec or FSRef of the folder to show initially, location is the (x, y) position on the screen where the dialog is shown, actionButtonLabel is a string to show instead of "Open" in the OK button, cancelButtonLabel is a string to show instead of "Cancel" in the cancel button, wanted is the type of value wanted as a return: str, unicode, FSSpec, FSRef and subtypes thereof are acceptable.

For a description of the other arguments please see the Apple Navigation Services documentation and the EasyDialogs source code.


Post a dialog asking the user for a file to save to, and return the file selected or None if the user cancelled. savedFileName is the default for the file name to save to (the return value). See AskFileForOpen( ) for a description of the other arguments.
AskFolder([, clientId[, windowTitle][, actionButtonLabel][, cancelButtonLabel][, preferenceKey][, popupExtension][, eventProc][, filterProc][, wanted])

Post a dialog asking the user to select a folder, and return the folder selected or None if the user cancelled. See AskFileForOpen() for a description of the arguments.

See Also:

Navigation Services Reference

Programmer's reference documentation for the Navigation Services, a part of the Carbon framework.

Subsections

- 2.7.1 ProgressBar Objects

Macintosh Library Modules

Previous: 2.6 findertools  Up: 2. MacPython Modules  Next: 2.7.1 ProgressBar Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
2.7.1 ProgressBar Objects

ProgressBar objects provide support for modeless progress-bar dialogs. Both determinate (thermometer style) and indeterminate (barber-pole style) progress bars are supported. The bar will be determinate if its maximum value is greater than zero; otherwise it will be indeterminate. Changed in version 2.2: Support for indeterminate-style progress bars was added.

The dialog is displayed immediately after creation. If the dialog's `Cancel` button is pressed, or if `Cmd-.` or `ESC` is typed, the dialog window is hidden and `KeyboardInterrupt` is raised (but note that this response does not occur until the progress bar is next updated, typically via a call to `inc()` or `set()`). Otherwise, the bar remains visible until the `ProgressBar` object is discarded.

`ProgressBar` objects possess the following attributes and methods:

**curval**
- The current value (of type integer or long integer) of the progress bar. The normal access methods coerce `curval` between 0 and `maxval`. This attribute should not be altered directly.

**maxval**
- The maximum value (of type integer or long integer) of the progress bar; the progress bar (thermometer style) is full when `curval` equals `maxval`. If `maxval` is 0, the bar will be indeterminate (barber-pole). This attribute should not be altered directly.

**title([newstr])**
- Sets the text in the title bar of the progress dialog to `newstr`.

**label([newstr])**
- Sets the text in the progress box of the progress dialog to `newstr`.

**set(value[, max])**
- Sets the progress bar's `curval` to `value`, and also `maxval` to `max` if the latter is provided. `value` is first coerced between 0 and `maxval`. The
thermometer bar is updated to reflect the changes, including a change from indeterminate to determinate or vice versa.

\texttt{inc([n])}

Increments the progress bar's \texttt{curval} by \texttt{n}, or by \texttt{1} if \texttt{n} is not provided. (Note that \texttt{n} may be negative, in which case the effect is a decrement.) The progress bar is updated to reflect the change. If the bar is indeterminate, this causes one ``spin'' of the barber pole. The resulting \texttt{curval} is coerced between 0 and \texttt{maxval} if incrementing causes it to fall outside this range.
2.8 FrameWork -- Interactive application framework

Availability: Macintosh.

The FrameWork module contains classes that together provide a framework for an interactive Macintosh application. The programmer builds an application by creating subclasses that override various methods of the bases classes, thereby implementing the functionality wanted. Overriding functionality can often be done on various different levels, i.e. to handle clicks in a single dialog window in a non-standard way it is not necessary to override the complete event handling.

Work on the FrameWork has pretty much stopped, now that PyObjC is available for full Cocoa access from Python, and the documentation describes only the most important functionality, and not in the most logical manner at that. Examine the source or the examples for more details. The following are some comments posted on the MacPython newsgroup about the strengths and limitations of FrameWork:

The strong point of FrameWork is that it allows you to break into the control-flow at many different places. W, for instance, uses a different way to enable/disable menus and that plugs right in leaving the rest intact. The weak points of FrameWork are that it has no abstract command interface (but that shouldn't be difficult), that its dialog support is minimal and that its control/toolbar support is non-existent.

The FrameWork module defines the following functions:

Application()
An object representing the complete application. See below for a description of the methods. The default __init__() routine creates an empty window dictionary and a menu bar with an apple menu.

MenuBar()
An object representing the menubar. This object is usually not created by
the user.

**Menu**(*bar, title[, after]*)

An object representing a menu. Upon creation you pass the MenuBar the menu appears in, the *title* string and a position (1-based) *after* where the menu should appear (default: at the end).

**MenuItem**(*menu, title[, shortcut, callback]*)

Create a menu item object. The arguments are the menu to create, the item title string and optionally the keyboard shortcut and a callback routine. The callback is called with the arguments menu-id, item number within menu (1-based), current front window and the event record.

Instead of a callable object the callback can also be a string. In this case menu selection causes the lookup of a method in the topmost window and the application. The method name is the callback string with 'domenu_' prepended.

Calling the MenuBar fixmenudimstate() method sets the correct dimming for all menu items based on the current front window.

**Separator**(*menu*)

Add a separator to the end of a menu.

**SubMenu**(*menu, label*)

Create a submenu named *label* under menu *menu*. The menu object is returned.

**Window**(*parent*)

Creates a (modeless) window. *Parent* is the application object to which the window belongs. The window is not displayed until later.

**DialogWindow**(*parent*)

Creates a modeless dialog window.

**windowbounds**(*width, height*)

Return a (*left, top, right, bottom*) tuple suitable for creation of a
window of given width and height. The window will be staggered with respect to previous windows, and an attempt is made to keep the whole window on-screen. However, the window will however always be the exact size given, so parts may be offscreen.

**setwatchcursor()**
Set the mouse cursor to a watch.

**setarrowcursor()**
Set the mouse cursor to an arrow.

---

**Subsections**

- [2.8.1 Application Objects](#)
- [2.8.2 Window Objects](#)
- [2.8.3 ControlsWindow Object](#)
- [2.8.4 ScrolledWindow Object](#)
- [2.8.5 DialogWindow Objects](#)

---

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
2.8.1 Application Objects

Application objects have the following methods, among others:

**makeusermenus()**
Override this method if you need menus in your application. Append the menus to the attribute `menubar`.

**getabouttext()**
Override this method to return a text string describing your application. Alternatively, override the `do_about()` method for more elaborate ```about``` messages.

**mainloop([`mask`, `wait`])**
This routine is the main event loop, call it to set your application rolling. `Mask` is the mask of events you want to handle, `wait` is the number of ticks you want to leave to other concurrent application (default 0, which is probably not a good idea). While raising `self` to exit the mainloop is still supported it is not recommended: call `self._quit()` instead.

The event loop is split into many small parts, each of which can be overridden. The default methods take care of dispatching events to windows and dialogs, handling drags and resizes, Apple Events, events for non-FrameWork windows, etc.

In general, all event handlers should return 1 if the event is fully handled and 0 otherwise (because the front window was not a FrameWork window, for instance). This is needed so that update events and such can be passed on to other windows like the Sioux console window. Calling `MacOS.HandleEvent()` is not allowed within `our_dispatch` or its callees, since this may result in an infinite loop if the code is called through the Python inner-loop event handler.

**asyncevents(`onoff`)**
Call this method with a nonzero parameter to enable asynchronous event handling. This will tell the inner interpreter loop to call the application
event handler *async_dispatch* whenever events are available. This will cause FrameWork window updates and the user interface to remain working during long computations, but will slow the interpreter down and may cause surprising results in non-reentrant code (such as FrameWork itself). By default *async_dispatch* will immediately call *our_dispatch* but you may override this to handle only certain events asynchronously. Events you do not handle will be passed to Sioux and such.

The old on/off value is returned.

**_quit()**

Terminate the running *mainloop()* call at the next convenient moment.

**do_char**(c, *event*)

The user typed character *c*. The complete details of the event can be found in the *event* structure. This method can also be provided in a *Window* object, which overrides the application-wide handler if the window is frontmost.

**do_dialogevent**(event)

Called early in the event loop to handle modeless dialog events. The default method simply dispatches the event to the relevant dialog (not through the *DialogWindow* object involved). Override if you need special handling of dialog events (keyboard shortcuts, etc).

**idle**(event)

Called by the main event loop when no events are available. The null-event is passed (so you can look at mouse position, etc).
2.8.2 Window Objects

Window objects have the following methods, among others:

**open()**
Override this method to open a window. Store the MacOS window-id in `self.wid` and call the `do_postopen()` method to register the window with the parent application.

**close()**
Override this method to do any special processing on window close. Call the `do_postclose()` method to cleanup the parent state.

**do_postresize(width, height, macoswindowid)**
Called after the window is resized. Override if more needs to be done than calling `InvalRect`.

**do_contentclick(local, modifiers, event)**
The user clicked in the content part of a window. The arguments are the coordinates (window-relative), the key modifiers and the raw event.

**do_update(macoswindowid, event)**
An update event for the window was received. Redraw the window.

**do_activate(activate, event)**
The window was activated (`activate == 1`) or deactivated (`activate == 0`). Handle things like focus highlighting, etc.
2.8.3 ControlsWindow Object

ControlsWindow objects have the following methods besides those of Window objects:

\textbf{do\_controlhit}( window, control, pcode, event)

Part \textit{pcode} of control \textit{control} was hit by the user. Tracking and such has already been taken care of.
2.8.4 ScrolledWindow Object

ScrolledWindow objects are ControlsWindow objects with the following extra methods:

**scrollbars**([wantx[, wanty]])
Create (or destroy) horizontal and vertical scrollbars. The arguments specify which you want (default: both). The scrollbars always have minimum 0 and maximum 32767.

**getscrollbarvalues()**
You must supply this method. It should return a tuple \((x, y)\) giving the current position of the scrollbars (between 0 and 32767). You can return None for either to indicate the whole document is visible in that direction.

**updatescrollbars()**
Call this method when the document has changed. It will call getscrollbarvalues() and update the scrollbars.

**scrollbar_callback**(which, what, value)
Supplied by you and called after user interaction. which will be 'x' or 'y', what will be '-', '--', 'set', '++' or '+'. For 'set', value will contain the new scrollbar position.

**scalebarvalues**(absmin, absmax, curmin, curmax)
Auxiliary method to help you calculate values to return from getscrollbarvalues(). You pass document minimum and maximum value and topmost (leftmost) and bottommost (rightmost) visible values and it returns the correct number or None.

**do_activate**(onoff, event)
Takes care of dimming/highlighting scrollbars when a window becomes frontmost. If you override this method, call this one at the end of your method.

**do_postresize**(width, height, window)
Moves scrollbars to the correct position. Call this method initially if you override it.

**do_controlhit** *(window, control, pcode, event)*

Handles scrollbar interaction. If you override it call this method first, a nonzero return value indicates the hit was in the scrollbars and has been handled.
2.8.5 DialogWindow Objects

DialogWindow objects have the following methods besides those of Window objects:

**open***( resid)  
Create the dialog window, from the DLOG resource with id resid. The dialog object is stored in self.wid.

**do_itemhit**( item, event)  
Item number item was hit. You are responsible for redrawing toggle buttons, etc.
2.9 autoGIL -- Global Interpreter Lock handling in event loops

Availability: Macintosh.

The autoGIL module provides a function installAutoGIL that automatically locks and unlocks Python's Global Interpreter Lock when running an event loop.

**exception AutoGILError**

Raised if the observer callback cannot be installed, for example because the current thread does not have a run loop.

**installAutoGIL()**

Install an observer callback in the event loop (CFRunLoop) for the current thread, that will lock and unlock the Global Interpreter Lock (GIL) at appropriate times, allowing other Python threads to run while the event loop is idle.

Availability: OSX 10.1 or later.
3. MacPython OSA Modules

This chapter describes the current implementation of the Open Scripting Architecture (OSA, also commonly referred to as AppleScript) for Python, allowing you to control scriptable applications from your Python program, and with a fairly pythonic interface. Development on this set of modules has stopped, and a replacement is expected for Python 2.5.

For a description of the various components of AppleScript and OSA, and to get an understanding of the architecture and terminology, you should read Apple's documentation. The "Applescript Language Guide" explains the conceptual model and the terminology, and documents the standard suite. The "Open Scripting Architecture" document explains how to use OSA from an application programmers point of view. In the Apple Help Viewer these books are located in the Developer Documentation, Core Technologies section.

As an example of scripting an application, the following piece of AppleScript will get the name of the frontmost Finder window and print it:

```applescript
tell application "Finder"
    get name of window 1
end tell
```

In Python, the following code fragment will do the same:

```python
import Finder

f = Finder.Finder()
print f.get(f.window(1).name)
```

As distributed the Python library includes packages that implement the standard suites, plus packages that interface to a small number of common applications.

To send AppleEvents to an application you must first create the Python package interfacing to the terminology of the application (what Script Editor calls the "Dictionary"). This can be done from within the PythonIDE or by running the gensuitemodule.py module as a standalone program from the command line.

The generated output is a package with a number of modules, one for every suite
used in the program plus an `__init__` module to glue it all together. The Python inheritance graph follows the AppleScript inheritance graph, so if a program's dictionary specifies that it includes support for the Standard Suite, but extends one or two verbs with extra arguments then the output suite will contain a module `Standard_Suite` that imports and re-exports everything from `StdSuites.Standard_Suite` but overrides the methods that have extra functionality. The output of `gensuitemodule` is pretty readable, and contains the documentation that was in the original AppleScript dictionary in Python docstrings, so reading it is a good source of documentation.

The output package implements a main class with the same name as the package which contains all the AppleScript verbs as methods, with the direct object as the first argument and all optional parameters as keyword arguments. AppleScript classes are also implemented as Python classes, as are comparisons and all the other thingies.

The main Python class implementing the verbs also allows access to the properties and elements declared in the AppleScript class "application". In the current release that is as far as the object orientation goes, so in the example above we need to use `f.get(f.window(1).name)` instead of the more Pythonic `f.window(1).name.get()`.

If an AppleScript identifier is not a Python identifier the name is mangled according to a small number of rules:

- spaces are replaced with underscores
- other non-alphanumeric characters are replaced with `_xx_` where `xx` is the hexadecimal character value
- any Python reserved word gets an underscore appended

Python also has support for creating scriptable applications in Python, but The following modules are relevant to MacPython AppleScript support:

- `gensuitemodule` Create a stub package from an OSA dictionary
- `aetools` Basic support for sending Apple Events
- `aepack` Conversion between Python variables and AppleEvent data containers.
- `aetypes` Python representation of the Apple Event Object Model.
MiniAEFrame

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

In addition, support modules have been pre-generated for Finder, Terminal, Explorer, Netscape, CodeWarrior, SystemEvents and StdSuites.
3.1 gensuitemodule -- Generate OSA stub packages

Availability: Macintosh.

The gensuitemodule module creates a Python package implementing stub code for the AppleScript suites that are implemented by a specific application, according to its AppleScript dictionary.

It is usually invoked by the user through the PythonIDE, but it can also be run as a script from the command line (pass --help for help on the options) or imported from Python code. For an example of its use see Mac/scripts/genallsuites.py in a source distribution, which generates the stub packages that are included in the standard library.

It defines the following public functions:

is_scriptable( application)

Returns true if application, which should be passed as a pathname, appears to be scriptable. Take the return value with a grain of salt: Internet Explorer appears not to be scriptable but definitely is.

processfile( application[, output, basepkgname, edit_modnames, creatorsignature, dump, verbose])

Create a stub package for application, which should be passed as a full pathname. For a .app bundle this is the pathname to the bundle, not to the executable inside the bundle; for an unbundled CFM application you pass the filename of the application binary.

This function asks the application for its OSA terminology resources, decodes these resources and uses the resultant data to create the Python code for the package implementing the client stubs.

output is the pathname where the resulting package is stored, if not specified a standard "save file as" dialog is presented to the user.
basepkgname is the base package on which this package will build, and defaults to StdSuites. Only when generating StdSuites itself do you need to specify this. edit_modnames is a dictionary that can be used to change modulenames that are too ugly after name mangling. creator_signature can be used to override the 4-char creator code, which is normally obtained from the PkgInfo file in the package or from the CFM file creator signature. When dump is given it should refer to a file object, and processfile will stop after decoding the resources and dump the Python representation of the terminology resources to this file. verbose should also be a file object, and specifying it will cause processfile to tell you what it is doing.

\[
\text{application}[, \text{output, basepkgname,}}
\]

\text{processfile_fromresource(} \text{edit_modnames, creatorsignature, dump,}
\]

\text{verbose])

This function does the same as processfile, except that it uses a different method to get the terminology resources. It opens application as a resource file and reads all "aete" and "aeut" resources from this file.
3.2 aetools -- OSA client support

Availability: Macintosh.

The aetools module contains the basic functionality on which Python AppleScript client support is built. It also imports and re-exports the core functionality of the aetypes and aepack modules. The stub packages generated by gensuitemodule import the relevant portions of aetools, so usually you do not need to import it yourself. The exception to this is when you cannot use a generated suite package and need lower-level access to scripting.

The aetools module itself uses the AppleEvent support provided by the Carbon.AE module. This has one drawback: you need access to the window manager, see section 1.1.2 for details. This restriction may be lifted in future releases.

The aetools module defines the following functions:

packevent( ae, parameters, attributes)

Stores parameters and attributes in a pre-created Carbon.AE.AEDesc object. parameters and attributes are dictionaries mapping 4-character OSA parameter keys to Python objects. The objects are packed using aepack.pack().

unpackevent( ae[, formodulename])

Recursively unpacks a Carbon.AE.AEDesc event to Python objects. The function returns the parameter dictionary and the attribute dictionary. The formodulename argument is used by generated stub packages to control where AppleScript classes are looked up.

keysubst( arguments, keydict)

Converts a Python keyword argument dictionary arguments to the format required by packevent by replacing the keys, which are Python identifiers, by the four-character OSA keys according to the mapping specified in keydict. Used by the generated suite packages.
enumsubst( arguments, key, edict)

If the arguments dictionary contains an entry for key convert the value for that entry according to dictionary edict. This converts human-readable Python enumeration names to the OSA 4-character codes. Used by the generated suite packages.

The aetools module defines the following class:

class TalkTo([signature=None, start=0, timeout=0])

Base class for the proxy used to talk to an application. signature overrides the class attribute _signature (which is usually set by subclasses) and is the 4-char creator code defining the application to talk to. start can be set to true to enable running the application on class instantiation. timeout can be specified to change the default timeout used while waiting for an AppleEvent reply.

_start()

Test whether the application is running, and attempt to start it if not.

send( code, subcode[, parameters, attributes])

Create the AppleEvent Carbon.AE.AEDesc for the verb with the OSA designation code, subcode (which are the usual 4-character strings), pack the parameters and attributes into it, send it to the target application, wait for the reply, unpack the reply with unpackevent and return the reply appleevent, the unpacked return values as a dictionary and the return attributes.
3.3 aepack -- Conversion between Python variables and AppleEvent data containers

Availability: Macintosh.

The aepack module defines functions for converting (packing) Python variables to AppleEvent descriptors and back (unpacking). Within Python the AppleEvent descriptor is handled by Python objects of built-in type AEDesc, defined in module Carbon.AE.

The aepack module defines the following functions:

`pack(x[, forctype])`

Returns an AEDesc object containing a conversion of Python value x. If forctype is provided it specifies the descriptor type of the result. Otherwise, a default mapping of Python types to Apple Event descriptor types is used, as follows:

<table>
<thead>
<tr>
<th>Python type</th>
<th>descriptor type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSSpec</td>
<td>typeFSS</td>
</tr>
<tr>
<td>FSRef</td>
<td>typeFSRef</td>
</tr>
<tr>
<td>Alias</td>
<td>typeAlias</td>
</tr>
<tr>
<td>integer</td>
<td>typeLong (32 bit integer)</td>
</tr>
<tr>
<td>float</td>
<td>typeFloat (64 bit floating point)</td>
</tr>
<tr>
<td>string</td>
<td>typeText</td>
</tr>
<tr>
<td>unicode</td>
<td>typeUnicodeText</td>
</tr>
<tr>
<td>list</td>
<td>typeAEList</td>
</tr>
<tr>
<td>dictionary</td>
<td>typeAERecord</td>
</tr>
<tr>
<td>instance</td>
<td>see below</td>
</tr>
</tbody>
</table>

If x is a Python instance then this function attempts to call an `__aepack__()` method. This method should return an AEDesc object.
If the conversion $x$ is not defined above, this function returns the Python string representation of a value (the `repr()` function) encoded as a text descriptor.

**unpack**(*$x$, *formodulename*)

$x$ must be an object of type `AEDesc`. This function returns a Python object representation of the data in the Apple Event descriptor $x$. Simple AppleEvent data types (integer, text, float) are returned as their obvious Python counterparts. Apple Event lists are returned as Python lists, and the list elements are recursively unpacked. Object references (ex. line 3 of document 1) are returned as instances of `aetypes.ObjectSpecifier`, unless `formodulename` is specified. AppleEvent descriptors with descriptor type typeFSS are returned as `FSSpec` objects. AppleEvent record descriptors are returned as Python dictionaries, with 4-character string keys and elements recursively unpacked.

The optional `formodulename` argument is used by the stub packages generated by `gensuitemodule`, and ensures that the OSA classes for object specifiers are looked up in the correct module. This ensures that if, say, the Finder returns an object specifier for a window you get an instance of `Finder.Window` and not a generic `aetypes.Window`. The former knows about all the properties and elements a window has in the Finder, while the latter knows no such things.

**See Also:**

Module **Carbon.AE**:

Built-in access to Apple Event Manager routines.

Module **aetypes**:

Python definitions of codes for Apple Event descriptor types.

*Inside Macintosh: Interapplication Communication*

Information about inter-process communications on the Macintosh.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
3.4 aetypes -- AppleEvent objects

Availability: Macintosh.

The aetypes defines classes used to represent Apple Event data descriptors and Apple Event object specifiers.

Apple Event data is contained in descriptors, and these descriptors are typed. For many descriptors the Python representation is simply the corresponding Python type: typeText in OSA is a Python string, typeFloat is a float, etc. For OSA types that have no direct Python counterpart this module declares classes. Packing and unpacking instances of these classes is handled automatically by aepack.

An object specifier is essentially an address of an object implemented in a Apple Event server. An Apple Event specifier is used as the direct object for an Apple Event or as the argument of an optional parameter. The aetypes module contains the base classes for OSA classes and properties, which are used by the packages generated by gensuitemodule to populate the classes and properties in a given suite.

For reasons of backward compatibility, and for cases where you need to script an application for which you have not generated the stub package this module also contains object specifiers for a number of common OSA classes such as Document, Window, Character, etc.

The AEObjects module defines the following classes to represent Apple Event descriptor data:

**class Unknown** (type, data)

The representation of OSA descriptor data for which the aepack and aetypes modules have no support, i.e. anything that is not represented by the other classes here and that is not equivalent to a simple Python value.

**class Enum** (enum)

An enumeration value with the given 4-character string value.
class InsertionLoc( of, pos)
    Position pos in object of.

class Boolean( bool)
    A boolean.

class StyledText( style, text)
    Text with style information (font, face, etc) included.

class AEText( script, style, text)
    Text with script system and style information included.

class IntlText( script, language, text)
    Text with script system and language information included.

class IntlWritingCode( script, language)
    Script system and language information.

class QDPoint( v, h)
    A quickdraw point.

class QDRectangle( v0, h0, v1, h1)
    A quickdraw rectangle.

class RGBColor( r, g, b)
    A color.

class Type( type)
    An OSA type value with the given 4-character name.

class Keyword( name)
    An OSA keyword with the given 4-character name.

class Range( start, stop)
    A range.
**class Ordinal**( abso)

Non-numeric absolute positions, such as "**firs**", first, or "**midd**", middle.

**class Logical**( logc, term)

The logical expression of applying operator logc to term.

**class Comparison**( obj1, relo, obj2)

The comparison relo of obj1 to obj2.

The following classes are used as base classes by the generated stub packages to represent AppleScript classes and properties in Python:

**class ComponentItem**( which[, fr])

Abstract baseclass for an OSA class. The subclass should set the class attribute want to the 4-character OSA class code. Instances of subclasses of this class are equivalent to AppleScript Object Specifiers. Upon instantiation you should pass a selector in which, and optionally a parent object in fr.

**class NProperty**( fr)

Abstract baseclass for an OSA property. The subclass should set the class attributes want and which to designate which property we are talking about. Instances of subclasses of this class are Object Specifiers.

**class ObjectSpecifier**( want, form, seld[, fr])

Base class of ComponentItem and NProperty, a general OSA Object Specifier. See the Apple Open Scripting Architecture documentation for the parameters. Note that this class is not abstract.
3.5 MiniAEFrame -- Open Scripting Architecture server support

Availability: **Macintosh.**

The module MiniAEFrame provides a framework for an application that can function as an Open Scripting Architecture (OSA) server, i.e. receive and process AppleEvents. It can be used in conjunction with [FrameWork](#) or standalone. As an example, it is used in [PythonCGISlave](#).

The MiniAEFrame module defines the following classes:

**class AEServer()**

A class that handles AppleEvent dispatch. Your application should subclass this class together with either MiniApplication or FrameWork.Application. Your `__init__()` method should call the `__init__()` method for both classes.

**class MiniApplication()**

A class that is more or less compatible with FrameWork.Application but with less functionality. Its event loop supports the apple menu, command-dot and AppleEvents; other events are passed on to the Python interpreter and/or Sioux. Useful if your application wants to use AEServer but does not provide its own windows, etc.
See About this document... for information on suggesting changes.
3.5.1 AEServer Objects

installaehandler(classe, type, callback)
Installs an AppleEvent handler. classe and type are the four-character OSA Class and Type designators, '****' wildcards are allowed. When a matching AppleEvent is received the parameters are decoded and your callback is invoked.

callback(_object, **kwargs)
Your callback is called with the OSA Direct Object as first positional parameter. The other parameters are passed as keyword arguments, with the 4-character designator as name. Three extra keyword parameters are passed: _class and _type are the Class and Type designators and _attributes is a dictionary with the AppleEvent attributes.

The return value of your method is packed with aetools.packevent() and sent as reply.

Note that there are some serious problems with the current design. AppleEvents which have non-identifier 4-character designators for arguments are not implementable, and it is not possible to return an error to the originator. This will be addressed in a future release.
4. MacOS Toolbox Modules

There are a set of modules that provide interfaces to various MacOS toolboxes. If applicable the module will define a number of Python objects for the various structures declared by the toolbox, and operations will be implemented as methods of the object. Other operations will be implemented as functions in the module. Not all operations possible in C will also be possible in Python (callbacks are often a problem), and parameters will occasionally be different in Python (input and output buffers, especially). All methods and functions have a `__doc__` string describing their arguments and return values, and for additional description you are referred to *Inside Macintosh* or similar works.

These modules all live in a package called Carbon. Despite that name they are not all part of the Carbon framework: CF is really in the CoreFoundation framework and Qt is in the QuickTime framework. The normal use pattern is

```python
from Carbon import AE
```

**Warning!** These modules are not yet documented. If you wish to contribute documentation of any of these modules, please get in touch with docs@python.org.

- **Carbon.AE**: Interface to the Apple Events toolbox.
- **Carbon.AH**: Interface to the Apple Help manager.
- **Carbon.App**: Interface to the Appearance Manager.
- **Carbon.CF**: Interface to the Core Foundation.
- **Carbon.CG**: Interface to the Component Manager.
- **Carbon.CarOnEvt**: Interface to the Carbon Event Manager.
- **Carbon.Cm**: Interface to the Component Manager.
- **Carbon.Ctl**: Interface to the Control Manager.
- **Carbon.Dlg**: Interface to the Dialog Manager.
- **Carbon.Evt**: Interface to the classic Event Manager.
- **Carbon.Fm**: Interface to the Font Manager.
- **Carbon.Folder**: Interface to the Folder Manager.
- **Carbon.Help**: Interface to the Carbon Help Manager.
Carbon.List  Interface to the List Manager.
Carbon.Menu  Interface to the Menu Manager.
Carbon.Mlte  Interface to the MultiLingual Text Editor.
Carbon.Qd  Interface to the QuickDraw toolbox.
Carbon.Qdoffs  Interface to the QuickDraw Offscreen APIs.
Carbon.Qt  Interface to the QuickTime toolbox.
Carbon.Scrap  Interface to the Carbon Scrap Manager.
Carbon.Snd  Interface to the Sound Manager.
Carbon.TE  Interface to TextEdit.
Carbon.Win  Interface to the Window Manager.
ColorPicker  Interface to the standard color selection dialog.

Macintosh Library Modules

Previous: 3.5.1 AEServer Objects  Up: Macintosh Library Modules  Next: 4.1 Carbon.AE

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.1 Carbon.AE -- Apple Events

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.2 Carbon.AH -- Apple Help

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.3 Carbon.App -- Appearance Manager

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.4 Carbon.CF -- Core Foundation

Availability: Macintosh.

The CFBaset, CFArray, CFData, CFDictionary, CFString and CFURL objects are supported, some only partially.
4.5 Carbon.CG -- Core Graphics

Availability: Macintosh.
4.6 Carbon.CarbonEvt -- Carbon Event Manager

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.7 Carbon.Cm -- Component Manager

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006. 
See About this document... for information on suggesting changes.
4.8 Carbon.Ctl -- Control Manager

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
4.9 Carbon.Dlg -- Dialog Manager

Availability: **Macintosh**.

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
4.10 Carbon.Evt -- Event Manager

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.11 Carbon.Fm -- Font Manager

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.12 Carbon.Folder -- Folder Manager

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.13 Carbon.Help -- Help Manager

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.14 Carbon.List -- List Manager

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
4.15 Carbon.Menu -- Menu Manager

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006. 
See About this document... for information on suggesting changes.
4.16 Carbon.Mlte -- MultiLingual Text Editor

Availability: Macintosh.
4.17 Carbon.Qd -- QuickDraw

Availability: Macintosh.
4.18 Carbon.Qdoffs -- QuickDraw Offscreen

Availability: Macintosh.
4.19 Carbon.Qt -- QuickTime

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.20 Carbon.Res -- Resource Manager and Handles

Availability: Macintosh.
4.21 Carbon.Scrap -- Scrap Manager

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.22 Carbon.Snd -- Sound Manager

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
4.23 Carbon . TE -- TextEdit

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.24 Carbon.Win -- Window Manager

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.25 ColorPicker -- Color selection dialog

Availability: Macintosh.

The ColorPicker module provides access to the standard color picker dialog.

**GetColor**(prompt, rgb)

Show a standard color selection dialog and allow the user to select a color. The user is given instruction by the prompt string, and the default color is set to rgb. rgb must be a tuple giving the red, green, and blue components of the color. GetColor() returns a tuple giving the user’s selected color and a flag indicating whether they accepted the selection of cancelled.

---

Release 2.5, documentation updated on 19th September, 2006.

See [About this document...](#) for information on suggesting changes.
5. Undocumented Modules

The modules in this chapter are poorly documented (if at all). If you wish to contribute documentation of any of these modules, please get in touch with docs@python.org.

- **applesingle**: Rudimentary decoder for AppleSingle format files.
- **buildtools**: Helper module for BuildApplet, BuildApplication and macfreeze.
- **cfmfile**: Code Fragment Resource module.
- **icopen**: Internet Config replacement for `open()`.
- **macerrors**: Constant definitions for many Mac OS error codes.
- **macresource**: Locate script resources.
- **Nav**: Interface to Navigation Services.
- **PixMapWrapper**: Wrapper for PixMap objects.
- **videoreader**: Read QuickTime movies frame by frame for further processing.
- **W**: Widgets for the Mac, built on top of **Framework**.
5.1 applesingle -- AppleSingle decoder

Availability: Macintosh.

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
5.2 buildtools -- Helper module for BuildApplet and Friends

Availability: Macintosh.

Deprecated since release 2.4. <

P>
5.3 cfmyfile -- Code Fragment Resource module

Availability: Macintosh.

cfmyfile is a module that understands Code Fragments and the accompanying `cfrg` resources. It can parse them and merge them, and is used by BuildApplication to combine all plugin modules to a single executable.

Deprecated since release 2.4. <

P>
5.4 icopen -- Internet Config replacement for open()

Availability: Macintosh.

Importing icopen will replace the built-in open() with a version that uses Internet Config to set file type and creator for new files.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.5 macerrors -- Mac OS Errors

Availability: Macintosh.

macerrors contains constant definitions for many Mac OS error codes.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.6 macresource -- Locate script resources

Availability: Macintosh.

macresource helps scripts finding their resources, such as dialogs and menus, without requiring special case code for when the script is run under MacPython, as a MacPython applet or under OSX Python.
5.7 Nav -- NavServices calls

Availability: Macintosh.

A low-level interface to Navigation Services.
5.8 PixMapWrapper -- Wrapper for PixMap objects

Availability: Macintosh.

PixMapWrapper wraps a PixMap object with a Python object that allows access to the fields by name. It also has methods to convert to and from PIL images.
5.9 videoreader -- Read QuickTime movies

Availability: Macintosh.

videoreader reads and decodes QuickTime movies and passes a stream of images to your program. It also provides some support for audio tracks.
5.10 W -- Widgets built on FrameWork

Availability: **Macintosh**.

The W widgets are used extensively in the **IDE**.
A. History and License

Subsections

- A.1 History of the software
- A.2 Terms and conditions for accessing or otherwise using Python
- A.3 Licenses and Acknowledgements for Incorporated Software
  - A.3.1 Mersenne Twister
  - A.3.2 Sockets
  - A.3.3 Floating point exception control
  - A.3.4 MD5 message digest algorithm
  - A.3.5 Asynchronous socket services
  - A.3.6 Cookie management
  - A.3.7 Profiling
  - A.3.8 Execution tracing
  - A.3.9 UUencode and UUdecode functions
  - A.3.10 XML Remote Procedure Calls

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
A.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

<table>
<thead>
<tr>
<th>Release</th>
<th>Derived from</th>
<th>Year</th>
<th>Owner</th>
<th>GPL compatible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.0 thru 1.2</td>
<td>n/a</td>
<td>1991-1995</td>
<td>CWI</td>
<td>yes</td>
</tr>
<tr>
<td>1.3 thru 1.5.2</td>
<td>1.2</td>
<td>1995-1999</td>
<td>CNRI</td>
<td>yes</td>
</tr>
<tr>
<td>1.6</td>
<td>1.5.2</td>
<td>2000</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.0</td>
<td>1.6</td>
<td>2000</td>
<td>BeOpen.com</td>
<td>no</td>
</tr>
<tr>
<td>1.6.1</td>
<td>1.6</td>
<td>2001</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>no</td>
</tr>
<tr>
<td>2.0.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.1</td>
<td>2.1+2.0.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2</td>
<td>2.1.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2.1.2</td>
<td>2.1.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.3</td>
<td>2.1.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.1</td>
<td>2.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.2</td>
<td>2.2.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.3</td>
<td>2.2.2</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3</td>
<td>2.2.2</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.1</td>
<td>2.3</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.2</td>
<td>2.3.1</td>
<td>2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.3</td>
<td>2.3.2</td>
<td>2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.4</td>
<td>2.3.3</td>
<td>2004</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.5</td>
<td>2.3.4</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4</td>
<td>2.3</td>
<td>2004</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.1</td>
<td>2.4</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.2</td>
<td>2.4.1</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.3</td>
<td>2.4.2</td>
<td>2006</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.5</td>
<td>2.4</td>
<td>2006</td>
<td>PSF</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Note:** GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.
A.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.5

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.5 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.5 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2006 Python Software Foundation; All Rights Reserved" are retained in Python 2.5 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.5 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.5.

4. PSF is making Python 2.5 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.5 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS
OF PYTHON 2.5 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.5, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.5, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com (``BeOpen''), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (``Licensee'') accessing and otherwise using this software in source or binary form and its associated documentation (``the Software '').

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an ``AS IS'' basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT
LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the ``BeOpen Python'' logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (``CNRI''), and the Individual or Organization (``Licensee'') accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to
reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): ``Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013.''

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an ``AS IS'' basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual
property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the ``ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR
ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
A.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

Subsections

- A.3.1 Mersenne Twister
- A.3.2 Sockets
- A.3.3 Floating point exception control
- A.3.4 MD5 message digest algorithm
- A.3.5 Asynchronous socket services
- A.3.6 Cookie management
- A.3.7 Profiling
- A.3.8 Execution tracing
- A.3.9 UUencode and UUdecode functions
- A.3.10 XML Remote Procedure Calls

Macintosh Library Modules

Previous: A.2 Terms and conditions Up: A. History and License Next: A.3.1 Mersenne Twister

Release 2.5, documentation updated on 19th September, 2006.
See About this document.. for information on suggesting changes.
A.3.1 Mersenne Twister

The _random module includes code based on a download from http://www.math.keio.ac.jp/~matumoto/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26. Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed) or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura, All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.
http://www.math.keio.ac.jp/matumoto/emt.html
email: matumoto@math.keio.ac.jp
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
A.3.2 Sockets

The socket module uses the functions, `getaddrinfo`, and `getnameinfo`, which are coded in separate source files from the WIDE Project, [http://www.wide.ad.jp/about/index.html](http://www.wide.ad.jp/about/index.html).

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
A.3.3 Floating point exception control

The source for the fpectl module includes the following notice:

---------------------------------------------------------------------
/ Copyright (c) 1996.
The Regents of the University of California.
All rights reserved.

Permission to use, copy, modify, and distribute this software
any purpose without fee is hereby granted, provided that this
tire notice is included in all copies of any software which i
cludes a copy or modification of this software and in
copies of the supporting documentation for such software.

This work was produced at the University of California, Lawr
Livermore National Laboratory under contract no. W-7405-EN
between the U.S. Department of Energy and The Regents of
University of California for the operation of UC LLNL.

DISCLAIMER

This software was prepared as an account of work sponsored by
agency of the United States Government. Neither the United St
Government nor the University of California nor any of their
ployees, makes any warranty, express or implied, or assumes
liability or responsibility for the accuracy, completeness
usefulness of any information, apparatus, product, or pro
 disclosed, or represents that its use would not infr
privately-owned rights. Reference herein to any specific com
cial products, process, or service by trade name, tradem
ufacturer, or otherwise, does not necessarily constitute
ply its endorsement, recommendation, or favoring by the Un
States Government or the University of California. The views
opinions of authors expressed herein do not necessarily stat
reflect those of the United States Government or the Univer
of California, and shall not be used for advertising or pro
endorsement purposes.

---------------------------------------------------------------------

Macintosh Library Modules

Previous: A.3.2 Sockets  Up: A.3 Licenses and Acknowledgements  Next: A.3.4
MD5 message digest

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
A.3.4 MD5 message digest algorithm

The source code for the md5 module contains the following notice:

Copyright (C) 1999, 2002 Aladdin Enterprises. All rights reserved

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose text is available at

http://www.ietf.org/rfc/rfc1321.txt

The code is derived from the text of the RFC, including the test suite (section A.5) but excluding the rest of Appendix A. It does not include any code or documentation that is identified in the RFC as being copyrighted.

The original and principal author of md5.h is L. Peter Deutsch <ghost@aladdin.com>. Other authors are noted in the change history that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed references to Ghostscript; clarified derivation from RFC 132.
2001-12-08 lpd Fixed typo in header comment (ansi2knr rather than ansi2knr).
2000-10-18 lpd Fixed typo in header comment (ansi2knr rather than ansi2knr).
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Added conditionalization for C++ compilation from Martin Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
A.3.5 Asynchronous socket services

The asynchat and asyncore modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
A.3.6 Cookie management

The Cookie module contains the following notice:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Timothy O'Malley not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
A.3.7 Profiling

The profile and pstats modules contain the following notice:

Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose (subject to the restriction in the following sentence) without fee is hereby granted provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear supporting documentation, and that the name of InfoSeek not be used advertising or publicity pertaining to distribution of the software without specific, written prior permission. This permission is explicitly restricted to the copying and modification of the software to remain in Python, compiled Python, or other languages (such as C wherein the modified or derived code is exclusively imported into a Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
A.3.8 Execution tracing

The `trace` module contains the following notice:

```python
portions copyright 2001, Autonomous Zones Industries, Inc., all rig
err... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights res
```

Permission to use, copy, modify, and distribute this Python softwar
its associated documentation for any purpose without fee is hereby
ganted, provided that the above copyright notice appears in all co
and that both that copyright notice and this permission notice appe
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertai
distribution of the software without specific, written prior permis
A.3.9 UUencode and UUdecode functions

The uu module contains the following notice:

Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
All Rights Reserved
Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and both that copyright notice and this permission notice appear in supporting documentation, and that the name of Lance Ellinghouse not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. Th
  version is still 5 times faster, though.
- Arguments more compliant with python standard
A.3.10 XML Remote Procedure Calls

The xmlrpclib module contains the following notice:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicit pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
Index

| a | b | c | d | e | f | g | i | k | l | m | n | o | p | q | r | s | t | u | v | w |
_ (underscore)

潜能 quit() (Application method)    开始 start() (TalkTo method)
aepack (standard module)
AEServer (class in MiniAEFrame)
AEText (class in aetypes)
aetools (standard module)
aetypes (standard module)
Alias Manager, Macintosh
AppleEvents, [Link]
applesingle (standard module)
Application() (in module FrameWork)
as_pathname() (FSSpec method)

as_tuple() (FSSpec method)
AskFileForOpen() (in module EasyDialogs)
AskFileForSave() (in module EasyDialogs)
AskFolder() (in module EasyDialogs)
AskPassword() (in module EasyDialogs)
AskString() (in module EasyDialogs)
AskYesNoCancel() (in module EasyDialogs)
asyncevents() (Application method)
autoGIL (extension module)
AutoGIL_Error
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean (class in aetypes)</td>
<td></td>
</tr>
<tr>
<td>BUFSIZ (in module macostools)</td>
<td></td>
</tr>
<tr>
<td>buildtools (standard module)</td>
<td></td>
</tr>
</tbody>
</table>
callback() (AEserver method)
Carbon.AE (standard module)
Carbon.AH (standard module)
Carbon.App (standard module)
Carbon.CaronEvt (standard module)
Carbon.CF (standard module)
Carbon.CG (standard module)
Carbon.Cm (standard module)
Carbon.Ctl (standard module)
Carbon.Dlg (standard module)
Carbon.Evt (standard module)
Carbon.Fm (standard module)
Carbon.Folder (standard module)
Carbon.Help (standard module)
Carbon.List (standard module)
Carbon.Menu (standard module)
Carbon.Mlte (standard module)
Carbon.Qd (built-in module)
Carbon.Qdoffs (built-in module)
Carbon.Qt (standard module)
Carbon.Res (standard module)
Carbon.Scrap (standard module)
Carbon.Snd (standard module)
Carbon.TE (standard module)
Carbon.Win (standard module)
cfmfile (standard module)
close() (Window method)
ColorPicker (extension module)
Comparison (class in aetypes)
ComponentItem (class in aetypes)
copy() (in module findertools)
copy() (in module macostools)
copytree() (in module macostools)
Creator (FInfo attribute)
curval (ProgressBar attribute)
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>data (Alias attribute)</td>
<td></td>
</tr>
<tr>
<td>data (FSSpec attribute)</td>
<td></td>
</tr>
<tr>
<td>DebugStr() (in module MacOS)</td>
<td></td>
</tr>
<tr>
<td>DialogWindow() (in module FrameWork)</td>
<td></td>
</tr>
<tr>
<td>do_activate() (IC method)</td>
<td></td>
</tr>
<tr>
<td>do_activate() (ScrolledWindow method)</td>
<td></td>
</tr>
<tr>
<td>do_char() (Application method)</td>
<td></td>
</tr>
<tr>
<td>do_contentclick() (Window method)</td>
<td></td>
</tr>
<tr>
<td>do_controlhit() (ControlsWindow method)</td>
<td></td>
</tr>
<tr>
<td>do_controlhit() (ScrolledWindow method)</td>
<td></td>
</tr>
<tr>
<td>do_dialogevent() (Application method)</td>
<td></td>
</tr>
<tr>
<td>do_itemhit() (DialogWindow method)</td>
<td></td>
</tr>
<tr>
<td>do_postresize() (ScrolledWindow method)</td>
<td></td>
</tr>
<tr>
<td>do_postresize() (Window method)</td>
<td></td>
</tr>
<tr>
<td>do_update() (Window method)</td>
<td></td>
</tr>
</tbody>
</table>
**EasyDialogs (standard module)**
**Enum (class in aetypes)**
**enums subst() (in module aetools)**

**Environment variables**

- PYTHONPATH
- Error
- error
<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>FindApplication()</td>
<td>(in module macfs)</td>
</tr>
<tr>
<td>findertools</td>
<td>(standard module)</td>
</tr>
<tr>
<td>FindFolder()</td>
<td>(in module macfs)</td>
</tr>
<tr>
<td>FInfo()</td>
<td>(in module macfs)</td>
</tr>
<tr>
<td>Flags (FInfo attribute)</td>
<td></td>
</tr>
<tr>
<td>Fldr (FInfo attribute)</td>
<td></td>
</tr>
<tr>
<td>FrameWork</td>
<td>(standard module), [Link]</td>
</tr>
<tr>
<td>FSSpec()</td>
<td>(in module macfs)</td>
</tr>
</tbody>
</table>
gensuitemodule (standard module)
getabouttext() (Application method)
GetArgv() (in module EasyDialogs)
GetColor() (in module ColorPicker)
GetCreatorAndType() (in module MacOS)
GetCreatorType() (FSSpec method)
GetDates() (FSSpec method)
GetDirectory() (in module macfs)
GetErrorString() (in module MacOS)
GetFInfo() (FSSpec method)
GetInfo() (Alias method)
getscrollbarvalues() (ScrolledWindow method)
GetTicks() (in module MacOS)
<table>
<thead>
<tr>
<th>ic (built-in module)</th>
<th>InsertionLoc (class in aetypes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC (class in ic)</td>
<td>installaehandler() (AEServer method)</td>
</tr>
<tr>
<td>icglue (built-in module)</td>
<td>installAutoGIL() (in module autoGIL)</td>
</tr>
<tr>
<td>icopen (standard module)</td>
<td>IntlText (class in aetypes)</td>
</tr>
<tr>
<td>idle() (Application method)</td>
<td>IntlWritingCode (class in aetypes)</td>
</tr>
<tr>
<td>inc() (ProgressBar method)</td>
<td>is_scriptable() (in module gensuitemodule)</td>
</tr>
</tbody>
</table>
keysubst() (in module aetools)   Keyword (class in aetypes)
<table>
<thead>
<tr>
<th>label() (ProgressBar method)</th>
<th>linkmodel (in module MacOS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>launch() (in module findertools)</td>
<td>Location (FInfo attribute)</td>
</tr>
<tr>
<td>launchurl() (IC method)</td>
<td>Logical (class in aetypes)</td>
</tr>
<tr>
<td>launchurl() (in module ic)</td>
<td></td>
</tr>
<tr>
<td>Nav (standard module)</td>
<td>NewAliasMinimal() (FSSpec method)</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Navigation Services</td>
<td>NewAliasMinimalFromFullPath() (in module macfs)</td>
</tr>
<tr>
<td>NewAlias() (FSSpec method)</td>
<td>NProperty (class in aetypes)</td>
</tr>
<tr>
<td>Class/Method/Function</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><code>ObjectSpecifier</code> (class in aetypes)</td>
<td></td>
</tr>
<tr>
<td><code>open()</code> (DialogWindow method)</td>
<td></td>
</tr>
<tr>
<td><code>open()</code> (Window method)</td>
<td></td>
</tr>
<tr>
<td>Open Scripting Architecture</td>
<td></td>
</tr>
<tr>
<td><code>openrf()</code> (in module MacOS)</td>
<td></td>
</tr>
<tr>
<td>Ordinal (class in aetypes)</td>
<td></td>
</tr>
</tbody>
</table>
pack() (in module aepack)
packevent() (in module aetools)
parseurl() (IC method)
parseurl() (in module ic)
PixmapWrapper (standard module)
Print() (in module findertools)

processfile() (in module gensuitemodule)
processfile_fromresource() (in module gensuitemodule)
ProgressBar() (in module EasyDialogs)
PromptGetFile() (in module macfs)
PYTHONPATH (environment variable)
<p>| QDPoint (class in aetypes) | QDRectangle (class in aetypes) |</p>
<table>
<thead>
<tr>
<th>Range (class in aetypes)</th>
<th>ResolveAliasFile() (in module macfs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RawAlias() (in module macfs)</td>
<td>restart() (in module findertools)</td>
</tr>
<tr>
<td>RawFSSpec() (in module macfs)</td>
<td>RGBColor (class in aetypes)</td>
</tr>
<tr>
<td>Resolve() (Alias method)</td>
<td>runtimemodel (in module MacOS)</td>
</tr>
</tbody>
</table>
scalebarvalues() (ScrollWindow method)
scrollbar_callback() (ScrollWindow method)
scrollbars() (ScrollWindow method)
send() (TalkTo method)
Separator() (in module FrameWork)
set() (ProgressBar method)
setarrowcursor() (in module FrameWork)
SetCreatorAndType() (in module MacOS)
SetCreatorType() (FSSpec method)
SetDates() (FSSpec method)
SetFInfo() (FSSpec method)
SetFolder() (in module macfs)

settypecreator() (IC method)
settypecreator() (in module ic)
setwatchcursor() (in module FrameWork)
shutdown() (in module findertools)
sleep() (in module findertools)
Standard File
StandardGetFile() (in module macfs)
StandardPutFile() (in module macfs)
StyledText (class in aetypes)
SubMenu() (in module FrameWork)
SysBeep() (in module MacOS)
TalkTo (class in aetools)
title() (ProgressBar method)
touched() (in module macostools)

Type (class in aetypes)
Type (FInfo attribute)
Unknown (class in aetypes)  
unpack() (in module aepack)  
unpackevent() (in module aetools)  
Update() (Alias method)  
updatescrollbars() (ScrolledWindow method)
Videoreader (standard module)
W (standard module)
Window() (in module FrameWork)

windowbounds() (in module FrameWork)
WMAvailable() (in module MacOS)
Front Matter

Contents

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

2. Defining New Types
   - 2.1 The Basics
2.1.1 Adding data and methods to the Basic example
2.1.2 Providing finer control over data attributes
2.1.3 Supporting cyclic garbage collection

2.2 Type Methods
2.2.1 Finalization and De-allocation
2.2.2 Object Presentation
2.2.3 Attribute Management
2.2.4 Object Comparison
2.2.5 Abstract Protocol Support
2.2.6 Weak Reference Support
2.2.7 More Suggestions

3. Building C and C++ Extensions with distutils
3.1 Distributing your extension modules

4. Building C and C++ Extensions on Windows
4.1 A Cookbook Approach
4.2 Differences Between UNIX and Windows
4.3 Using DLLs in Practice

5. Embedding Python in Another Application
5.1 Very High Level Embedding
5.2 Beyond Very High Level Embedding: An overview
5.3 Pure Embedding
5.4 Extending Embedded Python
5.5 Embedding Python in C++
5.6 Linking Requirements

A. Reporting Bugs

B. History and License
B.1 History of the software
B.2 Terms and conditions for accessing or otherwise using Python
B.3 Licenses and Acknowledgements for Incorporated Software
B.3.1 Mersenne Twister
B.3.2 Sockets
B.3.3 Floating point exception control
B.3.4 MD5 message digest algorithm
B.3.5 Asynchronous socket services
B.3.6 Cookie management
B.3.7 Profiling
B.3.8 Execution tracing
B.3.9 UUencode and UUdecode functions
B.3.10 XML Remote Procedure Calls
About this document...

Extending and Embedding the Python Interpreter

Up: Python Documentation Index  Next: Front Matter

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
1. Extending Python with C or C++

It is quite easy to add new built-in modules to Python, if you know how to program in C. Such extension modules can do two things that can't be done directly in Python: they can implement new built-in object types, and they can call C library functions and system calls.

To support extensions, the Python API (Application Programmers Interface) defines a set of functions, macros and variables that provide access to most aspects of the Python run-time system. The Python API is incorporated in a C source file by including the header "Python.h".

The compilation of an extension module depends on its intended use as well as on your system setup; details are given in later chapters.

Subsections

- 1.1 A Simple Example
- 1.2 Intermezzo: Errors and Exceptions
- 1.3 Back to the Example
- 1.4 The Module's Method Table and Initialization Function
- 1.5 Compilation and Linkage
- 1.6 Calling Python Functions from C
- 1.7 Extracting Parameters in Extension Functions
- 1.8 Keyword Parameters for Extension Functions
- 1.9 Building Arbitrary Values
- 1.10 Reference Counts
  - 1.10.1 Reference Counting in Python
  - 1.10.2 Ownership Rules
  - 1.10.3 Thin Ice
  - 1.10.4 NULL Pointers
- 1.11 Writing Extensions in C++
- 1.12 Providing a C API for an Extension Module
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1.1 A Simple Example

Let's create an extension module called "spam" (the favorite food of Monty Python fans...) and let's say we want to create a Python interface to the C library function `system()`. This function takes a null-terminated character string as argument and returns an integer. We want this function to be callable from Python as follows:

```python
>>> import spam
>>> status = spam.system("ls -l")
```

Begin by creating a file `spammodule.c`. (Historically, if a module is called "spam", the C file containing its implementation is called `spammodule.c`; if the module name is very long, like "spammify", the module name can be just `spammify.c`.)

The first line of our file can be:

```c
#include <Python.h>
```

which pulls in the Python API (you can add a comment describing the purpose of the module and a copyright notice if you like).

**Warning:** Since Python may define some pre-processor definitions which affect the standard headers on some systems, you must include `Python.h` before any standard headers are included.

All user-visible symbols defined by `Python.h` have a prefix of "Py" or "PY", except those defined in standard header files. For convenience, and since they are used extensively by the Python interpreter, "Python.h" includes a few standard header files: `<stdio.h>`, `<string.h>`, `<errno.h>`, and `<stdlib.h>`. If the latter header file does not exist on your system, it declares the functions `malloc()`, `free()` and `realloc()` directly.

The next thing we add to our module file is the C function that will be called
when the Python expression "spam.system(string)" is evaluated (we'll see shortly how it ends up being called):

```c
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", sts);
}
```

There is a straightforward translation from the argument list in Python (for example, the single expression "ls -l") to the arguments passed to the C function. The C function always has two arguments, conventionally named `self` and `args`.

The `self` argument is only used when the C function implements a built-in method, not a function. In the example, `self` will always be a NULL pointer, since we are defining a function, not a method. (This is done so that the interpreter doesn't have to understand two different types of C functions.)

The `args` argument will be a pointer to a Python tuple object containing the arguments. Each item of the tuple corresponds to an argument in the call's argument list. The arguments are Python objects -- in order to do anything with them in our C function we have to convert them to C values. The function `PyArg_ParseTuple() ` in the Python API checks the argument types and converts them to C values. It uses a template string to determine the required types of the arguments as well as the types of the C variables into which to store the converted values. More about this later.

`PyArg_ParseTuple() ` returns true (nonzero) if all arguments have the right type and its components have been stored in the variables whose addresses are passed. It returns false (zero) if an invalid argument list was passed. In the latter case it also raises an appropriate exception so the calling function can return NULL immediately (as we saw in the example).
Footnotes

...system().\footnote{1.1}

An interface for this function already exists in the standard module os -- it was chosen as a simple and straightforward example.
1.2 Intermezzo: Errors and Exceptions

An important convention throughout the Python interpreter is the following: when a function fails, it should set an exception condition and return an error value (usually a NULL pointer). Exceptions are stored in a static global variable inside the interpreter; if this variable is NULL no exception has occurred. A second global variable stores the “associated value” of the exception (the second argument to raise). A third variable contains the stack traceback in case the error originated in Python code. These three variables are the C equivalents of the Python variables sys.exc_type, sys.exc_value and sys.exc_traceback (see the section on module sys in the Python Library Reference). It is important to know about them to understand how errors are passed around.

The Python API defines a number of functions to set various types of exceptions.

The most common one is PyErr_SetString(). Its arguments are an exception object and a C string. The exception object is usually a predefined object like PyExc_ZeroDivisionError. The C string indicates the cause of the error and is converted to a Python string object and stored as the “associated value” of the exception.

Another useful function is PyErr_SetFromErrno(), which only takes an exception argument and constructs the associated value by inspection of the global variable errno. The most general function is PyErr_SetObject(), which takes two object arguments, the exception and its associated value. You don't need to Py_INCREF() the objects passed to any of these functions.

You can test non-destructively whether an exception has been set with PyErr_Occurred(). This returns the current exception object, or NULL if no exception has occurred. You normally don't need to call PyErr_Occurred() to see whether an error occurred in a function call, since you should be able to tell from the return value.

When a function \( f \) that calls another function \( g \) detects that the latter fails, \( f \)
should itself return an error value (usually NULL or -1). It should not call one of
the PyErr_*() functions -- one has already been called by g. f's caller is then
supposed to also return an error indication to its caller, again without calling
PyErr_*(), and so on -- the most detailed cause of the error was already
reported by the function that first detected it. Once the error reaches the Python
interpreter's main loop, this aborts the currently executing Python code and tries
to find an exception handler specified by the Python programmer.

(There are situations where a module can actually give a more detailed error
message by calling another PyErr_*() function, and in such cases it is fine to
do so. As a general rule, however, this is not necessary, and can cause
information about the cause of the error to be lost: most operations can fail for a
variety of reasons.)

To ignore an exception set by a function call that failed, the exception condition
must be cleared explicitly by calling PyErr_Clear(). The only time C code
should call PyErr_Clear() is if it doesn't want to pass the error on to the
interpreter but wants to handle it completely by itself (possibly by trying
something else, or pretending nothing went wrong).

Every failing malloc() call must be turned into an exception -- the direct
caller of malloc() (or realloc()) must call PyErr_NoMemory() and
return a failure indicator itself. All the object-creating functions (for example,
PyInt_FromLong()) already do this, so this note is only relevant to those
who call malloc() directly.

Also note that, with the important exception of PyArg_ParseTuple() and
friends, functions that return an integer status usually return a positive value or
zero for success and -1 for failure, like UNIX system calls.

Finally, be careful to clean up garbage (by making Py_XDECREF() or
Py_DECREF() calls for objects you have already created) when you return an
error indicator!

The choice of which exception to raise is entirely yours. There are predeclared C
objects corresponding to all built-in Python exceptions, such as
PyExc_ZeroDivisionError, which you can use directly. Of course, you
should choose exceptions wisely -- don't use PyExc_TypeError to mean that
a file couldn't be opened (that should probably be PyExc_IOError). If
something's wrong with the argument list, the PyArg_ParseTuple() function usually raises PyExc_TypeError. If you have an argument whose value must be in a particular range or must satisfy other conditions, PyExc_ValueError is appropriate.

You can also define a new exception that is unique to your module. For this, you usually declare a static object variable at the beginning of your file:

```c
static PyObject *SpamError;
```

and initialize it in your module's initialization function (initspam()) with an exception object (leaving out the error checking for now):

```c
PyMODINIT_FUNC
initspam(void)
{
    PyObject *m;

    m = Py_InitModule("spam", SpamMethods);
    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    Py_INCREF(SpamError);
    PyModule_AddObject(m, "error", SpamError);
}
```

Note that the Python name for the exception object is spam.error. The PyErr_NewException() function may create a class with the base class being Exception (unless another class is passed in instead of NULL), described in the Python Library Reference under ```Built-in Exceptions.''

Note also that the SpamError variable retains a reference to the newly created exception class; this is intentional! Since the exception could be removed from the module by external code, an owned reference to the class is needed to ensure that it will not be discarded, causing SpamError to become a dangling pointer. Should it become a dangling pointer, C code which raises the exception could cause a core dump or other unintended side effects.

We discuss the use of PyMODINIT_FUNC as a function return type later in this sample.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1.3 Back to the Example

Going back to our example function, you should now be able to understand this statement:

```c
if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;
```

It returns NULL (the error indicator for functions returning object pointers) if an error is detected in the argument list, relying on the exception set by `PyArg_ParseTuple()`. Otherwise the string value of the argument has been copied to the local variable `command`. This is a pointer assignment and you are not supposed to modify the string to which it points (so in Standard C, the variable `command` should properly be declared as "const char *command").

The next statement is a call to the UNIX function `system()`, passing it the string we just got from `PyArg_ParseTuple()`:

```c
sts = system(command);
```

Our `spam.system()` function must return the value of `sts` as a Python object. This is done using the function `Py_BuildValue()`, which is something like the inverse of `PyArg_ParseTuple()`: it takes a format string and an arbitrary number of C values, and returns a new Python object. More info on `Py_BuildValue()` is given later.

```c
return Py_BuildValue("i", sts);
```

In this case, it will return an integer object. (Yes, even integers are objects on the heap in Python!)

If you have a C function that returns no useful argument (a function returning `void`), the corresponding Python function must return `None`. You need this idiom to do so (which is implemented by the `Py_RETURN_NONE` macro):

```c
Py_INCREF(Py_None);
return Py_None;
```
Py_None is the C name for the special Python object None. It is a genuine Python object rather than a NULL pointer, which means ``error'' in most contexts, as we have seen.
1.4 The Module's Method Table and Initialization Function

I promised to show how spam_system() is called from Python programs. First, we need to list its name and address in a `method table`:

```c
static PyMethodDef SpamMethods[] = {
    ...
    {"system", spam_system, METH_VARARGS,
    "Execute a shell command."},
    ...
    {NULL, NULL, 0, NULL} /* Sentinel */
};
```

Note the third entry ("METH_VARARGS"). This is a flag telling the interpreter the calling convention to be used for the C function. It should normally always be "METH_VARARGS" or "METH_VARARGS | METH_KEYWORDS"; a value of 0 means that an obsolete variant of PyArg_ParseTuple() is used.

When using only "METH_VARARGS", the function should expect the Python-level parameters to be passed in as a tuple acceptable for parsing via PyArg_ParseTuple(); more information on this function is provided below.

The METH_KEYWORDS bit may be set in the third field if keyword arguments should be passed to the function. In this case, the C function should accept a third "PyObject **" parameter which will be a dictionary of keywords. Use PyArg_ParseTupleAndKeywords() to parse the arguments to such a function.

The method table must be passed to the interpreter in the module's initialization function. The initialization function must be named initname(), where name is the name of the module, and should be the only non-static item defined in the module file:

```c
PyMODINIT_FUNC
initspam(void)
{
    (void) Py_InitModule("spam", SpamMethods);
}
```
Note that PyMODINIT_FUNC declares the function as void return type, declares any special linkage declarations required by the platform, and for C++ declares the function as extern "C".

When the Python program imports module spam for the first time, \texttt{initspam()} is called. (See below for comments about embedding Python.) It calls \texttt{Py_InitModule()}, which creates a ``module object'' (which is inserted in the dictionary \texttt{sys.modules} under the key "spam"), and inserts built-in function objects into the newly created module based upon the table (an array of \texttt{PyMethodDef} structures) that was passed as its second argument. \texttt{Py_InitModule()} returns a pointer to the module object that it creates (which is unused here). It aborts with a fatal error if the module could not be initialized satisfactorily, so the caller doesn't need to check for errors.

When embedding Python, the \texttt{initspam()} function is not called automatically unless there's an entry in the \_\texttt{PyImport_Inittab} table. The easiest way to handle this is to statically initialize your statically-linked modules by directly calling \texttt{initspam()} after the call to \texttt{Py_Initialize()}:

```c
int main(int argc, char *argv[])
{
    /* Pass argv[0] to the Python interpreter */
    Py_SetProgramName(argv[0]);

    /* Initialize the Python interpreter. Required. */
    Py_Initialize();

    /* Add a static module */
    initspam();
}
```

An example may be found in the file \texttt{Demo/embed/demo.c} in the Python source distribution.

\textbf{Note:} Removing entries from \texttt{sys.modules} or importing compiled modules into multiple interpreters within a process (or following a \texttt{fork()} without an intervening \texttt{exec()}) can create problems for some extension modules. Extension module authors should exercise caution when initializing internal data structures. Note also that the \texttt{reload()} function can be used with extension modules, and will call the module initialization function (\texttt{initspam()} in the example), but will not load the module again if it was loaded from a dynamically
loadable object file (.so on UNIX, .dll on Windows).

A more substantial example module is included in the Python source distribution as Modules/xxmodule.c. This file may be used as a template or simply read as an example. The modulator.py script included in the source distribution or Windows install provides a simple graphical user interface for declaring the functions and objects which a module should implement, and can generate a template which can be filled in. The script lives in the Tools/modulator/ directory; see the README file there for more information.
1.5 Compilation and Linkage

There are two more things to do before you can use your new extension: compiling and linking it with the Python system. If you use dynamic loading, the details may depend on the style of dynamic loading your system uses; see the chapters about building extension modules (chapter 3) and additional information that pertains only to building on Windows (chapter 4) for more information about this.

If you can't use dynamic loading, or if you want to make your module a permanent part of the Python interpreter, you will have to change the configuration setup and rebuild the interpreter. Luckily, this is very simple on UNIX: just place your file (spammodule.c for example) in the Modules/ directory of an unpacked source distribution, add a line to the file Modules/Setup.local describing your file:

```
spam spammodule.o
```

and rebuild the interpreter by running `make` in the toplevel directory. You can also run `make` in the Modules/ subdirectory, but then you must first rebuild Makefile there by running `make Makefile`. (This is necessary each time you change the Setup file.)

If your module requires additional libraries to link with, these can be listed on the line in the configuration file as well, for instance:

```
spam spammodule.o -lX11
```
1.6 Calling Python Functions from C

So far we have concentrated on making C functions callable from Python. The reverse is also useful: calling Python functions from C. This is especially the case for libraries that support so-called ``callback'' functions. If a C interface makes use of callbacks, the equivalent Python often needs to provide a callback mechanism to the Python programmer; the implementation will require calling the Python callback functions from a C callback. Other uses are also imaginable.

Fortunately, the Python interpreter is easily called recursively, and there is a standard interface to call a Python function. (I won't dwell on how to call the Python parser with a particular string as input -- if you're interested, have a look at the implementation of the -c command line option in Python/pythonmain.c from the Python source code.)

Calling a Python function is easy. First, the Python program must somehow pass you the Python function object. You should provide a function (or some other interface) to do this. When this function is called, save a pointer to the Python function object (be careful to Py_INCREF() it!) in a global variable -- or wherever you see fit. For example, the following function might be part of a module definition:

```c
static PyObject *my_callback = NULL;

static PyObject *
my_set_callback(PyObject *dummy, PyObject *args)
{
    PyObject *result = NULL;
    PyObject *temp;

    if (PyArg_ParseTuple(args, "O:set_callback", &temp)) {
        if (!PyCallable_Check(temp)) {
            PyErr_SetString(PyExc_TypeError, "parameter must be callable");
            return NULL;
        }
        Py_XINCREF(temp); /* Add a reference to new callback
        Py_XDECREF(my_callback); /* Dispose of previous callback */
        my_callback = temp; /* Remember new callback */
        /* Boilerplate to return "None" */
        Py_INCREF(Py_None);
        result = Py_None;
    }
    return result;
}
```
return result;
}

This function must be registered with the interpreter using the METH_VARARGS flag; this is described in section 1.4, "The Module's Method Table and Initialization Function." The PyArg_ParseTuple() function and its arguments are documented in section 1.7, "Extracting Parameters in Extension Functions."

The macros Py_XINCREF() and Py_XDECREF() increment/decrement the reference count of an object and are safe in the presence of NULL pointers (but note that temp will not be NULL in this context). More info on them in section 1.10, "Reference Counts."

Later, when it is time to call the function, you call the C function PyEval_CallObject(). This function has two arguments, both pointers to arbitrary Python objects: the Python function, and the argument list. The argument list must always be a tuple object, whose length is the number of arguments. To call the Python function with no arguments, pass an empty tuple; to call it with one argument, pass a singleton tuple. Py_BuildValue() returns a tuple when its format string consists of zero or more format codes between parentheses. For example:

```c
int arg;
PyObject *arglist;
PyObject *result;
...
arg = 123;
...
/* Time to call the callback */
arglist = Py_BuildValue("(i)", arg);
result = PyEval_CallObject(my_callback, arglist);
Py_DECREF(arglist);
```

PyEval_CallObject() returns a Python object pointer: this is the return value of the Python function. PyEval_CallObject() is "reference-count-neutral" with respect to its arguments. In the example a new tuple was created to serve as the argument list, which is Py_DECREF()--ed immediately after the call.

The return value of PyEval_CallObject() is "new": either it is a brand new object, or it is an existing object whose reference count has been
incremented. So, unless you want to save it in a global variable, you should somehow `Py_DECREF()` the result, even (especially!) if you are not interested in its value.

Before you do this, however, it is important to check that the return value isn't `NULL`. If it is, the Python function terminated by raising an exception. If the C code that called `PyEval_CallObject()` is called from Python, it should now return an error indication to its Python caller, so the interpreter can print a stack trace, or the calling Python code can handle the exception. If this is not possible or desirable, the exception should be cleared by calling `PyErr_Clear()`. For example:

```c
if (result == NULL)
    return NULL; /* Pass error back */
...use result...
Py_DECREF(result);
```

Depending on the desired interface to the Python callback function, you may also have to provide an argument list to `PyEval_CallObject()`. In some cases the argument list is also provided by the Python program, through the same interface that specified the callback function. It can then be saved and used in the same manner as the function object. In other cases, you may have to construct a new tuple to pass as the argument list. The simplest way to do this is to call `Py_BuildValue()`. For example, if you want to pass an integral event code, you might use the following code:

```c
PyObject *arglist;
...arglist = Py_BuildValue("(l)", eventcode);
result = PyEval_CallObject(my_callback, arglist);
Py_DECREF(arglist);
if (result == NULL)
    return NULL; /* Pass error back */ /* Here maybe use the result */
Py_DECREF(result);
```

Note the placement of "`Py_DECREF(arglist)`" immediately after the call, before the error check! Also note that strictly spoken this code is not complete: `Py_BuildValue()` may run out of memory, and this should be checked.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1.7 Extracting Parameters in Extension Functions

The `PyArg_ParseTuple()` function is declared as follows:

```c
int PyArg_ParseTuple(PyObject *arg, char *format, ...);
```

The `arg` argument must be a tuple object containing an argument list passed from Python to a C function. The `format` argument must be a format string, whose syntax is explained in “Parsing arguments and building values” in the Python/C API Reference Manual. The remaining arguments must be addresses of variables whose type is determined by the format string.

Note that while `PyArg_ParseTuple()` checks that the Python arguments have the required types, it cannot check the validity of the addresses of C variables passed to the call: if you make mistakes there, your code will probably crash or at least overwrite random bits in memory. So be careful!

Note that any Python object references which are provided to the caller are borrowed references; do not decrement their reference count!

Some example calls:

```c
int ok;
int i, j;
long k, l;
const char *s;
int size;

ok = PyArg_ParseTuple(args, ""); /* No arguments */
     /* Python call: f() */

ok = PyArg_ParseTuple(args, "s", &s); /* A string */
     /* Possible Python call: f('whoops!') */

ok = PyArg_ParseTuple(args, "lls", &k, &l, &s); /* Two longs and */
     /* Possible Python call: f(1, 2, 'three') */

ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size);
     /* A pair of ints and a string, whose size is also returned */
     /* Possible Python call: f((1, 2), 'three') */
```
const char *file;
const char *mode = "r";
int bufsize = 0;
ok = PyArg_ParseTuple(args, "s|si", &file, &mode, &bufsize);
/* A string, and optionally another string and an integer */
/* Possible Python calls:
   f('spam')
   f('spam', 'w')
   f('spam', 'wb', 100000) */
}

int left, top, right, bottom, h, v;
ok = PyArg_ParseTuple(args, "((ii)(ii))(ii)",
&left, &top, &right, &bottom, &h, &v);
/* A rectangle and a point */
/* Possible Python call:
   f(((0, 0), (400, 300)), (10, 10)) */
}

Py_complex c;
ok = PyArg_ParseTuple(args, "D:myfunction", &c);
/* a complex, also providing a function name for errors */
/* Possible Python call: myfunction(1+2j) */
1.8 Keyword Parameters for Extension Functions

The `PyArg_ParseTupleAndKeywords()` function is declared as follows:

```c
int PyArg_ParseTupleAndKeywords(PyObject *arg, PyObject *kwdict, char *format, char *kwlist[], ...);
```

The `arg` and `format` parameters are identical to those of the `PyArg_ParseTuple()` function. The `kwdict` parameter is the dictionary of keywords received as the third parameter from the Python runtime. The `kwlist` parameter is a NULL-terminated list of strings which identify the parameters; the names are matched with the type information from `format` from left to right. On success, `PyArg_ParseTupleAndKeywords()` returns true, otherwise it returns false and raises an appropriate exception.

**Note:** Nested tuples cannot be parsed when using keyword arguments! Keyword parameters passed in which are not present in the `kwlist` will cause `TypeError` to be raised.

Here is an example module which uses keywords, based on an example by Geoff Philbrick (philbrick@hks.com):

```c
#include "Python.h"

static PyObject *
keywdarg_parrot(PyObject *self, PyObject *args, PyObject *keywds)
{
    int voltage;
    char *state = "a stiff";
    char *action = "voom";
    char *type = "Norwegian Blue";

    static char *kwlist[] = {"voltage", "state", "action", "type", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, keywds, "i|sss", kwlist, &voltage, &state, &action, &type))
        return NULL;

    printf("-- This parrot wouldn't %s if you put %i Volts through it.
           action, voltage);
    printf("-- Lovely plumage, the %s -- It's %s!
           type, state);
```
Py_INCREF(Py_None);
return Py_None;
}

static PyMethodDef keywdarg_methods[] = {
    /* The cast of the function is necessary since PyCFunction values
     * only take two PyObject* parameters, and keywdarg_parrot() takes
     * three.
     */
    {"parrot", (PyCFunction)keywdarg_parrot, METH_VARARGS | METH_KEYWORDS,
    "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL} /* sentinel */
};

void
initkeywdarg(void)
{
    /* Create the module and add the functions */
    Py_InitModule("keywdarg", keywdarg_methods);
}
1.9 Building Arbitrary Values

This function is the counterpart to PyArg_ParseTuple(). It is declared as follows:

```
PyObject *Py_BuildValue(char *format, ...);
```

It recognizes a set of format units similar to the ones recognized by PyArg_ParseTuple(), but the arguments (which are input to the function, not output) must not be pointers, just values. It returns a new Python object, suitable for returning from a C function called from Python.

One difference with PyArg_ParseTuple(): while the latter requires its first argument to be a tuple (since Python argument lists are always represented as tuples internally), Py_BuildValue() does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns None; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

Examples (to the left the call, to the right the resulting Python value):

```
Py_BuildValue(""")           None
Py_BuildValue("i", 123)      123
Py_BuildValue("iii", 123, 456, 789) (123, 456, 789)
Py_BuildValue("s", "hello")  "hello"
Py_BuildValue("ss", "hello", "world") ('hello', 'world')
Py_BuildValue("s#", "hello", 4)  'hell'
Py_BuildValue("()")          ()
Py_BuildValue("(i)", 123)     (123,)
Py_BuildValue("(ii)", 123, 456) (123, 456)
Py_BuildValue("(i,i)", 123, 456) (123, 456)
Py_BuildValue("[i,i]", 123, 456) [123, 456]
Py_BuildValue("{s,i:s:i}",  
     "abc", 123, "def", 456)  {'abc': 123, 'def': 456}
Py_BuildValue("((ii)(ii)) (ii)",
     1, 2, 3, 4, 5, 6)          (((1, 2), (3, 4)), (5, 6))
```

Extending and Embedding the Python Interpreter


toc contents
Release 2.5, documentation updated on 19th September, 2006. 
See About this document... for information on suggesting changes.
1.10 Reference Counts

In languages like C or C++, the programmer is responsible for dynamic allocation and deallocation of memory on the heap. In C, this is done using the functions `malloc()` and `free()`. In C++, the operators `new` and `delete` are used with essentially the same meaning and we'll restrict the following discussion to the C case.

Every block of memory allocated with `malloc()` should eventually be returned to the pool of available memory by exactly one call to `free()`. It is important to call `free()` at the right time. If a block's address is forgotten but `free()` is not called for it, the memory it occupies cannot be reused until the program terminates. This is called a memory leak. On the other hand, if a program calls `free()` for a block and then continues to use the block, it creates a conflict with re-use of the block through another `malloc()` call. This is called using freed memory. It has the same bad consequences as referencing uninitialized data -- core dumps, wrong results, mysterious crashes.

Common causes of memory leaks are unusual paths through the code. For instance, a function may allocate a block of memory, do some calculation, and then free the block again. Now a change in the requirements for the function may add a test to the calculation that detects an error condition and can return prematurely from the function. It's easy to forget to free the allocated memory block when taking this premature exit, especially when it is added later to the code. Such leaks, once introduced, often go undetected for a long time: the error exit is taken only in a small fraction of all calls, and most modern machines have plenty of virtual memory, so the leak only becomes apparent in a long-running process that uses the leaking function frequently. Therefore, it's important to prevent leaks from happening by having a coding convention or strategy that minimizes this kind of errors.

Since Python makes heavy use of `malloc()` and `free()`, it needs a strategy to avoid memory leaks as well as the use of freed memory. The chosen method is called reference counting. The principle is simple: every object contains a counter, which is incremented when a reference to the object is stored somewhere, and which is decremented when a reference to it is deleted. When the counter reaches zero, the last reference to the object has been deleted and the
object is freed.

An alternative strategy is called *automatic garbage collection*. (Sometimes, reference counting is also referred to as a garbage collection strategy, hence my use of ``automatic'' to distinguish the two.) The big advantage of automatic garbage collection is that the user doesn't need to call `free()` explicitly. (Another claimed advantage is an improvement in speed or memory usage -- this is no hard fact however.) The disadvantage is that for C, there is no truly portable automatic garbage collector, while reference counting can be implemented portably (as long as the functions `malloc()` and `free()` are available -- which the C Standard guarantees). Maybe some day a sufficiently portable automatic garbage collector will be available for C. Until then, we'll have to live with reference counts.

While Python uses the traditional reference counting implementation, it also offers a cycle detector that works to detect reference cycles. This allows applications to not worry about creating direct or indirect circular references; these are the weakness of garbage collection implemented using only reference counting. Reference cycles consist of objects which contain (possibly indirect) references to themselves, so that each object in the cycle has a reference count which is non-zero. Typical reference counting implementations are not able to reclaim the memory belonging to any objects in a reference cycle, or referenced from the objects in the cycle, even though there are no further references to the cycle itself.

The cycle detector is able to detect garbage cycles and can reclaim them so long as there are no finalizers implemented in Python (`__del__()` methods). When there are such finalizers, the detector exposes the cycles through the `gc` module (specifically, the `garbage` variable in that module). The `gc` module also exposes a way to run the detector (the `collect()` function), as well as configuration interfaces and the ability to disable the detector at runtime. The cycle detector is considered an optional component; though it is included by default, it can be disabled at build time using the `--without-cycle-gc` option to the `configure` script on UNIX platforms (including Mac OS X) or by removing the definition of `WITH_CYCLE_GC` in the `pyconfig.h` header on other platforms. If the cycle detector is disabled in this way, the `gc` module will not be available.
1.10.1 Reference Counting in Python
1.10.2 Ownership Rules
1.10.3 Thin Ice
1.10.4 NULL Pointers

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1.10.1 Reference Counting in Python

There are two macros, `Py_INCREF(x)` and `Py_DECREF(x)`, which handle the incrementing and decrementing of the reference count. `Py_DECREF()` also frees the object when the count reaches zero. For flexibility, it doesn't call `free()` directly -- rather, it makes a call through a function pointer in the object's type object. For this purpose (and others), every object also contains a pointer to its type object.

The big question now remains: when to use `Py_INCREF(x)` and `Py_DECREF(x)`? Let's first introduce some terms. Nobody "owns" an object; however, you can own a reference to an object. An object's reference count is now defined as the number of owned references to it. The owner of a reference is responsible for calling `Py_DECREF()` when the reference is no longer needed. Ownership of a reference can be transferred. There are three ways to dispose of an owned reference: pass it on, store it, or call `Py_DECREF()`. Forgetting to dispose of an owned reference creates a memory leak.

It is also possible to borrow a reference to an object. The borrower of a reference should not call `Py_DECREF()`. The borrower must not hold on to the object longer than the owner from which it was borrowed. Using a borrowed reference after the owner has disposed of it risks using freed memory and should be avoided completely.

The advantage of borrowing over owning a reference is that you don't need to take care of disposing of the reference on all possible paths through the code -- in other words, with a borrowed reference you don't run the risk of leaking when a premature exit is taken. The disadvantage of borrowing over leaking is that there are some subtle situations where in seemingly correct code a borrowed reference can be used after the owner from which it was borrowed has in fact disposed of it.

A borrowed reference can be changed into an owned reference by calling `Py_INCREF()`. This does not affect the status of the owner from which the reference was borrowed -- it creates a new owned reference, and gives full owner responsibilities (the new owner must dispose of the reference properly, as well as the previous owner).
Footnotes

...borrow\(^1\)\(^2\)

The metaphor of ``borrowing'' a reference is not completely correct: the owner still has a copy of the reference.

... completely.\(^1\)\(^3\)

Checking that the reference count is at least 1 does not work -- the reference count itself could be in freed memory and may thus be reused for another object!
1.10.2 Ownership Rules

Whenever an object reference is passed into or out of a function, it is part of the function's interface specification whether ownership is transferred with the reference or not.

Most functions that return a reference to an object pass on ownership with the reference. In particular, all functions whose function it is to create a new object, such as `PyInt_FromLong()` and `Py_BuildValue()`, pass ownership to the receiver. Even if the object is not actually new, you still receive ownership of a new reference to that object. For instance, `PyInt_FromLong()` maintains a cache of popular values and can return a reference to a cached item.

Many functions that extract objects from other objects also transfer ownership with the reference, for instance `PyObject_GetAttrString()`. The picture is less clear here, however, since a few common routines are exceptions: `PyTuple_GetItem()`, `PyList_GetItem()`, `PyDict_GetItem()`, and `PyDict_GetItemString()` all return references that you borrow from the tuple, list or dictionary.

The function `PyImport_AddModule()` also returns a borrowed reference, even though it may actually create the object it returns: this is possible because an owned reference to the object is stored in `sys.modules`.

When you pass an object reference into another function, in general, the function borrows the reference from you -- if it needs to store it, it will use `Py_INCREF()` to become an independent owner. There are exactly two important exceptions to this rule: ` PyTuple_SetItem()` and `PyList_SetItem()`. These functions take over ownership of the item passed to them -- even if they fail! (Note that `PyDict_SetItem()` and friends don't take over ownership -- they are ``normal."

When a C function is called from Python, it borrows references to its arguments from the caller. The caller owns a reference to the object, so the borrowed reference's lifetime is guaranteed until the function returns. Only when such a borrowed reference must be stored or passed on, it must be turned into an owned reference by calling `Py_INCREF()`.
The object reference returned from a C function that is called from Python must be an owned reference -- ownership is transferred from the function to its caller.
1.10.3 Thin Ice

There are a few situations where seemingly harmless use of a borrowed reference can lead to problems. These all have to do with implicit invocations of the interpreter, which can cause the owner of a reference to dispose of it.

The first and most important case to know about is using `Py_DECREF()` on an unrelated object while borrowing a reference to a list item. For instance:

```c
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    PyList_SetItem(list, 1, PyInt_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

This function first borrows a reference to `list[0]`, then replaces `list[1]` with the value 0, and finally prints the borrowed reference. Looks harmless, right? But it's not!

Let's follow the control flow into `PyList_SetItem()`. The list owns references to all its items, so when item 1 is replaced, it has to dispose of the original item 1. Now let's suppose the original item 1 was an instance of a user-defined class, and let's further suppose that the class defined a `__del__()` method. If this class instance has a reference count of 1, disposing of it will call its `__del__()` method.

Since it is written in Python, the `__del__()` method can execute arbitrary Python code. Could it perhaps do something to invalidate the reference to `item` in `bug()`? You bet! Assuming that the list passed into `bug()` is accessible to the `__del__()` method, it could execute a statement to the effect of `del list[0]`, and assuming this was the last reference to that object, it would free the memory associated with it, thereby invalidating `item`.

The solution, once you know the source of the problem, is easy: temporarily increment the reference count. The correct version of the function reads:

```c
void
```c
no_bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    Py_INCREF(item);
    PyList_SetItem(list, 1, PyInt_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

This is a true story. An older version of Python contained variants of this bug and someone spent a considerable amount of time in a C debugger to figure out why his `__del__()` methods would fail...

The second case of problems with a borrowed reference is a variant involving threads. Normally, multiple threads in the Python interpreter can't get in each other's way, because there is a global lock protecting Python's entire object space. However, it is possible to temporarily release this lock using the macro `Py_BEGIN_ALLOW_THREADS`, and to re-acquire it using `Py_END_ALLOW_THREADS`. This is common around blocking I/O calls, to let other threads use the processor while waiting for the I/O to complete. Obviously, the following function has the same problem as the previous one:

```c
void bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    Py_BEGIN_ALLOW_THREADS
    ...some blocking I/O call...
    Py_END_ALLOW_THREADS
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```
1.10.4 NULL Pointers

In general, functions that take object references as arguments do not expect you to pass them NULL pointers, and will dump core (or cause later core dumps) if you do so. Functions that return object references generally return NULL only to indicate that an exception occurred. The reason for not testing for NULL arguments is that functions often pass the objects they receive on to other function -- if each function were to test for NULL, there would be a lot of redundant tests and the code would run more slowly.

It is better to test for NULL only at the "source:" when a pointer that may be NULL is received, for example, from `malloc()` or from a function that may raise an exception.

The macros `Py_INCREF()` and `Py_DECREF()` do not check for NULL pointers -- however, their variants `Py_XINCREF()` and `Py_XDECREF()` do.

The macros for checking for a particular object type (`Pytype_Check()`) don't check for NULL pointers -- again, there is much code that calls several of these in a row to test an object against various different expected types, and this would generate redundant tests. There are no variants with NULL checking.

The C function calling mechanism guarantees that the argument list passed to C functions (args in the examples) is never NULL -- in fact it guarantees that it is always a tuple.\footnote{1.4}

It is a severe error to ever let a NULL pointer `"escape" to the Python user.

Footnotes

\footnote{1.4}{These guarantees don't hold when you use the "old" style calling convention -- this is still found in much existing code.}
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1.11 Writing Extensions in C++

It is possible to write extension modules in C++. Some restrictions apply. If the main program (the Python interpreter) is compiled and linked by the C compiler, global or static objects with constructors cannot be used. This is not a problem if the main program is linked by the C++ compiler. Functions that will be called by the Python interpreter (in particular, module initialization functions) have to be declared using `extern "C"`. It is unnecessary to enclose the Python header files in `extern "C" { ... }` -- they use this form already if the symbol "__cplusplus" is defined (all recent C++ compilers define this symbol).
1.12 Providing a C API for an Extension Module

Many extension modules just provide new functions and types to be used from Python, but sometimes the code in an extension module can be useful for other extension modules. For example, an extension module could implement a type `collection` which works like lists without order. Just like the standard Python list type has a C API which permits extension modules to create and manipulate lists, this new collection type should have a set of C functions for direct manipulation from other extension modules.

At first sight this seems easy: just write the functions (without declaring them static, of course), provide an appropriate header file, and document the C API. And in fact this would work if all extension modules were always linked statically with the Python interpreter. When modules are used as shared libraries, however, the symbols defined in one module may not be visible to another module. The details of visibility depend on the operating system; some systems use one global namespace for the Python interpreter and all extension modules (Windows, for example), whereas others require an explicit list of imported symbols at module link time (AIX is one example), or offer a choice of different strategies (most Unices). And even if symbols are globally visible, the module whose functions one wishes to call might not have been loaded yet!

Portability therefore requires not to make any assumptions about symbol visibility. This means that all symbols in extension modules should be declared static, except for the module's initialization function, in order to avoid name clashes with other extension modules (as discussed in section 1.4). And it means that symbols that should be accessible from other extension modules must be exported in a different way.

Python provides a special mechanism to pass C-level information (pointers) from one extension module to another one: CObjects. A CObject is a Python data type which stores a pointer (void *). CObjects can only be created and accessed via their C API, but they can be passed around like any other Python object. In particular, they can be assigned to a name in an extension module's namespace. Other extension modules can then import this module, retrieve the
value of this name, and then retrieve the pointer from the CObject.

There are many ways in which CObjects can be used to export the C API of an extension module. Each name could get its own CObject, or all C API pointers could be stored in an array whose address is published in a CObject. And the various tasks of storing and retrieving the pointers can be distributed in different ways between the module providing the code and the client modules.

The following example demonstrates an approach that puts most of the burden on the writer of the exporting module, which is appropriate for commonly used library modules. It stores all C API pointers (just one in the example!) in an array of void pointers which becomes the value of a CObject. The header file corresponding to the module provides a macro that takes care of importing the module and retrieving its C API pointers; client modules only have to call this macro before accessing the C API.

The exporting module is a modification of the spam module from section 1.1. The function spam.system() does not call the C library function system() directly, but a function PySpam_System(), which would of course do something more complicated in reality (such as adding ```spam``` to every command). This function PySpam_System() is also exported to other extension modules.

The function PySpam_System() is a plain C function, declared static like everything else:

```c
static int
PySpam_System(const char *command)
{
    return system(command);
}
```

The function spam_system() is modified in a trivial way:

```c
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = PySpam_System(command);
```
return Py_BuildValue("i", sts);
}

In the beginning of the module, right after the line

#include "Python.h"

two more lines must be added:

#define SPAM_MODULE
#include "spammodule.h"

The #define is used to tell the header file that it is being included in the
exporting module, not a client module. Finally, the module's initialization
function must take care of initializing the C API pointer array:

PyMODINIT_FUNC
initspam(void)
{
    PyObject *m;
    static void *PySpam_API[PySpam_API_pointers];
    PyObject *c_api_object;
    m = Py_InitModule("spam", SpamMethods);

    /* Initialize the C API pointer array */
    PySpam_API[PySpam_System_NUM] = (void *)PySpam_System;

    /* Create a CObject containing the API pointer array's address *
    c_api_object = PyCObject_FromVoidPtr((void *)PySpam_API, NULL);

    if (c_api_object != NULL)
        PyModule_AddObject(m, "_C_API", c_api_object);
}

Note that PySpam_API is declared static; otherwise the pointer array would
disappear when initspam() terminates!

The bulk of the work is in the header file spammodule.h, which looks like this:

#ifndef Py_SPAMMODULE_H
#define Py_SPAMMODULE_H
#ifdef __cplusplus
extern "C" {
#endif

/* Header file for spammodule */
/* C API functions */
#define PySpam_System_NUM 0
#define PySpam_System_RETURN int
#define PySpam_System_PROTO (const char *command)

/* Total number of C API pointers */
#define PySpam_API_pointers 1

#ifdef SPAM_MODULE
/* This section is used when compiling spammodule.c */
static PySpam_System_RETURN PySpam_System PySpam_System_PROTO;
#else
/* This section is used in modules that use spammodule's API */
static void **PySpam_API;
#endif
#define PySpam_System
(*(PySpam_System_RETURN (*)PySpam_System_PROTO) PySpam_API[PySpam_System_NUM])

/* Return -1 and set exception on error, 0 on success. */
static int import_spam(void)
{
    PyObject *module = PyImport_ImportModule("spam");

    if (module != NULL) {
        PyObject *c_api_object = PyObject_GetAttrString(module, ".C_
        if (c_api_object == NULL)
            return -1;
        if (PyCObject_Check(c_api_object))
            PySpam_API = (void **)PyCObject_AsVoidPtr(c_api_object);
        Py_DECREF(c_api_object);
    }
    return 0;
}
#endif

#ifdef __cplusplus
}
#endif
#endif /*!defined(Py_SPAMMODULE_H) */

All that a client module must do in order to have access to the function PySpam_System() is to call the function (or rather macro) import_spam() in its initialization function:
PyMODINIT_FUNC
initclient(void)
{
    PyObject *m;

    Py_InitModule("client", ClientMethods);
    if (import_spam() < 0)
        return;
    /* additional initialization can happen here */
}

The main disadvantage of this approach is that the file `spammodule.h` is rather complicated. However, the basic structure is the same for each function that is exported, so it has to be learned only once.

Finally it should be mentioned that CObjects offer additional functionality, which is especially useful for memory allocation and deallocation of the pointer stored in a CObject. The details are described in the `Python/C API Reference Manual` in the section ``CObjects`` and in the implementation of CObjects (files Include/cobject.h and Objects/cobject.c in the Python source code distribution).
2. Defining New Types

As mentioned in the last chapter, Python allows the writer of an extension module to define new types that can be manipulated from Python code, much like strings and lists in core Python.

This is not hard; the code for all extension types follows a pattern, but there are some details that you need to understand before you can get started.

Note: The way new types are defined changed dramatically (and for the better) in Python 2.2. This document documents how to define new types for Python 2.2 and later. If you need to support older versions of Python, you will need to refer to older versions of this documentation.

Subsections

- 2.1 The Basics
  - 2.1.1 Adding data and methods to the Basic example
  - 2.1.2 Providing finer control over data attributes
  - 2.1.3 Supporting cyclic garbage collection

- 2.2 Type Methods
  - 2.2.1 Finalization and De-allocation
  - 2.2.2 Object Presentation
  - 2.2.3 Attribute Management
    - 2.2.3.1 Generic Attribute Management
    - 2.2.3.2 Type-specific Attribute Management
  - 2.2.4 Object Comparison
  - 2.2.5 Abstract Protocol Support
  - 2.2.6 Weak Reference Support
  - 2.2.7 More Suggestions
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.

Adding data and
2.1 The Basics

The Python runtime sees all Python objects as variables of type `PyObject*`. A `PyObject` is not a very magnificent object - it just contains the refcount and a pointer to the object's ```type object```. This is where the action is; the type object determines which (C) functions get called when, for instance, an attribute gets looked up on an object or it is multiplied by another object. These C functions are called ```type methods``" to distinguish them from things like `[ ] . append` (which we call ```object methods``").

So, if you want to define a new object type, you need to create a new type object.

This sort of thing can only be explained by example, so here's a minimal, but complete, module that defines a new type:

```c
#include <Python.h>

typedef struct {
    PyObject_HEAD
    /* Type-specific fields go here. */
} noddy_NoddyObject;

static PyTypeObject noddy_NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0,          /*ob_size*/
    "noddy.Noddy", /*tp_name*/
    sizeof(noddy_NoddyObject), /*tp_basicsize*/
    0,          /*tp_itemsize*/
    0,          /*tp_dealloc*/
    0,          /*tp_print*/
    0,          /*tp_getattr*/
    0,          /*tp_setattr*/
    0,          /*tp_compare*/
    0,          /*tp_repr*/
    0,          /*tp_as_number*/
    0,          /*tp_as_sequence*/
    0,          /*tp_as_mapping*/
    0,          /*tp_hash */
    0,          /*tp_call*/
    0,          /*tp_str*/
    0,          /*tp_getattro*/
    0,          /*tp_setattro*/
    0,          /*tp_as_buffer*/
    Py_TPFLAGS_DEFAULT, /*tp_flags*/
    "Noddy objects",  /* tp_doc */
};
```
Now that's quite a bit to take in at once, but hopefully bits will seem familiar from the last chapter.

The first bit that will be new is:

```c
typedef struct {
    PyObject_HEAD
} noddy_NoddyObject;
```

This is what a Noddy object will contain—in this case, nothing more than every Python object contains, namely a refcount and a pointer to a type object. These are the fields the PyObject_HEAD macro brings in. The reason for the macro is to standardize the layout and to enable special debugging fields in debug builds. Note that there is no semicolon after the PyObject_HEAD macro; one is included in the macro definition. Be wary of adding one by accident; it's easy to do from habit, and your compiler might not complain, but someone else's probably will! (On Windows, MSVC is known to call this an error and refuse to compile the code.)
For contrast, let's take a look at the corresponding definition for standard Python integers:

```c
typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```

Moving on, we come to the crunch -- the type object.

```c
static PyTypeObject noddy_NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0,    /*ob_size*/
    "noddy.Noddy", /*tp_name*/
    sizeof(noddy_NoddyObject), /*tp_basicsize*/
    0, /*tp_itemsize*/
    0, /*tp_dealloc*/
    0, /*tp_print*/
    0, /*tp_getattr*/
    0, /*tp_setattr*/
    0, /*tp_compare*/
    0, /*tp_repr*/
    0, /*tp_as_number*/
    0, /*tp_as_sequence*/
    0, /*tp_as_mapping*/
    0, /*tp_hash */
    0, /*tp_call*/
    0, /*tp_str*/
    0, /*tp_getattro*/
    0, /*tp_setattro*/
    0, /*tp_as_buffer*/
    Py_TPFLAGS_DEFAULT, /*tp_flags*/
    "Noddy objects", /* tp_doc */
};
```

Now if you go and look up the definition of PyTypeObject in object.h you'll see that it has many more fields that the definition above. The remaining fields will be filled with zeros by the C compiler, and it's common practice to not specify them explicitly unless you need them.

This is so important that we're going to pick the top of it apart still further:

```c
PyObject_HEAD_INIT(NULL)
```

This line is a bit of a wart; what we'd like to write is:

```c
PyObject_HEAD_INIT(&PyType_Type)
```
as the type of a type object is ``type'', but this isn't strictly conforming C and some compilers complain. Fortunately, this member will be filled in for us by PyType_Ready().

    0, /* ob_size */

The ob_size field of the header is not used; its presence in the type structure is a historical artifact that is maintained for binary compatibility with extension modules compiled for older versions of Python. Always set this field to zero.

    "noddy.Noddy", /* tp_name */

The name of our type. This will appear in the default textual representation of our objects and in some error messages, for example:

    >>> "" + noddy.new_noddy()
    Traceback (most recent call last):
      File "<stdin>", line 1, in ?
    TypeError: cannot add type "noddy.Noddy" to string

Note that the name is a dotted name that includes both the module name and the name of the type within the module. The module in this case is noddy and the type is Noddy, so we set the type name to noddy.Noddy.

    sizeof(noddy_NoddyObject), /* tp_basicsize */

This is so that Python knows how much memory to allocate when you call PyObject_New().

**Note:** If you want your type to be subclassable from Python, and your type has the same tp_basicsize as its base type, you may have problems with multiple inheritance. A Python subclass of your type will have to list your type first in its __bases__, or else it will not be able to call your type's __new__ method without getting an error. You can avoid this problem by ensuring that your type has a larger value for tp_basicsize than its base type does. Most of the time, this will be true anyway, because either your base type will be object, or else you will be adding data members to your base type, and therefore increasing its size.

    0, /* tp_itemsize */

This has to do with variable length objects like lists and strings. Ignore this for
now.

Skipping a number of type methods that we don't provide, we set the class flags to Py_TPFLAGS_DEFAULT.

    Py_TPFLAGS_DEFAULT,     /*tp_flags*/

All types should include this constant in their flags. It enables all of the members defined by the current version of Python.

We provide a doc string for the type in tp_doc.

    "Noddy objects",       /* tp_doc */

Now we get into the type methods, the things that make your objects different from the others. We aren't going to implement any of these in this version of the module. We'll expand this example later to have more interesting behavior.

For now, all we want to be able to do is to create new Noddy objects. To enable object creation, we have to provide a tp_new implementation. In this case, we can just use the default implementation provided by the API function PyType_GenericNew(). We'd like to just assign this to the tp_new slot, but we can't, for portability sake, On some platforms or compilers, we can't statically initialize a structure member with a function defined in another C module, so, instead, we'll assign the tp_new slot in the module initialization function just before calling PyType_Ready():

    noddy_NoddyType.tp_new = PyType_GenericNew;
    if (PyType_Ready(&noddy_NoddyType) < 0)
        return;

All the other type methods are NULL, so we'll go over them later -- that's for a later section!

Everything else in the file should be familiar, except for some code in initnoddy():

    if (PyType_Ready(&noddy_NoddyType) < 0)
        return;

This initializes the Noddy type, filing in a number of members, including ob_type that we initially set to NULL.
PyModule_AddObject(m, "Noddy", (PyObject *)&noddy_NoddyType);

This adds the type to the module dictionary. This allows us to create Noddy instances by calling the Noddy class:

```python
>>> import noddy
>>> mynoddy = noddy.Noddy()
```

That's it! All that remains is to build it; put the above code in a file called noddy.c and

```python
from distutils.core import setup, Extension
setup(name="noddy", version="1.0",
      ext_modules=[Extension("noddy", ["noddy.c"])]
)
```

in a file called setup.py; then typing

```bash
$ python setup.py build
```

at a shell should produce a file `noddy.so` in a subdirectory; move to that directory and fire up Python -- you should be able to `import noddy` and play around with Noddy objects.

That wasn't so hard, was it?

Of course, the current Noddy type is pretty uninteresting. It has no data and doesn't do anything. It can't even be subclassed.

---

Subsections

- [2.1.1 Adding data and methods to the Basic example](#)
- [2.1.2 Providing finer control over data attributes](#)
- [2.1.3 Supporting cyclic garbage collection](#)
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
2.1.1 Adding data and methods to the Basic example

Let's expend the basic example to add some data and methods. Let's also make
the type usable as a base class. We'll create a new module, noddy2 that adds
these capabilities:

```c
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} Noddy;

static void
Noddy_dealloc(Noddy* self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    self->ob_type->tp_free((PyObject*)self);
}

static PyObject *
Noddy_new(PyTypeObject *type,
         PyObject *args,
         PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyString_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return self;
}
```
return (PyObject *)self;
}

static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, kwds, |OOi|, kwlist, &first, &last, &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }

    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }

    return 0;
}

static PyMemberDef Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,
        "first name"},
    {"last", T_OBJECT_EX, offsetof(Noddy, last), 0,
        "last name"},
    {"number", T_INT, offsetof(Noddy, number), 0,
        "noddy number"},
    {NULL} /* Sentinel */
};

static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
    }

    return (PyObject *)self;
}
if (format == NULL)
    return NULL;

if (self->first == NULL) {
    PyErr_SetString(PyExc_AttributeError, "first");
    return NULL;
}

if (self->last == NULL) {
    PyErr_SetString(PyExc_AttributeError, "last");
    return NULL;
}

args = Py_BuildValue("OO", self->first, self->last);
if (args == NULL)
    return NULL;

result = PyString_Format(format, args);
Py_DECREF(args);
return result;

static PyMethodDef Noddy_methods[] = {
    {"name", (PyCFunction)Noddy_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};

static PyTypeObject NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0,                  /*ob_size*/
    "noddy.Noddy",     /*tp_name*/
    sizeof(Noddy),     /*tp_basicsize*/
    0,                  /*tp_itemsize*/
    (destructor)Noddy_dealloc, /*tp_dealloc*/
    0,                  /*tp_print*/
    0,                  /*tp_getattr*/
    0,                  /*tp_setattr*/
    0,                  /*tp_compare*/
    0,                  /*tp_repr*/
    0,                  /*tp_as_number*/
    0,                  /*tp_as_sequence*/
    0,                  /*tp_as_mapping*/
    0,                  /*tp_hash */
    0,                  /*tp_call*/
    0,                  /*tp_str*/
    0,                  /*tp_getattro*/
    0,                  /*tp_setattro*/
    0,                  /*tp_as_buffer*/
};
This version of the module has a number of changes.

We've added an extra include:
#include "structmember.h"

This include provides declarations that we use to handle attributes, as described a bit later.

The name of the Noddy object structure has been shortened to Noddy. The type object name has been shortened to NoddyType.

The Noddy type now has three data attributes, first, last, and number. The first and last variables are Python strings containing first and last names. The number attribute is an integer.

The object structure is updated accordingly:

```c
typedef struct {
    PyObject_HEAD
    PyObject *first;
    PyObject *last;
    int number;
} Noddy;
```

Because we now have data to manage, we have to be more careful about object allocation and deallocation. At a minimum, we need a deallocation method:

```c
static void Noddy_dealloc(Noddy* self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    self->ob_type->tp_free((PyObject*)self);
}
```

which is assigned to the tp_dealloc member:

```c
(destructor)Noddy_dealloc, /*tp_dealloc*/
```

This method decrements the reference counts of the two Python attributes. We use Py_XDECREF() here because the first and last members could be NULL. It then calls the tp_free member of the object's type to free the object's memory. Note that the object's type might not be NoddyType, because the object may be an instance of a subclass.

We want to make sure that the first and last names are initialized to empty strings, so we provide a new method:
static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }

        self->last = PyString_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }

        self->number = 0;
    }

    return (PyObject *)self;
}

and install it in the tp_new member:

    Noddy_new,            /* tp_new */

The new member is responsible for creating (as opposed to initializing) objects of the type. It is exposed in Python as the __new__( ) method. See the paper titled "Unifying types and classes in Python" for a detailed discussion of the __new__( ) method. One reason to implement a new method is to assure the initial values of instance variables. In this case, we use the new method to make sure that the initial values of the members first and last are not NULL. If we didn't care whether the initial values were NULL, we could have used PyType_GenericNew() as our new method, as we did before. PyType_GenericNew() initializes all of the instance variable members to NULL.

The new method is a static method that is passed the type being instantiated and any arguments passed when the type was called, and that returns the new object created. New methods always accept positional and keyword arguments, but they often ignore the arguments, leaving the argument handling to initializer
methods. Note that if the type supports subclassing, the type passed may not be the type being defined. The new method calls the tp_alloc slot to allocate memory. We don't fill the tp_alloc slot ourselves. Rather PyType_Ready() fills it for us by inheriting it from our base class, which is object by default. Most types use the default allocation.

**Note:** If you are creating a co-operative tp_new (one that calls a base type's tp_new or __new__), you must not try to determine what method to call using method resolution order at runtime. Always statically determine what type you are going to call, and call its tp_new directly, or via type->tp_base->tp_new. If you do not do this, Python subclasses of your type that also inherit from other Python-defined classes may not work correctly. (Specifically, you may not be able to create instances of such subclasses without getting a TypeError.)

We provide an initialization function:

```c
static int Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist,
        &first, &last, &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XINCREF(tmp);
    }

    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XINCREF(tmp);
    }

    return 0;
}
```
by filling the `tp_init` slot.

```
(initproc)Noddy_init,       /* tp_init */
```

The `tp_init` slot is exposed in Python as the `__init__()` method. It is used to initialize an object after it's created. Unlike the new method, we can't guarantee that the initializer is called. The initializer isn't called when unpickling objects and it can be overridden. Our initializer accepts arguments to provide initial values for our instance. Initializers always accept positional and keyword arguments.

Initializers can be called multiple times. Anyone can call the `__init__()` method on our objects. For this reason, we have to be extra careful when assigning the new values. We might be tempted, for example to assign the `first` member like this:

```
if (first) {
    Py_XDECREF(self->first);
    Py_INCREF(first);
    self->first = first;
}
```

But this would be risky. Our type doesn't restrict the type of the `first` member, so it could be any kind of object. It could have a destructor that causes code to be executed that tries to access the `first` member. To be paranoid and protect ourselves against this possibility, we almost always reassign members before decrementing their reference counts. When don't we have to do this?

- when we absolutely know that the reference count is greater than 1
- when we know that deallocation of the object\(^{2,1}\) will not cause any calls back into our type's code
- when decrementing a reference count in a `tp_dealloc` handler when garbage-collections is not supported\(^{2,2}\)

We want to want to expose our instance variables as attributes. There are a number of ways to do that. The simplest way is to define member definitions:

```
static PyMemberDef Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(Noddy, last), 0,
```
"last name"},
{"number", T_INT, offsetof(Noddy, number), 0,
 "noddy number"},
{NULL} /* Sentinel */
}

and put the definitions in the tp_members slot:

Noddy_members,       /* tp_members */

Each member definition has a member name, type, offset, access flags and documentation string. See the "Generic Attribute Management" section below for details.

A disadvantage of this approach is that it doesn't provide a way to restrict the types of objects that can be assigned to the Python attributes. We expect the first and last names to be strings, but any Python objects can be assigned. Further, the attributes can be deleted, setting the C pointers to NULL. Even though we can make sure the members are initialized to non-NULL values, the members can be set to NULL if the attributes are deleted.

We define a single method, name, that outputs the objects name as the concatenation of the first and last names.

```c
static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }

    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);

```
if (args == NULL)
    return NULL;

result = PyString_Format(format, args);
Py_DECREF(args);

return result;
}

The method is implemented as a C function that takes a Noddy (or Noddy subclass) instance as the first argument. Methods always take an instance as the first argument. Methods often take positional and keyword arguments as well, but in this case we don't take any and don't need to accept a positional argument tuple or keyword argument dictionary. This method is equivalent to the Python method:

    def name(self):
        return "%s %s" % (self.first, self.last)

Note that we have to check for the possibility that our first and last members are NULL. This is because they can be deleted, in which case they are set to NULL. It would be better to prevent deletion of these attributes and to restrict the attribute values to be strings. We'll see how to do that in the next section.

Now that we've defined the method, we need to create an array of method definitions:

    static PyMethodDef Noddy_methods[] = {
        {"name", (PyCFunction)Noddy_name, METH_NOARGS, "Return the name, combining the first and last name"},
        {NULL} /* Sentinel */
    };

and assign them to the tp_methods slot:

    Noddy_methods, /* tp_methods */

Note that we used the METH_NOARGS flag to indicate that the method is passed no arguments.

Finally, we'll make our type usable as a base class. We've written our methods carefully so far so that they don't make any assumptions about the type of the
object being created or used, so all we need to do is to add the
Py_TPFLAGS_BASETYPE to our class flag definition:

    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /*tp_flags*/

We rename initnoddy() to initnoddy2() and update the module name
passed to Py_InitModule3().

Finally, we update our setup.py file to build the new module:

    from distutils.core import setup, Extension
    setup(name="noddy", version="1.0",
          ext_modules=[
            Extension("noddy", ["noddy.c"],
                       Extension("noddy2", ["noddy2.c"],
                       )
          ])"

---

**Footnotes**

... object\(^2.1\)

This is true when we know that the object is a basic type, like a string or a
float.

... supported\(^2.2\)

We relied on this in the tp_dealloc handler in this example, because our
type doesn't support garbage collection. Even if a type supports garbage
collection, there are calls that can be made to \``untrack\" the object from
garbage collection, however, these calls are advanced and not covered here.
2.1.2 Providing finer control over data attributes

In this section, we'll provide finer control over how the first and last attributes are set in the Noddy example. In the previous version of our module, the instance variables first and last could be set to non-string values or even deleted. We want to make sure that these attributes always contain strings.

```c
#include <Python.h>
#include "structmember.h"

typedef struct {
   PyObject_HEAD
   PyObject *first;
   PyObject *last;
   int number;
} Noddy;

static void
Noddy_dealloc(Noddy* self)
{
   Py_XDECREF(self->first);
   Py_XDECREF(self->last);
   self->ob_type->tp_free((PyObject*)self);
}

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
   Noddy *self;

   self = (Noddy *)type->tp_alloc(type, 0);
   if (self != NULL) {
      self->first = PyString_FromString("" );
      if (self->first == NULL) {
         Py_DECREF(self);
         return NULL;
      }
      self->last = PyString_FromString(""");
      if (self->last == NULL) {
         Py_DECREF(self);
         return NULL;
      }
      self->number = 0;
   }
   return (PyObject *)self;
}
```
return (PyObject *)self;
}

static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;
    static char *kwlist[]={"first", "last", "number", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|SSi", kwlist,
                                    &first, &last,
                                    &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }

    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }

    return 0;
}

static PyMemberDef Noddy_members[] = {
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL} /* Sentinel */
};

static PyObject *
Noddy_getfirst(Noddy *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Noddy_setfirst(Noddy *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attrib
        return -1;
    }
if (!PyString_Check(value)) {
    PyErr_SetString(PyExc_TypeError, "The first attribute value must be a string");
    return -1;
}

Py_DECREF(self->first);
Py_INCREF(value);
self->first = value;
return 0;
}

static PyObject *
Noddy_getlast(Noddy *self, void *closure)
{
    Py_INCREF(self->last);
    return self->last;
}

static int
Noddy_setlast(Noddy *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }

    if (!PyString_Check(value)) {
        PyErr_SetString(PyExc_TypeError, "The last attribute value must be a string");
        return -1;
    }

    Py_DECREF(self->last);
    Py_INCREF(value);
    self->last = value;

    return 0;
}

static PyGetSetDef Noddy_getseters[] = {
    {"first", (getter)Noddy_getfirst, (setter)Noddy_setfirst, "first name", NULL},
    {"last", (getter)Noddy_getlast, (setter)Noddy_setlast, "last name", NULL},
};
static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);
    return result;
}

static PyMethodDef Noddy_methods[] = {
    {"name", (PyCFunction)Noddy_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};

static PyTypeObject NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0, /*ob_size*/
    "noddy.Noddy", /*tp_name*/
    sizeof(Noddy), /*tp_basicsize*/
    0, /*tp_itemsize*/
    (destructor)Noddy_dealloc, /*tp_dealloc*/
    0, /*tp_print*/
    0, /*tp_getattr*/
    0, /*tp_setattr*/
    0, /*tp_compare*/
    0, /*tp_repr*/
    0, /*tp_as_number*/
    0, /*tp_as_sequence*/
    0, /*tp_as_mapping*/
    0, /*tp_hash */
    0, /*tp_call*/
    0, /*tp_str*/
    0, /*tp_getattro*/
    0, /*tp_setattro*/
};
To provide greater control, over the first and last attributes, we'll use custom getter and setter functions. Here are the functions for getting and setting
the first attribute:

```c
Noddy_getfirst(Noddy *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}
```

```c
static int
Noddy_setfirst(Noddy *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (! PyString_Check(value)) {
        PyErr_SetString(PyExc_TypeError, "The first attribute value must be a string");
        return -1;
    }
    Py_DECREF(self->first);
    Py_INCREF(value);
    self->first = value;
    return 0;
}
```

The getter function is passed a Noddy object and a ``closure'', which is void pointer. In this case, the closure is ignored. (The closure supports an advanced usage in which definition data is passed to the getter and setter. This could, for example, be used to allow a single set of getter and setter functions that decide the attribute to get or set based on data in the closure.)

The setter function is passed the Noddy object, the new value, and the closure. The new value may be NULL, in which case the attribute is being deleted. In our setter, we raise an error if the attribute is deleted or if the attribute value is not a string.

We create an array of PyGetSetDef structures:

```c
static PyGetSetDef Noddy_getseters[] = {
    "first",
    (getter)Noddy_getfirst, (setter)Noddy_setfirst,
    "first name",
    NULL,
};
```
The last item in a `PyGetSetDef` structure is the closure mentioned above. In this case, we aren't using the closure, so we just pass `NULL`.

We also remove the member definitions for these attributes:

```c
static PyMemberDef Noddy_members[] = {
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL} /* Sentinel */
};
```

We also need to update the `tp_init` handler to only allow strings to be passed:

```c
static int Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;
    static char *kwlist[] = {"first", "last", "number", NULL};
    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|SSi", kwlist, &first, &last, &self->number))
        return -1;
    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}
```
Py_INCREF(last);
    self->last = last;
    Py_DECREF(tmp);
}
    return 0;
}

With these changes, we can assure that the first and last members are never NULL so we can remove checks for NULL values in almost all cases. This means that most of the Py_XDECREF() calls can be converted to Py_DECREF() calls. The only place we can't change these calls is in the deallocator, where there is the possibility that the initialization of these members failed in the constructor.

We also rename the module initialization function and module name in the initialization function, as we did before, and we add an extra definition to the setup.py file.

Footnotes

... strings

We now know that the first and last members are strings, so perhaps we could be less careful about decrementing their reference counts, however, we accept instances of string subclasses. Even though deallocating normal strings won't call back into our objects, we can't guarantee that deallocating an instance of a string subclass won't. call back into out objects.
2.1.3 Supporting cyclic garbage collection

Python has a cyclic-garbage collector that can identify unneeded objects even when their reference counts are not zero. This can happen when objects are involved in cycles. For example, consider:

```python
>>> l = []
>>> l.append(l)
>>> del l
```

In this example, we create a list that contains itself. When we delete it, it still has a reference from itself. Its reference count doesn't drop to zero. Fortunately, Python's cyclic-garbage collector will eventually figure out that the list is garbage and free it.

In the second version of the Noddy example, we allowed any kind of object to be stored in the `first` or `last` attributes. This means that Noddy objects can participate in cycles:

```python
>>> import noddy2
>>> n = noddy2.Noddy()
>>> l = [n]
>>> n.first = l
```

This is pretty silly, but it gives us an excuse to add support for the cyclic-garbage collector to the Noddy example. To support cyclic garbage collection, types need to fill two slots and set a class flag that enables these slots:

```c
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first;
    PyObject *last;
    int number;
} Noddy;

static int
Noddy_traverse(Noddy *self, visitproc visit, void *arg)
{
    int vret;
    if (self->first) {
```
vret = visit(self->first, arg);
    if (vret != 0)
        return vret;
}
if (self->last) {
    vret = visit(self->last, arg);
    if (vret != 0)
        return vret;
}
return 0;
}

static int
Noddy_clear(Noddy *self)
{
    PyObject *tmp;

    tmp = self->first;
    self->first = NULL;
    Py_XDECREF(tmp);
    tmp = self->last;
    self->last = NULL;
    Py_XDECREF(tmp);
    return 0;
}

static void
Noddy_dealloc(Noddy* self)
{
    Noddy_clear(self);
    self->ob_type->tp_free((PyObject*)self);
}

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL)
            return NULL;
    }

    self->last = PyString_FromString("");
    if (self->last == NULL)
```c
{  
    Py_DECREF(self);  
    return NULL;  
}
self->number = 0;
return (PyObject *)self;
}
static int Noddy_init(Noddy *self, PyObject *args, PyObject *kwds) {
    PyObject *first=NULL, *last=NULL, *tmp;
    static char *kwlist[] = {"first", "last", "number", NULL};
    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist, 
                                      &first, &last, 
                                      &self->number))
        return -1;
    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }
    return 0;
}
static PyMemberDef Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,  
     "first name"},
    {"last", T_OBJECT_EX, offsetof(Noddy, last), 0,  
     "last name"},
    {"number", T_INT, offsetof(Noddy, number), 0,  
     "noddy number"},
    {NULL} /* Sentinel */
};
static PyObject * Noddy_name(Noddy* self)
```
{  static PyObject *format = NULL;
   PyObject *args, *result;

   if (format == NULL) {
      format = PyString_FromString("%s %s");
      if (format == NULL)
         return NULL;
   }

   if (self->first == NULL) {
      PyErr_SetString(PyExc_AttributeError, "first");
      return NULL;
   }

   if (self->last == NULL) {
      PyErr_SetString(PyExc_AttributeError, "last");
      return NULL;
   }

   args = Py_BuildValue("OO", self->first, self->last);
   if (args == NULL)
      return NULL;

   result = PyString_Format(format, args);
   Py_DECREF(args);

   return result;
}

static PyMethodDef Noddy_methods[] = {
   {"name", (PyCFunction)Noddy_name, METH_NOARGS,
    "Return the name, combining the first and last name"},
   {NULL} /* Sentinel */
};

static PyTypeObject NoddyType = {
   PyObject_HEAD_INIT(NULL)
   0, /*ob_size*/
   "noddy.Noddy", /*tp_name*/
   sizeof(Noddy), /*tp_basicsize*/
   0, /*tp_itemsize*/
   (destructor)Noddy_dealloc, /*tp_dealloc*/
   0, /*tp_print*/
   0, /*tp_getattr*/
   0, /*tp_setattr*/
   0, /*tp_compare*/
   0, /*tp_repr*/
   0, /*tp_as_number*/
   0, /*tp_as_sequence*/
   0, /*tp_as_mapping*/
};
0,       /*tp_hash */
0,       /*tp_call*/
0,       /*tp_str*/
0,       /*tp_getattro*/
0,       /*tp_setattro*/
0,       /*tp_as_buffer*/
Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,    
"Noddy objects",      /* tp_doc */
(traverseproc)Noddy_traverse,    /* tp_traverse */
(inquiry)Noddy_clear,        /* tp_clear */
0,       /* tp_richcompare */
0,       /* tp_weaklistoffset */
0,       /* tp_iter */
0,       /* tp_iternext */
Noddy_methods,       /* tp_methods */
Noddy_members,       /* tp_members */
0,       /* tp_getset */
0,       /* tp_base */
0,       /* tp_dict */
0,       /* tp_descr_get */
0,       /* tp_descr_set */
0,       /* tp_dictoffset */
(initproc)Noddy_init,    /* tp_init */
0,       /* tp_alloc */
Noddy_new,        /* tp_new */
};

static PyMethodDef module_methods[] = {
   {NULL}  /* Sentinel */
};

#ifndef PyMODINIT_FUNC  /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
initnoddy4(void)
{
    PyObject* m;

    if (PyType_Ready(&NoddyType) < 0)
        return;

    m = Py_InitModule3("noddy4", module_methods,
                      "Example module that creates an extension typ
                      
if (m == NULL)
            return;

    Py_INCREF(&NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&NoddyType);
}
The traversal method provides access to subobjects that could participate in cycles:

```c
static int
Noddy_traverse(Noddy *self, visitproc visit, void *arg)
{
    int vret;
    if (self->first) {
        vret = visit(self->first, arg);
        if (vret != 0)
            return vret;
    }
    if (self->last) {
        vret = visit(self->last, arg);
        if (vret != 0)
            return vret;
    }
    return 0;
}
```

For each subobject that can participate in cycles, we need to call the `visit()` function, which is passed to the traversal method. The `visit()` function takes as arguments the subobject and the extra argument `arg` passed to the traversal method. It returns an integer value that must be returned if it is non-zero.

Python 2.4 and higher provide a `Py_VISIT()` macro that automates calling visit functions. With `Py_VISIT()`, `Noddy_traverse()` can be simplified:

```c
static int
Noddy_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}
```

**Note:** Note that the `tp_traverse` implementation must name its arguments exactly `visit` and `arg` in order to use `Py_VISIT()`. This is to encourage uniformity across these boring implementations.

We also need to provide a method for clearing any subobjects that can participate in cycles. We implement the method and reimplement the deallocator to use it:
Notice the use of a temporary variable in `Noddy_clear()`. We use the temporary variable so that we can set each member to NULL before decrementing its reference count. We do this because, as was discussed earlier, if the reference count drops to zero, we might cause code to run that calls back into the object. In addition, because we now support garbage collection, we also have to worry about code being run that triggers garbage collection. If garbage collection is run, our `tp_traverse` handler could get called. We can't take a chance of having `Noddy_traverse()` called when a member's reference count has dropped to zero and its value hasn't been set to NULL.

Python 2.4 and higher provide a `Py_CLEAR()` that automates the careful decrementing of reference counts. With `Py_CLEAR()`, the `Noddy_clear()` function can be simplified:

```c
static int
Noddy_clear(Noddy *self)
{
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}
```

Finally, we add the `Py_TPFLAGS_HAVE_GC` flag to the class flags:
That's pretty much it. If we had written custom `tp_alloc` or `tp_free` slots, we'd need to modify them for cyclic-garbage collection. Most extensions will use the versions automatically provided.

---

Footnotes

... attributes.\textsuperscript{2.4}

Even in the third version, we aren't guaranteed to avoid cycles. Instances of string subclasses are allowed and string subclasses could allow cycles even if normal strings don't.
2.2 Type Methods

This section aims to give a quick fly-by on the various type methods you can implement and what they do.

Here is the definition of `PyTypeObject`, with some fields only used in debug builds omitted:

```c
typedef struct _typeobject {
    PyObject_VAR_HEAD
    char *tp_name; /* For printing, in format "<module>.<name>" */
    int tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */
    destructor tp_dealloc;
    printfunc tp_print;
   getattrfunc tp_getattr;
   setattrfunc tp_setattr;
    cmpfunc tp_compare;
    reprfunc tp_repr;

    /* Method suites for standard classes */
    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */
    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    long tp_flags;

    char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;
}
```
Now that's a lot of methods. Don't worry too much though - if you have a type you want to define, the chances are very good that you will only implement a handful of these.

As you probably expect by now, we're going to go over this and give more information about the various handlers. We won't go in the order they are defined in the structure, because there is a lot of historical baggage that impacts the ordering of the fields; be sure your type initialization keeps the fields in the right order! It's often easiest to find an example that includes all the fields you
need (even if they're initialized to 0) and then change the values to suit your new type.

```c
char *tp_name; /* For printing */
```

The name of the type - as mentioned in the last section, this will appear in various places, almost entirely for diagnostic purposes. Try to choose something that will be helpful in such a situation!

```c
int tp_basicsize, tp_itemsize; /* For allocation */
```

These fields tell the runtime how much memory to allocate when new objects of this type are created. Python has some built-in support for variable length structures (think: strings, lists) which is where the `tp_itemsize` field comes in. This will be dealt with later.

```c
char *tp_doc;
```

Here you can put a string (or its address) that you want returned when the Python script references `obj.__doc__` to retrieve the doc string.

Now we come to the basic type methods--the ones most extension types will implement.

---

**Subsections**

- [2.2.1 Finalization and De-allocation](#)
- [2.2.2 Object Presentation](#)
- [2.2.3 Attribute Management](#)
- [2.2.3.1 Generic Attribute Management](#)
- [2.2.3.2 Type-specific Attribute Management](#)
- [2.2.4 Object Comparison](#)
- [2.2.5 Abstract Protocol Support](#)
- [2.2.6 Weak Reference Support](#)
- [2.2.7 More Suggestions](#)
2.2.1 Finalization and De-allocation

destructor tp_dealloc;

This function is called when the reference count of the instance of your type is reduced to zero and the Python interpreter wants to reclaim it. If your type has memory to free or other clean-up to perform, put it here. The object itself needs to be freed here as well. Here is an example of this function:

```c
static void
newdatatype_dealloc(newdatatypeobject * obj)
{
    free(obj->obj_UnderlyingDatatypePtr);
    obj->ob_type->tp_free(obj);
}
```

One important requirement of the deallocator function is that it leaves any pending exceptions alone. This is important since deallocators are frequently called as the interpreter unwinds the Python stack; when the stack is unwound due to an exception (rather than normal returns), nothing is done to protect the deallocators from seeing that an exception has already been set. Any actions which a deallocator performs which may cause additional Python code to be executed may detect that an exception has been set. This can lead to misleading errors from the interpreter. The proper way to protect against this is to save a pending exception before performing the unsafe action, and restoring it when done. This can be done using the PyErr_Fetch() and PyErr_Restore() functions:

```c
static void
my_dealloc(PyObject *obj)
{
    MyObject *self = (MyObject *) obj;
    PyObject *cbresult;

    if (self->my_callback != NULL) {
        PyObject *err_type, *err_value, *err_traceback;
        int have_error = PyErr_Occurred() ? 1 : 0;

        if (have_error)
            PyErr_Fetch(&err_type, &err_value, &err_traceback);

        cbresult = PyObject_CallObject(self->my_callback, NULL);
        if (cbresult == NULL)
            PyErr_WriteUnraisable(self->my_callback);
```
else
    Py_DECREF(cbresult);

    if (have_error)
        PyErr_Restore(err_type, err_value, err_traceback);

    Py_DECREF(self->my_callback);
}
obj->ob_type->tp_free((PyObject*)self);
2.2.2 Object Presentation

In Python, there are three ways to generate a textual representation of an object: the `repr()` function (or equivalent back-tick syntax), the `str()` function, and the `print` statement. For most objects, the `print` statement is equivalent to the `str()` function, but it is possible to special-case printing to a `FILE*` if necessary; this should only be done if efficiency is identified as a problem and profiling suggests that creating a temporary string object to be written to a file is too expensive.

These handlers are all optional, and most types at most need to implement the `tp_str` and `tp_repr` handlers.

```c
reprfunc tp_repr;
reprfunc tp_str;
printfunc tp_print;
```

The `tp_repr` handler should return a string object containing a representation of the instance for which it is called. Here is a simple example:

```c
static PyObject *
newdatatype_repr(newdatatypeobject * obj)
{
    return PyString_FromFormat("Repr-ified_newdatatype{{size:\%d}}", obj->obj_UnderlyingDatatypePtr->size);
}
```

If no `tp_repr` handler is specified, the interpreter will supply a representation that uses the type's `tp_name` and a uniquely-identifying value for the object.

The `tp_str` handler is to `str()` what the `tp_repr` handler described above is to `repr()`; that is, it is called when Python code calls `str()` on an instance of your object. Its implementation is very similar to the `tp_repr` function, but the resulting string is intended for human consumption. If `tp_str` is not specified, the `tp_repr` handler is used instead.

Here is a simple example:

```c
static PyObject *
newdatatype_str(newdatatypeobject * obj)
{
```
The print function will be called whenever Python needs to "print" an instance of the type. For example, if 'node' is an instance of type TreeNode, then the print function is called when Python code calls:

```
print node
```

There is a flags argument and one flag, Py_PRINT_RAW, and it suggests that you print without string quotes and possibly without interpreting escape sequences.

The print function receives a file object as an argument. You will likely want to write to that file object.

Here is a sample print function:

```
static int
newdatatype_print(newdatatypeobject *obj, FILE *fp, int flags)
{
    if (flags & Py_PRINT_RAW) {
        fprintf(fp, "<{newdatatype object--size: %d}>",
                obj->obj_UnderlyingDatatypePtr->size);
    } else {
        fprintf(fp, ""{"<newdatatype object--size: %d}>"",
                obj->obj_UnderlyingDatatypePtr->size);
    }
    return 0;
}
```
2.2.3 Attribute Management

For every object which can support attributes, the corresponding type must provide the functions that control how the attributes are resolved. There needs to be a function which can retrieve attributes (if any are defined), and another to set attributes (if setting attributes is allowed). Removing an attribute is a special case, for which the new value passed to the handler is NULL.

Python supports two pairs of attribute handlers; a type that supports attributes only needs to implement the functions for one pair. The difference is that one pair takes the name of the attribute as a char*, while the other accepts a PyObject*. Each type can use whichever pair makes more sense for the implementation's convenience.

```c
getattrfunc   tp_getattr;    /* char * version */
setattrfunc    tp_setattr;
/* ... */
gettattrofunc  tp_getattrofunc;   /* PyObject * version */
setattrofunc  tp_setattrofunc;
```

If accessing attributes of an object is always a simple operation (this will be explained shortly), there are generic implementations which can be used to provide the PyObject* version of the attribute management functions. The actual need for type-specific attribute handlers almost completely disappeared starting with Python 2.2, though there are many examples which have not been updated to use some of the new generic mechanism that is available.

---

Subsections

- [2.2.3.1 Generic Attribute Management](#)
- [2.2.3.2 Type-specific Attribute Management](#)
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
## 2.2.4 Object Comparison

```c
    cmpfunc tp_compare;
```

The `tp_compare` handler is called when comparisons are needed and the object does not implement the specific rich comparison method which matches the requested comparison. (It is always used if defined and the `PyObject_Compare()` or `PyObject_Cmp()` functions are used, or if `cmp()` is used from Python.) It is analogous to the `__cmp__()` method. This function should return -1 if `obj1` is less than `obj2`, 0 if they are equal, and 1 if `obj1` is greater than `obj2`. (It was previously allowed to return arbitrary negative or positive integers for less than and greater than, respectively; as of Python 2.2, this is no longer allowed. In the future, other return values may be assigned a different meaning.)

A `tp_compare` handler may raise an exception. In this case it should return a negative value. The caller has to test for the exception using `PyErr_Occurred()`.

Here is a sample implementation:

```c
    static int newdatatype_compare(newdatatypeobject * obj1, newdatatypeobject * ob
    { long result;

    if (obj1->obj_UnderlyingDatatypePtr->size <
          obj2->obj_UnderlyingDatatypePtr->size) {
            result = -1;
    }
    else if (obj1->obj_UnderlyingDatatypePtr->size >
          obj2->obj_UnderlyingDatatypePtr->size) {
            result = 1;
    }
    else {
            result = 0;
    }
    return result;
    }
```

---

**Extending and Embedding the Python**
2.2.5 Abstract Protocol Support

Python supports a variety of abstract `protocols;' the specific interfaces provided to use these interfaces are documented in the *Python/C API Reference Manual* in the chapter ``Abstract Objects Layer.''

A number of these abstract interfaces were defined early in the development of the Python implementation. In particular, the number, mapping, and sequence protocols have been part of Python since the beginning. Other protocols have been added over time. For protocols which depend on several handler routines from the type implementation, the older protocols have been defined as optional blocks of handlers referenced by the type object. For newer protocols there are additional slots in the main type object, with a flag bit being set to indicate that the slots are present and should be checked by the interpreter. (The flag bit does not indicate that the slot values are non-NULL. The flag may be set to indicate the presence of a slot, but a slot may still be unfilled.)

```c
PyNumberMethods tp_as_number;
PySequenceMethods tp_as_sequence;
PyMappingMethods tp_as_mapping;
```

If you wish your object to be able to act like a number, a sequence, or a mapping object, then you place the address of a structure that implements the C type PyNumberMethods, PySequenceMethods, or PyMappingMethods, respectively. It is up to you to fill in this structure with appropriate values. You can find examples of the use of each of these in the Objects directory of the Python source distribution.

```c
hashfunc tp_hash;
```

This function, if you choose to provide it, should return a hash number for an instance of your data type. Here is a moderately pointless example:

```c
static long
newdatatypenumberobject_hash(newdatatypenumberobject *obj)
{
    long result;
    result = obj->obj_UnderlyingDatatypePtr->size;
    result = result * 3;
    return result;
}
```
ternaryfunc tp_call;

This function is called when an instance of your data type is "called", for example, if obj1 is an instance of your data type and the Python script contains obj1('hello'), the tp_call handler is invoked.

This function takes three arguments:

1. arg1 is the instance of the data type which is the subject of the call. If the call is obj1('hello'), then arg1 is obj1.

2. arg2 is a tuple containing the arguments to the call. You can use PyArg_ParseTuple() to extract the arguments.

3. arg3 is a dictionary of keyword arguments that were passed. If this is non-NULL and you support keyword arguments, use PyArg_ParseTupleAndKeywords() to extract the arguments. If you do not want to support keyword arguments and this is non-NULL, raise a TypeError with a message saying that keyword arguments are not supported.

Here is a desultory example of the implementation of the call function.

```c
/* Implement the call function.
 * obj1 is the instance receiving the call.
 * obj2 is a tuple containing the arguments to the call, in this case 3 strings.
 */
static PyObject *
newdatatype_call(newdatatypeobject *obj, PyObject *args, PyObject *other)
{
    PyObject *result;
    char *arg1;
    char *arg2;
    char *arg3;

    if (!PyArg_ParseTuple(args, "sss:call", &arg1, &arg2, &arg3)) {
        return NULL;
    }
    result = PyString_FromFormat(
        " obj->obj_UnderlyingDatatypePtr->size, arg1, arg2, arg3);
    printf("%s", PyString_AS_STRING(result));
    return result;
```
XXX some fields need to be added here...

    /* Added in release 2.2 */
    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

These functions provide support for the iterator protocol. Any object which
wishes to support iteration over its contents (which may be generated during
iteration) must implement the tp_iter handler. Objects which are returned by
a tp_iter handler must implement both the tp_iter and tp_iternext
handlers. Both handlers take exactly one parameter, the instance for which they
are being called, and return a new reference. In the case of an error, they should
set an exception and return NULL.

For an object which represents an iterable collection, the tp_iter handler
must return an iterator object. The iterator object is responsible for maintaining
the state of the iteration. For collections which can support multiple iterators
which do not interfere with each other (as lists and tuples do), a new iterator
should be created and returned. Objects which can only be iterated over once
(usually due to side effects of iteration) should implement this handler by
returning a new reference to themselves, and should also implement the
tp_iternext handler. File objects are an example of such an iterator.

Iterator objects should implement both handlers. The tp_iter handler should
return a new reference to the iterator (this is the same as the tp_iter handler
for objects which can only be iterated over destructively). The tp_iternext
handler should return a new reference to the next object in the iteration if there is
one. If the iteration has reached the end, it may return NULL without setting an
exception or it may set StopIteration; avoiding the exception can yield
slightly better performance. If an actual error occurs, it should set an exception
and return NULL.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
2.2.6 Weak Reference Support

One of the goals of Python's weak-reference implementation is to allow any type to participate in the weak reference mechanism without incurring the overhead on those objects which do not benefit by weak referencing (such as numbers).

For an object to be weakly referencable, the extension must include a PyObject* field in the instance structure for the use of the weak reference mechanism; it must be initialized to NULL by the object's constructor. It must also set the tp_weaklistoffset field of the corresponding type object to the offset of the field. For example, the instance type is defined with the following structure:

```
typedef struct {
    PyObject_HEAD
    PyClassObject *in_class;  /* The class object */
    PyObject * in_dict;     /* A dictionary */
    PyObject * in_weakreflist; /* List of weak references */
} PyInstanceObject;
```

The statically-declared type object for instances is defined this way:

```
PyTypeObject PyInstance_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "module.instance",
    /* Lots of stuff omitted for brevity... */
    Py_TPFLAGS_DEFAULT,       /* tp_flags */
    0,                        /* tp_doc */
    0,                        /* tp_traverse */
    0,                        /* tp_clear */
    0,                        /* tp_richcompare */
    offsetof(PyInstanceObject, in_weakreflist), /* tp_weaklistoffset */
};
```

The type constructor is responsible for initializing the weak reference list to NULL:

```
static PyObject *
instance_new() {
    /* Other initialization stuff omitted for brevity */
```
The only further addition is that the destructor needs to call the weak reference manager to clear any weak references. This should be done before any other parts of the destruction have occurred, but is only required if the weak reference list is non-NULL:

```c
static void
instance_dealloc(PyInstanceObject *inst)
{
    /* Allocate temporaries if needed, but do not begin
destruction just yet. */

    if (inst->in_weakreflist != NULL)
        PyObject_ClearWeakRefs((PyObject *) inst);

    /* Proceed with object destruction normally. */
}
```

Extending and Embedding the Python Interpreter

Previous: 2.2.5 Abstract Protocol Support  Up: 2.2 Type Methods  Next: 2.2.7

More Suggestions

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
2.2.7 More Suggestions

Remember that you can omit most of these functions, in which case you provide 0 as a value. There are type definitions for each of the functions you must provide. They are in object.h in the Python include directory that comes with the source distribution of Python.

In order to learn how to implement any specific method for your new data type, do the following: Download and unpack the Python source distribution. Go the Objects directory, then search the C source files for tp_ plus the function you want (for example, tp_print or tp_compare). You will find examples of the function you want to implement.

When you need to verify that an object is an instance of the type you are implementing, use the PyObject_TypeCheck function. A sample of its use might be something like the following:

```c
if (!PyObject_TypeCheck(some_object, &MyType)) {
    PyErr_SetString(PyExc_TypeError, "arg #1 not a mything");
    return NULL;
}
```

---

Release 2.5, documentation updated on 19th September, 2006.  
See About this document... for information on suggesting changes.
3. Building C and C++ Extensions with distutils

Starting in Python 1.4, Python provides, on UNIX, a special make file for building make files for building dynamically-linked extensions and custom interpreters. Starting with Python 2.0, this mechanism (known as related to Makefile.pre.in, and Setup files) is no longer supported. Building custom interpreters was rarely used, and extension modules can be built using distutils.

Building an extension module using distutils requires that distutils is installed on the build machine, which is included in Python 2.x and available separately for Python 1.5. Since distutils also supports creation of binary packages, users don't necessarily need a compiler and distutils to install the extension.

A distutils package contains a driver script, setup.py. This is a plain Python file, which, in the most simple case, could look like this:

```python
from distutils.core import setup, Extension

module1 = Extension('demo',
                      sources = ['demo.c'])

setup (name = 'PackageName',
       version = '1.0',
       description = 'This is a demo package',
       ext_modules = [module1])
```

With this setup.py, and a file demo.c, running

```bash
python setup.py build
```

will compile demo.c, and produce an extension module named "demo" in the build directory. Depending on the system, the module file will end up in a subdirectory build/lib.system, and may have a name like demo.so or demo.pyd.

In the setup.py, all execution is performed by calling the "setup" function. This takes a variable number of keyword arguments, of which the example above uses only a subset. Specifically, the example specifies meta-information to
build packages, and it specifies the contents of the package. Normally, a package will contain of addition modules, like Python source modules, documentation, subpackages, etc. Please refer to the distutils documentation in *Distributing Python Modules* to learn more about the features of distutils; this section explains building extension modules only.

It is common to pre-compute arguments to `setup`, to better structure the driver script. In the example above, the "ext_modules" argument to `setup` is a list of extension modules, each of which is an instance of the `Extension`. In the example, the instance defines an extension named "demo" which is build by compiling a single source file, demo.c.

In many cases, building an extension is more complex, since additional preprocessor defines and libraries may be needed. This is demonstrated in the example below.

```python
from distutils.core import setup, Extension

module1 = Extension('demo',
    define_macros = [('MAJOR_VERSION', '1'),
                     ('MINOR_VERSION', '0')],
    include_dirs = ['/usr/local/include'],
    libraries = ['tcl83'],
    library_dirs = ['/usr/local/lib'],
    sources = ['demo.c'])

setup (name = 'PackageName',
    version = '1.0',
    description = 'This is a demo package',
    author = 'Martin v. Loewis',
    author_email = 'martin@v.loewis.de',
    url = 'http://www.python.org/doc/current/ext/building.html',
    long_description = '''
This is really just a demo package.
''',
    ext_modules = [module1])
```

In this example, `setup` is called with additional meta-information, which is recommended when distribution packages have to be built. For the extension itself, it specifies preprocessor defines, include directories, library directories, and libraries. Depending on the compiler, distutils passes this information in different ways to the compiler. For example, on UNIX, this may result in the compilation commands

```bash
gcc -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -fPIC -DMAJOR_VERSION=
```
gcc -shared build/temp.linux-i686-2.2/demo.o -L/usr/local/lib -ltcl8

These lines are for demonstration purposes only; distutils users should trust that distutils gets the invocations right.

Subsections

- 3.1 Distributing your extension modules

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
3.1 Distributing your extension modules

When an extension has been successfully build, there are three ways to use it.

End-users will typically want to install the module, they do so by running

    python setup.py install

Module maintainers should produce source packages; to do so, they run

    python setup.py sdist

In some cases, additional files need to be included in a source distribution; this is done through a MANIFEST.in file; see the distutils documentation for details.

If the source distribution has been build successfully, maintainers can also create binary distributions. Depending on the platform, one of the following commands can be used to do so.

    python setup.py bdist_wininst
    python setup.py bdist_rpm
    python setup.py bdist_dumb

4. Building C and C++ Extensions on Windows

This chapter briefly explains how to create a Windows extension module for Python using Microsoft Visual C++, and follows with more detailed background information on how it works. The explanatory material is useful for both the Windows programmer learning to build Python extensions and the UNIX programmer interested in producing software which can be successfully built on both UNIX and Windows.

Module authors are encouraged to use the distutils approach for building extension modules, instead of the one described in this section. You will still need the C compiler that was used to build Python; typically Microsoft Visual C++.

Note: This chapter mentions a number of filenames that include an encoded Python version number. These filenames are represented with the version number shown as "XY"; in practice, "X" will be the major version number and "Y" will be the minor version number of the Python release you're working with. For example, if you are using Python 2.2.1, "XY" will actually be "22".

Subsections

- 4.1 A Cookbook Approach
- 4.2 Differences Between UNIX and Windows
- 4.3 Using DLLs in Practice

Extending and Embedding the Python Interpreter
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.1 A Cookbook Approach

There are two approaches to building extension modules on Windows, just as there are on UNIX: use the distutils package to control the build process, or do things manually. The distutils approach works well for most extensions; documentation on using distutils to build and package extension modules is available in Distributing Python Modules. This section describes the manual approach to building Python extensions written in C or C++.

To build extensions using these instructions, you need to have a copy of the Python sources of the same version as your installed Python. You will need Microsoft Visual C++ "Developer Studio"; project files are supplied for VC++ version 7.1, but you can use older versions of VC++. Notice that you should use the same version of VC++ that was used to build Python itself. The example files described here are distributed with the Python sources in the PC\example_nt directory.

1. **Copy the example files**
   The example_nt directory is a subdirectory of the PC directory, in order to keep all the PC-specific files under the same directory in the source distribution. However, the example_nt directory can't actually be used from this location. You first need to copy or move it up one level, so that example_nt is a sibling of the PC and Include directories. Do all your work from within this new location.

2. **Open the project**
   From VC++, use the File > Open Solution dialog (not File > Open!). Navigate to and select the file example.sln, in the copy of the example_nt directory you made above. Click Open.

3. **Build the example DLL**
   In order to check that everything is set up right, try building:

   1. Select a configuration. This step is optional. Choose Build > Configuration Manager > Active Solution Configuration and select either Release or Debug. If you skip this step, VC++ will use the Debug configuration by default.
2. Build the DLL. Choose Build > Build Solution. This creates all intermediate and result files in a subdirectory called either Debug or Release, depending on which configuration you selected in the preceding step.

4. Testing the debug-mode DLL
Once the Debug build has succeeded, bring up a DOS box, and change to the example_nt\Debug directory. You should now be able to repeat the following session (C> is the DOS prompt, >>> is the Python prompt; note that build information and various debug output from Python may not match this screen dump exactly):

```
C> ..\.\PCbuild\python_d
Adding parser accelerators ...
Done.
Python 2.2 (#28, Dec 19 2001, 23:26:37) [MSC 32 bit (Intel)] on
Type "copyright", "credits" or "license" for more information.
>>> import example
[4897 refs]
>>> example.foo()
Hello, world
[4903 refs]
>>>
```

Congratulations! You've successfully built your first Python extension module.

5. Creating your own project
Choose a name and create a directory for it. Copy your C sources into it. Note that the module source file name does not necessarily have to match the module name, but the name of the initialization function should match the module name -- you can only import a module spam if its initialization function is called initspam(), and it should call Py_InitModule() with the string "spam" as its first argument (use the minimal example.c in this directory as a guide). By convention, it lives in a file called spam.c or spammodule.c. The output file should be called spam.dll or spam.pyd (the latter is supported to avoid confusion with a system library spam.dll to which your module could be a Python interface) in Release mode, or spam_d.dll or spam_d.pyd in Debug mode.

Now your options are:
1. Copy `example.sln` and `example.vcproj`, rename them to `spam.*`, and edit them by hand, or
2. Create a brand new project; instructions are below.

In either case, copy `example nt\example.def` to `spam\spam.def`, and edit the new `spam.def` so its second line contains the string `\initspam`. If you created a new project yourself, add the file `spam.def` to the project now. (This is an annoying little file with only two lines. An alternative approach is to forget about the `.def` file, and add the option `/export:initspam` somewhere to the Link settings, by manually editing the setting in Project Properties dialog).

6. **Creating a brand new project**
   Use the File > New > Project dialog to create a new Project Workspace. Select Visual C++ Projects/Win32/ Win32 Project, enter the name ("spam"), and make sure the Location is set to parent of the spam directory you have created (which should be a direct subdirectory of the Python build tree, a sibling of `Include` and `PC`). Select Win32 as the platform (in my version, this is the only choice). Make sure the Create new workspace radio button is selected. Click OK.

   You should now create the file `spam.def` as instructed in the previous section. Add the source files to the project, using Project > Add Existing Item. Set the pattern to *. * and select both `spam.c` and `spam.def` and click OK. (Inserting them one by one is fine too.)

   Now open the Project > spam properties dialog. You only need to change a few settings. Make sure All Configurations is selected from the Settings for: dropdown list. Select the C/C++ tab. Choose the General category in the popup menu at the top. Type the following text in the entry box labeled Additional Include Directories:

   ```
   ..\Include, ..\PC
   ```

   Then, choose the General category in the Linker tab, and enter

   ```
   ..\PCbuild
   ```

   in the text box labelled Additional library Directories.
Now you need to add some mode-specific settings:

Select Release in the Configuration dropdown list. Choose the Link tab, choose the Input category, and append `pythonXY.lib` to the list in the Additional Dependencies box.

Select Debug in the Configuration dropdown list, and append `pythonXY_d.lib` to the list in the Additional Dependencies box. Then click the C/C++ tab, select Code Generation, and select Multi-threaded Debug DLL from the Runtime library dropdown list.

Select Release again from the Configuration dropdown list. Select Multi-threaded DLL from the Runtime library dropdown list.

If your module creates a new type, you may have trouble with this line:

```c
PyObject_HEAD_INIT(&PyType_Type)
```

Change it to:

```c
PyObject_HEAD_INIT(NULL)
```

and add the following to the module initialization function:

```c
MyObject_Type.ob_type = &PyType_Type;
```

Refer to section 3 of the [Python FAQ](#) for details on why you must do this.
4.2 Differences Between **UNIX** and **Windows**

**UNIX** and Windows use completely different paradigms for run-time loading of code. Before you try to build a module that can be dynamically loaded, be aware of how your system works.

In **UNIX**, a shared object (.so) file contains code to be used by the program, and also the names of functions and data that it expects to find in the program. When the file is joined to the program, all references to those functions and data in the file's code are changed to point to the actual locations in the program where the functions and data are placed in memory. This is basically a link operation.

In Windows, a dynamic-link library (.dll) file has no dangling references. Instead, an access to functions or data goes through a lookup table. So the DLL code does not have to be fixed up at runtime to refer to the program's memory; instead, the code already uses the DLL's lookup table, and the lookup table is modified at runtime to point to the functions and data.

In **UNIX**, there is only one type of library file (.a) which contains code from several object files (.o). During the link step to create a shared object file (.so), the linker may find that it doesn't know where an identifier is defined. The linker will look for it in the object files in the libraries; if it finds it, it will include all the code from that object file.

In Windows, there are two types of library, a static library and an import library (both called .lib). A static library is like a **UNIX** .a file; it contains code to be included as necessary. An import library is basically used only to reassure the linker that a certain identifier is legal, and will be present in the program when the DLL is loaded. So the linker uses the information from the import library to build the lookup table for using identifiers that are not included in the DLL. When an application or a DLL is linked, an import library may be generated, which will need to be used for all future DLLs that depend on the symbols in the application or DLL.

Suppose you are building two dynamic-load modules, B and C, which should
share another block of code A. On UNIX, you would *not* pass A.a to the linker for B.so and C.so; that would cause it to be included twice, so that B and C would each have their own copy. In Windows, building A.dll will also build A.lib. You *do* pass A.lib to the linker for B and C. A.lib does not contain code; it just contains information which will be used at runtime to access A's code.

In Windows, using an import library is sort of like using "import spam"; it gives you access to spam's names, but does not create a separate copy. On UNIX, linking with a library is more like "from spam import *"; it does create a separate copy.
4.3 Using DLLs in Practice

Windows Python is built in Microsoft Visual C++; using other compilers may or may not work (though Borland seems to). The rest of this section is MSVC++ specific.

When creating DLLs in Windows, you must pass pythonXY.lib to the linker. To build two DLLs, spam and ni (which uses C functions found in spam), you could use these commands:

```
cl /LD /I/python/include spam.c ../libs/pythonXY.lib
cl /LD /I/python/include ni.c spam.lib ../libs/pythonXY.lib
```

The first command created three files: spam.obj, spam.dll and spam.lib. Spam.dll does not contain any Python functions (such as PyArg_ParseTuple()), but it does know how to find the Python code thanks to pythonXY.lib.

The second command created ni.dll (and .obj and .lib), which knows how to find the necessary functions from spam, and also from the Python executable.

Not every identifier is exported to the lookup table. If you want any other modules (including Python) to be able to see your identifiers, you have to say "declspec(dllexport)", as in "void _declspec(dllexport) initspam(void)" or "PyObject _declspec(dllexport) * NiGetSpamData(void)".

Developer Studio will throw in a lot of import libraries that you do not really need, adding about 100K to your executable. To get rid of them, use the Project Settings dialog, Link tab, to specify ignore default libraries. Add the correct msvcrxx.lib to the list of libraries.
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
5. Embedding Python in Another Application

The previous chapters discussed how to extend Python, that is, how to extend the functionality of Python by attaching a library of C functions to it. It is also possible to do it the other way around: enrich your C/C++ application by embedding Python in it. Embedding provides your application with the ability to implement some of the functionality of your application in Python rather than C or C++. This can be used for many purposes; one example would be to allow users to tailor the application to their needs by writing some scripts in Python. You can also use it yourself if some of the functionality can be written in Python more easily.

Embedding Python is similar to extending it, but not quite. The difference is that when you extend Python, the main program of the application is still the Python interpreter, while if you embed Python, the main program may have nothing to do with Python -- instead, some parts of the application occasionally call the Python interpreter to run some Python code.

So if you are embedding Python, you are providing your own main program. One of the things this main program has to do is initialize the Python interpreter. At the very least, you have to call the function `Py_Initialize()` (on Mac OS, call `PyMac_Initialize()` instead). There are optional calls to pass command line arguments to Python. Then later you can call the interpreter from any part of the application.

There are several different ways to call the interpreter: you can pass a string containing Python statements to `PyRun_SimpleString()`, or you can pass a stdio file pointer and a file name (for identification in error messages only) to `PyRun_SimpleFile()`. You can also call the lower-level operations described in the previous chapters to construct and use Python objects.

A simple demo of embedding Python can be found in the directory `Demo/embed/` of the source distribution.
See Also:

*Python/C API Reference Manual*

The details of Python's C interface are given in this manual. A great deal of necessary information can be found here.

Subsections

- 5.1 Very High Level Embedding
- 5.2 Beyond Very High Level Embedding: An overview
- 5.3 Pure Embedding
- 5.4 Extending Embedded Python
- 5.5 Embedding Python in C++
- 5.6 Linking Requirements

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
5.1 Very High Level Embedding

The simplest form of embedding Python is the use of the very high level interface. This interface is intended to execute a Python script without needing to interact with the application directly. This can for example be used to perform some operation on a file.

```c
#include <Python.h>

int main(int argc, char *argv[])
{
    Py_Initialize();
    PyRun_SimpleString("from time import time,ctime\n":
                 "print 'Today is',ctime(time())\n");
    Py_Finalize();
    return 0;
}
```

The above code first initializes the Python interpreter with `Py_Initialize()`, followed by the execution of a hard-coded Python script that print the date and time. Afterwards, the `Py_Finalize()` call shuts the interpreter down, followed by the end of the program. In a real program, you may want to get the Python script from another source, perhaps a text-editor routine, a file, or a database. Getting the Python code from a file can better be done by using the `PyRun_SimpleFile()` function, which saves you the trouble of allocating memory space and loading the file contents.
5.2 Beyond Very High Level Embedding: An overview

The high level interface gives you the ability to execute arbitrary pieces of Python code from your application, but exchanging data values is quite cumbersome to say the least. If you want that, you should use lower level calls. At the cost of having to write more C code, you can achieve almost anything.

It should be noted that extending Python and embedding Python is quite the same activity, despite the different intent. Most topics discussed in the previous chapters are still valid. To show this, consider what the extension code from Python to C really does:

1. Convert data values from Python to C,
2. Perform a function call to a C routine using the converted values, and
3. Convert the data values from the call from C to Python.

When embedding Python, the interface code does:

1. Convert data values from C to Python,
2. Perform a function call to a Python interface routine using the converted values, and
3. Convert the data values from the call from Python to C.

As you can see, the data conversion steps are simply swapped to accommodate the different direction of the cross-language transfer. The only difference is the routine that you call between both data conversions. When extending, you call a C routine, when embedding, you call a Python routine.

This chapter will not discuss how to convert data from Python to C and vice versa. Also, proper use of references and dealing with errors is assumed to be understood. Since these aspects do not differ from extending the interpreter, you can refer to earlier chapters for the required information.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
The first program aims to execute a function in a Python script. Like in the section about the very high level interface, the Python interpreter does not directly interact with the application (but that will change in the next section).

The code to run a function defined in a Python script is:

```c
#include <Python.h>

int
main(int argc, char *argv[])
{
    PyObject *pName, *pModule, *pDict, *pFunc;
    PyObject *pArgs, *pValue;
    int i;

    if (argc < 3) {
        fprintf(stderr,"Usage: call pythonfile funcname [args]\n");
        return 1;
    }

    Py_Initialize();
    pName = PyString_FromString(argv[1]);
    /* Error checking of pName left out */

    pModule = PyImport_Import(pName);
    Py_DECREF(pName);

    if (pModule != NULL) {
        pFunc = PyObject_GetAttrString(pModule, argv[2]);
        /* pFunc is a new reference */

        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyInt_FromLong(atoi(argv[i + 3]));
                if (!pValue) {
                    Py_DECREF(pArgs);
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert argument\n");
                    return 1;
                }
                /* pValue reference stolen here: */
                PyTuple_SetItem(pArgs, i, pValue);
            }
            /* pValue reference stolen here: */
            PyObject_CallObject(pFunc, pArgs);
        }
    }
    pValue = PyObject_CallObject(pFunc, pArgs);
}```
Py_DECREF(pArgs);
if (pValue != NULL) {
    printf("Result of call: %ld\n", PyInt_AsLong(pValue))
    Py_DECREF(pValue);
} else {
    Py_DECREF(pFunc);
    Py_DECREF(pModule);
    PyErr_Print();
    fprintf(stderr,"Call failed\n");
    return 1;
}
} else {
    if (PyErr_Occurred())
        PyErr_Print();
    fprintf(stderr, "Cannot find function "\%s"\n", argv[2]);
    Py_XDECREF(pFunc);
    Py_DECREF(pModule);
} else {
    PyErr_Print();
    fprintf(stderr, "Failed to load "\%s"\n", argv[1]);
    return 1;
}
Py_Finalize();
return 0;

Download as text (original file name: run-func.c).

This code loads a Python script using argv[1], and calls the function named in argv[2]. Its integer arguments are the other values of the argv array. If you compile and link this program (let's call the finished executable call), and use it to execute a Python script, such as:

```python
def multiply(a,b):
    print "Will compute", a, "times", b
    c = 0
    for i in range(0, a):
        c = c + b
    return c
```

then the result should be:

```
$ call multiply multiply 3 2
Will compute 3 times 2
Result of call: 6
```
Although the program is quite large for its functionality, most of the code is for data conversion between Python and C, and for error reporting. The interesting part with respect to embedding Python starts with

```c
Py_Initialize();
pName = PyString_FromString(argv[1]);
/* Error checking of pName left out */
pModule = PyImport_Import(pName);
```

After initializing the interpreter, the script is loaded using `PyImport_Import()`. This routine needs a Python string as its argument, which is constructed using the `PyString_FromString()` data conversion routine.

```c
pFunc = PyObject_GetAttrString(pModule, argv[2]);
/* pFunc is a new reference */
if (pFunc && PyCallable_Check(pFunc)) {
    ...
}
Py_XDECREF(pFunc);
```

Once the script is loaded, the name we're looking for is retrieved using `PyObject_GetAttrString()`. If the name exists, and the object returned is callable, you can safely assume that it is a function. The program then proceeds by constructing a tuple of arguments as normal. The call to the Python function is then made with:

```c
pValue = PyObject_CallObject(pFunc, pArgs);
```

Upon return of the function, `pValue` is either `NULL` or it contains a reference to the return value of the function. Be sure to release the reference after examining the value.
5.4 Extending Embedded Python

Until now, the embedded Python interpreter had no access to functionality from the application itself. The Python API allows this by extending the embedded interpreter. That is, the embedded interpreter gets extended with routines provided by the application. While it sounds complex, it is not so bad. Simply forget for a while that the application starts the Python interpreter. Instead, consider the application to be a set of subroutines, and write some glue code that gives Python access to those routines, just like you would write a normal Python extension. For example:

```c
static int numargs=0;

/* Return the number of arguments of the application command line */
static PyObject*
emb_numargs(PyObject *self, PyObject *args)
{
    if(!PyArg_ParseTuple(args, ":numargs"))
        return NULL;
    return Py_BuildValue("i", numargs);
}

static PyMethodDef EmbMethods[] = {
    {"numargs", emb_numargs, METH_VARARGS,
     "Return the number of arguments received by the process."},
    {NULL, NULL, 0, NULL}
};
```

Insert the above code just above the `main()` function. Also, insert the following two statements directly after `Py_Initialize()`:

```c
numargs = argc;
Py_InitModule("emb", EmbMethods);
```

These two lines initialize the `numargs` variable, and make the `emb.numargs()` function accessible to the embedded Python interpreter. With these extensions, the Python script can do things like

```python
import emb
print "Number of arguments", emb.numargs()
```

In a real application, the methods will expose an API of the application to
Python.
5.5 Embedding Python in C++

It is also possible to embed Python in a C++ program; precisely how this is done will depend on the details of the C++ system used; in general you will need to write the main program in C++, and use the C++ compiler to compile and link your program. There is no need to recompile Python itself using C++.
5.6 Linking Requirements

While the `configure` script shipped with the Python sources will correctly build Python to export the symbols needed by dynamically linked extensions, this is not automatically inherited by applications which embed the Python library statically, at least on UNIX. This is an issue when the application is linked to the static runtime library (`libpython.a`) and needs to load dynamic extensions (implemented as .so files).

The problem is that some entry points are defined by the Python runtime solely for extension modules to use. If the embedding application does not use any of these entry points, some linkers will not include those entries in the symbol table of the finished executable. Some additional options are needed to inform the linker not to remove these symbols.

Determining the right options to use for any given platform can be quite difficult, but fortunately the Python configuration already has those values. To retrieve them from an installed Python interpreter, start an interactive interpreter and have a short session like this:

```python
>>> import distutils.sysconfig
>>> distutils.sysconfig.get_config_var('LINKFORSHARED')
'-Xlinker -export-dynamic'
```

The contents of the string presented will be the options that should be used. If the string is empty, there's no need to add any additional options. The `LINKFORSHARED` definition corresponds to the variable of the same name in Python's top-level `Makefile`. 

---

**Extending and Embedding the Python Interpreter**

*Previous: 5.5 Embedding Python in*  
*Up: 5. Embedding Python in*  
*Next: A. Reporting Bugs*

*Release 2.5, documentation updated on 19th September, 2006.*  
*See [About this document...](#) for information on suggesting changes.*
A. Reporting Bugs

Python is a mature programming language which has established a reputation for stability. In order to maintain this reputation, the developers would like to know of any deficiencies you find in Python or its documentation.

Before submitting a report, you will be required to log into SourceForge; this will make it possible for the developers to contact you for additional information if needed. It is not possible to submit a bug report anonymously.

All bug reports should be submitted via the Python Bug Tracker on SourceForge (http://sourceforge.net/bugs/?group_id=5470). The bug tracker offers a Web form which allows pertinent information to be entered and submitted to the developers.

The first step in filing a report is to determine whether the problem has already been reported. The advantage in doing so, aside from saving the developers time, is that you learn what has been done to fix it; it may be that the problem has already been fixed for the next release, or additional information is needed (in which case you are welcome to provide it if you can!). To do this, search the bug database using the search box on the left side of the page.

If the problem you're reporting is not already in the bug tracker, go back to the Python Bug Tracker (http://sourceforge.net/bugs/?group_id=5470). Select the ``Submit a Bug'' link at the top of the page to open the bug reporting form.

The submission form has a number of fields. The only fields that are required are the ``Summary'' and ``Details'' fields. For the summary, enter a very short description of the problem; less than ten words is good. In the Details field, describe the problem in detail, including what you expected to happen and what did happen. Be sure to include the version of Python you used, whether any extension modules were involved, and what hardware and software platform you were using (including version information as appropriate).

The only other field that you may want to set is the ``Category'' field, which allows you to place the bug report into a broad category (such as ``Documentation'' or ``Library'').
Each bug report will be assigned to a developer who will determine what needs to be done to correct the problem. You will receive an update each time action is taken on the bug.

See Also:

*How to Report Bugs Effectively*
Article which goes into some detail about how to create a useful bug report. This describes what kind of information is useful and why it is useful.

*Bug Writing Guidelines*
Information about writing a good bug report. Some of this is specific to the Mozilla project, but describes general good practices.
B. History and License

Subsections

- B.1 History of the software
- B.2 Terms and conditions for accessing or otherwise using Python
- B.3 Licenses and Acknowledgements for Incorporated Software
  - B.3.1 Mersenne Twister
  - B.3.2 Sockets
  - B.3.3 Floating point exception control
  - B.3.4 MD5 message digest algorithm
  - B.3.5 Asynchronous socket services
  - B.3.6 Cookie management
  - B.3.7 Profiling
  - B.3.8 Execution tracing
  - B.3.9 UUencode and UUdecode functions
  - B.3.10 XML Remote Procedure Calls

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
B.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see [http://www.cwi.nl/](http://www.cwi.nl/)) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see [http://www.cnri.reston.va.us/](http://www.cnri.reston.va.us/)) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see [http://www.zope.com/](http://www.zope.com/)). In 2001, the Python Software Foundation (PSF, see [http://www.python.org/psf/](http://www.python.org/psf/)) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see [http://www.opensource.org/](http://www.opensource.org/) for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

<table>
<thead>
<tr>
<th>Release</th>
<th>Derived from</th>
<th>Year</th>
<th>Owner</th>
<th>GPL compatible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.0 thru 1.2</td>
<td>n/a</td>
<td>1991-1995</td>
<td>CWI</td>
<td>yes</td>
</tr>
<tr>
<td>1.3 thru 1.5.2</td>
<td>1.2</td>
<td>1995-1999</td>
<td>CNRI</td>
<td>yes</td>
</tr>
<tr>
<td>1.6</td>
<td>1.5.2</td>
<td>2000</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.0</td>
<td>1.6</td>
<td>2000</td>
<td>BeOpen.com</td>
<td>no</td>
</tr>
<tr>
<td>1.6.1</td>
<td>1.6</td>
<td>2001</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>no</td>
</tr>
<tr>
<td>2.0.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.1</td>
<td>2.1+2.0.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2</td>
<td>2.1.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>Version</td>
<td>Previous</td>
<td>Year</td>
<td>License</td>
<td>Compatible</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>------------</td>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>2.1.2</td>
<td>2.1.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.3</td>
<td>2.1.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.1</td>
<td>2.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.2</td>
<td>2.2.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.3</td>
<td>2.2.2</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3</td>
<td>2.2.2</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.1</td>
<td>2.3</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.2</td>
<td>2.3.1</td>
<td>2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.3</td>
<td>2.3.2</td>
<td>2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.4</td>
<td>2.3.3</td>
<td>2004</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.5</td>
<td>2.3.4</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4</td>
<td>2.3</td>
<td>2004</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.1</td>
<td>2.4</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.2</td>
<td>2.4.1</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.3</td>
<td>2.4.2</td>
<td>2006</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.5</td>
<td>2.4</td>
<td>2006</td>
<td>PSF</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Note:** GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.
B.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.5

1. This LICENSE AGREEMENT is between the Python Software Foundation (``PSF''), and the Individual or Organization (``Licensee'') accessing and otherwise using Python 2.5 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.5 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., ``Copyright © 2001-2006 Python Software Foundation; All Rights Reserved'' are retained in Python 2.5 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.5 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.5.

4. PSF is making Python 2.5 available to Licensee on an ``AS IS'' basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.5 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS
OF PYTHON 2.5 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.5, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.5, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com (``BeOpen''), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (``Licensee'') accessing and otherwise using this software in source or binary form and its associated documentation (``the Software'').

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an ``AS IS'' basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT
LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the ``BeOpen Python'' logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

**CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1**

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (``CNRI''), and the Individual or Organization (``Licensee'') accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to
reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., ``Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): ``Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an ``AS IS'' basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual
property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the "``ACCEPT"" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR
ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
Mersenne Twister
B.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

Subsections

- B.3.1 Mersenne Twister
- B.3.2 Sockets
- B.3.3 Floating point exception control
- B.3.4 MD5 message digest algorithm
- B.3.5 Asynchronous socket services
- B.3.6 Cookie management
- B.3.7 Profiling
- B.3.8 Execution tracing
- B.3.9 UUencode and UUdecode functions
- B.3.10 XML Remote Procedure Calls
The _random module includes code based on a download from http://www.math.keio.ac.jp/~matumoto/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26. Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed) or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura, All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.
http://www.math.keio.ac.jp/matumoto/emt.html
email: matumoto@math.keio.ac.jp
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
B.3.2 Sockets

The socket module uses the functions, `getaddrinfo`, and `getnameinfo`, which are coded in separate source files from the WIDE Project, [http://www.wide.ad.jp/about/index.html](http://www.wide.ad.jp/about/index.html).

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
B.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

```
/  Copyright (c) 1996.
  The Regents of the University of California.
  All rights reserved.

Permission to use, copy, modify, and distribute this software
any purpose without fee is hereby granted, provided that this
tire notice is included in all copies of any software which i
cludes a copy or modification of this software and in
copies of the supporting documentation for such software.

This work was produced at the University of California, Lawr
Livermore National Laboratory under contract no. W-7405-EN
between the U.S. Department of Energy and The Regents of
University of California for the operation of UC LLNL.

DISCLAIMER

This software was prepared as an account of work sponsored b
agency of the United States Government. Neither the United St
Government nor the University of California nor any of their
ployees, makes any warranty, express or implied, or assumes
liability or responsibility for the accuracy, completeness
usefulness of any information, apparatus, product, or pro
disclosed, or represents that its use would not infr
privately-owned rights. Reference herein to any specific com
cial products, process, or service by trade name, tradem
ufacturer, or otherwise, does not necessarily constitute
ply its endorsement, recommendation, or favoring by the Un
States Government or the University of California. The views
opinions of authors expressed herein do not necessarily stat
reflect those of the United States Government or the Univer
of California, and shall not be used for advertising or pro
endorsement purposes.
```
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
B.3.4 MD5 message digest algorithm

The source code for the md5 module contains the following notice:

Copyright (C) 1999, 2002 Aladdin Enterprises. All rights reserved

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and not misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose text is available at

http://www.ietf.org/rfc/rfc1321.txt

The code is derived from the text of the RFC, including the test suite (section A.5) but excluding the rest of Appendix A. It does not include any code or documentation that is identified in the RFC as being copyrighted.

The original and principal author of md5.h is L. Peter Deutsch <ghost@aladdin.com>. Other authors are noted in the change history that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed references to Ghostscript; clarified derivation from RFC 132
now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction
1999-10-18 lpd Fixed typo in header comment (ans12knr rather than
added conditionalization for C++ compilation from Martin
Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
B.3.5 Asynchronous socket services

The asynchat and asyncore modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
B.3.6 Cookie management

The Cookie module contains the following notice:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Timothy O'Malley not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
B.3.7 Profiling

The profile and pstats modules contain the following notice:

Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose (subject to the restriction in the following sentence) without fee is hereby granted provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in all copies of this software, and that the name of InfoSeek not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. This permission is explicitly restricted to the copying and modification of the software to remain in Python, compiled Python, or other languages (such as C wherein the modified or derived code is exclusively imported into a Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
B.3.8 Execution tracing

The `trace` module contains the following notice:

```plaintext
portions copyright 2001, Autonomous Zones Industries, Inc., all rig
er... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights res
```

Permission to use, copy, modify, and distribute this Python softwar
its associated documentation for any purpose without fee is hereby gra
ted, provided that the above copyright notice appears in all co
and that both that copyright notice and this permission notice appe
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertai
distribution of the software without specific, written prior permis
B.3.9 UUencode and UUdecode functions

The uu module contains the following notice:

Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The
  version is still 5 times faster, though.
- Arguments more compliant with python standard
The `xmlrpclib` module contains the following notice:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicit pertaining to distribution of the software without specific, writte prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
Python/C API Reference Manual

Guido van Rossum

Python Software Foundation Email: docs@python.org

Fred L. Drake, Jr., editor

Release 2.5
19th September, 2006

- Front Matter
- Contents
- 1. Introduction
  - 1.1 Include Files
  - 1.2 Objects, Types and Reference Counts
    - 1.2.1 Reference Counts
    - 1.2.2 Types
  - 1.3 Exceptions
  - 1.4 Embedding Python
  - 1.5 Debugging Builds
- 2. The Very High Level Layer
- 3. Reference Counting
- 4. Exception Handling
  - 4.1 Standard Exceptions
  - 4.2 Deprecation of String Exceptions
- 5. Utilities
  - 5.1 Operating System Utilities
  - 5.2 Process Control
  - 5.3 Importing Modules
  - 5.4 Data marshalling support
  - 5.5 Parsing arguments and building values
- 6. Abstract Objects Layer
  - 6.1 Object Protocol
○ **6.2 Number Protocol**
○ **6.3 Sequence Protocol**
○ **6.4 Mapping Protocol**
○ **6.5 Iterator Protocol**
○ **6.6 Buffer Protocol**
● **7. Concrete Objects Layer**
  ○ **7.1 Fundamental Objects**
    ■ **7.1.1 Type Objects**
    ■ **7.1.2 The None Object**
  ○ **7.2 Numeric Objects**
    ■ **7.2.1 Plain Integer Objects**
    ■ **7.2.2 Boolean Objects**
    ■ **7.2.3 Long Integer Objects**
    ■ **7.2.4 Floating Point Objects**
    ■ **7.2.5 Complex Number Objects**
  ○ **7.3 Sequence Objects**
    ■ **7.3.1 String Objects**
    ■ **7.3.2 Unicode Objects**
    ■ **7.3.3 Buffer Objects**
    ■ **7.3.4 Tuple Objects**
    ■ **7.3.5 List Objects**
  ○ **7.4 Mapping Objects**
    ■ **7.4.1 Dictionary Objects**
  ○ **7.5 Other Objects**
    ■ **7.5.1 File Objects**
    ■ **7.5.2 Instance Objects**
    ■ **7.5.3 Function Objects**
    ■ **7.5.4 Method Objects**
    ■ **7.5.5 Module Objects**
    ■ **7.5.6 Iterator Objects**
    ■ **7.5.7 Descriptor Objects**
    ■ **7.5.8 Slice Objects**
    ■ **7.5.9 Weak Reference Objects**
    ■ **7.5.10 CObjects**
    ■ **7.5.11 Cell Objects**
    ■ **7.5.12 Generator Objects**
    ■ **7.5.13 DateTime Objects**
    ■ **7.5.14 Set Objects**
● **8. Initialization, Finalization, and Threads**
8.1 Thread State and the Global Interpreter Lock
8.2 Profiling and Tracing
8.3 Advanced Debugger Support

9. Memory Management
  9.1 Overview
  9.2 Memory Interface
  9.3 Examples

10. Object Implementation Support
  10.1 Allocating Objects on the Heap
  10.2 Common Object Structures
  10.3 Type Objects
  10.4 Mapping Object Structures
  10.5 Number Object Structures
  10.6 Sequence Object Structures
  10.7 Buffer Object Structures
  10.8 Supporting the Iterator Protocol
  10.9 Supporting Cyclic Garbage Collection

A. Reporting Bugs
B. History and License
  B.1 History of the software
  B.2 Terms and conditions for accessing or otherwise using Python
  B.3 Licenses and Acknowledgements for Incorporated Software
    B.3.1 Mersenne Twister
    B.3.2 Sockets
    B.3.3 Floating point exception control
    B.3.4 MD5 message digest algorithm
    B.3.5 Asynchronous socket services
    B.3.6 Cookie management
    B.3.7 Profiling
    B.3.8 Execution tracing
    B.3.9 UUencode and UUdecode functions
    B.3.10 XML Remote Procedure Calls

Index
About this document...
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1. Introduction

The Application Programmer's Interface to Python gives C and C++ programmers access to the Python interpreter at a variety of levels. The API is equally usable from C++, but for brevity it is generally referred to as the Python/C API. There are two fundamentally different reasons for using the Python/C API. The first reason is to write extension modules for specific purposes; these are C modules that extend the Python interpreter. This is probably the most common use. The second reason is to use Python as a component in a larger application; this technique is generally referred to as embedding Python in an application.

Writing an extension module is a relatively well-understood process, where a "cookbook" approach works well. There are several tools that automate the process to some extent. While people have embedded Python in other applications since its early existence, the process of embedding Python is less straightforward than writing an extension.

Many API functions are useful independent of whether you're embedding or extending Python; moreover, most applications that embed Python will need to provide a custom extension as well, so it's probably a good idea to become familiar with writing an extension before attempting to embed Python in a real application.

---

Subsections

- 1.1 Include Files
- 1.2 Objects, Types and Reference Counts
  - 1.2.1 Reference Counts
    - 1.2.1.1 Reference Count Details
  - 1.2.2 Types
- 1.3 Exceptions
- 1.4 Embedding Python
- 1.5 Debugging Builds
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1.1 Include Files

All function, type and macro definitions needed to use the Python/C API are included in your code by the following line:

```c
#include "Python.h"
```

This implies inclusion of the following standard headers: `<stdio.h>`, `<string.h>`, `<errno.h>`, `<limits.h>`, and `<stdlib.h>` (if available).

**Warning:** Since Python may define some pre-processor definitions which affect the standard headers on some systems, you must include `Python.h` before any standard headers are included.

All user visible names defined by `Python.h` (except those defined by the included standard headers) have one of the prefixes "Py" or "_Py". Names beginning with "_Py" are for internal use by the Python implementation and should not be used by extension writers. Structure member names do not have a reserved prefix.

**Important:** user code should never define names that begin with "Py" or "_Py". This confuses the reader, and jeopardizes the portability of the user code to future Python versions, which may define additional names beginning with one of these prefixes.

The header files are typically installed with Python. On UNIX, these are located in the directories `prefix/include/pythonversion/` and `exec_prefix/include/pythonversion/`, where `prefix` and `exec_prefix` are defined by the corresponding parameters to Python's `configure` script and `version` is `sys.version[:3]`. On Windows, the headers are installed in `prefix/include`, where `prefix` is the installation directory specified to the installer.

To include the headers, place both directories (if different) on your compiler's search path for includes. Do not place the parent directories on the search path.
and then use "#include <python2.5/Python.h>"; this will break on multi-platform builds since the platform independent headers under prefix include the platform specific headers from exec_prefix.

C++ users should note that though the API is defined entirely using C, the header files do properly declare the entry points to be extern "C", so there is no need to do anything special to use the API from C++.
1.2 Objects, Types and Reference Counts

Most Python/C API functions have one or more arguments as well as a return value of type PyObject*. This type is a pointer to an opaque data type representing an arbitrary Python object. Since all Python object types are treated the same way by the Python language in most situations (e.g., assignments, scope rules, and argument passing), it is only fitting that they should be represented by a single C type. Almost all Python objects live on the heap: you never declare an automatic or static variable of type PyObject, only pointer variables of type PyObject* can be declared. The sole exception are the type objects; since these must never be deallocated, they are typically static PyTypeObject objects.

All Python objects (even Python integers) have a type and a reference count. An object's type determines what kind of object it is (e.g., an integer, a list, or a user-defined function; there are many more as explained in the Python Reference Manual). For each of the well-known types there is a macro to check whether an object is of that type; for instance, "PyList_Check(a)" is true if (and only if) the object pointed to by a is a Python list.

Subsections

- 1.2.1 Reference Counts
  - 1.2.1.1 Reference Count Details
- 1.2.2 Types
1.2.1 Reference Counts

The reference count is important because today's computers have a finite (and often severely limited) memory size; it counts how many different places there are that have a reference to an object. Such a place could be another object, or a global (or static) C variable, or a local variable in some C function. When an object's reference count becomes zero, the object is deallocated. If it contains references to other objects, their reference count is decremented. Those other objects may be deallocated in turn, if this decrement makes their reference count become zero, and so on. (There's an obvious problem with objects that reference each other here; for now, the solution is ``don't do that.''

Reference counts are always manipulated explicitly. The normal way is to use the macro Py_INCREF() to increment an object's reference count by one, and Py_DECREF() to decrement it by one. The Py_DECREF() macro is considerably more complex than the incref one, since it must check whether the reference count becomes zero and then cause the object's deallocator to be called. The deallocator is a function pointer contained in the object's type structure. The type-specific deallocator takes care of decrementing the reference counts for other objects contained in the object if this is a compound object type, such as a list, as well as performing any additional finalization that's needed. There's no chance that the reference count can overflow; at least as many bits are used to hold the reference count as there are distinct memory locations in virtual memory (assuming sizeof(long) >= sizeof(char*)). Thus, the reference count increment is a simple operation.

It is not necessary to increment an object's reference count for every local variable that contains a pointer to an object. In theory, the object's reference count goes up by one when the variable is made to point to it and it goes down by one when the variable goes out of scope. However, these two cancel each other out, so at the end the reference count hasn't changed. The only real reason to use the reference count is to prevent the object from being deallocated as long as our variable is pointing to it. If we know that there is at least one other reference to the object that lives at least as long as our variable, there is no need to increment the reference count temporarily. An important situation where this arises is in objects that are passed as arguments to C functions in an extension module that are called from Python; the call mechanism guarantees to hold a
reference to every argument for the duration of the call.

However, a common pitfall is to extract an object from a list and hold on to it for a while without incrementing its reference count. Some other operation might conceivably remove the object from the list, decrementing its reference count and possible deallocating it. The real danger is that innocent-looking operations may invoke arbitrary Python code which could do this; there is a code path which allows control to flow back to the user from a `Py_DECREF()`, so almost any operation is potentially dangerous.

A safe approach is to always use the generic operations (functions whose name begins with "PyObject_", "PyNumber_", "PySequence_" or "PyMapping_"'). These operations always increment the reference count of the object they return. This leaves the caller with the responsibility to call `Py_DECREF()` when they are done with the result; this soon becomes second nature.

---

**Subsections**

- [1.2.1.1 Reference Count Details](#)

---

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
1.2.2 Types

There are few other data types that play a significant role in the Python/C API; most are simple C types such as `int`, `long`, `double` and `char*`. A few structure types are used to describe static tables used to list the functions exported by a module or the data attributes of a new object type, and another is used to describe the value of a complex number. These will be discussed together with the functions that use them.
1.3 Exceptions

The Python programmer only needs to deal with exceptions if specific error handling is required; unhandled exceptions are automatically propagated to the caller, then to the caller's caller, and so on, until they reach the top-level interpreter, where they are reported to the user accompanied by a stack traceback.

For C programmers, however, error checking always has to be explicit. All functions in the Python/C API can raise exceptions, unless an explicit claim is made otherwise in a function's documentation. In general, when a function encounters an error, it sets an exception, discards any object references that it owns, and returns an error indicator -- usually NULL or -1. A few functions return a Boolean true/false result, with false indicating an error. Very few functions return no explicit error indicator or have an ambiguous return value, and require explicit testing for errors with PyErr_Occurred().

Exception state is maintained in per-thread storage (this is equivalent to using global storage in an unthreaded application). A thread can be in one of two states: an exception has occurred, or not. The function PyErr_Occurred() can be used to check for this: it returns a borrowed reference to the exception type object when an exception has occurred, and NULL otherwise. There are a number of functions to set the exception state: PyErr_SetString() is the most common (though not the most general) function to set the exception state, and PyErr_Clear() clears the exception state.

The full exception state consists of three objects (all of which can be NULL): the exception type, the corresponding exception value, and the traceback. These have the same meanings as the Python objects sys.exc_type, sys.exc_value, and sys.exc_traceback; however, they are not the same: the Python objects represent the last exception being handled by a Python try ... except statement, while the C level exception state only exists while an exception is being passed on between C functions until it reaches the Python bytecode interpreter's main loop, which takes care of transferring it to sys.exc_type and friends.

Note that starting with Python 1.5, the preferred, thread-safe way to access the
exception state from Python code is to call the function `sys.exc_info()`, which returns the per-thread exception state for Python code. Also, the semantics of both ways to access the exception state have changed so that a function which catches an exception will save and restore its thread's exception state so as to preserve the exception state of its caller. This prevents common bugs in exception handling code caused by an innocent-looking function overwriting the exception being handled; it also reduces the often unwanted lifetime extension for objects that are referenced by the stack frames in the traceback.

As a general principle, a function that calls another function to perform some task should check whether the called function raised an exception, and if so, pass the exception state on to its caller. It should discard any object references that it owns, and return an error indicator, but it should not set another exception -- that would overwrite the exception that was just raised, and lose important information about the exact cause of the error.

A simple example of detecting exceptions and passing them on is shown in the `sum_sequence()` example above. It so happens that that example doesn't need to clean up any owned references when it detects an error. The following example function shows some error cleanup. First, to remind you why you like Python, we show the equivalent Python code:

```python
def incr_item(dict, key):
    try:
        item = dict[key]
    except KeyError:
        item = 0
    dict[key] = item + 1
```

Here is the corresponding C code, in all its glory:

```c
int
incr_item(PyObject *dict, PyObject *key)
{
    /* Objects all initialized to NULL for Py_XDECREF */
    PyObject *item = NULL, *const_one = NULL, *incremented_item = NULL;
    int rv = -1; /* Return value initialized to -1 (failure) */

    item = PyObject_GetItem(dict, key);
    if (item == NULL) {
        /* Handle KeyError only: */
        if (!PyErr_ExceptionMatches(PyExc_KeyError))
            goto error;
```
/* Clear the error and use zero: */
PyErr_Clear();
item = PyInt_FromLong(0L);
if (item == NULL)
goto error;
}
const_one = PyInt_FromLong(1L);
if (const_one == NULL)
goto error;

incremented_item = PyNumber_Add(item, const_one);
if (incremented_item == NULL)
goto error;

if (PyObject_SetItem(dict, key, incremented_item) < 0)
goto error;
rv = 0; /* Success */
/* Continue with cleanup code */

error:
/* Cleanup code, shared by success and failure path */

/* Use Py_XDecref() to ignore NULL references */
Py_XDecref(item);
Py_XDecref(const_one);
Py_XDecref(incremented_item);

return rv; /* -1 for error, 0 for success */
}

This example represents an endorsed use of the goto statement in C! It illustrates the use of PyErr_ExceptionMatches() and PyErr_Clear() to handle specific exceptions, and the use of Py_XDecref() to dispose of owned references that may be NULL (note the "X" in the name; Py_DECREF() would crash when confronted with a NULL reference). It is important that the variables used to hold owned references are initialized to NULL for this to work; likewise, the proposed return value is initialized to -1 (failure) and only set to success after the final call made is successful.
1.4 Embedding Python

The one important task that only embedders (as opposed to extension writers) of the Python interpreter have to worry about is the initialization, and possibly the finalization, of the Python interpreter. Most functionality of the interpreter can only be used after the interpreter has been initialized.

The basic initialization function is `Py_Initialize()`. This initializes the table of loaded modules, and creates the fundamental modules `__builtin__`, `__main__`, `sys`, and `exceptions`. It also initializes the module search path (`sys.path`).

`Py_Initialize()` does not set the "script argument list" (`sys.argv`). If this variable is needed by Python code that will be executed later, it must be set explicitly with a call to `PySys_SetArgv(argc, argv)` subsequent to the call to `Py_Initialize()`.

On most systems (in particular, on UNIX and Windows, although the details are slightly different), `Py_Initialize()` calculates the module search path based upon its best guess for the location of the standard Python interpreter executable, assuming that the Python library is found in a fixed location relative to the Python interpreter executable. In particular, it looks for a directory named `lib/python2.5` relative to the parent directory where the executable named `python` is found on the shell command search path (the environment variable `PATH`).

For instance, if the Python executable is found in `/usr/local/bin/python`, it will assume that the libraries are in `/usr/local/lib/python2.5`. (In fact, this particular path is also the "fallback" location, used when no executable file named `python` is found along `PATH`.) The user can override this behavior by setting the environment variable `PYTHONHOME`, or insert additional directories in front of the standard path by setting `PYTHONPATH`.

The embedding application can steer the search by calling `Py_SetProgramName(file) before calling Py_Initialize()`. Note that `PYTHONHOME` still overrides this and `PYTHONPATH` is still inserted in front of the standard path. An application that requires total control has to provide its

Sometimes, it is desirable to ``uninitialize'' Python. For instance, the application may want to start over (make another call to `Py_Initialize()`) or the application is simply done with its use of Python and wants to free memory allocated by Python. This can be accomplished by calling `Py_Finalize()`. The function `Py_IsInitialized()` returns true if Python is currently in the initialized state. More information about these functions is given in a later chapter. Notice that `Py_Finalize` does not free all memory allocated by the Python interpreter, e.g. memory allocated by extension modules currently cannot be released.
1.5 Debugging Builds

Python can be built with several macros to enable extra checks of the interpreter and extension modules. These checks tend to add a large amount of overhead to the runtime so they are not enabled by default.

A full list of the various types of debugging builds is in the file Misc/SpecialBuilds.txt in the Python source distribution. Builds are available that support tracing of reference counts, debugging the memory allocator, or low-level profiling of the main interpreter loop. Only the most frequently-used builds will be described in the remainder of this section.

Compiling the interpreter with the Py_DEBUG macro defined produces what is generally meant by "a debug build" of Python. Py_DEBUG is enabled in the UNIX build by adding --with-pydebug to the configure command. It is also implied by the presence of the not-Python-specific _DEBUG macro. When Py_DEBUG is enabled in the UNIX build, compiler optimization is disabled.

In addition to the reference count debugging described below, the following extra checks are performed:

- Extra checks are added to the object allocator.
- Extra checks are added to the parser and compiler.
- Downcasts from wide types to narrow types are checked for loss of information.
- A number of assertions are added to the dictionary and set implementations. In addition, the set object acquires a test_c_api method.
- Sanity checks of the input arguments are added to frame creation.
- The storage for long ints is initialized with a known invalid pattern to catch reference to uninitialized digits.
- Low-level tracing and extra exception checking are added to the runtime virtual machine.
- Extra checks are added to the memory arena implementation.
- Extra debugging is added to the thread module.

There may be additional checks not mentioned here.
Defining Py_TRACE_REFS enables reference tracing. When defined, a circular doubly linked list of active objects is maintained by adding two extra fields to every PyObject. Total allocations are tracked as well. Upon exit, all existing references are printed. (In interactive mode this happens after every statement run by the interpreter.) Implied by Py_DEBUG.

Please refer to Misc/SpecialBuilds.txt in the Python source distribution for more detailed information.
2. The Very High Level Layer

The functions in this chapter will let you execute Python source code given in a file or a buffer, but they will not let you interact in a more detailed way with the interpreter.

Several of these functions accept a start symbol from the grammar as a parameter. The available start symbols are `Py_eval_input`, `Py_file_input`, and `Py_single_input`. These are described following the functions which accept them as parameters.

Note also that several of these functions take `FILE*` parameters. On particular issue which needs to be handled carefully is that the `FILE` structure for different C libraries can be different and incompatible. Under Windows (at least), it is possible for dynamically linked extensions to actually use different libraries, so care should be taken that `FILE*` parameters are only passed to these functions if it is certain that they were created by the same library that the Python runtime is using.

```c
int Py_Main( int argc, char **argv)
```

The main program for the standard interpreter. This is made available for programs which embed Python. The `argc` and `argv` parameters should be prepared exactly as those which are passed to a C program's `main()` function. It is important to note that the argument list may be modified (but the contents of the strings pointed to by the argument list are not). The return value will be the integer passed to the `sys.exit()` function, 1 if the interpreter exits due to an exception, or 2 if the parameter list does not represent a valid Python command line.

```c
int PyRun_AnyFile( FILE *fp, const char *filename)
```

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving `closeit` set to 0 and `flags` set to NULL.

```c
int PyRun_AnyFileFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)
```

This is a simplified interface to `PyRun_AnyFileExFlags()` below,
leaving the `closeit` argument set to 0.

```c
int PyRun<AnyFileEx>( FILE *fp, const char *filename, int closeit)
This is a simplified interface to `PyRun<AnyFileExFlags>()` below,
leaving the `flags` argument set to NULL.

int PyRun<AnyFileExFlags>( FILE *fp, const char *filename, int closeit,
PyCompilerFlags *flags)
If `fp` refers to a file associated with an interactive device (console or
terminal input or Unix pseudo-terminal), return the value of
`PyRun_InteractiveLoop()`, otherwise return the result of
`PyRun_SimpleFile()`. If `filename` is NULL, this function uses "???

int PyRun_SimpleString( const char *command)
This is a simplified interface to `PyRun_SimpleStringFlags()`
below, leaving the `PyCompilerFlags*` argument set to NULL.

int PyRun_SimpleStringFlags( const char *command, PyCompilerFlags
*flags)
Executes the Python source code from `command` in the __main__ module
according to the `flags` argument. If __main__ does not already exist, it is
created. Returns 0 on success or -1 if an exception was raised. If there was
an error, there is no way to get the exception information. For the meaning
of `flags`, see below.

int PyRun_SimpleFile( FILE *fp, const char *filename)
This is a simplified interface to `PyRun_SimpleFileExFlags()`
below, leaving `closeit` set to 0 and `flags` set to NULL.

int PyRun_SimpleFileFlags( FILE *fp, const char *filename,
PyCompilerFlags *flags)
This is a simplified interface to `PyRun_SimpleFileExFlags()`
below, leaving `closeit` set to 0.

int PyRun_SimpleFileEx( FILE *fp, const char *filename, int closeit)
This is a simplified interface to `PyRun_SimpleFileExFlags()`
below, leaving flags set to NULL.

int PyRun_SimpleFileExFlags(FILE *fp, const char *filename, int closeit, PyCompilerFlags *flags)

Similar to PyRun_SimpleStringFlags(), but the Python source code is read from fp instead of an in-memory string. filename should be the name of the file. If closeit is true, the file is closed before PyRun_SimpleFileExFlags returns.

int PyRun_InteractiveOne(FILE *fp, const char *filename)

This is a simplified interface to PyRun_InteractiveOneFlags() below, leaving flags set to NULL.

int PyRun_InteractiveOneFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)

Read and execute a single statement from a file associated with an interactive device according to the flags argument. If filename is NULL, "???" is used instead. The user will be prompted using sys.ps1 and sys.ps2. Returns 0 when the input was executed successfully, -1 if there was an exception, or an error code from the errcode.h include file distributed as part of Python if there was a parse error. (Note that errcode.h is not included by Python.h, so must be included specifically if needed.)

int PyRun_InteractiveLoop(FILE *fp, const char *filename)

This is a simplified interface to PyRun_InteractiveLoopFlags() below, leaving flags set to NULL.

int PyRun_InteractiveLoopFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)

Read and execute statements from a file associated with an interactive device until EOF is reached. If filename is NULL, "???" is used instead. The user will be prompted using sys.ps1 and sys.ps2. Returns 0 at EOF.

struct _node* PyParser_SimpleParseString(const char *str, int start)

This is a simplified interface to PyParser_SimpleParseStringFlagsFilename() below,
leaving `filename` set to NULL and `flags` set to 0.

```
struct _node* PyParser_SimpleParseStringFlags(const char* str, int start, int flags)
```

This is a simplified interface to `PyParser_SimpleParseStringFlagsFilename()` below, leaving `filename` set to NULL.

```
struct _node* PyParser_SimpleParseStringFlagsFilename(char *filename, int start, int flags)
```

Parse Python source code from `str` using the start token `start` according to the `flags` argument. The result can be used to create a code object which can be evaluated efficiently. This is useful if a code fragment must be evaluated many times.

```
struct _node* PyParser_SimpleParseFile(FILE *fp, const char *filename, int start)
```

This is a simplified interface to `PyParser_SimpleParseFileFlags()` below, leaving `flags` set to 0.

```
struct _node* PyParser_SimpleParseFileFlags(FILE *fp, const char *filename, int start, int flags)
```

Similar to `PyParser_SimpleParseStringFlagsFilename()`, but the Python source code is read from `fp` instead of an in-memory string.

```
PyObject* PyRun_String(const char* str, int start, PyObject* globals, PyObject* locals)
```


This is a simplified interface to `PyRun_StringFlags()` below, leaving `flags` set to NULL.

```
PyObject* PyRun_StringFlags(*globals, PyObject* locals,
```

const char *str, int start, PyObject

PyObject* PyRun_StringFlags(*globals, PyObject *locals,
PyCompilerFlags *flags)

Execute Python source code from str in the context specified by the
dictionaries globals and locals with the compiler flags specified by flags.
The parameter start specifies the start token that should be used to parse the
source code.

Returns the result of executing the code as a Python object, or NULL if an
exception was raised.

PyObject* PyRun_File(FILE *fp, const char *filename, int start, PyObject
*globals, PyObject *locals)

This is a simplified interface to PyRun_FileExFlags() below, leaving
closeit set to 0 and flags set to NULL.

PyObject* PyRun_FileEx(FILE *fp, const char *filename, int start,
PyObject *globals, PyObject *locals, int closeit)

This is a simplified interface to PyRun_FileExFlags() below, leaving
flags set to NULL.

PyObject* PyRun_FileFlags(FILE *fp, const char *filename, int start,
PyObject *globals, PyObject *locals, PyCompilerFlags *flags)

This is a simplified interface to PyRun_FileExFlags() below, leaving
closeit set to 0.

PyObject* PyRun_FileExFlags(FILE *fp, const char *filename, int start,
PyObject *globals, PyObject *locals, int closeit, PyCompilerFlags *flags)

Similar to PyRun_StringFlags(), but the Python source code is read
from fp instead of an in-memory string. filename should be the name of the
file. If closeit is true, the file is closed before PyRun_FileExFlags() returns.
PyObject* `Py_CompileString` (const char * `str`, const char * `filename`, int start)

*Return value: New reference.*
This is a simplified interface to `Py_CompileStringFlags()` below, leaving `flags` set to NULL.

PyObject* `Py_CompileStringFlags` (const char * `str`, const char * `filename`, int start, PyCompilerFlags * `flags`)

*Return value: New reference.*
Parse and compile the Python source code in `str`, returning the resulting code object. The start token is given by `start`; this can be used to constrain the code which can be compiled and should be `Py_eval_input`, `Py_file_input`, or `Py_single_input`. The filename specified by `filename` is used to construct the code object and may appear in traceback or `SyntaxError` exception messages. This returns NULL if the code cannot be parsed or compiled.

`int Py_eval_input`
The start symbol from the Python grammar for isolated expressions; for use with `Py_CompileString()`.

`int Py_file_input`
The start symbol from the Python grammar for sequences of statements as read from a file or other source; for use with `Py_CompileString()`. This is the symbol to use when compiling arbitrarily long Python source code.

`int Py_single_input`
The start symbol from the Python grammar for a single statement; for use with `Py_CompileString()`. This is the symbol used for the interactive interpreter loop.

`struct PyCompilerFlags`
This is the structure used to hold compiler flags. In cases where code is only being compiled, it is passed as `int flags`, and in cases where code is being executed, it is passed as `PyCompilerFlags *flags`. In this case, `from __future__ import` can modify `flags`. 
Whenever `PyCompilerFlags *flags` is `NULL`, `cf_flags` is treated as equal to 0, and any modification due to `from __future__ import` is discarded.

```c
struct PyCompilerFlags {
    int cf_flags;
}
```

### int `CO_FUTURE_DIVISION`

This bit can be set in `flags` to cause division operator `/` to be interpreted as ``true division`` according to PEP 238.
3. Reference Counting

The macros in this section are used for managing reference counts of Python objects.

void Py_INCREF( PyObject *o)
Increment the reference count for object o. The object must not be NULL; if you aren't sure that it isn't NULL, use Py_XINCREF().

void Py_XINCREF( PyObject *o)
Increment the reference count for object o. The object may be NULL, in which case the macro has no effect.

void Py_DECREF( PyObject *o)
Decrement the reference count for object o. The object must not be NULL; if you aren't sure that it isn't NULL, use Py_XDECREF(). If the reference count reaches zero, the object's type's deallocation function (which must not be NULL) is invoked.

**Warning:** The deallocation function can cause arbitrary Python code to be invoked (e.g. when a class instance with a __del__() method is deallocated). While exceptions in such code are not propagated, the executed code has free access to all Python global variables. This means that any object that is reachable from a global variable should be in a consistent state before Py_DECREF() is invoked. For example, code to delete an object from a list should copy a reference to the deleted object in a temporary variable, update the list data structure, and then call Py_DECREF() for the temporary variable.

void Py_XDECREF( PyObject *o)
Decrement the reference count for object o. The object may be NULL, in which case the macro has no effect; otherwise the effect is the same as for Py_DECREF(), and the same warning applies.

void Py_CLEAR( PyObject *o)
Decrement the reference count for object o. The object may be NULL, in which case the macro has no effect; otherwise the effect is the same as for Py_DECREF(), except that the argument is also set to NULL. The warning for Py_DECREF() does not apply with respect to the object passed because the macro carefully uses a temporary variable and sets the argument to NULL before decrementing its reference count.

It is a good idea to use this macro whenever decrementing the value of a variable that might be traversed during garbage collection.

New in version 2.4.

The following functions are for runtime dynamic embedding of Python: Py_IncRef(PyObject *o), Py_DecRef(PyObject *o). They are simply exported function versions of Py_XINCREF() and Py_XDECREF(), respectively.

The following functions or macros are only for use within the interpreter core: _Py_Dealloc(), _Py_ForgetReference(), _Py_NewReference(), as well as the global variable _Py_RefTotal.
4. Exception Handling

The functions described in this chapter will let you handle and raise Python exceptions. It is important to understand some of the basics of Python exception handling. It works somewhat like the UNIX \texttt{errno} variable: there is a global indicator (per thread) of the last error that occurred. Most functions don't clear this on success, but will set it to indicate the cause of the error on failure. Most functions also return an error indicator, usually \texttt{NULL} if they are supposed to return a pointer, or \texttt{-1} if they return an integer (exception: the \texttt{PyArg_*()} functions return \texttt{1} for success and \texttt{0} for failure).

When a function must fail because some function it called failed, it generally doesn't set the error indicator; the function it called already set it. It is responsible for either handling the error and clearing the exception or returning after cleaning up any resources it holds (such as object references or memory allocations); it should \textit{not} continue normally if it is not prepared to handle the error. If returning due to an error, it is important to indicate to the caller that an error has been set. If the error is not handled or carefully propagated, additional calls into the Python/C API may not behave as intended and may fail in mysterious ways.

The error indicator consists of three Python objects corresponding to the Python variables \texttt{sys.exc_type}, \texttt{sys.exc_value} and \texttt{sys.exc_traceback}. API functions exist to interact with the error indicator in various ways. There is a separate error indicator for each thread.

\textbf{void} \texttt{PyErr_Print()} \\
Print a standard traceback to \texttt{sys.stderr} and clear the error indicator. Call this function only when the error indicator is set. (Otherwise it will cause a fatal error!)

\textbf{PyObject*} \texttt{PyErr_Occurred()} \\
\textit{Return value: Borrowed reference.} Test whether the error indicator is set. If set, return the exception \texttt{type} (the first argument to the last call to one of the \texttt{PyErr_Set*()} functions or to \texttt{PyErr_Restore()}). If not set, return \texttt{NULL}. You do not own a reference
to the return value, so you do not need to \texttt{Py\_DECREF()} it. \textbf{Note:} Do not compare the return value to a specific exception; use \texttt{PyErr\_ExceptionMatches()} instead, shown below. (The comparison could easily fail since the exception may be an instance instead of a class, in the case of a class exception, or it may the a subclass of the expected exception.)

\begin{verbatim}
int PyErr\_ExceptionMatches( PyObject *exc)
  Equivalent to
  "PyErr\_GivenExceptionMatches(PyErr\_Occurred(),
  exc)". This should only be called when an exception is actually set; a
memory access violation will occur if no exception has been raised.
\end{verbatim}

\begin{verbatim}
int PyErr\_GivenExceptionMatches( PyObject *given, PyObject *exc)
  Return true if the \textit{given} exception matches the exception in \textit{exc}. If \textit{exc} is a
class object, this also returns true when \textit{given} is an instance of a subclass. If
\textit{exc} is a tuple, all exceptions in the tuple (and recursively in subtuples) are
searched for a match. If \textit{given} is NULL, a memory access violation will occur.
\end{verbatim}

\begin{verbatim}
void PyErr\_NormalizeException( PyObject**exc, PyObject**val,
                                PyObject**tb)
  Under certain circumstances, the values returned by \texttt{PyErr\_Fetch()}
below can be ``unnormalized'', meaning that *\textit{exc} is a class object but *\textit{val}
is not an instance of the same class. This function can be used to instantiate
the class in that case. If the values are already normalized, nothing happens.
The delayed normalization is implemented to improve performance.
\end{verbatim}

\begin{verbatim}
void PyErr\_Clear()
  Clear the error indicator. If the error indicator is not set, there is no effect.
\end{verbatim}

\begin{verbatim}
void PyErr\_Fetch( PyObject **ptype, PyObject **pvalue,
                  PyObject **ptraceback)
  Retrieve the error indicator into three variables whose addresses are passed.
If the error indicator is not set, set all three variables to NULL. If it is set, it
will be cleared and you own a reference to each object retrieved. The value
and traceback object may be NULL even when the type object is not. \textbf{Note:}
\end{verbatim}
This function is normally only used by code that needs to handle exceptions or by code that needs to save and restore the error indicator temporarily.

```c
void PyErr_Restore(PyObject *type, PyObject *value, PyObject *traceback)
```

Set the error indicator from the three objects. If the error indicator is already set, it is cleared first. If the objects are NULL, the error indicator is cleared. Do not pass a NULL type and non-NULL value or traceback. The exception type should be a class. Do not pass an invalid exception type or value. (Violating these rules will cause subtle problems later.) This call takes away a reference to each object: you must own a reference to each object before the call and after the call you no longer own these references. (If you don't understand this, don't use this function. I warned you.) **Note:** This function is normally only used by code that needs to save and restore the error indicator temporarily; use PyErr_Fetch() to save the current exception state.

```c
void PyErr_SetString(PyObject *type, const char *message)
```

This is the most common way to set the error indicator. The first argument specifies the exception type; it is normally one of the standard exceptions, e.g. PyExc_RuntimeError. You need not increment its reference count. The second argument is an error message; it is converted to a string object.

```c
void PyErr_SetObject(PyObject *type, PyObject *value)
```

This function is similar to PyErr_SetString() but lets you specify an arbitrary Python object for the ``value'' of the exception.

```c
PyObject* PyErr_Format(PyObject *exception, const char *format, ...)
```

**Return value:** *Always NULL.*

This function sets the error indicator and returns NULL. exception should be a Python exception (class, not an instance). format should be a string, containing format codes, similar to printf(). The width.precision before a format code is parsed, but the width part is ignored.

<table>
<thead>
<tr>
<th>Table: Format Characters</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>%%</td>
<td>n/a</td>
<td>The literal % character.</td>
</tr>
<tr>
<td>Character</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>%c</td>
<td>int</td>
<td>A single character, represented as an C int.</td>
</tr>
<tr>
<td>%d</td>
<td>int</td>
<td>Exactly equivalent to <code>printf(&quot;%d&quot;)</code>.</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned int</td>
<td>Exactly equivalent to <code>printf(&quot;%u&quot;)</code>.</td>
</tr>
<tr>
<td>%ld</td>
<td>long</td>
<td>Exactly equivalent to <code>printf(&quot;%ld&quot;)</code>.</td>
</tr>
<tr>
<td>%lu</td>
<td>unsigned long</td>
<td>Exactly equivalent to <code>printf(&quot;%lu&quot;)</code>.</td>
</tr>
<tr>
<td>%zd</td>
<td>Py_ssize_t</td>
<td>Exactly equivalent to <code>printf(&quot;%zd&quot;)</code>.</td>
</tr>
<tr>
<td>%zu</td>
<td>size_t</td>
<td>Exactly equivalent to <code>printf(&quot;%zu&quot;)</code>.</td>
</tr>
<tr>
<td>%i</td>
<td>int</td>
<td>Exactly equivalent to <code>printf(&quot;%i&quot;)</code>.</td>
</tr>
<tr>
<td>%x</td>
<td>int</td>
<td>Exactly equivalent to <code>printf(&quot;%x&quot;)</code>.</td>
</tr>
<tr>
<td>%s</td>
<td>char*</td>
<td>A null-terminated C character array.</td>
</tr>
<tr>
<td>%p</td>
<td>void*</td>
<td>The hex representation of a C pointer. Mostly equivalent to <code>printf(&quot;%p&quot;)</code> except that it is guaranteed to start with the literal 0x regardless of what the platform's <code>printf</code> yields.</td>
</tr>
</tbody>
</table>

An unrecognized format character causes all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

```c
void PyErr_SetNone( PyObject *type)
{
    This is a shorthand for "PyErr_SetObject(type, Py_None)".
}
```

```c
int PyErr_BadArgument()
{
    This is a shorthand for "PyErr_SetString(PyExc_TypeError, message)"", where message indicates that a built-in operation was invoked with an illegal argument. It is mostly for internal use.
}
```

```c
PyObject* PyErr_NoMemory( )
{
    Return value: Always NULL.
    This is a shorthand for "PyErr_SetNone(PyExc_MemoryError)"; it returns NULL so an object allocation function can write "return PyErr_NoMemory();" when it runs out of memory.
}
```

```c
PyObject* PyErr_SetFromErrno( PyObject *type)
```
Return value: Always NULL.
This is a convenience function to raise an exception when a C library function has returned an error and set the C variable errno. It constructs a tuple object whose first item is the integer errno value and whose second item is the corresponding error message (gotten from strerror()), and then calls "PyErr_SetObject(type, object)". On UNIX, when the errno value is EINTR, indicating an interrupted system call, this calls PyErr_CheckSignals(), and if that set the error indicator, leaves it set to that. The function always returns NULL, so a wrapper function around a system call can write "return PyErr_SetFromErrno(type);" when the system call returns an error.

PyObject* PyErr_SetFromErrnoWithFilename(PyObject *type, const char *filename)
Return value: Always NULL.
Similar to PyErr_SetFromErrno(), with the additional behavior that if filename is not NULL, it is passed to the constructor of type as a third parameter. In the case of exceptions such as IOError and OSError, this is used to define the filename attribute of the exception instance.

PyObject* PyErr_SetFromWindowsErr(int ierr)
Return value: Always NULL.
This is a convenience function to raise WindowsError. If called with ierr of 0, the error code returned by a call to GetLastError() is used instead. It calls the Win32 function FormatMessage() to retrieve the Windows description of error code given by ierr or GetLastError(), then it constructs a tuple object whose first item is the ierr value and whose second item is the corresponding error message (gotten from FormatMessage()), and then calls "PyErr_SetObject(PyExc_WindowsError, object)". This function always returns NULL. Availability: Windows.

PyObject* PyErr_SetExcFromWindowsErr(PyObject *type, int ierr)
Return value: Always NULL.
Similar to PyErr_SetFromWindowsErr(), with an additional parameter specifying the exception type to be raised. Availability: Windows. New in version 2.3.
PyObject* `PyErr_SetFromWindowsErrWithFilename`( int `ierr`, const char* `filename`)  

Return value: *Always NULL.*
Similar to `PyErr_SetFromWindowsErr()` , with the additional behavior that if `filename` is not NULL, it is passed to the constructor of `WindowsError` as a third parameter. Availability: Windows.

PyObject* `PyErr_SetExcFromWindowsErrWithFilename`( PyObject* `type`, int `ierr`, char* `filename`)  

Return value: *Always NULL.*
Similar to `PyErr_SetFromWindowsErrWithFilename()` , with an additional parameter specifying the exception type to be raised. Availability: Windows. New in version 2.3.

void `PyErr_BadInternalCall`()  
This is a shorthand for "`PyErr_SetString(PyExc_TypeError, message)`", where `message` indicates that an internal operation (e.g. a Python/C API function) was invoked with an illegal argument. It is mostly for internal use.

int `PyErr_WarnEx`( PyObject* `category`, char* `message`, int `stacklevel`)  
Issue a warning message. The `category` argument is a warning category (see below) or NULL; the `message` argument is a message string. `stacklevel` is a positive number giving a number of stack frames; the warning will be issued from the currently executing line of code in that stack frame. A `stacklevel` of 1 is the function calling `PyErr_WarnEx()` , 2 is the function above that, and so forth.

This function normally prints a warning message to `sys.stderr`; however, it is also possible that the user has specified that warnings are to be turned into errors, and in that case this will raise an exception. It is also possible that the function raises an exception because of a problem with the warning machinery (the implementation imports the `warnings` module to do the heavy lifting). The return value is 0 if no exception is raised, or -1 if an exception is raised. (It is not possible to determine whether a warning
message is actually printed, nor what the reason is for the exception; this is intentional.) If an exception is raised, the caller should do its normal exception handling (for example, `Py_DECREF()` owned references and return an error value).

Warning categories must be subclasses of `Warning`; the default warning category is `RuntimeWarning`. The standard Python warning categories are available as global variables whose names are "PyExc_" followed by the Python exception name. These have the type `PyObject*`; they are all class objects. Their names are `PyExc.Warning`, `PyExc.UserWarning`, `PyExc.UnicodeWarning`, `PyExc.DeprecationWarning`, `PyExc_SyntaxWarning`, `PyExc_RuntimeWarning`, and `PyExc.FutureWarning`. `PyExc.Warning` is a subclass of `PyExc_Exception`; the other warning categories are subclasses of `PyExc.Warning`.

For information about warning control, see the documentation for the `warnings` module and the `-W` option in the command line documentation. There is no C API for warning control.

```c
int PyErr_Warn( PyObject *category, char *message )
```

Issue a warning message. The `category` argument is a warning category (see below) or `NULL`; the `message` argument is a message string. The warning will appear to be issued from the function calling `PyErr_Warn()`, equivalent to calling `PyErr_WarnEx()` with a `stacklevel` of 1.

Deprecated; use `PyErr_WarnEx()` instead.

```c
int PyErr_WarnExplicit( char *filename, int lineno, const char *module, PyObject *registry)
```

Issue a warning message with explicit control over all warning attributes. This is a straightforward wrapper around the Python function `warnings.warn_explicit()`, see there for more information. The `module` and `registry` arguments may be set to `NULL` to get the default effect described there.

```c
int PyErr_CheckSignals()
```
This function interacts with Python's signal handling. It checks whether a signal has been sent to the processes and if so, invokes the corresponding signal handler. If the signal module is supported, this can invoke a signal handler written in Python. In all cases, the default effect for SIGINT is to raise the KeyboardInterrupt exception. If an exception is raised the error indicator is set and the function returns 1; otherwise the function returns 0. The error indicator may or may not be cleared if it was previously set.

void **PyErr_SetInterrupt**()

This function simulates the effect of a SIGINT signal arriving -- the next time **PyErr_CheckSignals**() is called, KeyboardInterrupt will be raised. It may be called without holding the interpreter lock.

PyObject* **PyErr_NewException**( char *name, PyObject *base, PyObject *dict)

*Return value: New reference.*

This utility function creates and returns a new exception object. The *name* argument must be the name of the new exception, a C string of the form module.class. The *base* and *dict* arguments are normally NULL. This creates a class object derived from Exception (accessible in C as PyExc_Exception).

The __module__ attribute of the new class is set to the first part (up to the last dot) of the *name* argument, and the class name is set to the last part (after the last dot). The *base* argument can be used to specify alternate base classes; it can either be only one class or a tuple of classes. The *dict* argument can be used to specify a dictionary of class variables and methods.

void **PyErr_WriteUnraisable**( PyObject *obj)

This utility function prints a warning message to sys.stderr when an exception has been set but it is impossible for the interpreter to actually raise the exception. It is used, for example, when an exception occurs in an __del__() method.

The function is called with a single argument *obj* that identifies the context in which the unraisable exception occurred. The repr of *obj* will be printed
in the warning message.

Subsections

- 4.1 Standard Exceptions
- 4.2 Deprecation of String Exceptions

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.1 Standard Exceptions

All standard Python exceptions are available as global variables whose names are "PyExc_" followed by the Python exception name. These have the type `PyObject*`; they are all class objects. For completeness, here are all the variables:

<table>
<thead>
<tr>
<th>C Name</th>
<th>Python Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyExc_BaseException</td>
<td>BaseException</td>
<td>(1),(4)</td>
</tr>
<tr>
<td>PyExc_Exception</td>
<td>Exception</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_StandardError</td>
<td>StandardError</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_ArithmeticError</td>
<td>ArithmeticError</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_LookupError</td>
<td>LookupError</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_AssertionError</td>
<td>AssertionError</td>
<td></td>
</tr>
<tr>
<td>PyExc_AttributeError</td>
<td>AttributeError</td>
<td></td>
</tr>
<tr>
<td>PyExc_EOFError</td>
<td>EOFError</td>
<td></td>
</tr>
<tr>
<td>PyExc_EnvironmentError</td>
<td>EnvironmentError</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_FloatingPointError</td>
<td>FloatingPointError</td>
<td></td>
</tr>
<tr>
<td>PyExc_IOError</td>
<td>IOError</td>
<td></td>
</tr>
<tr>
<td>PyExc_ImportError</td>
<td>ImportError</td>
<td></td>
</tr>
<tr>
<td>PyExc_IndexError</td>
<td>IndexError</td>
<td></td>
</tr>
<tr>
<td>PyExc_KeyError</td>
<td>KeyError</td>
<td></td>
</tr>
<tr>
<td>PyExc_KeyboardInterrupt</td>
<td>KeyboardInterrupt</td>
<td></td>
</tr>
<tr>
<td>PyExc_MemoryError</td>
<td>MemoryError</td>
<td></td>
</tr>
<tr>
<td>PyExc_NameError</td>
<td>NameError</td>
<td></td>
</tr>
<tr>
<td>PyExc_NotImplementedError</td>
<td>NotImplementedError</td>
<td></td>
</tr>
<tr>
<td>PyExc_OSError</td>
<td>OSError</td>
<td></td>
</tr>
<tr>
<td>PyExc_OverflowError</td>
<td>OverflowError</td>
<td></td>
</tr>
<tr>
<td>PyExc_ReferenceError</td>
<td>ReferenceError</td>
<td>(2)</td>
</tr>
<tr>
<td>PyExc_RuntimeError</td>
<td>RuntimeError</td>
<td></td>
</tr>
<tr>
<td>PyExc_SyntaxError</td>
<td>SyntaxError</td>
<td></td>
</tr>
<tr>
<td>PyExc_SystemError</td>
<td>SystemError</td>
<td></td>
</tr>
<tr>
<td>PyExc_SystemExit</td>
<td>SystemExit</td>
<td></td>
</tr>
<tr>
<td>PyExc_TypeError</td>
<td>TypeError</td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------</td>
<td></td>
</tr>
<tr>
<td>PyExc_ValueError</td>
<td>ValueError</td>
<td></td>
</tr>
<tr>
<td>PyExc_WindowsError</td>
<td>WindowsError</td>
<td></td>
</tr>
<tr>
<td>PyExc_ZeroDivisionError</td>
<td>ZeroDivisionError</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1) This is a base class for other standard exceptions.

(2) This is the same as `weakref.ReferenceError`.

(3) Only defined on Windows; protect code that uses this by testing that the preprocessor macro `MS_WINDOWS` is defined.

(4) New in version 2.5.
4.2 Deprecation of String Exceptions

All exceptions built into Python or provided in the standard library are derived from `BaseException`.

String exceptions are still supported in the interpreter to allow existing code to run unmodified, but this will also change in a future release.
5. Utilities

The functions in this chapter perform various utility tasks, ranging from helping C code be more portable across platforms, using Python modules from C, and parsing function arguments and constructing Python values from C values.

Subsections

- 5.1 Operating System Utilities
- 5.2 Process Control
- 5.3 Importing Modules
- 5.4 Data marshalling support
- 5.5 Parsing arguments and building values
5.1 Operating System Utilities

```c
int Py_FdIsInteractive( FILE *fp, const char *filename )
    Return true (nonzero) if the standard I/O file fp with name filename is
deemed interactive. This is the case for files for which
"isatty(fileno(fp))" is true. If the global flag
Py_InteractiveFlag is true, this function also returns true if the
filename pointer is NULL or if the name is equal to one of the strings
'(<stdin>)' or '??'.
```

```c
long PyOS_GetLastModificationTime( char *filename )
    Return the time of last modification of the file filename. The result is
encoded in the same way as the timestamp returned by the standard C
library function time().
```

```c
void PyOS_AfterFork( )
    Function to update some internal state after a process fork; this should be
called in the new process if the Python interpreter will continue to be used.
If a new executable is loaded into the new process, this function does not
need to be called.
```

```c
int PyOS_CheckStack( )
    Return true when the interpreter runs out of stack space. This is a reliable
check, but is only available when USE_STACKCHECK is defined (currently
on Windows using the Microsoft Visual C++ compiler).
USE_STACKCHECK will be defined automatically; you should never
change the definition in your own code.
```

```c
PyOS_sighandler_t PyOS_getsig( int i )
    Return the current signal handler for signal i. This is a thin wrapper around
either sigaction() or signal(). Do not call those functions directly!
PyOS_sighandler_t is a typedef alias for void (*)(int).
```

```c
PyOS_sighandler_t PyOS_setsig( int i, PyOS_sighandler_t h )
    Set the signal handler for signal i to be h; return the old signal handler. This
```
is a thin wrapper around either `sigaction()` or `signal()`. Do not call those functions directly! `PyOS_sighandler_t` is a typedef alias for `void (*)(int)`.
5.2 Process Control

void Py_FatalError( const char *message)
    Print a fatal error message and kill the process. No cleanup is performed. This function should only be invoked when a condition is detected that would make it dangerous to continue using the Python interpreter; e.g., when the object administration appears to be corrupted. On UNIX, the standard C library function abort() is called which will attempt to produce a core file.

void Py_Exit( int status)
    Exit the current process. This calls Py_Finalize() and then calls the standard C library function exit(status).

int Py_AtExit( void (*func) ()
    Register a cleanup function to be called by Py_Finalize(). The cleanup function will be called with no arguments and should return no value. At most 32 cleanup functions can be registered. When the registration is successful, Py_AtExit() returns 0; on failure, it returns -1. The cleanup function registered last is called first. Each cleanup function will be called at most once. Since Python's internal finalization will have completed before the cleanup function, no Python APIs should be called by func.
5.3 Importing Modules

PyObject* `PyImport_ImportModule`( const char *name)

*Return value:* *New reference.*

This is a simplified interface to `PyImport_ImportModuleEx()` below, leaving the `globals` and `locals` arguments set to NULL. When the `name` argument contains a dot (when it specifies a submodule of a package), the `fromlist` argument is set to the list ['*'] so that the return value is the named module rather than the top-level package containing it as would otherwise be the case. (Unfortunately, this has an additional side effect when `name` in fact specifies a subpackage instead of a submodule: the submodules specified in the package's `__all__` variable are loaded.)

Return a new reference to the imported module, or NULL with an exception set on failure. Before Python 2.4, the module may still be created in the failure case -- examine `sys.modules` to find out. Starting with Python 2.4, a failing import of a module no longer leaves the module in `sys.modules`. Changed in version 2.4: failing imports remove incomplete module objects.

PyObject* `PyImport_ImportModuleEx`( PyObject *globals, PyObject *locals, PyObject *fromlist)

*Return value:* *New reference.*

Import a module. This is best described by referring to the built-in Python function `__import__()`, as the standard `__import__()` function calls this function directly.

The return value is a new reference to the imported module or top-level package, or NULL with an exception set on failure (before Python 2.4, the module may still be created in this case). Like for `__import__()`, the return value when a submodule of a package was requested is normally the top-level package, unless a non-empty `fromlist` was given. Changed in version 2.4: failing imports remove incomplete module objects.

PyObject* `PyImport_Import`( PyObject *name)

*Return value:* *New reference.*
This is a higher-level interface that calls the current "import hook function". It invokes the \_\_import\_\_() function from the \_\_builtins\_\_ of the current globals. This means that the import is done using whatever import hooks are installed in the current environment, e.g. by reexec or ihooks.

PyObject* **PyImport\_ReloadModule** (PyObject *m)

*Return value: New reference.*

Reload a module. This is best described by referring to the built-in Python function reload(), as the standard reload() function calls this function directly. Return a new reference to the reloaded module, or NULL with an exception set on failure (the module still exists in this case).

PyObject* **PyImport\_AddModule** (const char *name)

*Return value: Borrowed reference.*

Return the module object corresponding to a module name. The name argument may be of the form package.module. First check the modules dictionary if there’s one there, and if not, create a new one and insert it in the modules dictionary. Return NULL with an exception set on failure.  
**Note:** This function does not load or import the module; if the module wasn’t already loaded, you will get an empty module object. Use **PyImport\_ImportModule**() or one of its variants to import a module. Package structures implied by a dotted name for name are not created if not already present.

PyObject* **PyImport\_ExecCodeModule** (char *name, PyObject *co)

*Return value: New reference.*

Given a module name (possibly of the form package.module) and a code object read from a Python bytecode file or obtained from the built-in function compile(), load the module. Return a new reference to the module object, or NULL with an exception set if an error occurred. Before Python 2.4, the module could still be created in error cases. Starting with Python 2.4, name is removed from sys.modules in error cases, and even if name was already in sys.modules on entry to **PyImport\_ExecCodeModule**(). Leaving incompletely initialized modules in sys.modules is dangerous, as imports of such modules have no way to know that the module object is an unknown (and probably
damaged with respect to the module author's intents) state.

This function will reload the module if it was already imported. See `PyImport_ReloadModule()` for the intended way to reload a module.

If `name` points to a dotted name of the form `package.module`, any package structures not already created will still not be created.

Changed in version 2.4: `name` is removed from `sys.modules` in error cases.

```
long PyImport_GetMagicNumber()
Return the magic number for Python bytecode files (a.k.a. .pyc and .pyo files). The magic number should be present in the first four bytes of the bytecode file, in little-endian byte order.
```

```
PyObject* PyImport_GetModuleDict()
Return value: Borrowed reference.
Return the dictionary used for the module administration (a.k.a. `sys.modules`). Note that this is a per-interpreter variable.
```

```
void _PyImport_Init()
Initialize the import mechanism. For internal use only.
```

```
void PyImport_Cleanup()
Empty the module table. For internal use only.
```

```
void _PyImport_Fini()
Finalize the import mechanism. For internal use only.
```

```
PyObject* _PyImport_FindExtension( char *, char *)
Return value: Borrowed reference.
For internal use only.
```

```
PyObject* _PyImport_FixupExtension( char *, char *)
For internal use only.
```

```
int PyImport_ImportFrozenModule( char *name)
```
Load a frozen module named *name*. Return 1 for success, 0 if the module is not found, and -1 with an exception set if the initialization failed. To access the imported module on a successful load, use \texttt{PyImport_ImportModule()}. (Note the misnomer -- this function would reload the module if it was already imported.)

\textbf{struct } _\texttt{frozen}

This is the structure type definition for frozen module descriptors, as generated by the freeze utility (see Tools/freeze/ in the Python source distribution). Its definition, found in \texttt{Include/import.h}, is:

\begin{verbatim}
struct _frozen {
    char *name;
    unsigned char *code;
    int size;
};
\end{verbatim}

\texttt{struct } _\texttt{frozen}* \texttt{PyImport_FrozenModules}

This pointer is initialized to point to an array of \texttt{struct } _\texttt{frozen} records, terminated by one whose members are all NULL or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

\texttt{int } \texttt{PyImport_AppendInittab}( char *\texttt{name}, void (*\texttt{initfunc})(void))

Add a single module to the existing table of built-in modules. This is a convenience wrapper around \texttt{PyImport_ExtendInittab()}, returning -1 if the table could not be extended. The new module can be imported by the name \texttt{name}, and uses the function \texttt{initfunc} as the initialization function called on the first attempted import. This should be called before \texttt{Py_Initialize()}.

\textbf{struct } _\texttt{inittab}

Structure describing a single entry in the list of built-in modules. Each of these structures gives the name and initialization function for a module built into the interpreter. Programs which embed Python may use an array of these structures in conjunction with \texttt{PyImport_ExtendInittab()} to provide additional built-in modules. The structure is defined in \texttt{Include/import.h} as:
struct _inittab {
    char *name;
    void (*initfunc)(void);
};

int PyImport_ExtendInittab( struct _inittab *newtab )

    Add a collection of modules to the table of built-in modules. The newtab
    array must end with a sentinel entry which contains NULL for the name
    field; failure to provide the sentinel value can result in a memory fault.
    Returns 0 on success or -1 if insufficient memory could be allocated to
    extend the internal table. In the event of failure, no modules are added to
    the internal table. This should be called before Py_Initialize().
5.4 Data marshalling support

These routines allow C code to work with serialized objects using the same data format as the `marshal` module. There are functions to write data into the serialization format, and additional functions that can be used to read the data back. Files used to store marshalled data must be opened in binary mode.

Numeric values are stored with the least significant byte first.

The module supports two versions of the data format: version 0 is the historical version, version 1 (new in Python 2.4) shares interned strings in the file, and upon unmarshalling, `Py_MARSHAL_VERSION` indicates the current file format (currently 1).

```c
void PyMarshal_WriteLongToFile( long value, FILE *file, int version)
    Marshal a `long` integer, `value`, to `file`. This will only write the least-significant 32 bits of `value`; regardless of the size of the native `long` type.

    Changed in version 2.4: `version` indicates the file format.

void PyMarshal_WriteObjectToFile( PyObject *value, FILE *file, int version)
    Marshal a Python object, `value`, to `file`.

    Changed in version 2.4: `version` indicates the file format.

PyObject* PyMarshal_WriteObjectToString( PyObject *value, int version)
    Return a string object containing the marshalled representation of `value`.

    Changed in version 2.4: `version` indicates the file format.
```

The following functions allow marshalled values to be read back in.

XXX What about error detection? It appears that reading past the end of the file will always result in a negative numeric value (where that's relevant), but it's not
clear that negative values won't be handled properly when there's no error. What's the right way to tell? Should only non-negative values be written using these routines?

long **PyMarshal_ReadLongFromFile**( FILE *file)

    Return a C long from the data stream in a FILE* opened for reading. Only a 32-bit value can be read in using this function, regardless of the native size of long.

int **PyMarshal_ReadShortFromFile**( FILE *file)

    Return a C short from the data stream in a FILE* opened for reading. Only a 16-bit value can be read in using this function, regardless of the native size of short.

**PyObject**\* **PyMarshal_ReadObjectFromFile**( FILE *file)


    Return a Python object from the data stream in a FILE* opened for reading. On error, sets the appropriate exception (EOFError or TypeError) and returns NULL.

**PyObject**\* **PyMarshal_ReadLastObjectFromFile**( FILE *file)


    Return a Python object from the data stream in a FILE* opened for reading. Unlike PyMarshal_ReadObjectFromFile(), this function assumes that no further objects will be read from the file, allowing it to aggressively load file data into memory so that the de-serialization can operate from data in memory rather than reading a byte at a time from the file. Only use these variant if you are certain that you won't be reading anything else from the file. On error, sets the appropriate exception (EOFError or TypeError) and returns NULL.

**PyObject**\* **PyMarshal_ReadObjectFromString**( char *string,

Py_ssize_t len)


    Return a Python object from the data stream in a character buffer containing len bytes pointed to by string. On error, sets the appropriate exception (EOFError or TypeError) and returns NULL.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.5 Parsing arguments and building values

These functions are useful when creating your own extensions functions and methods. Additional information and examples are available in *Extending and Embedding the Python Interpreter*.

The first three of these functions described, `PyArg_ParseTuple()`, `PyArg_ParseTupleAndKeywords()`, and `PyArg_Parse()`, all use *format strings* which are used to tell the function about the expected arguments. The format strings use the same syntax for each of these functions.

A format string consists of zero or more ``format units.`` A format unit describes one Python object; it is usually a single character or a parenthesized sequence of format units. With a few exceptions, a format unit that is not a parenthesized sequence normally corresponds to a single address argument to these functions. In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that matches the format unit; and the entry in [square] brackets is the type of the C variable(s) whose address should be passed.

"s" (string or Unicode object) [const char *]

Convert a Python string or Unicode object to a C pointer to a character string. You must not provide storage for the string itself; a pointer to an existing string is stored into the character pointer variable whose address you pass. The C string is NUL-terminated. The Python string must not contain embedded NUL bytes; if it does, a *TypeError* exception is raised. Unicode objects are converted to C strings using the default encoding. If this conversion fails, a *UnicodeError* is raised.

"s#" (string, Unicode or any read buffer compatible object) [const char *, int]

This variant on "s" stores into two C variables, the first one a pointer to a character string, the second one its length. In this case the Python string may contain embedded null bytes. Unicode objects pass back a pointer to the default encoded string version of the object if such a conversion is
possible. All other read-buffer compatible objects pass back a reference to the raw internal data representation.

"z" (string or None) [const char *]
Like "s", but the Python object may also be None, in which case the C pointer is set to NULL.

"z#" (string or None or any read buffer compatible object) [const char *, int]
This is to "s#" as "z" is to "s".

"u" (Unicode object) [Py_UNICODE *]
Convert a Python Unicode object to a C pointer to a NUL-terminated buffer of 16-bit Unicode (UTF-16) data. As with "s", there is no need to provide storage for the Unicode data buffer; a pointer to the existing Unicode data is stored into the Py_UNICODE pointer variable whose address you pass.

"u#" (Unicode object) [Py_UNICODE *, int]
This variant on "u" stores into two C variables, the first one a pointer to a Unicode data buffer, the second one its length. Non-Unicode objects are handled by interpreting their read-buffer pointer as pointer to a Py_UNICODE array.

"es" (string, Unicode object or character buffer compatible object) [const char *encoding, char **buffer]
This variant on "s" is used for encoding Unicode and objects convertible to Unicode into a character buffer. It only works for encoded data without embedded NUL bytes.

This format requires two arguments. The first is only used as input, and must be a const char* which points to the name of an encoding as a NUL-terminated string, or NULL, in which case the default encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a char**; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument.

PyArg_ParseTuple() will allocate a buffer of the needed size, copy the encoded data into this buffer and adjust *buffer to reference the newly
allocated storage. The caller is responsible for calling `PyMem_Free()` to free the allocated buffer after use.

"et" (string, Unicode object or character buffer compatible object) [const char *encoding, char **buffer]
Same as "es" except that 8-bit string objects are passed through without recoding them. Instead, the implementation assumes that the string object uses the encoding passed in as parameter.

"es#" (string, Unicode object or character buffer compatible object) [const char *encoding, char **buffer, int *buffer_length]
This variant on "s#" is used for encoding Unicode and objects convertible to Unicode into a character buffer. Unlike the "es" format, this variant allows input data which contains NUL characters.

It requires three arguments. The first is only used as input, and must be a const char* which points to the name of an encoding as a NUL-terminated string, or NULL, in which case the default encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a char**; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument. The third argument must be a pointer to an integer; the referenced integer will be set to the number of bytes in the output buffer.

There are two modes of operation:

If *buffer points a NULL pointer, the function will allocate a buffer of the needed size, copy the encoded data into this buffer and set *buffer to reference the newly allocated storage. The caller is responsible for calling `PyMem_Free()` to free the allocated buffer after usage.

If *buffer points to a non-NULL pointer (an already allocated buffer), `PyArg_ParseTuple()` will use this location as the buffer and interpret the initial value of *buffer_length as the buffer size. It will then copy the encoded data into the buffer and NUL-terminate it. If the buffer is not large enough, a `ValueError` will be set.

In both cases, *buffer_length is set to the length of the encoded data without
the trailing NUL byte.

"et#" (string, Unicode object or character buffer compatible object) [const char *encoding, char **buffer]
Same as "es#" except that string objects are passed through without recoding them. Instead, the implementation assumes that the string object uses the encoding passed in as parameter.

"b" (integer) [char]
Convert a Python integer to a tiny int, stored in a C char.

"B" (integer) [unsigned char]
Convert a Python integer to a tiny int without overflow checking, stored in a C unsigned char. New in version 2.3.

"h" (integer) [short int]
Convert a Python integer to a C short int.

"H" (integer) [unsigned short int]
Convert a Python integer to a C unsigned short int, without overflow checking. New in version 2.3.

"i" (integer) [int]
Convert a Python integer to a plain C int.

"I" (integer) [unsigned int]
Convert a Python integer to a C unsigned int, without overflow checking. New in version 2.3.

"l" (integer) [long int]
Convert a Python integer to a C long int.

"k" (integer) [unsigned long]
Convert a Python integer or long integer to a C unsigned long without overflow checking. New in version 2.3.

"L" (integer) [PY_LONG_LONG]
Convert a Python integer to a C long long. This format is only available on platforms that support long long (or _int64 on Windows).
"K" (integer) [unsigned PY_LONG_LONG]
Convert a Python integer or long integer to a C unsigned long long without overflow checking. This format is only available on platforms that support unsigned long long (or unsigned _int64 on Windows). New in version 2.3.

"n" (integer) [Py_ssize_t]
Convert a Python integer or long integer to a C Py_ssize_t. New in version 2.5.

"c" (string of length 1) [char]
Convert a Python character, represented as a string of length 1, to a C char.

"f" (float) [float]
Convert a Python floating point number to a C float.

"d" (float) [double]
Convert a Python floating point number to a C double.

"D" (complex) [Py_complex]
Convert a Python complex number to a C Py_complex structure.

"O" (object) [PyObject *]
Store a Python object (without any conversion) in a C object pointer. The C program thus receives the actual object that was passed. The object's reference count is not increased. The pointer stored is not NULL.

"O!" (object) [typeobject, PyObject *]
Store a Python object in a C object pointer. This is similar to "O", but takes two C arguments: the first is the address of a Python type object, the second is the address of the C variable (of type PyObject*) into which the object pointer is stored. If the Python object does not have the required type, TypeError is raised.

"O&" (object) [converter, anything]
Convert a Python object to a C variable through a converter function. This takes two arguments: the first is a function, the second is the address of a C variable (of arbitrary type), converted to void *. The converter function
in turn is called as follows:

\[\text{status} = \text{converter(object, address)};\]

where \textit{object} is the Python object to be converted and \textit{address} is the \texttt{void*} argument that was passed to the \texttt{PyArg_Parse*()} function. The returned \textit{status} should be 1 for a successful conversion and 0 if the conversion has failed. When the conversion fails, the \textit{converter} function should raise an exception.

"S" (string) [PyStringObject *]
Like "O" but requires that the Python object is a string object. Raises \texttt{TypeError} if the object is not a string object. The C variable may also be declared as \texttt{PyObject*}.

"U" (Unicode string) [PyUnicodeObject *]
Like "O" but requires that the Python object is a Unicode object. Raises \texttt{TypeError} if the object is not a Unicode object. The C variable may also be declared as \texttt{PyObject*}.

"t#" (read-only character buffer) [char *, int]
Like "s#", but accepts any object which implements the read-only buffer interface. The \texttt{char *} variable is set to point to the first byte of the buffer, and the \texttt{int} is set to the length of the buffer. Only single-segment buffer objects are accepted; \texttt{TypeError} is raised for all others.

"w" (read-write character buffer) [char *]
Similar to "s", but accepts any object which implements the read-write buffer interface. The caller must determine the length of the buffer by other means, or use "w#" instead. Only single-segment buffer objects are accepted; \texttt{TypeError} is raised for all others.

"w#" (read-write character buffer) [char *, int]
Like "s#", but accepts any object which implements the read-write buffer interface. The \texttt{char *} variable is set to point to the first byte of the buffer, and the \texttt{int} is set to the length of the buffer. Only single-segment buffer objects are accepted; \texttt{TypeError} is raised for all others.

"(items)" (tuple) [matching-items]
The object must be a Python sequence whose length is the number of format units in \textit{items}. The C arguments must correspond to the individual format units in \textit{items}. Format units for sequences may be nested.

\textbf{Note:} Prior to Python version 1.5.2, this format specifier only accepted a tuple containing the individual parameters, not an arbitrary sequence. Code which previously caused \texttt{TypeError} to be raised here may now proceed without an exception. This is not expected to be a problem for existing code.

It is possible to pass Python long integers where integers are requested; however no proper range checking is done -- the most significant bits are silently truncated when the receiving field is too small to receive the value (actually, the semantics are inherited from downcasts in C -- your mileage may vary).

A few other characters have a meaning in a format string. These may not occur inside nested parentheses. They are:

\texttt{"|"}

Indicates that the remaining arguments in the Python argument list are optional. The C variables corresponding to optional arguments should be initialized to their default value -- when an optional argument is not specified, \texttt{PyArg_ParseTuple()} does not touch the contents of the corresponding C variable(s).

\texttt{";"}

The list of format units ends here; the string after the colon is used as the function name in error messages (the ``associated value'' of the exception that \texttt{PyArg_ParseTuple()} raises).

\texttt{";\;"}

The list of format units ends here; the string after the semicolon is used as the error message \textit{instead} of the default error message. Clearly, ";" and ";\;" mutually exclude each other.

Note that any Python object references which are provided to the caller are \textit{borrowed} references; do not decrement their reference count!

Additional arguments passed to these functions must be addresses of variables
whose type is determined by the format string; these are used to store values from the input tuple. There are a few cases, as described in the list of format units above, where these parameters are used as input values; they should match what is specified for the corresponding format unit in that case.

For the conversion to succeed, the arg object must match the format and the format must be exhausted. On success, the PyArg_Parse*() functions return true, otherwise they return false and raise an appropriate exception.

```c
int PyArg_ParseTuple(PyObject *args, const char *format, ...) 
Parse the parameters of a function that takes only positional parameters into local variables. Returns true on success; on failure, it returns false and raises the appropriate exception.

int PyArg_VaParse(PyObject *args, const char *format, va_list vargs) 
Identical to PyArg_ParseTuple(), except that it accepts a va_list rather than a variable number of arguments.
```

```c
int PyArg_ParseTupleAndKeywords(PyObject *args, PyObject *kw, const char *format, char *keywords[], ...) 
Parse the parameters of a function that takes both positional and keyword parameters into local variables. Returns true on success; on failure, it returns false and raises the appropriate exception.

int PyArg_VaParseTupleAndKeywords(PyObject *args, PyObject *kw, const char *format, char *keywords[], va_list vargs) 
Identical to PyArg_ParseTupleAndKeywords(), except that it accepts a va_list rather than a variable number of arguments.
```

```c
int PyArg_Parse(PyObject *args, const char *format, ...) 
Function used to deconstruct the argument lists of "old-style" functions -- these are functions which use the METH_OLDARGS parameter parsing method. This is not recommended for use in parameter parsing in new code, and most code in the standard interpreter has been modified to no longer use this for that purpose. It does remain a convenient way to decompose other tuples, however, and may continue to be used for that purpose.
```
int PyArg_UnpackTuple(PyObject *args, const char *name, Py_ssize_t min, Py_ssize_t max, ...)

A simpler form of parameter retrieval which does not use a format string to specify the types of the arguments. Functions which use this method to retrieve their parameters should be declared as METH_VARARGS in function or method tables. The tuple containing the actual parameters should be passed as args; it must actually be a tuple. The length of the tuple must be at least min and no more than max; min and max may be equal. Additional arguments must be passed to the function, each of which should be a pointer to a PyObject* variable; these will be filled in with the values from args; they will contain borrowed references. The variables which correspond to optional parameters not given by args will not be filled in; these should be initialized by the caller. This function returns true on success and false if args is not a tuple or contains the wrong number of elements; an exception will be set if there was a failure.

This is an example of the use of this function, taken from the sources for the _weakref helper module for weak references:

```c
static PyObject *
weakref_ref(PyObject *self, PyObject *args)
{
    PyObject *object;
    PyObject *callback = NULL;
    PyObject *result = NULL;

    if (PyArg_UnpackTuple(args, "ref", 1, 2, &object, &callback))
    {
        result = PyWeakref_NewRef(object, callback);
    }
    return result;
}
```

The call to PyArg_UnpackTuple() in this example is entirely equivalent to this call to PyArg_ParseTuple():

```c
PyArg_ParseTuple(args, "O|O:ref", &object, &callback)
```

New in version 2.2.

PyObject* Py_BuildValue( const char *format, ...)  

_Return value:_ **New reference.**  
Create a new value based on a format string similar to those accepted by the
PyArg_Parse*() family of functions and a sequence of values. Returns the value or NULL in the case of an error; an exception will be raised if NULL is returned.

Py_BuildValue() does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns None; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

When memory buffers are passed as parameters to supply data to build objects, as for the "s" and "s#" formats, the required data is copied. Buffers provided by the caller are never referenced by the objects created by Py_BuildValue(). In other words, if your code invokes malloc() and passes the allocated memory to Py_BuildValue(), your code is responsible for calling free() for that memory once Py_BuildValue() returns.

In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that the format unit will return; and the entry in [square] brackets is the type of the C value(s) to be passed.

The characters space, tab, colon and comma are ignored in format strings (but not within format units such as "s#"). This can be used to make long format strings a tad more readable.

"s" (string) [char *]
Convert a null-terminated C string to a Python object. If the C string pointer is NULL, None is used.

"s#" (string) [char *, int]
Convert a C string and its length to a Python object. If the C string pointer is NULL, the length is ignored and None is returned.

"z" (string or None) [char *]
Same as "s".

"z#" (string or None) [char *, int]
Same as "s#".

"u" (Unicode string) [Py_UNICODE *]
Convert a null-terminated buffer of Unicode (UCS-2 or UCS-4) data to a Python Unicode object. If the Unicode buffer pointer is NULL, None is returned.

"u#" (Unicode string) [Py_UNICODE *, int]
Convert a Unicode (UCS-2 or UCS-4) data buffer and its length to a Python Unicode object. If the Unicode buffer pointer is NULL, the length is ignored and None is returned.

"i" (integer) [int]
Convert a plain C int to a Python integer object.

"b" (integer) [char]
Convert a plain C char to a Python integer object.

"h" (integer) [short int]
Convert a plain C short int to a Python integer object.

"l" (integer) [long int]
Convert a C long int to a Python integer object.

"B" (integer) [unsigned char]
Convert a C unsigned char to a Python integer object.

"H" (integer) [unsigned short int]
Convert a C unsigned short int to a Python integer object.

"I" (integer/long) [unsigned int]
Convert a C unsigned int to a Python integer object or a Python long integer object, if it is larger than sys.maxint.

"k" (integer/long) [unsigned long]
Convert a C unsigned long to a Python integer object or a Python long integer object, if it is larger than sys.maxint.

"L" (long) [PY_LONG_LONG]
Convert a C `long long` to a Python long integer object. Only available on platforms that support `long long`.

"K" (long) [unsigned PY_LONG_LONG]
Convert a C unsigned `long long` to a Python long integer object. Only available on platforms that support unsigned `long long`.

"n" (int) [Py_ssize_t]
Convert a C `Py_ssize_t` to a Python integer or long integer. New in version 2.5.

"c" (string of length 1) [char]
Convert a C `int` representing a character to a Python string of length 1.

"d" (float) [double]
Convert a C `double` to a Python floating point number.

"f" (float) [float]
Same as "d".

"D" (complex) [Py_complex *]
Convert a C `Py_complex` structure to a Python complex number.

"O" (object) [PyObject *]
Pass a Python object untouched (except for its reference count, which is incremented by one). If the object passed in is a NULL pointer, it is assumed that this was caused because the call producing the argument found an error and set an exception. Therefore, `Py_BuildValue()` will return NULL but won't raise an exception. If no exception has been raised yet, `SystemError` is set.

"S" (object) [PyObject *]
Same as "O".

"N" (object) [PyObject *]
Same as "O", except it doesn't increment the reference count on the object. Useful when the object is created by a call to an object
constructor in the argument list.

"O" (object) [converter, anything]
Convert anything to a Python object through a converter function. The function is called with anything (which should be compatible with void *) as its argument and should return a "new" Python object, or NULL if an error occurred.

"(items)" (tuple) [matching-items]
Convert a sequence of C values to a Python tuple with the same number of items.

"[items]" (list) [matching-items]
Convert a sequence of C values to a Python list with the same number of items.

"{items}" (dictionary) [matching-items]
Convert a sequence of C values to a Python dictionary. Each pair of consecutive C values adds one item to the dictionary, serving as key and value, respectively.

If there is an error in the format string, the SystemError exception is set and NULL returned.
6. Abstract Objects Layer

The functions in this chapter interact with Python objects regardless of their type, or with wide classes of object types (e.g. all numerical types, or all sequence types). When used on object types for which they do not apply, they will raise a Python exception.

It is not possible to use these functions on objects that are not properly initialized, such as a list object that has been created by PyList_New(), but whose items have not been set to some non-NULL value yet.

Subsections

- 6.1 Object Protocol
- 6.2 Number Protocol
- 6.3 Sequence Protocol
- 6.4 Mapping Protocol
- 6.5 Iterator Protocol
- 6.6 Buffer Protocol

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
6.1 Object Protocol

int **PyObject_Print**( PyObject *o, FILE *fp, int flags)
Print an object o, on file fp. Returns -1 on error. The flags argument is used
to enable certain printing options. The only option currently supported is
Py_PRINT_RAW; if given, the str() of the object is written instead of
the repr().

int **PyObject_HasAttrString**( PyObject *o, const char *attr_name)
Returns 1 if o has the attribute attr_name, and 0 otherwise. This is
equivalent to the Python expression "hasattr(o, attr_name)". This
function always succeeds.

PyObject* **PyObject_GetAttrString**( PyObject *o, const char
*attr_name)
Return value: **New reference.**
Retrieve an attribute named attr_name from object o. Returns the attribute
value on success, or NULL on failure. This is the equivalent of the Python
expression "o.attr_name".

int **PyObject_HasAttr**( PyObject *o, PyObject *attr_name)
Returns 1 if o has the attribute attr_name, and 0 otherwise. This is
equivalent to the Python expression "hasattr(o, attr_name)". This
function always succeeds.

PyObject* **PyObject_GetAttr**( PyObject *o, PyObject *attr_name)
Return value: **New reference.**
Retrieve an attribute named attr_name from object o. Returns the attribute
value on success, or NULL on failure. This is the equivalent of the Python
expression "o.attr_name".

int **PyObject_SetAttrString**( PyObject *o, const char *attr_name,
PyObject *v)
Set the value of the attribute named attr_name, for object o, to the value v.
Returns -1 on failure. This is the equivalent of the Python statement
"o.attr_name = v".

int PyObject_SetAttr( PyObject *o, PyObject *attr_name, PyObject *v)
    Set the value of the attribute named attr_name, for object o, to the value v. Returns -1 on failure. This is the equivalent of the Python statement "o.attr_name = v".

int PyObject_DelAttrString( PyObject *o, const char *attr_name)
    Delete attribute named attr_name, for object o. Returns -1 on failure. This is the equivalent of the Python statement: "del o.attr_name".

int PyObject_DelAttr( PyObject *o, PyObject *attr_name)
    Delete attribute named attr_name, for object o. Returns -1 on failure. This is the equivalent of the Python statement "del o.attr_name".

PyObject* PyObject_RichCompare( PyObject *o1, PyObject *o2, int opid)
Compare the values of o1 and o2 using the operation specified by opid, which must be one of Py_LT, Py_LE, Py_EQ, Py_NE, Py_GT, or Py_GE, corresponding to <, <=, ==, !=, >, or >= respectively. This is the equivalent of the Python expression "o1 op o2", where op is the operator corresponding to opid. Returns the value of the comparison on success, or NULL on failure.

int PyObject_RichCompareBool( PyObject *o1, PyObject *o2, int opid)
Compare the values of o1 and o2 using the operation specified by opid, which must be one of Py_LT, Py_LE, Py_EQ, Py_NE, Py_GT, or Py_GE, corresponding to <, <=, ==, !=, >, or >= respectively. Returns -1 on error, 0 if the result is false, 1 otherwise. This is the equivalent of the Python expression "o1 op o2", where op is the operator corresponding to opid.

int PyObject_Cmp( PyObject *o1, PyObject *o2, int *result)
Compare the values of o1 and o2 using a routine provided by o1, if one exists, otherwise with a routine provided by o2. The result of the comparison is returned in result. Returns -1 on failure. This is the
equivalent of the Python statement "result = cmp(o1, o2)".

int **PyObject_Compare**( PyObject *o1, PyObject *o2)

Compare the values of o1 and o2 using a routine provided by o1, if one exists, otherwise with a routine provided by o2. Returns the result of the comparison on success. On error, the value returned is undefined; use PyErr_Occurred() to detect an error. This is equivalent to the Python expression "cmp(o1, o2)".

PyObject* **PyObject_Repr**( PyObject *o)

*Return value: New reference.*

Compute a string representation of object o. Returns the string representation on success, NULL on failure. This is the equivalent of the Python expression "repr(o)". Called by the repr() built-in function and by reverse quotes.

PyObject* **PyObject_Str**( PyObject *o)

*Return value: New reference.*

Compute a string representation of object o. Returns the string representation on success, NULL on failure. This is the equivalent of the Python expression "str(o)". Called by the str() built-in function and by the print statement.

PyObject* **PyObject_Unicode**( PyObject *o)

*Return value: New reference.*

Compute a Unicode string representation of object o. Returns the Unicode string representation on success, NULL on failure. This is the equivalent of the Python expression "unicode(o)". Called by the unicode() built-in function.

int **PyObject_IsInstance**( PyObject *inst, PyObject *cls)

Returns 1 if inst is an instance of the class cls or a subclass of cls, or 0 if not. On error, returns -1 and sets an exception. If cls is a type object rather than a class object, PyObject_IsInstance() returns 1 if inst is of type cls. If cls is a tuple, the check will be done against every entry in cls. The result will be 1 when at least one of the checks returns 1, otherwise it will be 0. If inst is not a class instance and cls is neither a type object, nor a
class object, nor a tuple, *inst* must have a __class__ attribute -- the class relationship of the value of that attribute with *cls* will be used to determine the result of this function. New in version 2.1. Changed in version 2.2: Support for a tuple as the second argument added.

Subclass determination is done in a fairly straightforward way, but includes a wrinkle that implementors of extensions to the class system may want to be aware of. If *A* and *B* are class objects, *B* is a subclass of *A* if it inherits from *A* either directly or indirectly. If either is not a class object, a more general mechanism is used to determine the class relationship of the two objects. When testing if *B* is a subclass of *A*, if *A* is *B*, *PyObject_IsSubclass()* returns true. If *A* and *B* are different objects, *B*’s __bases__ attribute is searched in a depth-first fashion for *A* -- the presence of the __bases__ attribute is considered sufficient for this determination.

```c
int PyObject_IsSubclass(PyObject *derived, PyObject *cls)
```

Returns 1 if the class *derived* is identical to or derived from the class *cls*, otherwise returns 0. In case of an error, returns -1. If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be 1 when at least one of the checks returns 1, otherwise it will be 0. If either *derived* or *cls* is not an actual class object (or tuple), this function uses the generic algorithm described above. New in version 2.1. Changed in version 2.3: Older versions of Python did not support a tuple as the second argument.

```c
int PyCallable_Check(PyObject *o)
```

Determine if the object *o* is callable. Return 1 if the object is callable and 0 otherwise. This function always succeeds.

```c
PyObject* PyObject_Call(PyObject *callable_object, PyObject *args, PyObject *kw)
```

*Return value: New reference.*

Call a callable Python object *callable_object*, with arguments given by the tuple *args*, and named arguments given by the dictionary *kw*. If no named arguments are needed, *kw* may be NULL. *args* must not be NULL, use an empty tuple if no arguments are needed. Returns the result of the call on success, or NULL on failure. This is the equivalent of the Python expression "apply(*callable_object*, *args, **kw)" or "*callable_object*(*args, **kw)". New in version 2.2.
PyObject* **PyObject_CallObject**( PyObject *callable_object, PyObject *args)

*Return value: New reference.*
Call a callable Python object `callable_object`, with arguments given by the tuple `args`. If no arguments are needed, then `args` may be NULL. Returns the result of the call on success, or NULL on failure. This is the equivalent of the Python expression "apply(`callable_object`, `args`)" or "`callable_object(*args)`".

PyObject* **PyObject_CallFunction**( PyObject *callable, char *format, ...)

*Return value: New reference.*
Call a callable Python object `callable`, with a variable number of C arguments. The C arguments are described using a `Py_BuildValue()` style format string. The format may be NULL, indicating that no arguments are provided. Returns the result of the call on success, or NULL on failure. This is the equivalent of the Python expression "apply(`callable`, `args`)" or "`callable(*args)`". Note that if you only pass `PyObject * args`, `PyObject_CallFunctionObjArgs` is a faster alternative.

PyObject* **PyObject_CallMethod**( PyObject *o, char *method, char *format, ...)

*Return value: New reference.*
Call the method named `method` of object `o` with a variable number of C arguments. The C arguments are described by a `Py_BuildValue()` format string that should produce a tuple. The format may be NULL, indicating that no arguments are provided. Returns the result of the call on success, or NULL on failure. This is the equivalent of the Python expression "o.method(`args`)". Note that if you only pass `PyObject * args`, `PyObject_CallMethodObjArgs` is a faster alternative.

PyObject* **PyObject_CallFunctionObjArgs**( PyObject *callable, ..., NULL)

*Return value: New reference.*
Call a callable Python object `callable`, with a variable number of `PyObject*` arguments. The arguments are provided as a variable number of parameters followed by NULL. Returns the result of the call on success,
or NULL on failure. New in version 2.2.

PyObject* **PyObject_CallMethodObjArgs**( PyObject *o, PyObject *name, ..., NULL)

*Return value: New reference.*
Calls a method of the object o, where the name of the method is given as a Python string object in name. It is called with a variable number of PyObject* arguments. The arguments are provided as a variable number of parameters followed by NULL. Returns the result of the call on success, or NULL on failure. New in version 2.2.

long **PyObject_Hash**( PyObject *o)
Compute and return the hash value of an object o. On failure, return -1. This is the equivalent of the Python expression "hash(o)".

int **PyObject_IsTrue**( PyObject *o)
Returns 1 if the object o is considered to be true, and 0 otherwise. This is equivalent to the Python expression "not not o". On failure, return -1.

int **PyObject_Not**( PyObject *o)
Returns 0 if the object o is considered to be true, and 1 otherwise. This is equivalent to the Python expression "not o". On failure, return -1.

PyObject* **PyObject_Type**( PyObject *o)
*Return value: New reference.*
When o is non-NULL, returns a type object corresponding to the object type of object o. On failure, raises SystemError and returns NULL. This is equivalent to the Python expression type(o). This function increments the reference count of the return value. There's really no reason to use this function instead of the common expression o->ob_type, which returns a pointer of type PyTypeObject*, except when the incremented reference count is needed.

int **PyObject_TypeCheck**( PyObject *o, PyTypeObject *type)
Return true if the object o is of type type or a subtype of type. Both parameters must be non-NULL. New in version 2.2.
Py_ssize_t PyObject_Length( PyObject *o)

Py_ssize_t PyObject_Size( PyObject *o)

Return the length of object o. If the object o provides either the sequence and mapping protocols, the sequence length is returned. On error, -1 is returned. This is the equivalent to the Python expression "len(o)".

PyObject* PyObject_GetItem( PyObject *o, PyObject *key)


Return element of o corresponding to the object key or NULL on failure. This is the equivalent of the Python expression "o[key]".

int PyObject_SetItem( PyObject *o, PyObject *key, PyObject *v)

Map the object key to the value v. Returns -1 on failure. This is the equivalent of the Python statement "o[key] = v".

int PyObject_DelItem( PyObject *o, PyObject *key)

Delete the mapping for key from o. Returns -1 on failure. This is the equivalent of the Python statement "del o[key]".

int PyObject_AsFileDescriptor( PyObject *o)

Derives a file-descriptor from a Python object. If the object is an integer or long integer, its value is returned. If not, the object's fileno() method is called if it exists; the method must return an integer or long integer, which is returned as the file descriptor value. Returns -1 on failure.

PyObject* PyObject_Dir( PyObject *o)


This is equivalent to the Python expression "dir(o)" , returning a (possibly empty) list of strings appropriate for the object argument, or NULL if there was an error. If the argument is NULL, this is like the Python "dir()", returning the names of the current locals; in this case, if no execution frame is active then NULL is returned but PyErr_Occurred() will return false.

PyObject* PyObject_GetIter( PyObject *o)

This is equivalent to the Python expression \texttt{"iter(o)"}. It returns a new
iterator for the object argument, or the object itself if the object is already
an iterator. Raises \texttt{TypeError} and returns NULL if the object cannot be
iterated.
6.2 Number Protocol

int PyNumber_Check(PyObject *o)
    Returns 1 if the object o provides numeric protocols, and false otherwise. This function always succeeds.

PyObject* PyNumber_Add(PyObject *o1, PyObject *o2)
    Returns the result of adding o1 and o2, or NULL on failure. This is the equivalent of the Python expression "o1 + o2".

PyObject* PyNumber_Subtract(PyObject *o1, PyObject *o2)
    Returns the result of subtracting o2 from o1, or NULL on failure. This is the equivalent of the Python expression "o1 - o2".

PyObject* PyNumber_Multiply(PyObject *o1, PyObject *o2)
    Returns the result of multiplying o1 and o2, or NULL on failure. This is the equivalent of the Python expression "o1 * o2".

PyObject* PyNumber_Divide(PyObject *o1, PyObject *o2)
    Returns the result of dividing o1 by o2, or NULL on failure. This is the equivalent of the Python expression "o1 / o2".

PyObject* PyNumber_FloorDivide(PyObject *o1, PyObject *o2)
    Return the floor of o1 divided by o2, or NULL on failure. This is equivalent to the "classic" division of integers. New in version 2.2.

PyObject* PyNumber_TrueDivide(PyObject *o1, PyObject *o2)
    Return a reasonable approximation for the mathematical value of o1 divided by o2, or NULL on failure. The return value is "approximate"
because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers. New in version 2.2.

PyObject* **PyNumber_Remainder**( PyObject *o1, PyObject *o2)

*Return value: New reference.*

Returns the remainder of dividing o1 by o2, or NULL on failure. This is the equivalent of the Python expression "o1 % o2".

PyObject* **PyNumber_Divmod**( PyObject *o1, PyObject *o2)

*Return value: New reference.*

See the built-in function `divmod()` function. Returns NULL on failure. This is the equivalent of the Python expression "divmod(o1, o2)".

PyObject* **PyNumber_Power**( PyObject *o1, PyObject *o2, PyObject *o3)

*Return value: New reference.*

See the built-in function `pow()` function. Returns NULL on failure. This is the equivalent of the Python expression "pow(o1, o2, o3)", where o3 is optional. If o3 is to be ignored, pass `Py_None` in its place (passing NULL for o3 would cause an illegal memory access).

PyObject* **PyNumber_Negative**( PyObject *o)

*Return value: New reference.*

Returns the negation of o on success, or NULL on failure. This is the equivalent of the Python expression "-o".

PyObject* **PyNumber_Positive**( PyObject *o)

*Return value: New reference.*

Returns o on success, or NULL on failure. This is the equivalent of the Python expression "+o".

PyObject* **PyNumber.Absolute**( PyObject *o)

*Return value: New reference.*

Returns the absolute value of o, or NULL on failure. This is the equivalent of the Python expression "abs(o)".

PyObject* **PyNumber_Invert**( PyObject *o)
Returns the bitwise negation of o on success, or NULL on failure. This is the equivalent of the Python expression "~o".

PyObject* PyNumber_Lshift( PyObject *o1, PyObject *o2)
Returns the result of left shifting o1 by o2 on success, or NULL on failure. This is the equivalent of the Python expression "o1 << o2".

PyObject* PyNumber_Rshift( PyObject *o1, PyObject *o2)
Returns the result of right shifting o1 by o2 on success, or NULL on failure. This is the equivalent of the Python expression "o1 >> o2".

PyObject* PyNumber_And( PyObject *o1, PyObject *o2)
Returns the "bitwise and" of o1 and o2 on success and NULL on failure. This is the equivalent of the Python expression "o1 & o2".

PyObject* PyNumber_Xor( PyObject *o1, PyObject *o2)
Returns the "bitwise exclusive or" of o1 by o2 on success, or NULL on failure. This is the equivalent of the Python expression "o1 ^ o2".

PyObject* PyNumber_Or( PyObject *o1, PyObject *o2)
Returns the "bitwise or" of o1 and o2 on success, or NULL on failure. This is the equivalent of the Python expression "o1 | o2".

PyObject* PyNumber_InPlaceAdd( PyObject *o1, PyObject *o2)
Returns the result of adding o1 and o2, or NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement "o1 += o2".

PyObject* PyNumber_InPlaceSubtract( PyObject *o1, PyObject *o2)
Returns the result of subtracting o2 from o1, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement "o1 -= o2".

PyObject* **PyNumber_InPlaceMultiply**( PyObject *o1, PyObject *o2)  
*Return value: New reference.*  
Returns the result of multiplying o1 and o2, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement "o1 *= o2".

PyObject* **PyNumber_InPlaceDivide**( PyObject *o1, PyObject *o2)  
*Return value: New reference.*  
Returns the result of dividing o1 by o2, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement "o1 /= o2".

PyObject* **PyNumber_InPlaceFloorDivide**( PyObject *o1, PyObject *o2)  
*Return value: New reference.*  
Returns the mathematical floor of dividing o1 by o2, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement "o1 //= o2". New in version 2.2.

PyObject* **PyNumber_InPlaceTrueDivide**( PyObject *o1, PyObject *o2)  
*Return value: New reference.*  
Return a reasonable approximation for the mathematical value of o1 divided by o2, or NULL on failure. The return value is "approximate" because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers. The operation is done *in-place* when o1 supports it. New in version 2.2.

PyObject* **PyNumber_InPlaceRemainder**( PyObject *o1, PyObject *o2)  
*Return value: New reference.*  
Returns the remainder of dividing o1 by o2, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the
Python statement "o1 %= o2".

PyObject* **PyNumber_InPlacePower**( PyObject *o1, PyObject *o2, PyObject *o3)

*Return value: New reference.*
See the built-in function pow(). Returns NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement "o1 **= o2" when o3 is Py_None, or an in-place variant of "pow(o1, o2, o3)" otherwise. If o3 is to be ignored, pass Py_None in its place (passing NULL for o3 would cause an illegal memory access).

PyObject* **PyNumber_InPlaceLshift**( PyObject *o1, PyObject *o2)

*Return value: New reference.*
Returns the result of left shifting o1 by o2 on success, or NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement "o1 <<= o2".

PyObject* **PyNumber_InPlaceRshift**( PyObject *o1, PyObject *o2)

*Return value: New reference.*
Returns the result of right shifting o1 by o2 on success, or NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement "o1 >>= o2".

PyObject* **PyNumber_InPlaceAnd**( PyObject *o1, PyObject *o2)

*Return value: New reference.*
Returns the "bitwise and" of o1 and o2 on success and NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement "o1 &= o2".

PyObject* **PyNumber_InPlaceXor**( PyObject *o1, PyObject *o2)

*Return value: New reference.*
Returns the "bitwise exclusive or" of o1 by o2 on success, or NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement "o1 ^= o2".

PyObject* **PyNumber_InPlaceOr**( PyObject *o1, PyObject *o2)

*Return value: New reference.*
Returns the "bitwise or" of o1 and o2 on success, or NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement "o1 | = o2".

```c
int PyNumber_Coerce( PyObject **p1, PyObject **p2)
```
This function takes the addresses of two variables of type PyObject*. If the objects pointed to by *p1 and *p2 have the same type, increment their reference count and return 0 (success). If the objects can be converted to a common numeric type, replace *p1 and *p2 by their converted value (with 'new' reference counts), and return 0. If no conversion is possible, or if some other error occurs, return -1 (failure) and don't increment the reference counts. The call PyNumber_Coerce(&o1, &o2) is equivalent to the Python statement "o1, o2 = coerce(o1, o2)".

```c
PyObject* PyNumber_Int( PyObject *o)
```
*Return value: New reference.*
Returns the o converted to an integer object on success, or NULL on failure. If the argument is outside the integer range a long object will be returned instead. This is the equivalent of the Python expression "int(o)".

```c
PyObject* PyNumber_Long( PyObject *o)
```
*Return value: New reference.*
Returns the o converted to a long integer object on success, or NULL on failure. This is the equivalent of the Python expression "long(o)".

```c
PyObject* PyNumber_Float( PyObject *o)
```
*Return value: New reference.*
Returns the o converted to a float object on success, or NULL on failure. This is the equivalent of the Python expression "float(o)".

```c
PyObject* PyNumber_Index( PyObject *o)
```
Returns the o converted to a Python int or long on success or NULL with a TypeError exception raised on failure. New in version 2.5.

```c
Py_ssize_t PyNumber_AsSsize_t( PyObject *o, PyObject *exc)
```
Returns o converted to a Py_ssize_t value if o can be interpreted as an integer. If o can be converted to a Python int or long but the attempt to
convert to a `Py_ssize_t` value would raise an `OverflowError`, then the `exc` argument is the type of exception that will be raised (usually `IndexError` or `OverflowError`). If `exc` is `NULL`, then the exception is cleared and the value is clipped to `PY_SSIZE_T_MIN` for a negative integer or `PY_SSIZE_T_MAX` for a positive integer. New in version 2.5.

```c
int PyIndex_Check( PyObject *o )
```

Returns True if `o` is an index integer (has the `nb_index` slot of the `tp_as_number` structure filled in). New in version 2.5.
6.3 Sequence Protocol

int **PySequence_Check**( PyObject *o)

Return 1 if the object provides sequence protocol, and 0 otherwise. This function always succeeds.

**Py_ssize_t PySequence_Size**( PyObject *o)

Returns the number of objects in sequence \( o \) on success, and -1 on failure. For objects that do not provide sequence protocol, this is equivalent to the Python expression "\( \text{len}(o) \)".

**Py_ssize_t PySequence_Length**( PyObject *o)

Alternate name for **PySequence_Size()**.

**PyObject* PySequence_Concat**( PyObject *o1, PyObject *o2)

*Return value: New reference.*

Return the concatenation of \( o1 \) and \( o2 \) on success, and NULL on failure. This is the equivalent of the Python expression "\( o1 + o2 \)".

**PyObject* PySequence_Repeat**( PyObject *o, Py_ssize_t count)

*Return value: New reference.*

Return the result of repeating sequence object \( o \) \( \text{count} \) times, or NULL on failure. This is the equivalent of the Python expression "\( o \* \text{count} \)".

**PyObject* PySequence_InPlaceConcat**( PyObject *o1, PyObject *o2)

*Return value: New reference.*

Return the concatenation of \( o1 \) and \( o2 \) on success, and NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python expression "\( o1 += o2 \)".

**PyObject* PySequence_InPlaceRepeat**( PyObject *o, Py_ssize_t count)

*Return value: New reference.*

Return the result of repeating sequence object \( o \) \( \text{count} \) times, or NULL on failure. The operation is done in-place when \( o \) supports it. This is the equivalent of the Python expression "\( o *= \text{count} \)".
PyObject* **PySequence_GetItem**( PyObject *o, Py_ssize_t i)

*Return value: New reference.*

Return the i\textsuperscript{th} element of o, or NULL on failure. This is the equivalent of the Python expression "o[i]".

PyObject* **PySequence_GetSlice**( PyObject *o, Py_ssize_t i1, Py_ssize_t i2)

*Return value: New reference.*

Return the slice of sequence object o between i1 and i2, or NULL on failure. This is the equivalent of the Python expression "o[i1:i2]".

int **PySequence_SetItem**( PyObject *o, Py_ssize_t i, PyObject *v)

Assign object v to the i\textsuperscript{th} element of o. Returns -1 on failure. This is the equivalent of the Python statement "o[i] = v". This function does not steal a reference to v.

int **PySequence_DelItem**( PyObject *o, Py_ssize_t i)

Delete the i\textsuperscript{th} element of object o. Returns -1 on failure. This is the equivalent of the Python statement "del o[i]".

int **PySequence_SetSlice**( PyObject *o, Py_ssize_t i1, Py_ssize_t i2, PyObject *v)

Assign the sequence object v to the slice in sequence object o from i1 to i2. This is the equivalent of the Python statement "o[i1:i2] = v".

int **PySequence_DelSlice**( PyObject *o, Py_ssize_t i1, Py_ssize_t i2)

Delete the slice in sequence object o from i1 to i2. Returns -1 on failure. This is the equivalent of the Python statement "del o[i1:i2]".

int **PySequence_Count**( PyObject *o, PyObject *value)

Return the number of occurrences of value in o, that is, return the number of keys for which o[key] == value. On failure, return -1. This is equivalent to the Python expression "o.count(value)".

int **PySequence_Contains**( PyObject *o, PyObject *value)

Determine if o contains value. If an item in o is equal to value, return 1, otherwise return 0. On error, return -1. This is equivalent to the Python
expression "value in o".

int PySequence_Index( PyObject *o, PyObject *value)
    Return the first index $i$ for which $o[i] == value$. On error, return -1. This is equivalent to the Python expression "o.index(value)".

PyObject* PySequence_List( PyObject *o)
    Return a list object with the same contents as the arbitrary sequence o. The returned list is guaranteed to be new.

PyObject* PySequence_Tuple( PyObject *o)
    Return a tuple object with the same contents as the arbitrary sequence o or NULL on failure. If o is a tuple, a new reference will be returned, otherwise a tuple will be constructed with the appropriate contents. This is equivalent to the Python expression "tuple(o)".

PyObject* PySequence_Fast( PyObject *o, const char *m)
    Returns the sequence o as a tuple, unless it is already a tuple or list, in which case o is returned. Use PySequence_Fast_GET_ITEM() to access the members of the result. Returns NULL on failure. If the object is not a sequence, raises TypeError with m as the message text.

PyObject* PySequence_Fast_GET_ITEM( PyObject *o, Py_ssize_t i)
    Return value: Borrowed reference.
    Return the $i$th element of o, assuming that o was returned by PySequence_Fast(), o is not NULL, and that $i$ is within bounds.

PyObject** PySequence_Fast_ITEMS( PyObject *o)
    Return the underlying array of PyObject pointers. Assumes that o was returned by PySequence_Fast() and o is not NULL. New in version 2.4.

PyObject* PySequence_ITEM( PyObject *o, Py_ssize_t i)
Return the $i$th element of $o$ or NULL on failure. Macro form of PySequence_GetItem() but without checking that PySequence_Check($o$) is true and without adjustment for negative indices. New in version 2.3.

int **PySequence_Fast_GET_SIZE**( PyObject *$o$)

Returns the length of $o$, assuming that $o$ was returned by PySequence_Fast() and that $o$ is not NULL. The size can also be gotten by calling PySequence_Size() on $o$, but PySequence_Fast_GET_SIZE() is faster because it can assume $o$ is a list or tuple.
6.4 Mapping Protocol

int **PyMapping_Check**( PyObject *\( o \))
Return 1 if the object provides mapping protocol, and 0 otherwise. This function always succeeds.

Py_ssize_t **PyMapping_Length**( PyObject *\( o \))
Returns the number of keys in object \( o \) on success, and -1 on failure. For objects that do not provide mapping protocol, this is equivalent to the Python expression "\len(o)\".

int **PyMapping_DelItemString**( PyObject *\( o \), char *\( key \))
Remove the mapping for object \( key \) from the object \( o \). Return -1 on failure. This is equivalent to the Python statement "\del o[\key]\".

int **PyMapping_DelItem**( PyObject *\( o \), PyObject *\( key \))
Remove the mapping for object \( key \) from the object \( o \). Return -1 on failure. This is equivalent to the Python statement "\del o[\key]\".

int **PyMapping_HasKeyString**( PyObject *\( o \), char *\( key \))
On success, return 1 if the mapping object has the key \( key \) and 0 otherwise. This is equivalent to the Python expression "\o .\has_key(\key)\". This function always succeeds.

int **PyMapping_HasKey**( PyObject *\( o \), PyObject *\( key \))
Return 1 if the mapping object has the key \( key \) and 0 otherwise. This is equivalent to the Python expression "\o .\has_key(\key)\". This function always succeeds.

PyObject* **PyMapping_Keys**( PyObject *\( o \))
Return value: **New reference**.
On success, return a list of the keys in object \( o \). On failure, return NULL. This is equivalent to the Python expression "\o .\keys()\".

PyObject* **PyMapping_Values**( PyObject *\( o \))
Return value: **New reference.**
On success, return a list of the values in object o. On failure, return NULL. This is equivalent to the Python expression "o.values()".

PyObject* **PyMapping_Items** (PyObject *o)

*Return value: New reference.*
On success, return a list of the items in object o, where each item is a tuple containing a key-value pair. On failure, return NULL. This is equivalent to the Python expression "o.items()".

PyObject* **PyMapping_GetItemString** (PyObject *o, char *key)

*Return value: New reference.*
Return element of o corresponding to the object key or NULL on failure. This is the equivalent of the Python expression "o[key]".

int **PyMapping_SetItemString** (PyObject *o, char *key, PyObject *v)

Map the object key to the value v in object o. Returns -1 on failure. This is the equivalent of the Python statement "o[key] = v".
6.5 Iterator Protocol

New in version 2.2.

There are only a couple of functions specifically for working with iterators.

```c
int PyIter_Check( PyObject *o)
   Return true if the object o supports the iterator protocol.

PyObject* PyIter_Next( PyObject *o)
   Return the next value from the iteration o. If the object is an iterator, this
   retrieves the next value from the iteration, and returns NULL with no
   exception set if there are no remaining items. If the object is not an iterator,
   TypeError is raised, or if there is an error in retrieving the item, returns
   NULL and passes along the exception.
```

To write a loop which iterates over an iterator, the C code should look something
like this:

```c
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;

if (iterator == NULL) {
   /* propagate error */
}

while (item = PyIter_Next(iterator)) {
   /* do something with item */
   ...
   /* release reference when done */
   Py_DECREF(item);
}

Py_DECREF(iterator);

if (PyErr_Occurred()) {
   /* propagate error */
}
else {
   /* continue doing useful work */
}
```
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
6.6 Buffer Protocol

int **buffer**

int `PyObject_AsCharBuffer`( PyObject *obj, const char **buffer, Py_ssize_t *buffer_len)

Returns a pointer to a read-only memory location useable as character-based input. The `obj` argument must support the single-segment character buffer interface. On success, returns 0, sets `buffer` to the memory location and `buffer_len` to the buffer length. Returns -1 and sets a `TypeError` on error. New in version 1.6.

int `PyObject_AsReadBuffer`( PyObject *obj, const void **buffer, Py_ssize_t *buffer_len)

Returns a pointer to a read-only memory location containing arbitrary data. The `obj` argument must support the single-segment readable buffer interface. On success, returns 0, sets `buffer` to the memory location and `buffer_len` to the buffer length. Returns -1 and sets a `TypeError` on error. New in version 1.6.

int `PyObject_CheckReadBuffer`( PyObject *o)

Returns 1 if `o` supports the single-segment readable buffer interface. Otherwise returns 0. New in version 2.2.

int `PyObject_AsWriteBuffer`( PyObject *obj, void **buffer, Py_ssize_t *buffer_len)

Returns a pointer to a writeable memory location. The `obj` argument must support the single-segment, character buffer interface. On success, returns 0, sets `buffer` to the memory location and `buffer_len` to the buffer length. Returns -1 and sets a `TypeError` on error. New in version 1.6.
7. Concrete Objects Layer

The functions in this chapter are specific to certain Python object types. Passing them an object of the wrong type is not a good idea; if you receive an object from a Python program and you are not sure that it has the right type, you must perform a type check first; for example, to check that an object is a dictionary, use `PyDict_Check()`. The chapter is structured like the "family tree" of Python object types.

**Warning:** While the functions described in this chapter carefully check the type of the objects which are passed in, many of them do not check for NULL being passed instead of a valid object. Allowing NULL to be passed in can cause memory access violations and immediate termination of the interpreter.

---

**Subsections**

- **7.1 Fundamental Objects**
  - 7.1.1 Type Objects
  - 7.1.2 The None Object
- **7.2 Numeric Objects**
  - 7.2.1 Plain Integer Objects
  - 7.2.2 Boolean Objects
  - 7.2.3 Long Integer Objects
  - 7.2.4 Floating Point Objects
  - 7.2.5 Complex Number Objects
    - 7.2.5.1 Complex Numbers as C Structures
    - 7.2.5.2 Complex Numbers as Python Objects
- **7.3 Sequence Objects**
  - 7.3.1 String Objects
  - 7.3.2 Unicode Objects
    - 7.3.2.1 Built-in Codecs
    - 7.3.2.2 Methods and Slot Functions
  - 7.3.3 Buffer Objects
  - 7.3.4 Tuple Objects
7.3.5 List Objects

7.4 Mapping Objects
  7.4.1 Dictionary Objects

7.5 Other Objects
  7.5.1 File Objects
  7.5.2 Instance Objects
  7.5.3 Function Objects
  7.5.4 Method Objects
  7.5.5 Module Objects
  7.5.6 Iterator Objects
  7.5.7 Descriptor Objects
  7.5.8 Slice Objects
  7.5.9 Weak Reference Objects
  7.5.10 CObjects
  7.5.11 Cell Objects
  7.5.12 Generator Objects
  7.5.13 DateTime Objects
  7.5.14 Set Objects
7.1 Fundamental Objects

This section describes Python type objects and the singleton object None.

Subsections

- 7.1.1 Type Objects
- 7.1.2 The None Object

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
The None Object
7.1.1 Type Objects

PyTypeObject
The C structure of the objects used to describe built-in types.

PyObject* PyType_Type
This is the type object for type objects; it is the same object as type and types.TypeType in the Python layer.

int PyType_Check( PyObject *o)
Return true if the object o is a type object, including instances of types derived from the standard type object. Return false in all other cases.

int PyType_CheckExact( PyObject *o)
Return true if the object o is a type object, but not a subtype of the standard type object. Return false in all other cases. New in version 2.2.

int PyType_HasFeature( PyObject *o, int feature)
Return true if the type object o sets the feature feature. Type features are denoted by single bit flags.

int PyType_IS_GC( PyObject *o)
Return true if the type object includes support for the cycle detector; this tests the type flag Py_TPFLAGS_HAVE_GC. New in version 2.0.

int PyType_IsSubtype( PyTypeObject *a, PyTypeObject *b)
Return true if a is a subtype of b. New in version 2.2.

PyObject* PyType_GenericAlloc( PyTypeObject *type, Py_ssize_t nitems)
New in version 2.2.

PyObject* PyType_GenericNew( PyTypeObject *type, PyObject *args, PyObject *kwds)
New in version 2.2.

```c
int PyType_Ready(PyTypeObject *type)
```

Finalize a type object. This should be called on all type objects to finish their initialization. This function is responsible for adding inherited slots from a type's base class. Return 0 on success, or return -1 and sets an exception on error. New in version 2.2.
7.1.2 The None Object

Note that the PyTypeObject for None is not directly exposed in the Python/C API. Since None is a singleton, testing for object identity (using "==" in C) is sufficient. There is no PyNone_Check() function for the same reason.

PyObject* Py_None

The Python None object, denoting lack of value. This object has no methods. It needs to be treated just like any other object with respect to reference counts.

Py_RETURN_NONE

Properly handle returning Py_None from within a C function.
7.2 Numeric Objects

Subsections

- 7.2.1 Plain Integer Objects
- 7.2.2 Boolean Objects
- 7.2.3 Long Integer Objects
- 7.2.4 Floating Point Objects
- 7.2.5 Complex Number Objects
  - 7.2.5.1 Complex Numbers as C Structures
  - 7.2.5.2 Complex Numbers as Python Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.2.1 Plain Integer Objects

**PyIntObject**
This subtype of PyObject represents a Python integer object.

PyTypeObject **PyInt_Type**
This instance of PyTypeObject represents the Python plain integer type. This is the same object as int and types.IntType.

```c
int PyInt_Check( PyObject *o )
Return true if o is of type PyInt_Type or a subtype of PyInt_Type.
Changed in version 2.2: Allowed subtypes to be accepted.
```

```c
int PyInt_CheckExact( PyObject *o )
Return true if o is of type PyInt_Type, but not a subtype of PyInt_Type. New in version 2.2.
```

```c
PyObject* PyInt_FromString( char *str, char **pend, int base )
*Return value: New reference.*
Return a new PyIntObject or PyLongObject based on the string value in str, which is interpreted according to the radix in base. If pend is non-NULL, *pend will point to the first character in str which follows the representation of the number. If base is 0, the radix will be determined based on the leading characters of str: if str starts with '0x' or '0X', radix 16 will be used; if str starts with '0', radix 8 will be used; otherwise radix 10 will be used. If base is not 0, it must be between 2 and 36, inclusive. Leading spaces are ignored. If there are no digits, ValueError will be raised. If the string represents a number too large to be contained within the machine's long int type and overflow warnings are being suppressed, a PyLongObject will be returned. If overflow warnings are not being suppressed, NULL will be returned in this case.
```

```c
PyObject* PyInt_FromLong( long ival )
*Return value: New reference.*
Create a new integer object with a value of ival.
The current implementation keeps an array of integer objects for all integers between -5 and 256, when you create an int in that range you actually just get back a reference to the existing object. So it should be possible to change the value of 1. I suspect the behaviour of Python in this case is undefined. :-) 

PyObject* PyInt_FromSsize_t( Py_ssize_t ival)
Create a new integer object with a value of ival. If the value exceeds LONG_MAX, a long integer object is returned.

New in version 2.5.

long PyInt_AsLong( PyObject *io)
Will first attempt to cast the object to a PyIntObject, if it is not already one, and then return its value. If there is an error, -1 is returned, and the caller should check PyErr_Occurred() to find out whether there was an error, or whether the value just happened to be -1.

long PyInt_AS_LONG( PyObject *io)
Return the value of the object io. No error checking is performed.

unsigned long PyInt_AsUnsignedLongMask( PyObject *io)
Will first attempt to cast the object to a PyIntObject or PyLongObject, if it is not already one, and then return its value as unsigned long. This function does not check for overflow. New in version 2.3.

unsigned PY_LONG_LONG PyInt_AsUnsignedLongLongMask( PyObject *io)
Will first attempt to cast the object to a PyIntObject or PyLongObject, if it is not already one, and then return its value as unsigned long long, without checking for overflow. New in version 2.3.

Py_ssize_t PyInt_AsSsize_t( PyObject *io)
Will first attempt to cast the object to a PyIntObject or PyLongObject, if it is not already one, and then return its value as
Py_ssize_t. New in version 2.5.

long **PyInt_GetMax**()

Return the system's idea of the largest integer it can handle (LONG_MAX, as defined in the system header files).
7.2.2 Boolean Objects

Booleans in Python are implemented as a subclass of integers. There are only two booleans, Py_False and Py_True. As such, the normal creation and deletion functions don't apply to booleans. The following macros are available, however.

```c
int PyBool_Check( PyObject *o)
    Return true if o is of type PyBool_Type. New in version 2.3.
```

```c
PyObject* Py_False
    The Python False object. This object has no methods. It needs to be treated just like any other object with respect to reference counts.
```

```c
PyObject* Py_True
    The Python True object. This object has no methods. It needs to be treated just like any other object with respect to reference counts.
```

```c
Py_RETURN_FALSE
    Return Py_False from a function, properly incrementing its reference count. New in version 2.4.
```

```c
Py_RETURN_TRUE
    Return Py_True from a function, properly incrementing its reference count. New in version 2.4.
```

```c
PyObject* PyBool_FromLong( long v)
    Return a new reference to Py_True or Py_False depending on the truth value of v. New in version 2.3.
```
See About this document... for information on suggesting changes.
7.2.3 Long Integer Objects

**PyLongObject**
This subtype of PyObject represents a Python long integer object.

**PyTypeObject PyLong_Type**
This instance of PyTypeObject represents the Python long integer type. This is the same object as long and types.LongType.

```c
int PyLong_Check( PyObject *p)
```
Return true if its argument is a PyLongObject or a subtype of PyLongObject. Changed in version 2.2: Allowed subtypes to be accepted.

```c
int PyLong_CheckExact( PyObject *p)
```
Return true if its argument is a PyLongObject, but not a subtype of PyLongObject. New in version 2.2.

```c
PyObject* PyLong_FromLong( long v)
```
Return value: *New reference.*
Return a new PyLongObject object from v, or NULL on failure.

```c
PyObject* PyLong_FromUnsignedLong( unsigned long v)
```
Return value: *New reference.*
Return a new PyLongObject object from a C unsigned long, or NULL on failure.

```c
PyObject* PyLong_FromLongLong( PY_LONG_LONG v)
```
Return value: *New reference.*
Return a new PyLongObject object from a C long long, or NULL on failure.

```c
PyObject* PyLong_FromUnsignedLongLong( unsigned PY_LONG_LONG v)
```
Return value: *New reference.*
Return a new PyLongObject object from a C unsigned long long.
long, or NULL on failure.

PyObject* PyLong_FromDouble( double v)

Return a new PyLongObject object from the integer part of v, or NULL
on failure.

PyObject* PyLong_FromString( char *str, char **pend, int base)

Return a new PyLongObject based on the string value in str, which is
interpreted according to the radix in base. If pend is non-NULL, *pend will
point to the first character in str which follows the representation of the
number. If base is 0, the radix will be determined based on the leading
characters of str: if str starts with '0x' or '0X', radix 16 will be used; if
str starts with '0', radix 8 will be used; otherwise radix 10 will be used. If
base is not 0, it must be between 2 and 36, inclusive. Leading spaces are
ignored. If there are no digits, ValueError will be raised.

PyObject* PyLong_FromUnicode(Py_UNICODE *u, Py_ssize_t length, int base)

Convert a sequence of Unicode digits to a Python long integer value. The
first parameter, u, points to the first character of the Unicode string, length
gives the number of characters, and base is the radix for the conversion.
The radix must be in the range [2, 36]; if it is out of range, ValueError
will be raised. New in version 1.6.

PyObject* PyLong_FromVoidPtr( void *p)

Create a Python integer or long integer from the pointer p. The pointer
value can be retrieved from the resulting value using
PyLong_AsVoidPtr(). New in version 1.5.2. Changed in version 2.5:
If the integer is larger than LONG_MAX, a positive long integer is
returned.

long PyLong_AsLong( PyObject *pylong)

Return a C long representation of the contents of pylong. If pylong is
greater than LONG_MAX, an OverflowError is raised.

unsigned long **PyLong_AsUnsignedLong** (PyObject *pylong)
Return a C unsigned long representation of the contents of pylong. If pylong is greater than ULONG_MAX, an OverflowError is raised.

**PY_LONG_LONG PyLong_AsLongLong** (PyObject *pylong)
Return a C long long from a Python long integer. If pylong cannot be represented as a long long, an OverflowError will be raised. New in version 2.2.

unsigned PY_LONG_LONG **PyLong_AsUnsignedLongLong** (PyObject *pylong)
Return a C unsigned long long from a Python long integer. If pylong cannot be represented as an unsigned long long, an OverflowError will be raised if the value is positive, or a TypeError will be raised if the value is negative. New in version 2.2.

unsigned long **PyLong_AsUnsignedLongMask** (PyObject *io)
Return a C unsigned long long from a Python long integer, without checking for overflow. New in version 2.3.

unsigned PY_LONG_LONG **PyLong_AsUnsignedLongLongMask** (PyObject *io)
Return a C unsigned long long from a Python long integer, without checking for overflow. New in version 2.3.

double **PyLong_AsDouble** (PyObject *pylong)
Return a C double representation of the contents of pylong. If pylong cannot be approximately represented as a double, an OverflowError exception is raised and -1.0 will be returned.

void* **PyLong_AsVoidPtr** (PyObject *pylong)
Convert a Python integer or long integer pylong to a C void pointer. If pylong cannot be converted, an OverflowError will be raised. This is only assured to produce a usable void pointer for values created with PyLong_FromVoidPtr(). New in version 1.5.2. Changed in version
2.5: For values outside 0..LONG_MAX, both signed and unsigned integers are accepted.
7.2.4 Floating Point Objects

**PyFloatObject**
This subtype of PyObject represents a Python floating point object.

PyTypeObject **PyFloat_Type**
This instance of PyTypeObject represents the Python floating point type. This is the same object as float and types.FloatType.

int **PyFloat_Check**( PyObject *p)
Return true if its argument is a PyFloatObject or a subtype of PyFloatObject. Changed in version 2.2: Allowed subtypes to be accepted.

int **PyFloat_CheckExact**( PyObject *p)
Return true if its argument is a PyFloatObject, but not a subtype of PyFloatObject. New in version 2.2.

PyObject* **PyFloat_FromString**( PyObject *str, char **pend)
Return value: *New reference.*
Create a PyFloatObject object based on the string value in str, or NULL on failure. The pend argument is ignored. It remains only for backward compatibility.

PyObject* **PyFloat_FromDouble**( double v)
Return value: *New reference.*
Create a PyFloatObject object from v, or NULL on failure.

double **PyFloat_AsDouble**( PyObject *pyfloat)
Return a C double representation of the contents of pyfloat.

double **PyFloat_AS_DOUBLE**( PyObject *pyfloat)
Return a C double representation of the contents of pyfloat, but without error checking.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.2.5 Complex Number Objects

Python's complex number objects are implemented as two distinct types when viewed from the C API: one is the Python object exposed to Python programs, and the other is a C structure which represents the actual complex number value. The API provides functions for working with both.

Subsections

- 7.2.5.1 Complex Numbers as C Structures
- 7.2.5.2 Complex Numbers as Python Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.3 Sequence Objects

Generic operations on sequence objects were discussed in the previous chapter; this section deals with the specific kinds of sequence objects that are intrinsic to the Python language.

Subsections

- 7.3.1 String Objects
- 7.3.2 Unicode Objects
  - 7.3.2.1 Built-in Codecs
  - 7.3.2.2 Methods and Slot Functions
- 7.3.3 Buffer Objects
- 7.3.4 Tuple Objects
- 7.3.5 List Objects
7.3.1 String Objects

These functions raise `TypeError` when expecting a string parameter and are called with a non-string parameter.

**PyStringObject**
This subtype of `PyObject` represents a Python string object.

**PyTypeError**
This instance of `PyTypeObject` represents the Python string type; it is the same object as `str` and `types.StringType` in the Python layer.

```c
int PyString_Check( PyObject *o )
Return true if the object o is a string object or an instance of a subtype of the string type. Changed in version 2.2: Allowed subtypes to be accepted.
```

```c
int PyString_CheckExact( PyObject *o )
Return true if the object o is a string object, but not an instance of a subtype of the string type. New in version 2.2.
```

```c
PyObject* PyString_FromString( const char *v )
Return a new string object with the value v on success, and NULL on failure. The parameter v must not be NULL; it will not be checked.
```

```c
PyObject* PyString_FromStringAndSize( const char *v, Py_ssize_t len )
Return a new string object with the value v and length len on success, and NULL on failure. If v is NULL, the contents of the string are uninitialized.
```

```c
PyObject* PyString_FromFormat( const char *format, ... )
Take a C printf()-style format string and a variable number of arguments, calculate the size of the resulting Python string and return a string with the values formatted into it. The variable arguments must be C
types and must correspond exactly to the format characters in the *format* string. The following format characters are allowed:

<table>
<thead>
<tr>
<th>Format Characters</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>n/a</td>
<td>The literal % character.</td>
</tr>
<tr>
<td>%c</td>
<td>int</td>
<td>A single character, represented as an C int.</td>
</tr>
<tr>
<td>%d</td>
<td>int</td>
<td>Exactly equivalent to <code>printf(&quot;%d&quot;)</code>.</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned int</td>
<td>Exactly equivalent to <code>printf(&quot;%u&quot;)</code>.</td>
</tr>
<tr>
<td>%ld</td>
<td>long</td>
<td>Exactly equivalent to <code>printf(&quot;%ld&quot;)</code>.</td>
</tr>
<tr>
<td>%lu</td>
<td>unsigned long</td>
<td>Exactly equivalent to <code>printf(&quot;%lu&quot;)</code>.</td>
</tr>
<tr>
<td>%zd</td>
<td>Py_ssize_t</td>
<td>Exactly equivalent to <code>printf(&quot;%zd&quot;)</code>.</td>
</tr>
<tr>
<td>%zu</td>
<td>size_t</td>
<td>Exactly equivalent to <code>printf(&quot;%zu&quot;)</code>.</td>
</tr>
<tr>
<td>%i</td>
<td>int</td>
<td>Exactly equivalent to <code>printf(&quot;%i&quot;)</code>.</td>
</tr>
<tr>
<td>%x</td>
<td>int</td>
<td>Exactly equivalent to <code>printf(&quot;%x&quot;)</code>.</td>
</tr>
<tr>
<td>%s</td>
<td>char*</td>
<td>A null-terminated C character array.</td>
</tr>
<tr>
<td>%p</td>
<td>void*</td>
<td>The hex representation of a C pointer. Mostly equivalent to <code>printf(&quot;%p&quot;)</code> except that it is guaranteed to start with the literal 0x regardless of what the platform's <code>printf</code> yields.</td>
</tr>
</tbody>
</table>

An unrecognized format character causes all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

**PyObject** _**PyString_FromFormatV**( const char *format, va_list args)  
*Return value: New reference.*  
Identical to _PyString_FromFormat()_ except that it takes exactly two arguments.

**Py_ssize_t** _PyString_Size_( PyObject *string)  
Return the length of the string in string object *string*.

**Py_ssize_t** _PyString_GET_SIZE_( PyObject *string)
Macro form of `PyString_Size()` but without error checking.

```c
char* PyString_AsString( PyObject *string )
```

Return a NUL-terminated representation of the contents of `string`. The pointer refers to the internal buffer of `string`, not a copy. The data must not be modified in any way, unless the string was just created using `PyString_FromStringAndSize(NULL, size)`. It must not be deallocated. If `string` is a Unicode object, this function computes the default encoding of `string` and operates on that. If `string` is not a string object at all, `PyString_AsString()` returns NULL and raises `TypeError`.

```c
char* PyString_AS_STRING( PyObject *string )
```

Macro form of `PyString_AsString()` but without error checking. Only string objects are supported; no Unicode objects should be passed.

```c
int PyString_AsStringAndSize( PyObject *obj, char **buffer, Py_ssize_t *length )
```

Return a NUL-terminated representation of the contents of the object `obj` through the output variables `buffer` and `length`.

The function accepts both string and Unicode objects as input. For Unicode objects it returns the default encoded version of the object. If `length` is NULL, the resulting buffer may not contain NUL characters; if it does, the function returns -1 and a `TypeError` is raised.

The buffer refers to an internal string buffer of `obj`, not a copy. The data must not be modified in any way, unless the string was just created using `PyString_FromStringAndSize(NULL, size)`. It must not be deallocated. If `string` is a Unicode object, this function computes the default encoding of `string` and operates on that. If `string` is not a string object at all, `PyString_AsStringAndSize()` returns -1 and raises `TypeError`.

```c
void PyString_Concat( PyObject **string, PyObject *newpart )
```

Create a new string object in `*string` containing the contents of `newpart` appended to `string`; the caller will own the new reference. The reference to the old value of `string` will be stolen. If the new string cannot be created, the old reference to `string` will still be discarded and the value of `*string` will be set to NULL; the appropriate exception will be set.
void **PyString_ConcatAndDel( PyObject **string, PyObject *newpart)
    Create a new string object in *string containing the contents of newpart appended to string. This version decrements the reference count of newpart.

int _PyString_Resize( PyObject **string, Py_ssize_t newsize)
    A way to resize a string object even though it is ``immutable''. Only use this to build up a brand new string object; don't use this if the string may already be known in other parts of the code. It is an error to call this function if the refcount on the input string object is not one. Pass the address of an existing string object as an lvalue (it may be written into), and the new size desired. On success, *string holds the resized string object and 0 is returned; the address in *string may differ from its input value. If the reallocation fails, the original string object at *string is deallocated, *string is set to NULL, a memory exception is set, and -1 is returned.

PyObject* PyString_Format( PyObject *format, PyObject *args)
    Return a new string object from format and args. Analogous to format % args. The args argument must be a tuple.

void PyString_InternInPlace( PyObject **string)
    Intern the argument *string in place. The argument must be the address of a pointer variable pointing to a Python string object. If there is an existing interned string that is the same as *string, it sets *string to it (decrementing the reference count of the old string object and incrementing the reference count of the interned string object), otherwise it leaves *string alone and interns it (incrementing its reference count). (Clarification: even though there is a lot of talk about reference counts, think of this function as reference-count-neutral; you own the object after the call if and only if you owned it before the call.)

PyObject* PyString_InternFromString( const char *v)
    A combination of PyString_FromString() and PyString_InternInPlace(), returning either a new string object that has been interned, or a new (``owned'') reference to an earlier interned string object with the same value.
PyObject* **PyString.Decode** (const char *s, Py_ssize_t size, const char *encoding, const char *errors)

*Return value: New reference.*
Create an object by decoding size bytes of the encoded buffer s using the codec registered for encoding. encoding and errors have the same meaning as the parameters of the same name in the `unicode()` built-in function. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

PyObject* **PyString.AsDecodedObject** (PyObject *str, const char *encoding, const char *errors)

*Return value: New reference.*
Decode a string object by passing it to the codec registered for encoding and return the result as Python object. encoding and errors have the same meaning as the parameters of the same name in the `encode()` method. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

PyObject* **PyString.Encode** (const char *s, Py_ssize_t size, const char *encoding, const char *errors)

*Return value: New reference.*
Encode the char buffer of the given size by passing it to the codec registered for encoding and return a Python object. encoding and errors have the same meaning as the parameters of the same name in the `encode()` method. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

PyObject* **PyString.AsEncodedObject** (PyObject *str, const char *encoding, const char *errors)

*Return value: New reference.*
Encode a string object using the codec registered for encoding and return the result as Python object. encoding and errors have the same meaning as the parameters of the same name in the `encode()` method. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.3.2 Unicode Objects

These are the basic Unicode object types used for the Unicode implementation in Python:

**Py_UNICODE**

This type represents the storage type which is used by Python internally as basis for holding Unicode ordinals. Python's default builds use a 16-bit type for Py_UNICODE and store Unicode values internally as UCS2. It is also possible to build a UCS4 version of Python (most recent Linux distributions come with UCS4 builds of Python). These builds then use a 32-bit type for Py_UNICODE and store Unicode data internally as UCS4. On platforms where wchar_t is available and compatible with the chosen Python Unicode build variant, Py_UNICODE is a typedef alias for wchar_t to enhance native platform compatibility. On all other platforms, Py_UNICODE is a typedef alias for either unsigned short (UCS2) or unsigned long (UCS4).

Note that UCS2 and UCS4 Python builds are not binary compatible. Please keep this in mind when writing extensions or interfaces.

**PyUnicodeObject**

This subtype of PyObject represents a Python Unicode object.

**PyTypeObject PyUnicode_Type**

This instance of PyTypeObject represents the Python Unicode type. It is exposed to Python code as unicode and types.UnicodeType.

The following APIs are really C macros and can be used to do fast checks and to access internal read-only data of Unicode objects:

```c
int PyUnicode_Check( PyObject *o)

Return true if the object o is a Unicode object or an instance of a Unicode subtype. Changed in version 2.2: Allowed subtypes to be accepted.
```

```c
int PyUnicode_CheckExact( PyObject *o)

Return true if the object o is a Unicode object, but not an instance of a
```
Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration.

int `Py_UNICODE_ISSPACE`( Py_UNICODE ch)
    Return 1 or 0 depending on whether `ch` is a whitespace character.

int `Py_UNICODE_ISLOWER`( Py_UNICODE ch)
    Return 1 or 0 depending on whether `ch` is a lowercase character.

int `Py_UNICODE_ISUPPER`( Py_UNICODE ch)
    Return 1 or 0 depending on whether `ch` is an uppercase character.

int `Py_UNICODE_ISTITLE`( Py_UNICODE ch)
    Return 1 or 0 depending on whether `ch` is a titlecase character.

int `Py_UNICODE_ISLINEBREAK`( Py_UNICODE ch)
    Return 1 or 0 depending on whether `ch` is a linebreak character.
int Py_UNICODE_ISDECIMAL(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a decimal character.

int Py_UNICODE_ISDIGIT(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a digit character.

int Py_UNICODE_ISNUMERIC(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a numeric character.

int Py_UNICODE_ISALPHA(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is an alphabetic character.

int Py_UNICODE_ISALNUM(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is an alphanumeric character.

These APIs can be used for fast direct character conversions:

Py_UNICODE Py_UNICODE_TOLOWER(Py_UNICODE ch)
    Return the character ch converted to lower case.

Py_UNICODE Py_UNICODE_TOUPPER(Py_UNICODE ch)
    Return the character ch converted to upper case.

Py_UNICODE Py_UNICODE_TOTITLE(Py_UNICODE ch)
    Return the character ch converted to title case.

int Py_UNICODE_TODECIMAL(Py_UNICODE ch)
    Return the character ch converted to a decimal positive integer. Return -1 if this is not possible. This macro does not raise exceptions.

int Py_UNICODE_TODIGIT(Py_UNICODE ch)
    Return the character ch converted to a single digit integer. Return -1 if this is not possible. This macro does not raise exceptions.

double Py_UNICODE_TONUMERIC(Py_UNICODE ch)
    Return the character ch converted to a double. Return -1.0 if this is not possible. This macro does not raise exceptions.
To create Unicode objects and access their basic sequence properties, use these APIs:

PyObject* `PyUnicode_FromUnicode` (const Py_UNICODE *u, Py_ssize_t size)

*Return value: *New reference.*
Create a Unicode Object from the Py_UNICODE buffer u of the given size. u may be NULL which causes the contents to be undefined. It is the user's responsibility to fill in the needed data. The buffer is copied into the new object. If the buffer is not NULL, the return value might be a shared object. Therefore, modification of the resulting Unicode object is only allowed when u is NULL.

Py_UNICODE* `PyUnicode_AsUnicode` (PyObject *unicode)
Return a read-only pointer to the Unicode object's internal Py_UNICODE buffer, NULL if unicode is not a Unicode object.

Py_ssize_t `PyUnicode_GetSize` (PyObject *unicode)
Return the length of the Unicode object.

PyObject* `PyUnicode_FromEncodedObject` (PyObject *obj, const char *encoding, const char *errors)

*Return value: *New reference.*
Coerce an encoded object obj to an Unicode object and return a reference with incremented refcount.

String and other char buffer compatible objects are decoded according to the given encoding and using the error handling defined by errors. Both can be NULL to have the interface use the default values (see the next section for details).

All other objects, including Unicode objects, cause a TypeError to be set.

The API returns NULL if there was an error. The caller is responsible for decref'ing the returned objects.
PyObject* \textbf{PyUnicode\_FromObject}( PyObject *obj)

\textit{Return value: \textit{New reference.}}

Shortcut for \texttt{PyUnicode\_FromEncodedObject(obj, NULL, "strict")} which is used throughout the interpreter whenever coercion to Unicode is needed.

If the platform supports \texttt{wchar\_t} and provides a header file \texttt{wchar.h}, Python can interface directly to this type using the following functions. Support is optimized if Python's own \texttt{Py\_UNICODE} type is identical to the system's \texttt{wchar\_t}.

PyObject* \textbf{PyUnicode\_FromWideChar}( const wchar\_t *w, Py\_ssize\_t size)

\textit{Return value: \textit{New reference.}}

Create a Unicode object from the \texttt{wchar\_t} buffer \texttt{w} of the given size. Return NULL on failure.

Py\_ssize\_t \textbf{PyUnicode\_AsWideChar}( PyUnicodeObject *unicode, wchar\_t *w, Py\_ssize\_t size)

Copy the Unicode object contents into the \texttt{wchar\_t} buffer \texttt{w}. At most \textit{size} \texttt{wchar\_t} characters are copied (excluding a possibly trailing 0-termination character). Return the number of \texttt{wchar\_t} characters copied or -1 in case of an error. Note that the resulting \texttt{wchar\_t} string may or may not be 0-terminated. It is the responsibility of the caller to make sure that the \texttt{wchar\_t} string is 0-terminated in case this is required by the application.

\section*{Subsections}

- \textbf{7.3.2.1} Built-in Codecs
- \textbf{7.3.2.2} Methods and Slot Functions
7.3.3 Buffer Objects

Python objects implemented in C can export a group of functions called the `buffer interface.` These functions can be used by an object to expose its data in a raw, byte-oriented format. Clients of the object can use the buffer interface to access the object data directly, without needing to copy it first.

Two examples of objects that support the buffer interface are strings and arrays. The string object exposes the character contents in the buffer interface's byte-oriented form. An array can also expose its contents, but it should be noted that array elements may be multi-byte values.

An example user of the buffer interface is the file object's `write()` method. Any object that can export a series of bytes through the buffer interface can be written to a file. There are a number of format codes to `PyArg_ParseTuple()` that operate against an object's buffer interface, returning data from the target object.

More information on the buffer interface is provided in the section `Buffer Object Structures" (section 10.7), under the description for `PyBufferProcs`.

A `buffer object" is defined in the `bufferobject.h` header (included by `Python.h`). These objects look very similar to string objects at the Python programming level: they support slicing, indexing, concatenation, and some other standard string operations. However, their data can come from one of two sources: from a block of memory, or from another object which exports the buffer interface.

Buffer objects are useful as a way to expose the data from another object's buffer interface to the Python programmer. They can also be used as a zero-copy slicing mechanism. Using their ability to reference a block of memory, it is possible to expose any data to the Python programmer quite easily. The memory could be a large, constant array in a C extension, it could be a raw block of memory for manipulation before passing to an operating system library, or it could be used to pass around structured data in its native, in-memory format.

**PyBufferObject**

This subtype of `PyObject` represents a buffer object.
PyTypeObject PyBuffer_Type
The instance of PyTypeObject which represents the Python buffer type; it is the same object as buffer and types.BufferType in the Python layer.

int Py_END_OF_BUFFER
This constant may be passed as the size parameter to PyBuffer_FromObject() or PyBuffer_FromReadWriteObject(). It indicates that the new PyBufferObject should refer to base object from the specified offset to the end of its exported buffer. Using this enables the caller to avoid querying the base object for its length.

int PyBuffer_Check( PyObject *p)
Return true if the argument has type PyBuffer_Type.

PyObject* PyBuffer_FromObject( PyObject *base, Py_ssize_t offset,
Py_ssize_t size)

Return a new read-only buffer object. This raises TypeError if base doesn't support the read-only buffer protocol or doesn't provide exactly one buffer segment, or it raises ValueError if offset is less than zero. The buffer will hold a reference to the base object, and the buffer's contents will refer to the base object's buffer interface, starting as position offset and extending for size bytes. If size is Py_END_OF_BUFFER, then the new buffer's contents extend to the length of the base object's exported buffer data.

PyObject* PyBuffer_FromReadWriteObject( Py_ssize_t offset,
Py_ssize_t size)

Return a new writable buffer object. Parameters and exceptions are similar to those for PyBuffer_FromObject(). If the base object does not export the writable buffer protocol, then TypeError is raised.

PyObject* PyBuffer_FromMemory( void *ptr, Py_ssize_t size)
Return a new read-only buffer object that reads from a specified location in memory, with a specified size. The caller is responsible for ensuring that the memory buffer, passed in as \textit{ptr}, is not deallocated while the returned buffer object exists. Raises \texttt{ValueError} if \textit{size} is less than zero. Note that \texttt{Py\_END\_OF\_BUFFER} may \textit{not} be passed for the \textit{size} parameter; \texttt{ValueError} will be raised in that case.

\begin{verbatim}
PyObject* PyBuffer_FromReadWriteMemory( void *ptr, Py_ssize_t size)

Return value: \textit{New reference.}
Similar to \texttt{PyBuffer\_FromMemory()}, but the returned buffer is writable.
\end{verbatim}

\begin{verbatim}
PyObject* PyBuffer\_New( Py_ssize_t size)

Return value: \textit{New reference.}
Return a new writable buffer object that maintains its own memory buffer of \textit{size} bytes. \texttt{ValueError} is returned if \textit{size} is not zero or positive. Note that the memory buffer (as returned by \texttt{PyObject\_AsWriteBuffer()}) is not specifically aligned.
\end{verbatim}
7.3.4 Tuple Objects

**PyTupleObject**
This subtype of PyObject represents a Python tuple object.

PyTypeObject **PyTuple_Type**
This instance of PyTypeObject represents the Python tuple type; it is the same object as `tuple` and `types.TupleType` in the Python layer.

```c
int PyTuple_Check( PyObject *p)
Return true if p is a tuple object or an instance of a subtype of the tuple type. Changed in version 2.2: Allowed subtypes to be accepted.
```

```c
int PyTuple_CheckExact( PyObject *p)
Return true if p is a tuple object, but not an instance of a subtype of the tuple type. New in version 2.2.
```

**PyObject** * PyTuple_New( Py_ssize_t len)
*Return value: New reference.*
Return a new tuple object of size `len`, or NULL on failure.

```c
PyObject* PyTuple_Pack( Py_ssize_t n, ...)
Return a new tuple object of size `n`, or NULL on failure. The tuple values are initialized to the subsequent `n` C arguments pointing to Python objects. "PyTuple_Pack(2, a, b)" is equivalent to "Py_BuildValue(" (00)", a, b)". New in version 2.4.
```

```c
int PyTuple_Size( PyObject *p)
Take a pointer to a tuple object, and return the size of that tuple.
```

```c
int PyTuple_GET_SIZE( PyObject *p)
Return the size of the tuple `p`, which must be non-NULL and point to a tuple; no error checking is performed.
```

**PyObject** * PyTuple_GetItem( PyObject *p, Py_ssize_t pos)
Return value: **Borrowed reference.**
Return the object at position *pos* in the tuple pointed to by *p*. If *pos* is out of bounds, return NULL and sets an *IndexError* exception.

PyObject* **PyTuple_GET_ITEM**( PyObject * *p*, Py_ssize_t *pos*)

Return value: **Borrowed reference.**
Like *PyTuple_GetItem()* , but does no checking of its arguments.

PyObject* **PyTuple_GetSlice**( PyObject * *p*, Py_ssize_t *low*, Py_ssize_t *high*)

Return value: **New reference.**
Take a slice of the tuple pointed to by *p* from *low* to *high* and return it as a new tuple.

int **PyTuple_SetItem**( PyObject * *p*, Py_ssize_t *pos*, PyObject * *o*)

Insert a reference to object *o* at position *pos* of the tuple pointed to by *p*. Return 0 on success. **Note:** This function \``steals\'' a reference to *o*.

void **PyTuple_SET_ITEM**( PyObject * *p*, Py_ssize_t *pos*, PyObject * *o*)

Like *PyTuple_SetItem()* , but does no error checking, and should only be used to fill in brand new tuples. **Note:** This function \``steals\'' a reference to *o*.

int **_PyTuple_Resize**( PyObject ***p*, Py_ssize_t *newsize*)

Can be used to resize a tuple. *newsize* will be the new length of the tuple. Because tuples are *supposed* to be immutable, this should only be used if there is only one reference to the object. Do *not* use this if the tuple may already be known to some other part of the code. The tuple will always grow or shrink at the end. Think of this as destroying the old tuple and creating a new one, only more efficiently. Returns 0 on success. Client code should never assume that the resulting value of *p* will be the same as before calling this function. If the object referenced by *p* is replaced, the original *p* is destroyed. On failure, returns -1 and sets *p* to NULL, and raises *MemoryError* or *SystemError*. Changed in version 2.2: Removed unused third parameter, *last_is_sticky*. 
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.3.5 List Objects

**PyListObject**
This subtype of *PyObject* represents a Python list object.

**PyTypeObject PyList_Type**
This instance of *PyTypeObject* represents the Python list type. This is the same object as *list* and *types.ListType* in the Python layer.

```c
int PyList_Check( PyObject *p)
```
Return true if *p* is a list object or an instance of a subtype of the list type. Changed in version 2.2: Allowed subtypes to be accepted.

```c
int PyList_CheckExact( PyObject *p)
```
Return true if *p* is a list object, but not an instance of a subtype of the list type. New in version 2.2.

```c
PyObject* PyList_New( Py_ssize_t len)
```
*Return value: New reference.*
Return a new list of length *len* on success, or NULL on failure. **Note:** If *length* is greater than zero, the returned list object's items are set to NULL. Thus you cannot use abstract API functions such as *PySequence_SetItem()* or expose the object to Python code before setting all items to a real object with *PyList_SetItem()*.

```c
Py_ssize_t PyList_Size( PyObject *list)
```
Return the length of the list object in *list*; this is equivalent to "\len(list)\" on a list object.

```c
Py_ssize_t PyList_GET_SIZE( PyObject *list)
```
Macro form of *PyList_Size()* without error checking.

```c
PyObject* PyList_GetItem( PyObject *list, Py_ssize_t index)
```
*Return value: Borrowed reference.*
Return the object at position *pos* in the list pointed to by *p*. The position must be positive, indexing from the end of the list is not supported. If *pos* is
out of bounds, return NULL and set an IndexError exception.

PyObject* **PyList_GET_ITEM** (PyObject *list, Py_ssize_t i)

*Return value: Borrowed reference.*
Macro form of PyList_GetItem() without error checking.

int **PyList_SetItem** (PyObject *list, Py_ssize_t index, PyObject *item)

Set the item at index index in list to item. Return 0 on success or -1 on failure. **Note:** This function `"steals"` a reference to item and discards a reference to an item already in the list at the affected position.

void **PyList_SET_ITEM** (PyObject *list, Py_ssize_t i, PyObject *o)

Macro form of PyList_SetItem() without error checking. This is normally only used to fill in new lists where there is no previous content. **Note:** This function `"steals"` a reference to item, and, unlike PyList_SetItem(), does not discard a reference to any item that it being replaced; any reference in list at position i will be leaked.

int **PyList_Insert** (PyObject *list, Py_ssize_t index, PyObject *item)

Insert the item item into list list in front of index index. Return 0 if successful; return -1 and set an exception if unsuccessful. Analogous to list.insert(index, item).

int **PyList_Append** (PyObject *list, PyObject *item)

Append the object item at the end of list list. Return 0 if successful; return -1 and set an exception if unsuccessful. Analogous to list.append(item).

PyObject* **PyList_GetSlice** (PyObject *list, Py_ssize_t low, Py_ssize_t high)

*Return value: New reference.*
Return a list of the objects in list containing the objects between low and high. Return NULL and set an exception if unsuccessful. Analogous to list[low:high].

int **PyList_SetSlice** (PyObject *list, Py_ssize_t low, Py_ssize_t high, PyObject *itemlist)
Set the slice of list between low and high to the contents of itemlist. Analogous to list[low:high] = itemlist. The itemlist may be NULL, indicating the assignment of an empty list (slice deletion). Return 0 on success, -1 on failure.

int PyList_Sort( PyObject *list)
Sort the items of list in place. Return 0 on success, -1 on failure. This is equivalent to "list.sort()".

int PyList_Reverse( PyObject *list)
Reverse the items of list in place. Return 0 on success, -1 on failure. This is the equivalent of "list.reverse()".

PyObject* PyList_AsTuple( PyObject *list)
Return a new tuple object containing the contents of list; equivalent to "tuple(list)".
7.4 Mapping Objects

Subsections

- 7.4.1 Dictionary Objects

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
# 7.4.1 Dictionary Objects

**PyDictObject**
This subtype of PyObject represents a Python dictionary object.

**PyTypeObject PyDict_Type**
This instance of PyTypeObject represents the Python dictionary type. This is exposed to Python programs as dict and types.DictType.

```c
int PyDict_Check( PyObject *p )
    Return true if p is a dict object or an instance of a subtype of the dict type. Changed in version 2.2: Allowed subtypes to be accepted.
```

```c
int PyDict_CheckExact( PyObject *p )
    Return true if p is a dict object, but not an instance of a subtype of the dict type. New in version 2.4.
```

```c
PyObject* PyDict_New()
    Return a new empty dictionary, or NULL on failure.
```

```c
PyObject* PyDictProxy_New( PyObject *dict )
    Return a proxy object for a mapping which enforces read-only behavior. This is normally used to create a proxy to prevent modification of the dictionary for non-dynamic class types. New in version 2.2.
```

```c
void PyDict_Clear( PyObject *p )
    Empty an existing dictionary of all key-value pairs.
```

```c
int PyDict_Contains( PyObject *p, PyObject *key )
    Determine if dictionary p contains key. If an item in p is matches key, return 1, otherwise return 0. On error, return -1. This is equivalent to the Python expression "key in p". New in version 2.4.
```

```c
PyObject* PyDict_Copy( PyObject *p )
```
Return value: *New reference.*
Return a new dictionary that contains the same key-value pairs as \( p \). New in version 1.6.

```c
int PyDict_SetItem( PyObject *p, PyObject *key, PyObject *val )
```

Insert value into the dictionary \( p \) with a key of \( key \). \( key \) must be hashable; if it isn't, TypeError will be raised. Return 0 on success or -1 on failure.

```c
int PyDict_SetItemString( PyObject *p, const char *key, PyObject *val )
```

Insert value into the dictionary \( p \) using \( key \) as a key. \( key \) should be a char*. The key object is created using `PyString_FromString(key)`. Return 0 on success or -1 on failure.

```c
int PyDict_DelItem( PyObject *p, PyObject *key )
```

Remove the entry in dictionary \( p \) with key \( key \). \( key \) must be hashable; if it isn't, TypeError is raised. Return 0 on success or -1 on failure.

```c
int PyDict_DelItemString( PyObject *p, char *key )
```

Remove the entry in dictionary \( p \) which has a key specified by the string \( key \). Return 0 on success or -1 on failure.

```c
PyObject* PyDict_GetItem( PyObject *p, PyObject *key )
```

*Return value: *Borrowed reference.*
Return the object from dictionary \( p \) which has a key \( key \). Return NULL if the key \( key \) is not present, but without setting an exception.

```c
PyObject* PyDict_GetItemString( PyObject *p, const char *key )
```

*Return value: *Borrowed reference.*
This is the same as `PyDict_GetItem()`, but \( key \) is specified as a char*, rather than a PyObject*.

```c
PyObject* PyDict_Items( PyObject *p )
```

*Return value: *New reference.*
Return a PyListObject containing all the items from the dictionary, as in the dictionary method `items()` (see the [Python Library Reference](https://docs.python.org/)).

```c
PyObject* PyDict_Keys( PyObject *p )
```
Return value: **New reference.**
Return a **PyListObject** containing all the keys from the dictionary, as in the dictionary method `keys()` (see the [Python Library Reference](#)).

**PyObject* PyDict_Values**( PyObject *p)

*Return value: New reference.*
Return a **PyListObject** containing all the values from the dictionary `p`, as in the dictionary method `values()` (see the [Python Library Reference](#)).

**Py_ssize_t PyDict_Size**( PyObject *p)

Return the number of items in the dictionary. This is equivalent to "len(p)" on a dictionary.

**int PyDict_Next**( PyObject *p, Py_ssize_t *ppos, PyObject **pkey, PyObject **pvalue)

Iterate over all key-value pairs in the dictionary `p`. The **int** referred to by `ppos` must be initialized to 0 prior to the first call to this function to start the iteration; the function returns true for each pair in the dictionary, and false once all pairs have been reported. The parameters `pkey` and `pvalue` should either point to **PyObject*** variables that will be filled in with each key and value, respectively, or may be NULL. Any references returned through them are borrowed. `ppos` should not be altered during iteration. Its value represents offsets within the internal dictionary structure, and since the structure is sparse, the offsets are not consecutive.

For example:

```c
PyObject *key, *value;
int pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}
```

The dictionary `p` should not be mutated during iteration. It is safe (since Python 2.1) to modify the values of the keys as you iterate over the dictionary, but only so long as the set of keys does not change. For example:
PyObject *key, *value;
int pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    int i = PyInt_AS_LONG(value) + 1;
    PyObject *o = PyInt_FromLong(i);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
    Py_DECREF(o);
}

int PyDict_Merge( PyObject *a, PyObject *b, int override)
    Iterate over mapping object b adding key-value pairs to dictionary a. b may
    be a dictionary, or any object supporting PyMapping_Keys() and
    PyObject_GetItem(). If override is true, existing pairs in a will be
    replaced if a matching key is found in b, otherwise pairs will only be added
    if there is not a matching key in a. Return 0 on success or -1 if an
    exception was raised. New in version 2.2.

int PyDict_Update( PyObject *a, PyObject *b)
    This is the same as PyDict_Merge(a, b, 1) in C, or a.update(b)
    in Python. Return 0 on success or -1 if an exception was raised. New in
    version 2.2.

int PyDict_MergeFromSeq2( PyObject *a, PyObject *seq2, int override)
    Update or merge into dictionary a, from the key-value pairs in seq2. seq2
    must be an iterable object producing iterable objects of length 2, viewed as
    key-value pairs. In case of duplicate keys, the last wins if override is true,
    else the first wins. Return 0 on success or -1 if an exception was raised.
    Equivalent Python (except for the return value):

    def PyDict_MergeFromSeq2(a, seq2, override):
        for key, value in seq2:
            if override or key not in a:
                a[key] = value

    New in version 2.2.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.5 Other Objects

Subsections

- 7.5.1 File Objects
- 7.5.2 Instance Objects
- 7.5.3 Function Objects
- 7.5.4 Method Objects
- 7.5.5 Module Objects
- 7.5.6 Iterator Objects
- 7.5.7 Descriptor Objects
- 7.5.8 Slice Objects
- 7.5.9 Weak Reference Objects
- 7.5.10 CObjects
- 7.5.11 Cell Objects
- 7.5.12 Generator Objects
- 7.5.13 DateTime Objects
- 7.5.14 Set Objects

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.5.1 File Objects

Python's built-in file objects are implemented entirely on the FILE* support from the C standard library. This is an implementation detail and may change in future releases of Python.

**PyFileObject**
This subtype of PyObject represents a Python file object.

**PyTypeObject PyFile_Type**
This instance of PyTypeObject represents the Python file type. This is exposed to Python programs as file and types.FileType.

```c
int PyFile_Check(PyObject *p)

Return true if its argument is a PyFileObject or a subtype of PyFileObject. Changed in version 2.2: Allowed subtypes to be accepted.
```

```c
int PyFile_CheckExact(PyObject *p)

Return true if its argument is a PyFileObject, but not a subtype of PyFileObject. New in version 2.2.
```

```c
PyObject* PyFile_FromString(char *filename, char *mode)

On success, return a new file object that is opened on the file given by filename, with a file mode given by mode, where mode has the same semantics as the standard C routine fopen(). On failure, return NULL.
```

```c
PyObject* PyFile_FromFile(FILE *fp, char *name, char *mode, int (*close)(FILE*))

Create a new PyFileObject from the already-open standard C file pointer, fp. The function close will be called when the file should be closed. Return NULL on failure.
```

```c
FILE* PyFile_AsFile(PyObject *p)
```
Return the file object associated with \( p \) as a `FILE*`.

**PyObject**\* `PyFile_GetLine`( PyObject \*\( p \), int \( n \))

*Return value: New reference.*

Equivalent to \( p.\text{readline}([n]) \), this function reads one line from the object \( p \). \( p \) may be a file object or any object with a `readline()` method. If \( n \) is \( 0 \), exactly one line is read, regardless of the length of the line. If \( n \) is greater than \( 0 \), no more than \( n \) bytes will be read from the file; a partial line can be returned. In both cases, an empty string is returned if the end of the file is reached immediately. If \( n \) is less than \( 0 \), however, one line is read regardless of length, but `EOFError` is raised if the end of the file is reached immediately.

**PyObject**\* `PyFile_Name`( PyObject \*\( p \))

*Return value: Borrowed reference.*

Return the name of the file specified by \( p \) as a string object.

**void** `PyFile_SetBufSize`( PyFileObject \*\( p \), int \( n \))

Available on systems with `setvbuf()` only. This should only be called immediately after file object creation.

**int** `PyFile_Encoding`( PyFileObject \*\( p \), char \*\( enc \))

Set the file's encoding for Unicode output to \( enc \). Return 1 on success and 0 on failure. New in version 2.3.

**int** `PyFile_SoftSpace`( PyObject \*\( p \), int \( newflag \))

This function exists for internal use by the interpreter. Set the `softspace` attribute of \( p \) to \( newflag \) and return the previous value. \( p \) does not have to be a file object for this function to work properly; any object is supported (thought its only interesting if the `softspace` attribute can be set). This function clears any errors, and will return \( 0 \) as the previous value if the attribute either does not exist or if there were errors in retrieving it. There is no way to detect errors from this function, but doing so should not be needed.

**int** `PyFile_WriteObject`( PyObject \*\( obj \), PyObject \*\( p \), int \( flags \))

Write object \( obj \) to file object \( p \). The only supported flag for \( flags \) is
Py_PRINT_RAW; if given, the str() of the object is written instead of the repr(). Return 0 on success or -1 on failure; the appropriate exception will be set.

int PyFile_WriteString( const char *s, PyObject *p)
Write string s to file object p. Return 0 on success or -1 on failure; the appropriate exception will be set.
7.5.2 Instance Objects

There are very few functions specific to instance objects.

PyTypeObject **PyInstance_Type**
Type object for class instances.

int **PyInstance_Check**( PyObject *obj)
Return true if obj is an instance.

PyObject* **PyInstance_New**( PyObject *class, PyObject *arg, PyObject *
*kw)
*Return value: New reference.*
Create a new instance of a specific class. The parameters arg and kw are used as the positional and keyword parameters to the object's constructor.

PyObject* **PyInstance_NewRaw**( PyObject *class, PyObject *
dict)
*Return value: New reference.*
Create a new instance of a specific class without calling its constructor. class is the class of new object. The dict parameter will be used as the object's __dict__; if NULL, a new dictionary will be created for the instance.
7.5.3 Function Objects

There are a few functions specific to Python functions.

**PyFunctionObject**
The C structure used for functions.

PyTypeObject **PyFunction_Type**
This is an instance of PyTypeObject and represents the Python function type. It is exposed to Python programmers as `types.FunctionType`.

```c
int PyFunction_Check( PyObject *o)
Return true if `o` is a function object (has type `PyFunction_Type`). The parameter must not be NULL.
```

```c
PyObject* PyFunction_New( PyObject *code, PyObject *globals)
Return a new function object associated with the code object `code`. `globals` must be a dictionary with the global variables accessible to the function.

The function's docstring, name and `__module__` are retrieved from the code object, the argument defaults and closure are set to NULL.
```

```c
PyObject* PyFunction_GetCode( PyObject *op)
Return value: Borrowed reference.
Return the code object associated with the function object `op`.
```

```c
PyObject* PyFunction_GetGlobals( PyObject *op)
Return value: Borrowed reference.
Return the globals dictionary associated with the function object `op`.
```

```c
PyObject* PyFunction_GetModule( PyObject *op)
Return value: Borrowed reference.
Return the `__module__` attribute of the function object `op`. This is normally a string containing the module name, but can be set to any other object by Python code.
PyObject* **PyFunction_GetDefaults**( PyObject *op)

*Return value: Borrowed reference.*

Return the argument default values of the function object *op*. This can be a tuple of arguments or NULL.

int **PyFunction_SetDefaults**( PyObject *op, PyObject *defaults)

Set the argument default values for the function object *op*. *defaults* must be *Py_None* or a tuple.

Raises SystemError and returns -1 on failure.

PyObject* **PyFunction_GetClosure**( PyObject *op)

*Return value: Borrowed reference.*

Return the closure associated with the function object *op*. This can be NULL or a tuple of cell objects.

int **PyFunction_SetClosure**( PyObject *op, PyObject *closure)

Set the closure associated with the function object *op*. *closure* must be *Py_None* or a tuple of cell objects.

Raises SystemError and returns -1 on failure.
7.5.4 Method Objects

There are some useful functions that are useful for working with method objects.

PyTypeObject **PyMethod_Type**

This instance of PyTypeObject represents the Python method type. This is exposed to Python programs as *types.MethodType*.

`int PyMethod_Check( PyObject *o)`

Return true if o is a method object (has type PyMethod_Type). The parameter must not be NULL.

`PyObject* PyMethod_New( PyObject *func, PyObject *self, PyObject *class)`

*Return value: New reference.*

Return a new method object, with *func* being any callable object; this is the function that will be called when the method is called. If this method should be bound to an instance, *self* should be the instance and *class* should be the class of *self*, otherwise *self* should be NULL and *class* should be the class which provides the unbound method..

`PyObject* PyMethod_Class( PyObject *meth)`

*Return value: Borrowed reference.*

Return the class object from which the method *meth* was created; if this was created from an instance, it will be the class of the instance.

`PyObject* PyMethod_GET_CLASS( PyObject *meth)`

*Return value: Borrowed reference.*

Macro version of PyMethod_Class() which avoids error checking.

`PyObject* PyMethod_Function( PyObject *meth)`

*Return value: Borrowed reference.*

Return the function object associated with the method *meth*.

`PyObject* PyMethod_GET_FUNCTION( PyObject *meth)`

*Return value: Borrowed reference.*

Macro version of PyMethod_Function() which avoids error checking.
(PyObject* PyMethod_Self( PyObject *meth))

Return value: Borrowed reference.
Return the instance associated with the method meth if it is bound, otherwise return NULL.

(PyObject* PyMethod_GET_SELF( PyObject *meth))

Return value: Borrowed reference.
Macro version of PyMethod_Self() which avoids error checking.
7.5.5 Module Objects

There are only a few functions special to module objects.

PyTypeObject PyModule_Type
   This instance of PyTypeObject represents the Python module type. This is exposed to Python programs as types.ModuleType.

int PyModule_Check( PyObject *p)
   Return true if p is a module object, or a subtype of a module object. Changed in version 2.2: Allowed subtypes to be accepted.

int PyModule_CheckExact( PyObject *p)
   Return true if p is a module object, but not a subtype of PyModule_Type. New in version 2.2.

PyObject* PyModule_New( const char *name)
   Return a new module object with the __name__ attribute set to name. Only the module's __doc__ and __name__ attributes are filled in; the caller is responsible for providing a __file__ attribute.

PyObject* PyModule_GetDict( PyObject *module)
   Return value: Borrowed reference.
   Return the dictionary object that implements module's namespace; this object is the same as the __dict__ attribute of the module object. This function never fails. It is recommended extensions use other PyModule_*() and PyObject_*() functions rather than directly manipulate a module's __dict__.

char* PyModule_GetName( PyObject *module)
   Return module's __name__ value. If the module does not provide one, or if it is not a string, SystemError is raised and NULL is returned.

char* PyModule_GetFilename( PyObject *module)
   Return the name of the file from which module was loaded using module's
__file__ attribute. If this is not defined, or if it is not a string, raise SystemError and return NULL.

int PyModule_AddObject(PyObject *module, const char *name, PyObject *value)

Add an object to module as name. This is a convenience function which can be used from the module's initialization function. This steals a reference to value. Return -1 on error, 0 on success. New in version 2.0.

int PyModule_AddIntConstant(PyObject *module, const char *name, long value)

Add an integer constant to module as name. This convenience function can be used from the module's initialization function. Return -1 on error, 0 on success. New in version 2.0.

int PyModule_AddStringConstant(PyObject *module, const char *name, const char *value)

Add a string constant to module as name. This convenience function can be used from the module's initialization function. The string value must be null-terminated. Return -1 on error, 0 on success. New in version 2.0.
7.5.6 Iterator Objects

Python provides two general-purpose iterator objects. The first, a sequence iterator, works with an arbitrary sequence supporting the `__getitem__()` method. The second works with a callable object and a sentinel value, calling the callable for each item in the sequence, and ending the iteration when the sentinel value is returned.

PyTypeObject **PySeqIter_Type**

Type object for iterator objects returned by `PySeqIter_New()` and the one-argument form of the `iter()` built-in function for built-in sequence types. New in version 2.2.

```
int PySeqIter_Check( op)
Return true if the type of op is PySeqIter_Type. New in version 2.2.
```

```
PyObject* PySeqIter_New( PyObject *seq)
Return an iterator that works with a general sequence object, seq. The iteration ends when the sequence raises IndexError for the subscripting operation. New in version 2.2.
```

PyTypeObject **PyCallIter_Type**

Type object for iterator objects returned by `PyCallIter_New()` and the two-argument form of the `iter()` built-in function. New in version 2.2.

```
int PyCallIter_Check( op)
Return true if the type of op is PyCallIter_Type. New in version 2.2.
```

```
PyObject* PyCallIter_New( PyObject *callable, PyObject *sentinel)
Return a new iterator. The first parameter, callable, can be any Python callable object that can be called with no parameters; each call to it should return the next item in the iteration. When callable returns a value equal to sentinel, the iteration will be terminated. New in version 2.2.
```
7.5.7 Descriptor Objects

`Descriptors" are objects that describe some attribute of an object. They are found in the dictionary of type objects.

PyTypeObject **PyProperty>Type**
The type object for the built-in descriptor types. New in version 2.2.

PyObject* **PyDescr_NewGetSet**(PyTypeObject *type, struct PyGetSetDef *getset)

*Return value: New reference.*
New in version 2.2.

PyObject* **PyDescr_NewMember**(PyTypeObject *type, struct PyMemberDef *meth)

*Return value: New reference.*
New in version 2.2.

PyObject* **PyDescr_NewMethod**(PyTypeObject *type, struct PyMethodDef *meth)

*Return value: New reference.*
New in version 2.2.

PyObject* **PyDescr_NewWrapper**(PyTypeObject *type, struct wrapperbase *wrapper, void *wrapped)

*Return value: New reference.*
New in version 2.2.

PyObject* **PyDescr_NewClassMethod**(PyTypeObject *type, PyMethodDef *method)

*Return value: New reference.*
New in version 2.3.

int **PyDescr_IsData**(PyObject *descr)

Return true if the descriptor objects *descr* describes a data attribute, or false if it describes a method. *descr* must be a descriptor object; there is no error
checking. New in version 2.2.

PyObject* PyWrapper_New( PyObject *, PyObject *)

New in version 2.2.
7.5.8 Slice Objects

PyTypeObject **PySlice_Type**
The type object for slice objects. This is the same as `slice` and `types.SliceType`.

int **PySlice_Check**( PyObject *ob)
Return true if `ob` is a slice object; `ob` must not be NULL.

PyObject* **PySlice_New**( PyObject *start, PyObject *stop, PyObject *step)

*Return value: New reference.*

Return a new slice object with the given values. The `start`, `stop`, and `step` parameters are used as the values of the slice object attributes of the same names. Any of the values may be NULL, in which case the `None` will be used for the corresponding attribute. Return NULL if the new object could not be allocated.

PySliceObject *slice, Py_ssize_t length,
int **PySlice_GetIndices**( Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t *step)

Retrieve the start, stop and step indices from the slice object `slice`, assuming a sequence of length `length`. Treats indices greater than `length` as errors.

Returns 0 on success and -1 on error with no exception set (unless one of the indices was not `None` and failed to be converted to an integer, in which case -1 is returned with an exception set).

You probably do not want to use this function. If you want to use slice objects in versions of Python prior to 2.3, you would probably do well to incorporate the source of `PySlice_GetIndicesEx`, suitably renamed, in the source of your extension.

PySliceObject *slice, Py_ssize_t length,
int **PySlice_GetIndicesEx**( Py_ssize_t *start, Py_ssize_t *stop,
    Py_ssize_t *step, Py_ssize_t *slicelength)

Usable replacement for `PySlice_GetIndices`. Retrieve the start, stop,
and step indices from the slice object \textit{slice} assuming a sequence of length \textit{length}, and store the length of the slice in \textit{slicelength}. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Returns 0 on success and -1 on error with exception set.

New in version 2.3.
7.5.9 Weak Reference Objects

Python supports weak references as first-class objects. There are two specific object types which directly implement weak references. The first is a simple reference object, and the second acts as a proxy for the original object as much as it can.

```c
int PyWeakref_Check( ob)
    Return true if ob is either a reference or proxy object. New in version 2.2.

int PyWeakref_CheckRef( ob)
    Return true if ob is a reference object. New in version 2.2.

int PyWeakref_CheckProxy( ob)
    Return true if ob is a proxy object. New in version 2.2.
```

```c
PyObject* PyWeakref_NewRef( PyObject *ob, PyObject *callback)
    Return a weak reference object for the object ob. This will always return a new reference, but is not guaranteed to create a new object; an existing reference object may be returned. The second parameter, callback, can be a callable object that receives notification when ob is garbage collected; it should accept a single parameter, which will be the weak reference object itself. callback may also be None or NULL. If ob is not a weakly-referencable object, or if callback is not callable, None, or NULL, this will return NULL and raise TypeError. New in version 2.2.
```

```c
PyObject* PyWeakref_NewProxy( PyObject *ob, PyObject *callback)
    Return a weak reference proxy object for the object ob. This will always return a new reference, but is not guaranteed to create a new object; an existing proxy object may be returned. The second parameter, callback, can be a callable object that receives notification when ob is garbage collected; it should accept a single parameter, which will be the weak reference object itself. callback may also be None or NULL. If ob is not a weakly-referencable object, or if callback is not callable, None, or NULL, this will
return NULL and raise TypeError. New in version 2.2.

PyObject* **PyWeakref_GetObject** (PyObject *ref)

*Return value: Borrowed reference.*

Return the referenced object from a weak reference, *ref*. If the referent is no longer live, returns None. New in version 2.2.

PyObject* **PyWeakref_GET_OBJECT** (PyObject *ref)

*Return value: Borrowed reference.*

Similar to PyWeakref_GetObject(), but implemented as a macro that does no error checking. New in version 2.2.
7.5.10 CObjects

Refer to *Extending and Embedding the Python Interpreter*, section 1.12, "Providing a C API for an Extension Module," for more information on using these objects.

**PyCObject**

This subtype of PyObject represents an opaque value, useful for C extension modules who need to pass an opaque value (as a void* pointer) through Python code to other C code. It is often used to make a C function pointer defined in one module available to other modules, so the regular import mechanism can be used to access C APIs defined in dynamically loaded modules.

int `PyCObject_Check`( PyObject *p)

Return true if its argument is a PyCObject.

PyObject* `PyCObject_FromVoidPtr`( void* obj, void (*destr)(void *))

*Return value: New reference.*

Create a PyCObject from the void *obj. The destr function will be called when the object is reclaimed, unless it is NULL.

PyObject* `PyCObject_FromVoidPtrAndDesc`( void *obj, void* desc, void (*destr)(void *, void *))

*Return value: New reference.*

Create a PyCObject from the void *obj. The destr function will be called when the object is reclaimed. The desc argument can be used to pass extra callback data for the destructor function.

void* `PyCObject_AsVoidPtr`( PyObject* self)

Return the object void * that the PyCObject self was created with.

void* `PyCObject_GetDesc`( PyObject* self)

Return the description void * that the PyCObject self was created with.
int PyCObject_SetVoidPtr(PyObject* self, void* cobj)

Set the void pointer inside *self* to *cobj*. The PyCObject must not have an
associated destructor. Return true on success, false on failure.
7.5.11 Cell Objects

``Cell'' objects are used to implement variables referenced by multiple scopes. For each such variable, a cell object is created to store the value; the local variables of each stack frame that references the value contains a reference to the cells from outer scopes which also use that variable. When the value is accessed, the value contained in the cell is used instead of the cell object itself. This de-referencing of the cell object requires support from the generated byte-code; these are not automatically de-referenced when accessed. Cell objects are not likely to be useful elsewhere.

**PyCellObject**
The C structure used for cell objects.

PyTypeObject **PyCell_Type**
The type object corresponding to cell objects.

```c
int PyCell_Check( PyObject *ob )
    Return true if ob is a cell object; ob must not be NULL.
```

```c
PyObject* PyCell_New( PyObject *ob )
    Create and return a new cell object containing the value ob. The parameter may be NULL.
```

```c
PyObject* PyCell_Get( PyObject *cell )
    Return the contents of the cell cell.
```

```c
PyObject* PyCell_GET( PyObject *cell )
    Return value: Borrowed reference.
    Return the contents of the cell cell, but without checking that cell is non-NULL and a cell object.
```

```c
int PyCell_Set( PyObject *cell, PyObject *value )
    Set the contents of the cell object cell to value. This releases the reference
to any current content of the cell. value may be NULL. cell must be non-NULL; if it is not a cell object, -1 will be returned. On success, 0 will be returned.

void **PyCell_SET**( PyObject *cell, PyObject *value)
Sets the value of the cell object cell to value. No reference counts are adjusted, and no checks are made for safety; cell must be non-NULL and must be a cell object.
7.5.12 Generator Objects

Generator objects are what Python uses to implement generator iterators. They are normally created by iterating over a function that yields values, rather than explicitly calling PyGen_New.

**PyGenObject**

The C structure used for generator objects.

**PyTypeObject PyGen_Type**

The type object corresponding to generator objects

```c
int PyGen_Check( ob)
    Return true if ob is a generator object; ob must not be NULL.
```

```c
int PyGen_CheckExact( ob)
    Return true if ob's type is PyGen_Type is a generator object; ob must not be NULL.
```

```c
PyObject* PyGen_New( PyFrameObject *frame)
    Create and return a new generator object based on the frame object. A reference to frame is stolen by this function. The parameter must not be NULL.
```
7.5.13 DateTime Objects

Various date and time objects are supplied by the datetime module. Before using any of these functions, the header file datetime.h must be included in your source (note that this is not include by Python.h), and macro PyDateTime_IMPORT() must be invoked. The macro arranges to put a pointer to a C structure in a static variable PyDateTimeAPI, which is used by the following macros.

Type-check macros:

int PyDate_Check( PyObject *ob)
    Return true if ob is of type PyDateTime_DateType or a subtype of PyDateTime_DateType. ob must not be NULL. New in version 2.4.

int PyDate_CheckExact( PyObject *ob)
    Return true if ob is of type PyDateTime_DateType. ob must not be NULL. New in version 2.4.

int PyDateTime_Check( PyObject *ob)
    Return true if ob is of type PyDateTime_DateTimeType or a subtype of PyDateTime_DateTimeType. ob must not be NULL. New in version 2.4.

int PyDateTime_CheckExact( PyObject *ob)
    Return true if ob is of type PyDateTime_DateTimeType. ob must not be NULL. New in version 2.4.

int PyTime_Check( PyObject *ob)
    Return true if ob is of type PyDateTime_TimeType or a subtype of PyDateTime_TimeType. ob must not be NULL. New in version 2.4.

int PyTime_CheckExact( PyObject *ob)
    Return true if ob is of type PyDateTime_TimeType. ob must not be NULL. New in version 2.4.
int **PyDelta**.Check( PyObject *ob)
Return true if ob is of type **PyDateTime**.DeltaType or a subtype of
**PyDateTime**.DeltaType. ob must not be NULL. New in version 2.4.

int **PyDelta**.CheckExact( PyObject *ob)
Return true if ob is of type **PyDateTime**.DeltaType. ob must not be
NULL. New in version 2.4.

int **PyTZInfo**.Check( PyObject *ob)
Return true if ob is of type **PyDateTime**.TZInfoType or a subtype of
**PyDateTime**.TZInfoType. ob must not be NULL. New in version 2.4.

int **PyTZInfo**.CheckExact( PyObject *ob)
Return true if ob is of type **PyDateTime**.TZInfoType. ob must not be
NULL. New in version 2.4.

Macros to create objects:

PyObject* **PyDate**.FromDate( int year, int month, int day)
Return a datetime.date object with the specified year, month and day.
New in version 2.4.

PyObject* **PyDateTime**.FromDateAndTime( int hour, int minute, int
second, int usecond)
Return a datetime.datetime object with the specified year, month,
day, hour, minute, second and microsecond. New in version 2.4.

PyObject* **PyTime**.FromTime( int hour, int minute, int second, int usecond)
Return a datetime.time object with the specified hour, minute, second
and microsecond. New in version 2.4.

PyObject* **PyDelta**.FromDSU( int days, int seconds, int useconds)
Return a `datetime.timedelta` object representing the given number of days, seconds and microseconds. Normalization is performed so that the resulting number of microseconds and seconds lie in the ranges documented for `datetime.timedelta` objects. New in version 2.4.

Macros to extract fields from date objects. The argument must be an instance of `PyDateTime_Date`, including subclasses (such as `PyDateTime_DateTime`). The argument must not be `NULL`, and the type is not checked:

```c
int PyDateTime_GET_YEAR( PyDateTime_Date *o)
    return the year, as a positive int. New in version 2.4.
```

```c
int PyDateTime_GET_MONTH( PyDateTime_Date *o)
    return the month, as an int from 1 through 12. New in version 2.4.
```

```c
int PyDateTime_GET_DAY( PyDateTime_Date *o)
    return the day, as an int from 1 through 31. New in version 2.4.
```

Macros to extract fields from datetime objects. The argument must be an instance of `PyDateTime_DateTime`, including subclasses. The argument must not be `NULL`, and the type is not checked:

```c
int PyDateTime_DATE_GET_HOUR( PyDateTime_DateTime *o)
    return the hour, as an int from 0 through 23. New in version 2.4.
```

```c
int PyDateTime_DATE_GET_MINUTE( PyDateTime_DateTime *o)
    return the minute, as an int from 0 through 59. New in version 2.4.
```

```c
int PyDateTime_DATE_GET_SECOND( PyDateTime_DateTime *o)
    return the second, as an int from 0 through 59. New in version 2.4.
```

```c
int PyDateTime_DATE_GET_MICROSECOND( PyDateTime_DateTime *o)
    return the microsecond, as an int from 0 through 999999. New in version 2.4.
```

Macros to extract fields from time objects. The argument must be an instance of `PyDateTime_Time`, including subclasses. The argument must not be `NULL`,
and the type is not checked:

```c
int PyDateTime_TIME_GET_HOUR( PyDateTime_Time *o)
    Return the hour, as an int from 0 through 23. New in version 2.4.
```

```c
int PyDateTime_TIME_GET_MINUTE( PyDateTime_Time *o)
    Return the minute, as an int from 0 through 59. New in version 2.4.
```

```c
int PyDateTime_TIME_GET_SECOND( PyDateTime_Time *o)
    Return the second, as an int from 0 through 59. New in version 2.4.
```

```c
int PyDateTime_TIME_GET_MICROSECOND( PyDateTime_Time *o)
    Return the microsecond, as an int from 0 through 999999. New in version 2.4.
```

Macros for the convenience of modules implementing the DB API:

```c
PyObject* PyDateTime_FromTimestamp( PyObject *args)
    Create and return a new datetime.datetime object given an argument tuple suitable for passing to
datetime.datetime.fromtimestamp(). New in version 2.4.
```

```c
PyObject* PyDate_FromTimestamp( PyObject *args)
    Create and return a new datetime.date object given an argument tuple suitable for passing to
datetime.date.fromtimestamp(). New in version 2.4.
```
7.5.14 Set Objects

New in version 2.5.

This section details the public API for set and frozenset objects. Any functionality not listed below is best accessed using the either the abstract object protocol (including PyObject_CallMethod(), PyObject_RichCompareBool(), PyObject_Repr(), PyObject_IsTrue(), PyObject_Print(), and PyObject_GetIter()) or the abstract number protocol (including PyNumber_Add(), PyNumber_Subtract(), PyNumber_Or(), PyNumber_Xor(), PyNumber_InPlaceAdd(), PyNumber_InPlaceSubtract(), PyNumber_InPlaceOr(), and PyNumber_InPlaceXor()).

**PySetObject**
This subtype of PyObject is used to hold the internal data for both set and frozenset objects. It is like a PyDictObject in that it is a fixed size for small sets (much like tuple storage) and will point to a separate, variable sized block of memory for medium and large sized sets (much like list storage). None of the fields of this structure should be considered public and are subject to change. All access should be done through the documented API rather than by manipulating the values in the structure.

PyTypeObject **PySet_Type**
This is an instance of PyTypeObject representing the Python set type.

PyTypeObject **PyFrozenSet_Type**
This is an instance of PyTypeObject representing the Python frozenset type.

The following type check macros work on pointers to any Python object. Likewise, the constructor functions work with any iterable Python object.

```
int PyAnySet_Check( PyObject *p)
Return true if p is a set object, a frozenset object, or an instance of a subtype.
```

int PyAnySet_CheckExact( PyObject *p)
    Return true if p is a set object or a frozenset object but not an instance of a subtype.

int PyFrozenSet_CheckExact( PyObject *p)
    Return true if p is a frozenset object but not an instance of a subtype.

PyObject* PySet_New( PyObject *iterable)
    Return a new set containing objects returned by the iterable. The iterable may be NULL to create a new empty set. Return the new set on success or NULL on failure. Raise TypeError if iterable is not actually iterable. The constructor is also useful for copying a set (c=set(s)).

PyObject* PyFrozenSet_New( PyObject *iterable)
    Return a new frozenset containing objects returned by the iterable. The iterable may be NULL to create a new empty frozenset. Return the new set on success or NULL on failure. Raise TypeError if iterable is not actually iterable.

The following functions and macros are available for instances of set or frozenset or instances of their subtypes.

int PySet_Size( PyObject *anyset)
    Return the length of a set or frozenset object. Equivalent to "len(anyset)". Raises a PyExc_SystemError if anyset is not a set, frozenset, or an instance of a subtype.

int PySet_GET_SIZE( PyObject *anyset)
    Macro form of PySet_Size() without error checking.

int PySet_Contains( PyObject *anyset, PyObject *key)
    Return 1 if found, 0 if not found, and -1 if an error is encountered. Unlike the Python __contains__() method, this function does not automatically convert unhashable sets into temporary frozensets. Raise a TypeError if the key is unhashable. Raise PyExc_SystemError if
anynet is not a set, frozenset, or an instance of a subtype.

The following functions are available for instances of set or its subtypes but not for instances of frozenset or its subtypes.

int PySet_Add( PyObject *set, PyObject *key)

Add key to a set instance. Does not apply to frozenset instances. Return 0 on success or -1 on failure. Raise a TypeError if the key is unhashable. Raise a MemoryError if there is no room to grow. Raise a SystemError if set is an not an instance of set or its subtype.

int PySet_Discard( PyObject *set, PyObject *key)

Return 1 if found and removed, 0 if not found (no action taken), and -1 if an error is encountered. Does not raise KeyError for missing keys. Raise a TypeError if the key is unhashable. Unlike the Python discard() method, this function does not automatically convert unhashable sets into temporary frozensets. Raise PyExc_SystemError if set is an not an instance of set or its subtype.

PyObject* PySet_Pop( PyObject *set)

Return a new reference to an arbitrary object in the set, and removes the object from the set. Return NULL on failure. Raise KeyError if the set is empty. Raise a SystemError if set is an not an instance of set or its subtype.

int PySet_Clear( PyObject *set)

Empty an existing set of all elements.
8. Initialization, Finalization, and Threads

void **Py_Initialize**()

Initialize the Python interpreter. In an application embedding Python, this should be called before using any other Python/C API functions; with the exception of `Py_SetProgramName()`, `PyEval_InitThreads()`, `PyEval_ReleaseLock()`, and `PyEval_AcquireLock()`. This initializes the table of loaded modules (`sys.modules`), and creates the fundamental modules `__builtin__`, `__main__` and `sys`. It also initializes the module search path (`sys.path`). It does not set `sys.argv`; use `PySys_SetArgv()` for that. This is a no-op when called for a second time (without calling `Py_Finalize()` first). There is no return value; it is a fatal error if the initialization fails.

void **Py_InitializeEx**( int initsigs)

This function works like `Py_Initialize()` if `initsigs` is 1. If `initsigs` is 0, it skips initialization registration of signal handlers, which might be useful when Python is embedded. New in version 2.4.

int **Py_IsInitialized**()

Return true (nonzero) when the Python interpreter has been initialized, false (zero) if not. After `Py_Finalize()` is called, this returns false until `Py_Initialize()` is called again.

void **Py_Finalize**()

Undo all initializations made by `Py_Initialize()` and subsequent use of Python/C API functions, and destroy all sub-interpreters (see `Py_NewInterpreter()` below) that were created and not yet destroyed since the last call to `Py_Initialize()`. Ideally, this frees all memory allocated by the Python interpreter. This is a no-op when called for a second time (without calling `Py_Initialize()` again first). There is no return value; errors during finalization are ignored.
This function is provided for a number of reasons. An embedding application might want to restart Python without having to restart the application itself. An application that has loaded the Python interpreter from a dynamically loadable library (or DLL) might want to free all memory allocated by Python before unloading the DLL. During a hunt for memory leaks in an application a developer might want to free all memory allocated by Python before exiting from the application.

**Bugs and caveats:** The destruction of modules and objects in modules is done in random order; this may cause destructors (__del__ methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once; this can happen if an application calls Py_Initialize() and Py_Finalize() more than once.

**PyThreadState** *Py_NewInterpreter()**

Create a new sub-interpreter. This is an (almost) totally separate environment for the execution of Python code. In particular, the new interpreter has separate, independent versions of all imported modules, including the fundamental modules __builtin__, __main__ and sys. The table of loaded modules (sys.modules) and the module search path (sys.path) are also separate. The new environment has no sys.argv variable. It has new standard I/O stream file objects sys.stdin, sys.stdout and sys.stderr (however these refer to the same underlying FILE structures in the C library).

The return value points to the first thread state created in the new sub-interpreter. This thread state is made in the current thread state. Note that no actual thread is created; see the discussion of thread states below. If creation of the new interpreter is unsuccessful, NULL is returned; no exception is set since the exception state is stored in the current thread state and there may not be a current thread state. (Like all other Python/C API functions, the global interpreter lock must be held before calling this function and is still
Extension modules are shared between (sub-)interpreters as follows: the first time a particular extension is imported, it is initialized normally, and a (shallow) copy of its module's dictionary is squirreled away. When the same extension is imported by another (sub-)interpreter, a new module is initialized and filled with the contents of this copy; the extension's \texttt{init} function is not called. Note that this is different from what happens when an extension is imported after the interpreter has been completely re-initialized by calling \texttt{Py_Finalize()} and \texttt{Py_Initialize()}; in that case, the extension's \texttt{initmodule} function is called again.

\textbf{Bugs and caveats:} Because sub-interpreters (and the main interpreter) are part of the same process, the insulation between them isn't perfect -- for example, using low-level file operations like \texttt{os.close()} they can (accidentally or maliciously) affect each other's open files. Because of the way extensions are shared between (sub-)interpreters, some extensions may not work properly; this is especially likely when the extension makes use of (static) global variables, or when the extension manipulates its module's dictionary after its initialization. It is possible to insert objects created in one sub-interpreter into a namespace of another sub-interpreter; this should be done with great care to avoid sharing user-defined functions, methods, instances or classes between sub-interpreters, since import operations executed by such objects may affect the wrong (sub-)interpreter's dictionary of loaded modules. (XXX This is a hard-to-fix bug that will be addressed in a future release.)

Also note that the use of this functionality is incompatible with extension modules such as PyObjC and ctypes that use the \texttt{PyGILState_*} APIs (and this is inherent in the way the \texttt{PyGILState_*} functions work). Simple things may work, but confusing behavior will always be near.

void \texttt{Py_EndInterpreter( PyThreadState *tstate)}

Destroy the (sub-)interpreter represented by the given thread state. The given thread state must be the current thread state. See the discussion of thread states below. When the call returns, the current thread state is \texttt{NULL}. All thread states associated with this interpreter are destroyed. (The global interpreter lock must be held before calling this function and is still held
when it returns.) \texttt{Py_Finalize()} will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

\textbf{void \texttt{Py_SetProgramName}( char *name)}

This function should be called before \texttt{Py_Initialize()} is called for the first time, if it is called at all. It tells the interpreter the value of the \texttt{argv[0]} argument to the \texttt{main()} function of the program. This is used by \texttt{Py_GetPath()} and some other functions below to find the Python run-time libraries relative to the interpreter executable. The default value is 'python'. The argument should point to a zero-terminated character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

\textbf{char* \texttt{Py_GetProgramName}()}

Return the program name set with \texttt{Py_SetProgramName()}, or the default. The returned string points into static storage; the caller should not modify its value.

\textbf{char* \texttt{Py_GetPrefix}()}

Return the \textit{prefix} for installed platform-independent files. This is derived through a number of complicated rules from the program name set with \texttt{Py_SetProgramName()} and some environment variables; for example, if the program name is '/usr/local/bin/python', the prefix is '/usr/local'. The returned string points into static storage; the caller should not modify its value. This corresponds to the prefix variable in the top-level Makefile and the --prefix argument to the configure script at build time. The value is available to Python code as \texttt{sys.prefix}. It is only useful on UNIX. See also the next function.

\textbf{char* \texttt{Py_GetExecPrefix}()}

Return the \textit{exec-prefix} for installed platform-dependent files. This is derived through a number of complicated rules from the program name set with \texttt{Py_SetProgramName()} and some environment variables; for example, if the program name is '/usr/local/bin/python', the exec-prefix is '/usr/local'. The returned string points into static storage; the caller should not modify its value. This corresponds to the exec_prefix variable in
the top-level Makefile and the --exec-prefix argument to the configure script at build time. The value is available to Python code as sys.exec_prefix. It is only useful on Unix.

Background: The exec-prefix differs from the prefix when platform dependent files (such as executables and shared libraries) are installed in a different directory tree. In a typical installation, platform dependent files may be installed in the /usr/local/plat subtree while platform independent may be installed in /usr/local.

Generally speaking, a platform is a combination of hardware and software families, e.g. Sparc machines running the Solaris 2.x operating system are considered the same platform, but Intel machines running Solaris 2.x are another platform, and Intel machines running Linux are yet another platform. Different major revisions of the same operating system generally also form different platforms. Non-Unix operating systems are a different story; the installation strategies on those systems are so different that the prefix and exec-prefix are meaningless, and set to the empty string. Note that compiled Python bytecode files are platform independent (but not independent from the Python version by which they were compiled!).

System administrators will know how to configure the mount or automount programs to share /usr/local between platforms while having /usr/local/plat be a different filesystem for each platform.

char* Py_GetProgramFullPath()

Return the full program name of the Python executable; this is computed as a side-effect of deriving the default module search path from the program name (set by Py_SetProgramName() above). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as sys.executable.

char* Py_GetPath()

Return the default module search path; this is computed from the program name (set by Py_SetProgramName() above) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is "::" on Unix and Mac OS X, ";" on Windows. The returned
string points into static storage; the caller should not modify its value. The value is available to Python code as the list `sys.path`, which may be modified to change the future search path for loaded modules.

```c
const char* Py_GetVersion()
```

Return the version of this Python interpreter. This is a string that looks something like

```
"1.5 (#67, Dec 31 1997, 22:34:28) [GCC 2.7.2.2]"
```

The first word (up to the first space character) is the current Python version; the first three characters are the major and minor version separated by a period. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.version`.

```c
const char* Py_GetBuildNumber()
```

Return a string representing the Subversion revision that this Python executable was built from. This number is a string because it may contain a trailing 'M' if Python was built from a mixed revision source tree. New in version 2.5.

```c
const char* Py_GetPlatform()
```

Return the platform identifier for the current platform. On UNIX, this is formed from the ``official'' name of the operating system, converted to lower case, followed by the major revision number; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is 'sunos5'. On Mac OS X, it is 'darwin'. On Windows, it is 'win'. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.platform`.

```c
const char* Py_GetCopyright()
```

Return the official copyright string for the current Python version, for example

```
'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'
```

The returned string points into static storage; the caller should not modify
its value. The value is available to Python code as `sys.copyright`.

```
const char* Py_GetCompiler()
```

Return an indication of the compiler used to build the current Python version, in square brackets, for example:

```
"[GCC 2.7.2.2]"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

```
const char* Py_GetBuildInfo()
```

Return information about the sequence number and build date and time of the current Python interpreter instance, for example

```
"#67, Aug  1 1997, 22:34:28"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

```
void PySys_SetArgv( int argc, char **argv)
```

Set `sys.argv` based on `argc` and `argv`. These parameters are similar to those passed to the program's `main()` function with the difference that the first entry should refer to the script file to be executed rather than the executable hosting the Python interpreter. If there isn't a script that will be run, the first entry in `argv` can be an empty string. If this function fails to initialize `sys.argv`, a fatal condition is signalled using `Py_FatalError()`.

---

**Subsections**

- [8.1 Thread State and the Global Interpreter Lock](#)
- [8.2 Profiling and Tracing](#)
- [8.3 Advanced Debugger Support](#)
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
8.1 Thread State and the Global Interpreter Lock

The Python interpreter is not fully thread safe. In order to support multi-threaded Python programs, there’s a global lock that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

Therefore, the rule exists that only the thread that has acquired the global interpreter lock may operate on Python objects or call Python/C API functions. In order to support multi-threaded Python programs, the interpreter regularly releases and reacquires the lock -- by default, every 100 bytecode instructions (this can be changed with `sys.setcheckinterval()`). The lock is also released and reacquired around potentially blocking I/O operations like reading or writing a file, so that other threads can run while the thread that requests the I/O is waiting for the I/O operation to complete.

The Python interpreter needs to keep some bookkeeping information separate per thread -- for this it uses a data structure called `PyThreadState`. There's one global variable, however: the pointer to the current `PyThreadState` structure. While most thread packages have a way to store "per-thread global data," Python's internal platform independent thread abstraction doesn't support this yet. Therefore, the current thread state must be manipulated explicitly.

This is easy enough in most cases. Most code manipulating the global interpreter lock has the following simple structure:

```
Save the thread state in a local variable.
Release the interpreter lock.
...Do some blocking I/O operation...
Reacquire the interpreter lock.
Restore the thread state from the local variable.
```

This is so common that a pair of macros exists to simplify it:
The `Py_BEGIN_ALLOW_THREADS` macro opens a new block and declares a hidden local variable; the `Py_END_ALLOW_THREADS` macro closes the block. Another advantage of using these two macros is that when Python is compiled without thread support, they are defined empty, thus saving the thread state and lock manipulations.

When thread support is enabled, the block above expands to the following code:

```c
PyThreadState *_save;
_save = PyEval_SaveThread();
...Do some blocking I/O operation...
PyEval_RestoreThread(_save);
```

Using even lower level primitives, we can get roughly the same effect as follows:

```c
PyThreadState * _save;
_save = PyThreadState_Swap(NULL);
PyEval_ReleaseLock();
...Do some blocking I/O operation...
PyEval_AcquireLock();
PyThreadState_Swap(_save);
```

There are some subtle differences; in particular, `PyEval_RestoreThread()` saves and restores the value of the global variable `errno`, since the lock manipulation does not guarantee that `errno` is left alone. Also, when thread support is disabled, `PyEval_SaveThread()` and `PyEval_RestoreThread()` don't manipulate the lock; in this case, `PyEval_ReleaseLock()` and `PyEval_AcquireLock()` are not available. This is done so that dynamically loaded extensions compiled with thread support enabled can be loaded by an interpreter that was compiled with disabled thread support.

The global interpreter lock is used to protect the pointer to the current thread state. When releasing the lock and saving the thread state, the current thread state pointer must be retrieved before the lock is released (since another thread could immediately acquire the lock and store its own thread state in the global
variable). Conversely, when acquiring the lock and restoring the thread state, the lock must be acquired before storing the thread state pointer.

Why am I going on with so much detail about this? Because when threads are created from C, they don’t have the global interpreter lock, nor is there a thread state data structure for them. Such threads must bootstrap themselves into existence, by first creating a thread state data structure, then acquiring the lock, and finally storing their thread state pointer, before they can start using the Python/C API. When they are done, they should reset the thread state pointer, release the lock, and finally free their thread state data structure.

Beginning with version 2.3, threads can now take advantage of the PyGILState_*() functions to do all of the above automatically. The typical idiom for calling into Python from a C thread is now:

```c
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result */

/* Release the thread. No Python API allowed beyond this point. */
PyGILState_Release(gstate);
```

Note that the PyGILState_*() functions assume there is only one global interpreter (created automatically by Py_Initialize()). Python still supports the creation of additional interpreters (using Py_NewInterpreter()), but mixing multiple interpreters and the PyGILState_*() API is unsupported.

**PyInterpreterState**

This data structure represents the state shared by a number of cooperating threads. Threads belonging to the same interpreter share their module administration and a few other internal items. There are no public members in this structure.

Threads belonging to different interpreters initially share nothing, except process state like available memory, open file descriptors and such. The global interpreter lock is also shared by all threads, regardless of to which interpreter they belong.
**PyThreadState**

This data structure represents the state of a single thread. The only public data member is `PyInterpreterState * interp`, which points to this thread's interpreter state.

```c
void PyEval_InitThreads()
```

Initialize and acquire the global interpreter lock. It should be called in the main thread before creating a second thread or engaging in any other thread operations such as `PyEval_ReleaseLock()` or `PyEval_ReleaseThread(tstate)`. It is not needed before calling `PyEval_SaveThread()` or `PyEval_RestoreThread()`.

This is a no-op when called for a second time. It is safe to call this function before calling `Py_Initialize()`.

When only the main thread exists, no lock operations are needed. This is a common situation (most Python programs do not use threads), and the lock operations slow the interpreter down a bit. Therefore, the lock is not created initially. This situation is equivalent to having acquired the lock: when there is only a single thread, all object accesses are safe. Therefore, when this function initializes the lock, it also acquires it. Before the Python thread module creates a new thread, knowing that either it has the lock or the lock hasn't been created yet, it calls `PyEval_InitThreads()`. When this call returns, it is guaranteed that the lock has been created and that the calling thread has acquired it.

It is **not** safe to call this function when it is unknown which thread (if any) currently has the global interpreter lock.

This function is not available when thread support is disabled at compile time.

```c
int PyEval_ThreadsInitialized()
```

Returns a non-zero value if `PyEval_InitThreads()` has been called. This function can be called without holding the lock, and therefore can be used to avoid calls to the locking API when running single-threaded. This function is not available when thread support is disabled at compile time. New in version 2.4.
void PyEval_AcquireLock()

Acquire the global interpreter lock. The lock must have been created earlier. If this thread already has the lock, a deadlock ensues. This function is not available when thread support is disabled at compile time.

void PyEval_ReleaseLock()

Release the global interpreter lock. The lock must have been created earlier. This function is not available when thread support is disabled at compile time.

void PyEval_AcquireThread( PyThreadState *tstate)

Acquire the global interpreter lock and set the current thread state to tstate, which should not be NULL. The lock must have been created earlier. If this thread already has the lock, deadlock ensues. This function is not available when thread support is disabled at compile time.

void PyEval_ReleaseThread( PyThreadState *tstate)

Reset the current thread state to NULL and release the global interpreter lock. The lock must have been created earlier and must be held by the current thread. The tstate argument, which must not be NULL, is only used to check that it represents the current thread state -- if it isn't, a fatal error is reported. This function is not available when thread support is disabled at compile time.

PyThreadState* PyEval_SaveThread()

Release the interpreter lock (if it has been created and thread support is enabled) and reset the thread state to NULL, returning the previous thread state (which is not NULL). If the lock has been created, the current thread must have acquired it. (This function is available even when thread support is disabled at compile time.)

void PyEval_RestoreThread( PyThreadState *tstate)

Acquire the interpreter lock (if it has been created and thread support is enabled) and set the thread state to tstate, which must not be NULL. If the lock has been created, the current thread must not have acquired it, otherwise deadlock ensues. (This function is available even when thread support is disabled at compile time.)
The following macros are normally used without a trailing semicolon; look for example usage in the Python source distribution.

**Py_BEGIN_ALLOW_THREADS**
This macro expands to "\{ PyThreadState * _save; _save = PyEval_SaveThread(); \}". Note that it contains an opening brace; it must be matched with a following Py_END_ALLOW_THREADS macro. See above for further discussion of this macro. It is a no-op when thread support is disabled at compile time.

**Py_END_ALLOW_THREADS**
This macro expands to "PyEval_RestoreThread(_save); \}". Note that it contains a closing brace; it must be matched with an earlier Py_BEGIN_ALLOW_THREADS macro. See above for further discussion of this macro. It is a no-op when thread support is disabled at compile time.

**Py_BLOCK_THREADS**
This macro expands to "PyEval_RestoreThread(_save); ": it is equivalent to Py_END_ALLOW_THREADS without the closing brace. It is a no-op when thread support is disabled at compile time.

**Py_UNBLOCK_THREADS**
This macro expands to "_save = PyEval_SaveThread(); ": it is equivalent to Py_BEGIN_ALLOW_THREADS without the opening brace and variable declaration. It is a no-op when thread support is disabled at compile time.

All of the following functions are only available when thread support is enabled at compile time, and must be called only when the interpreter lock has been created.

PyInterpreterState* **PyInterpreterState_New( )**
Create a new interpreter state object. The interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

void **PyInterpreterState_Clear( PyInterpreterState *interp)**
Reset all information in an interpreter state object. The interpreter lock must be held.
void **PyInterpreterState_Delete** (PyInterpreterState *interp)

Destroy an interpreter state object. The interpreter lock need not be held. The interpreter state must have been reset with a previous call to **PyInterpreterState_Clear()**.

PyThreadState* **PyThreadState_New** (PyInterpreterState *interp)

Create a new thread state object belonging to the given interpreter object. The interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

void **PyThreadState_Clear** (PyThreadState *tstate)

Reset all information in a thread state object. The interpreter lock must be held.

void **PyThreadState_Delete** (PyThreadState *tstate)

Destroy a thread state object. The interpreter lock need not be held. The thread state must have been reset with a previous call to **PyThreadState_Clear()**.

PyThreadState* **PyThreadState_Get** ()

Return the current thread state. The interpreter lock must be held. When the current thread state is NULL, this issues a fatal error (so that the caller needn't check for NULL).

PyThreadState* **PyThreadState_Swap** (PyThreadState *tstate)

Swap the current thread state with the thread state given by the argument tstate, which may be NULL. The interpreter lock must be held.

PyObject* **PyThreadState_GetDict** ()

*Return value: Borrowed reference.*

Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no current thread state is available. If this function returns NULL, no exception has been raised and the caller should assume no current thread state is available. Changed in version 2.3: Previously this could only be called when a current thread is active, and NULL meant that an exception was raised.
int PyThreadState_SetAsyncExc( long id, PyObject *exc)

Asynchronously raise an exception in a thread. The "id" argument is the thread id of the target thread; "exc" is the exception object to be raised. This function does not steal any references to "exc". To prevent naive misuse, you must write your own C extension to call this. Must be called with the GIL held. Returns the number of thread states modified; this is normally one, but will be zero if the thread id isn't found. If "exc" is NULL, the pending exception (if any) for the thread is cleared. This raises no exceptions. New in version 2.3.

PyGILState_STATE PyGILState_Ensure()

Ensure that the current thread is ready to call the Python C API regardless of the current state of Python, or of its thread lock. This may be called as many times as desired by a thread as long as each call is matched with a call to PyGILState_Release(). In general, other thread-related APIs may be used between PyGILState_Ensure() and PyGILState_Release() calls as long as the thread state is restored to its previous state before the Release(). For example, normal usage of the Py_BEGIN_ALLOW_THREADS and Py_END_ALLOW_THREADS macros is acceptable.

The return value is an opaque "handle" to the thread state when PyGILState_Acquire() was called, and must be passed to PyGILState_Release() to ensure Python is left in the same state. Even though recursive calls are allowed, these handles cannot be shared - each unique call to PyGILState_Ensure must save the handle for its call to PyGILState_Release.

When the function returns, the current thread will hold the GIL. Failure is a fatal error. New in version 2.3.

void PyGILState_Release( PyGILState_STATE)

Release any resources previously acquired. After this call, Python's state will be the same as it was prior to the corresponding PyGILState_Ensure call (but generally this state will be unknown to the caller, hence the use of the GILState API.)

Every call to PyGILState_Ensure() must be matched by a call to
PyGILState_Release() on the same thread. New in version 2.3.
8.2 Profiling and Tracing

The Python interpreter provides some low-level support for attaching profiling and execution tracing facilities. These are used for profiling, debugging, and coverage analysis tools.

Starting with Python 2.2, the implementation of this facility was substantially revised, and an interface from C was added. This C interface allows the profiling or tracing code to avoid the overhead of calling through Python-level callable objects, making a direct C function call instead. The essential attributes of the facility have not changed; the interface allows trace functions to be installed per-thread, and the basic events reported to the trace function are the same as had been reported to the Python-level trace functions in previous versions.

```c
int (*Py_tracefunc)(PyObject *obj, PyFrameObject *frame, int what, PyObject *arg)
```

The type of the trace function registered using `PyEval_SetProfile()` and `PyEval_SetTrace()`. The first parameter is the object passed to the registration function as `obj`, `frame` is the frame object to which the event pertains, `what` is one of the constants `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION`, or `PyTrace_C_RETURN`, and `arg` depends on the value of `what`:

<table>
<thead>
<tr>
<th>Value of <code>what</code></th>
<th>Meaning of <code>arg</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>PyTrace_CALL</td>
<td>Always NULL</td>
</tr>
<tr>
<td>PyTrace_EXCEPTION</td>
<td>Exception information as returned by <code>sys.exc_info()</code></td>
</tr>
<tr>
<td>PyTrace_LINE</td>
<td>Always NULL</td>
</tr>
<tr>
<td>PyTrace_RETURN</td>
<td>Value being returned to the caller</td>
</tr>
<tr>
<td>PyTrace_C_CALL</td>
<td>Name of function being called</td>
</tr>
<tr>
<td>PyTrace_C_EXCEPTION</td>
<td>Always NULL</td>
</tr>
<tr>
<td>PyTrace_C_RETURN</td>
<td>Always NULL</td>
</tr>
</tbody>
</table>

```c
int PyTrace_CALL
```
The value of the *what* parameter to a `Py_tracefunc` function when a new call to a function or method is being reported, or a new entry into a generator. Note that the creation of the iterator for a generator function is not reported as there is no control transfer to the Python bytecode in the corresponding frame.

`int PyTrace_EXCEPTION`
The value of the *what* parameter to a `Py_tracefunc` function when an exception has been raised. The callback function is called with this value for *what* when after any bytecode is processed after which the exception becomes set within the frame being executed. The effect of this is that as exception propagation causes the Python stack to unwind, the callback is called upon return to each frame as the exception propagates. Only trace functions receives these events; they are not needed by the profiler.

`int PyTrace_LINE`
The value passed as the *what* parameter to a trace function (but not a profiling function) when a line-number event is being reported.

`int PyTrace_RETURN`
The value for the *what* parameter to `Py_tracefunc` functions when a call is returning without propagating an exception.

`int PyTrace_C_CALL`
The value for the *what* parameter to `Py_tracefunc` functions when a C function is about to be called.

`int PyTrace_C_EXCEPTION`
The value for the *what* parameter to `Py_tracefunc` functions when a C function has thrown an exception.

`int PyTrace_C_RETURN`
The value for the *what* parameter to `Py_tracefunc` functions when a C function has returned.

`void PyEval_SetProfile( Py_tracefunc func, PyObject *obj)`
Set the profiler function to *func*. The *obj* parameter is passed to the function as its first parameter, and may be any Python object, or NULL. If the profile function needs to maintain state, using a different value for *obj* for each
thread provides a convenient and thread-safe place to store it. The profile function is called for all monitored events except the line-number events.

void **PyEval_SetTrace**( Py_tracefunc *func, PyObject *obj)
Set the tracing function to *func*. This is similar to PyEval_SetProfile(), except the tracing function does receive line-number events.

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
8.3 Advanced Debugger Support

These functions are only intended to be used by advanced debugging tools.

PyInterpreterState* PyInterpreterState_Head()
    Return the interpreter state object at the head of the list of all such objects. New in version 2.2.

PyInterpreterState* PyInterpreterState_Next(PyInterpreterState *interp)
    Return the next interpreter state object after interp from the list of all such objects. New in version 2.2.

PyThreadState * PyInterpreterState_ThreadHead(PyInterpreterState *interp)
    Return the a pointer to the first PyThreadState object in the list of threads associated with the interpreter interp. New in version 2.2.

PyThreadState* PyThreadState_Next(PyThreadState *tstate)
    Return the next thread state object after tstate from the list of all such objects belonging to the same PyInterpreterState object. New in version 2.2.
9. Memory Management

Subsections

- 9.1 Overview
- 9.2 Memory Interface
- 9.3 Examples

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
9.1 Overview

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the *Python memory manager*. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if she regularly manipulates object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

To avoid memory corruption, extension writers should never try to operate on Python objects with the functions exported by the C library: `malloc()`, `calloc()`, `realloc()` and `free()`. This will result in mixed calls between the C allocator and the Python memory manager with fatal consequences, because they implement different algorithms and operate on different heaps. However, one may safely allocate and release memory blocks with the C library allocator for individual purposes, as shown in the following example:

```c
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */
if (buf == NULL)
```

```c
```
In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the string object returned as a result.

In most situations, however, it is recommended to allocate memory from the Python heap specifically because the latter is under control of the Python memory manager. For example, this is required when the interpreter is extended with new object types written in C. Another reason for using the Python heap is the desire to inform the Python memory manager about the memory needs of the extension module. Even when the requested memory is used exclusively for internal, highly-specific purposes, delegating all memory requests to the Python memory manager causes the interpreter to have a more accurate image of its memory footprint as a whole. Consequently, under certain circumstances, the Python memory manager may or may not trigger appropriate actions, like garbage collection, memory compaction or other preventive procedures. Note that by using the C library allocator as shown in the previous example, the allocated memory for the I/O buffer escapes completely the Python memory manager.
9.2 Memory Interface

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap:

```c
void* PyMemMalloc(size_t n)
Allocates n bytes and returns a pointer of type void* to the allocated memory, or NULL if the request fails. Requesting zero bytes returns a distinct non-NULL pointer if possible, as if PyMemMalloc(1) had been called instead. The memory will not have been initialized in any way.

void* PyMemRealloc(void* p, size_t n)
Resizes the memory block pointed to by p to n bytes. The contents will be unchanged to the minimum of the old and the new sizes. If p is NULL, the call is equivalent to PyMemMalloc(n); else if n is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-NULL. Unless p is NULL, it must have been returned by a previous call to PyMemMalloc() or PyMemRealloc().

void PyMemFree(void* p)
Frees the memory block pointed to by p, which must have been returned by a previous call to PyMemMalloc() or PyMemRealloc(). Otherwise, or if PyMemFree(p) has been called before, undefined behavior occurs. If p is NULL, no operation is performed.
```

The following type-oriented macros are provided for convenience. Note that TYPE refers to any C type.

```c
TYPE* PyMemNew(TYPE, size_t n)
Same as PyMemMalloc(), but allocates (n * sizeof(TYPE)) bytes of memory. Returns a pointer cast to TYPE*. The memory will not have been initialized in any way.

TYPE* PyMemResize(void* p, TYPE, size_t n)
```
Same as `PyMem_Realloc()`, but the memory block is resized to \((n \times \text{sizeof}(\text{TYPE}))\) bytes. Returns a pointer cast to `TYPE*`.

```c
void PyMem_Del(void *p)
```

Same as `PyMem_Free()`.

In addition, the following macro sets are provided for calling the Python memory allocator directly, without involving the C API functions listed above. However, note that their use does not preserve binary compatibility across Python versions and is therefore deprecated in extension modules.

`PyMem_MALLOC()`, `PyMem_REALLOC()`, `PyMem_FREE()`.

`PyMem_NEW()`, `PyMem_RESIZE()`, `PyMem_DEL()`.
9.3 Examples

Here is the example from section 9.1, rewritten so that the I/O buffer is allocated from the Python heap by using the first function set:

```c
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyString_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

The same code using the type-oriented function set:

```c
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyString_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

Note that in the two examples above, the buffer is always manipulated via functions belonging to the same set. Indeed, it is required to use the same memory API family for a given memory block, so that the risk of mixing different allocators is reduced to a minimum. The following code sequence contains two errors, one of which is labeled as fatal because it mixes two different allocators operating on different heaps.

```c
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3); /* Wrong -- should be PyMem_Free() */
free(buf2);      /* Right -- allocated via malloc() */
free(buf1);      /* Fatal -- should be PyMem_Del() */
```

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with
PyObject_New(), PyObject_NewVar() and PyObject_Del().

These will be explained in the next chapter on defining and implementing new object types in C.
10. Object Implementation Support

This chapter describes the functions, types, and macros used when defining new object types.

Subsections

- 10.1 Allocating Objects on the Heap
- 10.2 Common Object Structures
- 10.3 Type Objects
- 10.4 Mapping Object Structures
- 10.5 Number Object Structures
- 10.6 Sequence Object Structures
- 10.7 Buffer Object Structures
- 10.8 Supporting the Iterator Protocol
- 10.9 Supporting Cyclic Garbage Collection

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
10.1 Allocating Objects on the Heap

PyObject* _PyObject_New( PyTypeObject *type)

PyVarObject* _PyObject_NewVar( PyTypeObject *type, Py_ssize_t size)

void _PyObject_Del( PyObject *op)

PyObject* PyObject_Init( PyObject *op, PyTypeObject *type)
Return value: Borrowed reference.
Initialize a newly-allocated object op with its type and initial reference.
Returns the initialized object. If type indicates that the object participates in
the cyclic garbage detector, it is added to the detector's set of observed
objects. Other fields of the object are not affected.

PyVarObject* PyObject_InitVar( PyVarObject *op, PyTypeObject *type,
Py_ssize_t size)
This does everything PyObject_Init() does, and also initializes the
length information for a variable-size object.

TYPE* PyObject_New( TYPE, PyTypeObject *type)
Allocate a new Python object using the C structure type TYPE and the
Python type object type. Fields not defined by the Python object header are
not initialized; the object's reference count will be one. The size of the
memory allocation is determined from the tp_basicsize field of the
type object.

TYPE* PyObject_NewVar( TYPE, PyTypeObject *type, Py_ssize_t size)
Allocate a new Python object using the C structure type TYPE and the
Python type object type. Fields not defined by the Python object header are
not initialized. The allocated memory allows for the TYPE structure plus
size fields of the size given by the tp_itemsize field of type. This is useful for implementing objects like tuples, which are able to determine their size at construction time. Embedding the array of fields into the same allocation decreases the number of allocations, improving the memory management efficiency.

void **PyObject** _**Del***( PyObject *op)
Releases memory allocated to an object using **PyObject** _**New***( ) or **PyObject** _**NewVar***( ). This is normally called from the tp_dealloc handler specified in the object's type. The fields of the object should not be accessed after this call as the memory is no longer a valid Python object.

PyObject* **Py_InitModule** *( char *name, PyMethodDef *methods)
*Return value: Borrowed reference.*
Create a new module object based on a name and table of functions, returning the new module object.

Changed in version 2.3: Older versions of Python did not support NULL as the value for the methods argument.

PyObject* **Py_InitModule3** *( char *name, PyMethodDef *methods, char *doc)
*Return value: Borrowed reference.*
Create a new module object based on a name and table of functions, returning the new module object. If doc is non-NULL, it will be used to define the docstring for the module.

Changed in version 2.3: Older versions of Python did not support NULL as the value for the methods argument.

PyObject* **Py_InitModule4** *( char *name, PyMethodDef *methods, char *doc, PyObject *self, int apiver)
*Return value: Borrowed reference.*
Create a new module object based on a name and table of functions, returning the new module object. If doc is non-NULL, it will be used to define the docstring for the module. If self is non-NULL, it will passed to the functions of the module as their (otherwise NULL) first parameter. (This was added as an experimental feature, and there are no known uses in the
current version of Python.) For \textit{apiver}, the only value which should be passed is defined by the constant \texttt{PYTHON\_API\_VERSION}.

\textbf{Note:} Most uses of this function should probably be using the \texttt{Py\_InitModule3()} instead; only use this if you are sure you need it.

Changed in version 2.3: Older versions of Python did not support NULL as the value for the \textit{methods} argument.

\texttt{DL\_IMPORT}

\texttt{PyObject \_Py\_NoneStruct}

Object which is visible in Python as \texttt{None}. This should only be accessed using the \texttt{Py\_None} macro, which evaluates to a pointer to this object.
10.2 Common Object Structures

There are a large number of structures which are used in the definition of object types for Python. This section describes these structures and how they are used.

All Python objects ultimately share a small number of fields at the beginning of the object's representation in memory. These are represented by the PyObject and PyVarObject types, which are defined, in turn, by the expansions of some macros also used, whether directly or indirectly, in the definition of all other Python objects.

PyObject

All object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In a normal "release" build, it contains only the objects reference count and a pointer to the corresponding type object. It corresponds to the fields defined by the expansion of the PyObject_HEAD macro.

PyVarObject

This is an extension of PyObject that adds the ob_size field. This is only used for objects that have some notion of length. This type does not often appear in the Python/C API. It corresponds to the fields defined by the expansion of the PyObject_VAR_HEAD macro.

These macros are used in the definition of PyObject and PyVarObject:

PyObject_HEAD

This is a macro which expands to the declarations of the fields of the PyObject type; it is used when declaring new types which represent objects without a varying length. The specific fields it expands to depend on the definition of Py_TRACE_REFS. By default, that macro is not defined, and PyObject_HEAD expands to:

```c
Py_ssize_t ob_refcnt;
PyTypeObject *ob_type;
```

When Py_TRACE_REFS is defined, it expands to:

```c
PyObject *__ob_next, *__ob_prev;
```
Py_ssize_t ob_refcnt;
PyTypeObject *ob_type;

PyObject_VAR_HEAD

This is a macro which expands to the declarations of the fields of the PyVarObject type; it is used when declaring new types which represent objects with a length that varies from instance to instance. This macro always expands to:

PyObject_HEAD
Py_ssize_t ob_size;

Note that PyObject_HEAD is part of the expansion, and that its own expansion varies depending on the definition of Py_TRACE_REFS.

PyObject_HEAD_INIT

PyCFunction

Type of the functions used to implement most Python callables in C. Functions of this type take two PyObject* parameters and return one such value. If the return value is NULL, an exception shall have been set. If not NULL, the return value is interpreted as the return value of the function as exposed in Python. The function must return a new reference.

PyMethodDef

Structure used to describe a method of an extension type. This structure has four fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>C Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ml_name</td>
<td>char *</td>
<td>name of the method</td>
</tr>
<tr>
<td>ml_meth</td>
<td>PyCFunction</td>
<td>pointer to the C implementation</td>
</tr>
<tr>
<td>ml_flags</td>
<td>int</td>
<td>flag bits indicating how the call should be constructed</td>
</tr>
<tr>
<td>ml_doc</td>
<td>char *</td>
<td>points to the contents of the docstring</td>
</tr>
</tbody>
</table>

The ml_meth is a C function pointer. The functions may be of different types, but they always return PyObject*. If the function is not of the PyCFunction, the compiler will require a cast in the method table. Even though PyCFunction defines the first parameter as PyObject*, it is common that the method implementation uses a the specific C type of the self
The `ml_flags` field is a bitfield which can include the following flags. The individual flags indicate either a calling convention or a binding convention. Of the calling convention flags, only `METH_VARARGS` and `METH_KEYWORDS` can be combined (but note that `METH_KEYWORDS` alone is equivalent to `METH_VARARGS | METH_KEYWORDS`). Any of the calling convention flags can be combined with a binding flag.

**METH_VARARGS**
This is the typical calling convention, where the methods have the type `PyCFunction`. The function expects two `PyObject*` values. The first one is the `self` object for methods; for module functions, it has the value given to `Py_InitModule4()` (or NULL if `Py_InitModule()` was used). The second parameter (often called `args`) is a tuple object representing all arguments. This parameter is typically processed using `PyArg_ParseTuple()` or `PyArg_UnpackTuple()`.

**METH_KEYWORDS**
Methods with these flags must be of type `PyCFunctionWithKeywords`. The function expects three parameters: `self`, `args`, and a dictionary of all the keyword arguments. The flag is typically combined with `METH_VARARGS`, and the parameters are typically processed using `PyArg_ParseTupleAndKeywords()`.

**METH_NOARGS**
Methods without parameters don't need to check whether arguments are given if they are listed with the `METH_NOARGS` flag. They need to be of type `PyCFunction`. When used with object methods, the first parameter is typically named `self` and will hold a reference to the object instance. In all cases the second parameter will be NULL.

**METH_O**
Methods with a single object argument can be listed with the `METH_O` flag, instead of invoking `PyArg_ParseTuple()` with a "O" argument. They have the type `PyCFunction`, with the `self` parameter, and a `PyObject*` parameter representing the single argument.

**METH_OLDARGS**
This calling convention is deprecated. The method must be of type `PyCFunction`. The second argument is `NULL` if no arguments are given, a single object if exactly one argument is given, and a tuple of objects if more than one argument is given. There is no way for a function using this convention to distinguish between a call with multiple arguments and a call with a tuple as the only argument.

These two constants are not used to indicate the calling convention but the binding when use with methods of classes. These may not be used for functions defined for modules. At most one of these flags may be set for any given method.

**METH_CLASS**

The method will be passed the type object as the first parameter rather than an instance of the type. This is used to create class methods, similar to what is created when using the `classmethod()` built-in function. New in version 2.3.

**METH_STATIC**

The method will be passed `NULL` as the first parameter rather than an instance of the type. This is used to create static methods, similar to what is created when using the `staticmethod()` built-in function. New in version 2.3.

One other constant controls whether a method is loaded in place of another definition with the same method name.

**METH_COEXIST**

The method will be loaded in place of existing definitions. Without `METH_COEXIST`, the default is to skip repeated definitions. Since slot wrappers are loaded before the method table, the existence of a `sq_contains` slot, for example, would generate a wrapped method named `__contains__()` and preclude the loading of a corresponding `PyCFunction` with the same name. With the flag defined, the `PyCFunction` will be loaded in place of the wrapper object and will co-exist with the slot. This is helpful because calls to `PyCFunctions` are optimized more than wrapper object calls. New in version 2.4.

```c
PyMethodDef table[], PyObject *ob, char
```
PyObject* **Py_FindMethod**( *name*)

*Return value: New reference.*

Return a bound method object for an extension type implemented in C. This can be useful in the implementation of a `tp_getattro` or `tp_getattr` handler that does not use the `PyObject_GenericGetAttr()` function.
### 10.3 Type Objects

Perhaps one of the most important structures of the Python object system is the structure that defines a new type: the `PyTypeObject` structure. Type objects can be handled using any of the `PyObject_*()` or `PyType_*()` functions, but do not offer much that's interesting to most Python applications. These objects are fundamental to how objects behave, so they are very important to the interpreter itself and to any extension module that implements new types.

Type objects are fairly large compared to most of the standard types. The reason for the size is that each type object stores a large number of values, mostly C function pointers, each of which implements a small part of the type's functionality. The fields of the type object are examined in detail in this section. The fields will be described in the order in which they occur in the structure.

Typedefs: unaryfunc, binaryfunc, ternaryfunc, inquiry, coercion, intargfunc, intintargfunc, intobjargproc, intintobjargproc, objobjargproc, destructor, freefunc, printfunc, setattrfunc, getattrofunc, setattrofunc, cmpfunc, reprfunc, hashfunc

The structure definition for `PyTypeObject` can be found in `Include/object.h`. For convenience of reference, this repeats the definition found there:

```c
typedef struct _typeobject {
    PyObject_VAR_HEAD
    char *tp_name; /* For printing, in format "<module>.<name>" */
    int tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

destructor tp_dealloc;
printfunc tp_print;
getattrfunc tp_getattr;
setattrfunc tp_setattr;
cmpfunc tp_compare;
reprfunc tp_repr;

    /* Method suites for standard classes */

PyNumberMethods *tp_as_number;
PySequenceMethods *tp_as_sequence;
PyMappingMethods *tp_as_mapping;
```
/* More standard operations (here for binary compatibility) */

hashfunc tp_hash;
ternaryfunc tp_call;
reprfunc tp_str;
getattrofunc tp_getattro;
setattrofunc tp_setattro;

/* Functions to access object as input/output buffer */
PyBufferProcs *tp_as_buffer;

/* Flags to define presence of optional/expanded features */
long tp_flags;

char *tp_doc; /* Documentation string */

/* Assigned meaning in release 2.0 */
/* call function for all accessible objects */
traverseproc tp_traverse;

/* delete references to contained objects */
inquiry tp_clear;

/* Assigned meaning in release 2.1 */
/* rich comparisons */
richcmpfunc tp_richcompare;

/* weak reference enabler */
long tp_weaklistoffset;

/* Added in release 2.2 */
/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;

/* Attribute descriptor and subclassing stuff */
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
struct _typeobject *tp_base;
PyObject *tp_dict;
descrgetfunc tp_descr_get;
desrcsetfunc tp_descr_set;
long tp_dictoffset;
initproc tp_init;
allocfunc tp_alloc;
newfunc tp_new; /* Low-level free-memory routine */
inquiry tp_is_gc; /* For PyObject_IS_GC */
PyObject *tp_bases;
PyObject *tp_mro; /* method resolution order */
The type object structure extends the `PyVarObject` structure. The `ob_size` field is used for dynamic types (created by `type_new()`, usually called from a class statement). Note that `PyType_Type` (the metatype) initializes `tp_itemsize`, which means that its instances (i.e. type objects) must have the `ob_size` field.

**PyObject* _ob_next**

**PyObject* _ob_prev**

These fields are only present when the macro `Py_TRACE_REFS` is defined. Their initialization to NULL is taken care of by the `PyObject_HEAD_INIT` macro. For statically allocated objects, these fields always remain NULL. For dynamically allocated objects, these two fields are used to link the object into a doubly-linked list of all live objects on the heap. This could be used for various debugging purposes; currently the only use is to print the objects that are still alive at the end of a run when the environment variable `PYTHONDUMPREFS` is set.

These fields are not inherited by subtypes.

**Py_ssize_t ob_refcnt**

This is the type object's reference count, initialized to 1 by the `PyObject_HEAD_INIT` macro. Note that for statically allocated type objects, the type's instances (objects whose `ob_type` points back to the type) do not count as references. But for dynamically allocated type objects, the instances do count as references.

This field is not inherited by subtypes.

**PyTypeObject* ob_type**

This is the type's type, in other words its metatype. It is initialized by the argument to the `PyObject_HEAD_INIT` macro, and its value should normally be `&PyType_Type`. However, for dynamically loadable
extension modules that must be usable on Windows (at least), the compiler
complains that this is not a valid initializer. Therefore, the convention is to
pass NULL to the PyObject_HEAD_INIT macro and to initialize this
field explicitly at the start of the module's initialization function, before
doing anything else. This is typically done like this:

    Foo_Type.ob_type = &PyType_Type;

This should be done before any instances of the type are created.
PyType_Ready() checks if ob_type is NULL, and if so, initializes it:
in Python 2.2, it is set to &PyType_Type; in Python 2.2.1 and later it is
initialized to the ob_type field of the base class. PyType_Ready() will
not change this field if it is non-zero.

In Python 2.2, this field is not inherited by subtypes. In 2.2.1, and in 2.3
and beyond, it is inherited by subtypes.

Py_ssize_t ob_size
    For statically allocated type objects, this should be initialized to zero. For
dynamically allocated type objects, this field has a special internal meaning.

    This field is not inherited by subtypes.

char* tp_name
    Pointer to a NUL-terminated string containing the name of the type. For
types that are accessible as module globals, the string should be the full
module name, followed by a dot, followed by the type name; for built-in
types, it should be just the type name. If the module is a submodule of a
package, the full package name is part of the full module name. For
example, a type named T defined in module M in subpackage Q in package
P should have the tp_name initializer "P.Q.M.T".

    For dynamically allocated type objects, this should just be the type name,
    and the module name explicitly stored in the type dict as the value for key
    '__module__'.

    For statically allocated type objects, the tp_name field should contain a dot.
    Everything before the last dot is made accessible as the __module__
    attribute, and everything after the last dot is made accessible as the
__name__ attribute.

If no dot is present, the entire tp_name field is made accessible as the __name__ attribute, and the __module__ attribute is undefined (unless explicitly set in the dictionary, as explained above). This means your type will be impossible to pickle.

This field is not inherited by subtypes.

Py_ssize_t tp_basicsize
Py_ssize_t tp_itemsize

These fields allow calculating the size in bytes of instances of the type.

There are two kinds of types: types with fixed-length instances have a zero tp_itemsize field, types with variable-length instances have a non-zero tp_itemsize field. For a type with fixed-length instances, all instances have the same size, given in tp_basicsize.

For a type with variable-length instances, the instances must have an ob_size field, and the instance size is tp_basicsize plus N times tp_itemsize, where N is the "length" of the object. The value of N is typically stored in the instance's ob_size field. There are exceptions: for example, long ints use a negative ob_size to indicate a negative number, and N is abs(ob_size) there. Also, the presence of an ob_size field in the instance layout doesn't mean that the instance structure is variable-length (for example, the structure for the list type has fixed-length instances, yet those instances have a meaningful ob_size field).

The basic size includes the fields in the instance declared by the macro PyObject_HEAD or PyObject_VAR_HEAD (whichever is used to declare the instance struct) and this in turn includes the _ob_prev and _ob_next fields if they are present. This means that the only correct way to get an initializer for the tp_basicsize is to use the sizeof operator on the struct used to declare the instance layout. The basic size does not include the GC header size (this is new in Python 2.2; in 2.1 and 2.0, the GC header size was included in tp_basicsize).

These fields are inherited separately by subtypes. If the base type has a non-zero tp_itemsize, it is generally not safe to set tp_itemsize to a
different non-zero value in a subtype (though this depends on the implementation of the base type).

A note about alignment: if the variable items require a particular alignment, this should be taken care of by the value of \texttt{tp\_basicsize}. Example: suppose a type implements an array of \texttt{double}. \texttt{tp\_itemsize} is \texttt{sizeof(double)}. It is the programmer's responsibility that \texttt{tp\_basicsize} is a multiple of \texttt{sizeof(double)} (assuming this is the alignment requirement for \texttt{double}).

destructor \texttt{tp\_dealloc}

A pointer to the instance destructor function. This function must be defined unless the type guarantees that its instances will never be deallocated (as is the case for the singletons \texttt{None} and \texttt{Ellipsis}).

The destructor function is called by the \texttt{Py\_DECREF()} and \texttt{Py\_XDECREF()} macros when the new reference count is zero. At this point, the instance is still in existence, but there are no references to it. The destructor function should free all references which the instance owns, free all memory buffers owned by the instance (using the freeing function corresponding to the allocation function used to allocate the buffer), and finally (as its last action) call the type's \texttt{tp\_free} function. If the type is not subtypable (doesn't have the \texttt{Py\_TPFLAGS\_BASETYPE} flag bit set), it is permissible to call the object deallocator directly instead of via \texttt{tp\_free}. The object deallocator should be the one used to allocate the instance; this is normally \texttt{PyObject\_Del()} if the instance was allocated using \texttt{PyObject\_New()} or \texttt{PyObject\_VarNew()}, or \texttt{PyObject\_GC\_Del()} if the instance was allocated using \texttt{PyObject\_GC\_New()} or \texttt{PyObject\_GC\_VarNew()}.

This field is inherited by subtypes.

printfunc \texttt{tp\_print}

An optional pointer to the instance print function.

The print function is only called when the instance is printed to a \textit{real} file; when it is printed to a pseudo-file (like a \texttt{StringIO} instance), the instance's \texttt{tp\_repr} or \texttt{tp\_str} function is called to convert it to a string. These are also called when the type's \texttt{tp\_print} field is \texttt{NULL}. A type
should never implement `tp_print` in a way that produces different output than `tp_repr` or `tp_str` would.

The print function is called with the same signature as `PyObject_Print()`. The `self` argument is the instance to be printed. The `file` argument is the stdio file to which it is to be printed. The `flags` argument is composed of flag bits. The only flag bit currently defined is `Py_PRINT_RAW`. When the `Py_PRINT_RAW` flag bit is set, the instance should be printed the same way as `tp_str` would format it; when the `Py_PRINT_RAW` flag bit is clear, the instance should be printed the same was as `tp_repr` would format it. It should return `-1` and set an exception condition when an error occurred during the comparison.

It is possible that the `tp_print` field will be deprecated. In any case, it is recommended not to define `tp_print`, but instead to rely on `tp_repr` and `tp_str` for printing.

This field is inherited by subtypes.

**getattribfunc** `tp_getattr`

An optional pointer to the get-attribute-string function.

This field is deprecated. When it is defined, it should point to a function that acts the same as the `tp_getattro` function, but taking a C string instead of a Python string object to give the attribute name. The signature is the same as for `PyObject_GetAttrString()`.

This field is inherited by subtypes together with `tp_getattro`: a subtype inherits both `tp_getattr` and `tp_getattro` from its base type when the subtype's `tp_getattr` and `tp_getattro` are both NULL.

**setattribfunc** `tp_setattr`

An optional pointer to the set-attribute-string function.

This field is deprecated. When it is defined, it should point to a function that acts the same as the `tp_setattro` function, but taking a C string instead of a Python string object to give the attribute name. The signature is the same as for `PyObject_SetAttrString()`.
This field is inherited by subtypes together with \texttt{tp\_setattro}: a subtype inherits both \texttt{tp\_setattr} and \texttt{tp\_setattro} from its base type when the subtype's \texttt{tp\_setattr} and \texttt{tp\_setattro} are both \texttt{NULL}.

\textbf{cmpfunc} \texttt{tp\_compare}

An optional pointer to the three-way comparison function.

The signature is the same as for \texttt{PyObject\_Compare()}. The function should return \texttt{1} if \texttt{self} greater than \texttt{other}, \texttt{0} if \texttt{self} is equal to \texttt{other}, and \texttt{-1} if \texttt{self} less than \texttt{other}. It should return \texttt{-1} and set an exception condition when an error occurred during the comparison.

This field is inherited by subtypes together with \texttt{tp\_richcompare} and \texttt{tp\_hash}: a subtypes inherits all three of \texttt{tp\_compare}, \texttt{tp\_richcompare}, and \texttt{tp\_hash} when the subtype's \texttt{tp\_compare}, \texttt{tp\_richcompare}, and \texttt{tp\_hash} are all \texttt{NULL}.

\textbf{reprfunc} \texttt{tp\_repr}

An optional pointer to a function that implements the built-in function \texttt{repr()}. The signature is the same as for \texttt{PyObject\_Repr()}; it must return a string or a Unicode object. Ideally, this function should return a string that, when passed to \texttt{eval()}, given a suitable environment, returns an object with the same value. If this is not feasible, it should return a string starting with "<" and ending with ">", from which both the type and the value of the object can be deduced.

When this field is not set, a string of the form "<\texttt{%s object at %p}>" is returned, where \texttt{%s} is replaced by the type name, and \texttt{%p} by the object's memory address.

This field is inherited by subtypes.

\texttt{PyNumberMethods *}\texttt{tp\_as\_number};

\texttt{XXX}

\texttt{PySequenceMethods *}\texttt{tp\_as\_sequence};
PyMappingMethods *tp_as_mapping;

hashfunc tp_hash
An optional pointer to a function that implements the built-in function hash().

The signature is the same as for PyObject_Hash(); it must return a C long. The value -1 should not be returned as a normal return value; when an error occurs during the computation of the hash value, the function should set an exception and return -1.

When this field is not set, two possibilities exist: if the tp_compare and tp_richcompare fields are both NULL, a default hash value based on the object's address is returned; otherwise, a TypeError is raised.

This field is inherited by subtypes together with tp_richcompare and tp_compare: a subtypes inherits all three of tp_compare, tp_richcompare, and tp_hash, when the subtype's tp_compare, tp_richcompare and tp_hash are all NULL.

ternaryfunc tp_call
An optional pointer to a function that implements calling the object. This should be NULL if the object is not callable. The signature is the same as for PyObject_Call().

This field is inherited by subtypes.

reprfunc tp_str
An optional pointer to a function that implements the built-in operation str(). (Note that str is a type now, and str() calls the constructor for that type. This constructor calls PyObject_Str() to do the actual work, and PyObject_Str() will call this handler.)

The signature is the same as for PyObject_Str(); it must return a string or a Unicode object. This function should return a ``friendly'' string
representation of the object, as this is the representation that will be used by the print statement.

When this field is not set, \code{PyObject_Repr()} is called to return a string representation.

This field is inherited by subtypes.

\textbf{getattrofunc \code{tp_getattro}}

An optional pointer to the get-attribute function.

The signature is the same as for \code{PyObject_GetAttr()}. It is usually convenient to set this field to \code{PyObject_GenericGetAttr()}, which implements the normal way of looking for object attributes.

This field is inherited by subtypes together with \code{tp_getattr}: a subtype inherits both \code{tp_getattr} and \code{tp_getattro} from its base type when the subtype's \code{tp_getattr} and \code{tp_getattro} are both NULL.

\textbf{setattrofunc \code{tp_setattro}}

An optional pointer to the set-attribute function.

The signature is the same as for \code{PyObject_SetAttr()}. It is usually convenient to set this field to \code{PyObject_GenericSetAttr()}, which implements the normal way of setting object attributes.

This field is inherited by subtypes together with \code{tp_setattr}: a subtype inherits both \code{tp_setattr} and \code{tp_setattro} from its base type when the subtype's \code{tp_setattr} and \code{tp_setattro} are both NULL.

\textbf{PyBufferProcs* \code{tp_as_buffer}}

Pointer to an additional structure that contains fields relevant only to objects which implement the buffer interface. These fields are documented in "Buffer Object Structures" (section 10.7).

The \code{tp_as_buffer} field is not inherited, but the contained fields are inherited individually.

\textbf{long \code{tp_flags}}
This field is a bit mask of various flags. Some flags indicate variant semantics for certain situations; others are used to indicate that certain fields in the type object (or in the extension structures referenced via \texttt{tp\_as\_number}, \texttt{tp\_as\_sequence}, \texttt{tp\_as\_mapping}, and \texttt{tp\_as\_buffer}) that were historically not always present are valid; if such a flag bit is clear, the type fields it guards must not be accessed and must be considered to have a zero or NULL value instead.

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type’s value of the flag bit is copied into the subtype together with a pointer to the extension structure. The \texttt{Py\_TPFLAGS\_HAVE\_GC} flag bit is inherited together with the \texttt{tp\_traverse} and \texttt{tp\_clear} fields, i.e. if the \texttt{Py\_TPFLAGS\_HAVE\_GC} flag bit is clear in the subtype and the \texttt{tp\_traverse} and \texttt{tp\_clear} fields in the subtype exist (as indicated by the \texttt{Py\_TPFLAGS\_HAVE\_RICHCOMPARE} flag bit) and have NULL values.

The following bit masks are currently defined; these can be or-ed together using the \texttt{|} operator to form the value of the \texttt{tp\_flags} field. The macro \texttt{PyType\_HasFeature()} takes a type and a flags value, \texttt{tp} and \texttt{f}, and checks whether \texttt{tp\->tp\_flags \& f} is non-zero.

\textbf{Py\_TPFLAGS\_HAVE\_GETCHARBUFFER}

If this bit is set, the \texttt{PyBufferProcs} struct referenced by \texttt{tp\_as\_buffer} has the \texttt{bf\_getcharbuffer} field.

\textbf{Py\_TPFLAGS\_HAVE\_SEQUENCE\_IN}

If this bit is set, the \texttt{PySequenceMethods} struct referenced by \texttt{tp\_as\_sequence} has the \texttt{sq\_contains} field.

\textbf{Py\_TPFLAGS\_GC}

This bit is obsolete. The bit it used to name is no longer in use. The symbol is now defined as zero.

\textbf{Py\_TPFLAGS\_HAVE\_INPLACEOPS}

If this bit is set, the \texttt{PySequenceMethods} struct referenced by
tp_as_sequence and the PyNumberMethods structure referenced by tp_as_number contain the fields for in-place operators. In particular, this means that the PyNumberMethods structure has the fields nb_inplace_add, nb_inplace_subtract, nb_inplace_multiply, nb_inplace_divide, nb_inplace_remainder, nb_inplace_power, nb_inplace_lshift, nb_inplace_rshift, nb_inplace_and, nb_inplace_xor, and nb_inplace_or; and the PySequenceMethods struct has the fields sq_inplace_concat and sq_inplace_repeat.

**Py_TPFLAGS_CHECKTYPES**
If this bit is set, the binary and ternary operations in the PyNumberMethods structure referenced by tp_as_number accept arguments of arbitrary object types, and do their own type conversions if needed. If this bit is clear, those operations require that all arguments have the current type as their type, and the caller is supposed to perform a coercion operation first. This applies to nb_add, nb_subtract, nb_multiply, nb_divide, nb_remainder, nb_divmod, nb_power, nb_lshift, nb_rshift, nb_and, nb_xor, and nb_or.

**Py_TPFLAGS_HAVE_RICHCOMPARE**
If this bit is set, the type object has the tp_richcompare field, as well as the tp_traverse and the tp_clear fields.

**Py_TPFLAGS_HAVE_WEAKREFS**
If this bit is set, the tp_weaklistoffset field is defined. Instances of a type are weakly referenceable if the type's tp_weaklistoffset field has a value greater than zero.

**Py_TPFLAGS_HAVE_ITER**
If this bit is set, the type object has the tp_iter and tp_iternext fields.

**Py_TPFLAGS_HAVE_CLASS**
If this bit is set, the type object has several new fields defined starting in Python 2.2: tp_methods, tp_members, tp_getset,
tp_base, tp_dict, tp_descr_get, tp_descr_set, 
tp_dictoffset, tp_init, tp_alloc, tp_new, tp_free, 
.tp_is_gc, tp_bases, tp_mro, tp_cache, tp_subclasses, 
and tp_weaklist.

**Py_TPFLAGS_HEAPTYPE**
This bit is set when the type object itself is allocated on the heap. In
this case, the ob_type field of its instances is considered a reference
to the type, and the type object is INCREF’ed when a new instance is
created, andDECREF’ed when an instance is destroyed (this does not
apply to instances of subtypes; only the type referenced by the
instance's ob_type gets INCREF’ed orDECREF’ed).

**Py_TPFLAGS_BASETYPE**
This bit is set when the type can be used as the base type of another
type. If this bit is clear, the type cannot be subtyped (similar to a
"final" class in Java).

**Py_TPFLAGS_READY**
This bit is set when the type object has been fully initialized by
PyType_Ready().

**Py_TPFLAGS_READYING**
This bit is set while PyType_Ready() is in the process of
initializing the type object.

**Py_TPFLAGS_HAVE_GC**
This bit is set when the object supports garbage collection. If this bit is
set, instances must be created using PyObject_GC_New() and
destroyed using PyObject_GC_Del(). More information in section
XXX about garbage collection. This bit also implies that the GC-
related fields tp_traverse and tp_clear are present in the type
object; but those fields also exist when Py_TPFLAGS_HAVE_GC is
clear but Py_TPFLAGS_HAVE_RICHCOMPARE is set.

**Py_TPFLAGS_DEFAULT**
This is a bitmask of all the bits that pertain to the existence of certain
fields in the type object and its extension structures. Currently, it
includes the following bits:
```c
static int
local_traverse(localobject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

Note that `Py_VISIT()` is called only on those members that can participate in reference cycles. Although there is also a "self->key" member, it can only be `NULL` or a Python string and therefore cannot be part of a reference cycle.
On the other hand, even if you know a member can never be part of a cycle, as a debugging aid you may want to visit it anyway just so the gc module's get_referents() function will include it.

Note that Py_VISIT() requires the visit and arg parameters to local_traverse to have these specific names; don't name them just anything.

This field is inherited by subtypes together with tp_clear and the Py_TPFLAGS_HAVE_GC flag bit: the flag bit, tp_traverse, and tp_clear are all inherited from the base type if they are all zero in the subtype and the subtype has the Py_TPFLAGS_HAVE_RICHCOMPARE flag bit set.

inquiry **tp_clear**
An optional pointer to a clear function for the garbage collector. This is only used if the Py_TPFLAGS_HAVE_GC flag bit is set.

The tp_clear member function is used to break reference cycles in cyclic garbage detected by the garbage collector. Taken together, all tp_clear functions in the system must combine to break all reference cycles. This is subtle, and if in any doubt supply a tp_clear function. For example, the tuple type does not implement a tp_clear function, because it's possible to prove that no reference cycle can be composed entirely of tuples. Therefore the tp_clear functions of other types must be sufficient to break any cycle containing a tuple. This isn't immediately obvious, and there's rarely a good reason to avoid implementing tp_clear.

Implementations of tp_clear should drop the instance's references to those of its members that may be Python objects, and set its pointers to those members to NULL, as in the following example:

```c
static int
local_clear(localobject *self)
{
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```
The Py_CLEAR() macro should be used, because clearing references is delicate: the reference to the contained object must not be decremented until after the pointer to the contained object is set to NULL. This is because decrementing the reference count may cause the contained object to become trash, triggering a chain of reclamation activity that may include invoking arbitrary Python code (due to finalizers, or weakref callbacks, associated with the contained object). If it's possible for such code to reference self again, it's important that the pointer to the contained object be NULL at that time, so that self knows the contained object can no longer be used. The Py_CLEAR() macro performs the operations in a safe order.

Because the goal of tp_clear functions is to break reference cycles, it's not necessary to clear contained objects like Python strings or Python integers, which can't participate in reference cycles. On the other hand, it may be convenient to clear all contained Python objects, and write the type's tp_dealloc function to invoke tp_clear.

More information about Python's garbage collection scheme can be found in section 10.9.

This field is inherited by subtypes together with tp_traverse and the Py_TPFLAGS_HAVE_GC flag bit: the flag bit, tp_traverse, and tp_clear are all inherited from the base type if they are all zero in the subtype and the subtype has the Py_TPFLAGS_HAVE_RICHCOMPARE flag bit set.

richcmpfunc tp_richcompare
An optional pointer to the rich comparison function.

The signature is the same as for PyObject_RichCompare(). The function should return the result of the comparison (usually Py_True or Py_False). If the comparison is undefined, it must return Py_NotImplemented, if another error occurred it must return NULL and set an exception condition.

This field is inherited by subtypes together with tp_compare and tp_hash: a subtype inherits all three of tp_compare,
tp_richcompare, and tp_hash, when the subtype's tp_compare, tp_richcompare, and tp_hash are all NULL.

The following constants are defined to be used as the third argument for tp_richcompare and for PyObject_RichCompare():

<table>
<thead>
<tr>
<th>Constant</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Py_LT</td>
<td>&lt;</td>
</tr>
<tr>
<td>Py_LE</td>
<td>&lt;=</td>
</tr>
<tr>
<td>Py_EQ</td>
<td>==</td>
</tr>
<tr>
<td>Py_NE</td>
<td>!=</td>
</tr>
<tr>
<td>Py_GT</td>
<td>&gt;</td>
</tr>
<tr>
<td>Py_GE</td>
<td>&gt;=</td>
</tr>
</tbody>
</table>

The next field only exists if the Py_TPFLAGS_HAVE_WEAKREFS flag bit is set.

long **tp_weaklistoffset**

If the instances of this type are weakly referenceable, this field is greater than zero and contains the offset in the instance structure of the weak reference list head (ignoring the GC header, if present); this offset is used by PyObject_ClearWeakRefs() and the PyWeakref_*() functions. The instance structure needs to include a field of type PyObject* which is initialized to NULL.

Do not confuse this field with tp_weaklist; that is the list head for weak references to the type object itself.

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype uses a different weak reference list head than the base type. Since the list head is always found via tp_weaklistoffset, this should not be a problem.

When a type defined by a class statement has no **slots** declaration, and none of its base types are weakly referenceable, the type is made weakly referenceable by adding a weak reference list head slot to the instance layout and setting the tp_weaklistoffset of that slot's offset.
When a type's __slots__ declaration contains a slot named __weakref__, that slot becomes the weak reference list head for instances of the type, and the slot's offset is stored in the type's tp_weaklistoffset.

When a type's __slots__ declaration does not contain a slot named __weakref__, the type inherits its tp_weaklistoffset from its base type.

The next two fields only exist if the Py_TPFLAGS_HAVE_CLASS flag bit is set.

getiterfunc tp_iter
An optional pointer to a function that returns an iterator for the object. Its presence normally signals that the instances of this type are iterable (although sequences may be iterable without this function, and classic instances always have this function, even if they don't define an __iter__() method).

This function has the same signature as PyObject_GetIter().

This field is inherited by subtypes.

iternextfunc tp_iternext
An optional pointer to a function that returns the next item in an iterator, or raises StopIteration when the iterator is exhausted. Its presence normally signals that the instances of this type are iterators (although classic instances always have this function, even if they don't define a next() method).

Iterator types should also define the tp_iter function, and that function should return the iterator instance itself (not a new iterator instance).

This function has the same signature as PyIter_Next().

This field is inherited by subtypes.

The next fields, up to and including tp_weaklist, only exist if the Py_TPFLAGS_HAVE_CLASS flag bit is set.
struct PyMethodDef* **tp_methods**
An optional pointer to a static NULL-terminated array of PyMethodDef structures, declaring regular methods of this type.

For each entry in the array, an entry is added to the type's dictionary (see **tp_dict** below) containing a method descriptor.

This field is not inherited by subtypes (methods are inherited through a different mechanism).

struct PyMemberDef* **tp_members**
An optional pointer to a static NULL-terminated array of PyMemberDef structures, declaring regular data members (fields or slots) of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see **tp_dict** below) containing a member descriptor.

This field is not inherited by subtypes (members are inherited through a different mechanism).

struct PyGetSetDef* **tp_getset**
An optional pointer to a static NULL-terminated array of PyGetSetDef structures, declaring computed attributes of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see **tp_dict** below) containing a getset descriptor.

This field is not inherited by subtypes (computed attributes are inherited through a different mechanism).

Docs for PyGetSetDef (XXX belong elsewhere):

```c
typedef PyObject *(*getter)(PyObject *, void *);
typedef int (*setter)(PyObject *, PyObject *, void *);

typedef struct PyGetSetDef {
    char *name;     /* attribute name */
    getter get;     /* C function to get the attribute */
    setter set;     /* C function to set the attribute */
    char *doc;      /* optional doc string */
    void *closure;  /* optional additional data for getter and setter */
} PyGetSetDef;
```
An optional pointer to a base type from which type properties are inherited. At this level, only single inheritance is supported; multiple inheritance require dynamically creating a type object by calling the metatype.

This field is not inherited by subtypes (obviously), but it defaults to &PyBaseObject_Type (which to Python programmers is known as the type object).

The type's dictionary is stored here by PyType_Ready().

This field should normally be initialized to NULL before PyType_Ready is called; it may also be initialized to a dictionary containing initial attributes for the type. Once PyType_Ready() has initialized the type, extra attributes for the type may be added to this dictionary only if they don't correspond to overloaded operations (like __add__( )).

This field is not inherited by subtypes (though the attributes defined in here are inherited through a different mechanism).

descrgetfunc tp_descr_get
An optional pointer to a "descriptor get" function.

The function signature is

    PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject

    XXX blah, blah.

This field is inherited by subtypes.

descrsetfunc tp_descr_set
An optional pointer to a "descriptor set" function.

The function signature is

    int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
This field is inherited by subtypes.

XXX blah, blah.

**long tp_dictoffset**

If the instances of this type have a dictionary containing instance variables, this field is non-zero and contains the offset in the instances of the type of the instance variable dictionary; this offset is used by PyObject_GenericGetAttr().

Do not confuse this field with tp_dict; that is the dictionary for attributes of the type object itself.

If the value of this field is greater than zero, it specifies the offset from the start of the instance structure. If the value is less than zero, it specifies the offset from the end of the instance structure. A negative offset is more expensive to use, and should only be used when the instance structure contains a variable-length part. This is used for example to add an instance variable dictionary to subtypes of str or tuple. Note that the tp_basicsize field should account for the dictionary added to the end in that case, even though the dictionary is not included in the basic object layout. On a system with a pointer size of 4 bytes, tp_dictoffset should be set to -4 to indicate that the dictionary is at the very end of the structure.

The real dictionary offset in an instance can be computed from a negative tp_dictoffset as follows:

\[
\text{dictoffset} = \text{tp_basicsize} + \text{abs}(\text{ob_size}) \times \text{tp_itemsize} + \text{tp_dictoffset}
\]

if dictoffset is not aligned on sizeof(void*):
    round up to sizeof(void*)

where tp_basicsize, tp_itemsize and tp_dictoffset are taken from the type object, and ob_size is taken from the instance. The absolute value is taken because long ints use the sign of ob_size to store the sign of the number. (There's never a need to do this calculation yourself; it is done for you by _PyObject_GetDictPtr().)

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype instances store the
dictionary at a difference offset than the base type. Since the dictionary is always found via `tp_dictoffset`, this should not be a problem.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types has an instance variable dictionary, a dictionary slot is added to the instance layout and the `tp_dictoffset` is set to that slot's offset.

When a type defined by a class statement has a `__slots__` declaration, the type inherits its `tp_dictoffset` from its base type.

(Adding a slot named `__dict__` to the `__slots__` declaration does not have the expected effect, it just causes confusion. Maybe this should be added as a feature just like `__weakref__` though.)

**initproc tp_init**  
An optional pointer to an instance initialization function.

This function corresponds to the `__init__() method of classes. Like `__init__()`, it is possible to create an instance without calling `__init__()`, and it is possible to reinitialize an instance by calling its `__init__()` method again.

The function signature is

```
    int tp_init(PyObject *self, PyObject *args, PyObject *kwds)
```

The self argument is the instance to be initialized; the `args` and `kwds` arguments represent positional and keyword arguments of the call to `__init__()`. The `tp_init` function, if not NULL, is called when an instance is created normally by calling its type, after the type's `tp_new` function has returned an instance of the type. If the `tp_new` function returns an instance of some other type that is not a subtype of the original type, no `tp_init` function is called; if `tp_new` returns an instance of a subtype of the original type, the subtype's `tp_init` is called. (VERSION NOTE: described here is what is implemented in Python 2.2.1 and later. In Python 2.2, the `tp_init` of the type of the object returned by `tp_new` was always called, if not
This field is inherited by subtypes.

.allocfunc **tp_alloc**
An optional pointer to an instance allocation function.

The function signature is

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems)
```

The purpose of this function is to separate memory allocation from memory initialization. It should return a pointer to a block of memory of adequate length for the instance, suitably aligned, and initialized to zeros, but with **ob_refcnt** set to 1 and **ob_type** set to the type argument. If the type's **tp_itemsize** is non-zero, the object's **ob_size** field should be initialized to **nitems** and the length of the allocated memory block should be **tp_basicsize** + **nitems*** **tp_itemsize**, rounded up to a multiple of **sizeof(void*)**; otherwise, **nitems** is not used and the length of the block should be **tp_basicsize**.

Do not use this function to do any other instance initialization, not even to allocate additional memory; that should be done by **tp_new**.

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement); in the latter, this field is always set to **PyType_GenericAlloc()**, to force a standard heap allocation strategy. That is also the recommended value for statically defined types.

.newfunc **tp_new**
An optional pointer to an instance creation function.

If this function is **NULL** for a particular type, that type cannot be called to create new instances; presumably there is some other way to create instances, like a factory function.

The function signature is

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds)
```
The subtype argument is the type of the object being created; the args and kwds arguments represent positional and keyword arguments of the call to the type. Note that subtype doesn't have to equal the type whose tp_new function is called; it may be a subtype of that type (but not an unrelated type).

The tp_new function should call subtype->tp_alloc(subtype, nitems) to allocate space for the object, and then do only as much further initialization as is absolutely necessary. Initialization that can safely be ignored or repeated should be placed in the tp_init handler. A good rule of thumb is that for immutable types, all initialization should take place in tp_new, while for mutable types, most initialization should be deferred to tp_init.

This field is inherited by subtypes, except it is not inherited by static types whose tp_base is NULL or &PyBaseObject_Type. The latter exception is a precaution so that old extension types don't become callable simply by being linked with Python 2.2.

destructor tp_free
An optional pointer to an instance deallocation function.

The signature of this function has changed slightly: in Python 2.2 and 2.2.1, its signature is destructor:

    void tp_free(PyObject *)

In Python 2.3 and beyond, its signature is freecfunc:

    void tp_free(void *)

The only initializer that is compatible with both versions is _PyObject_Del, whose definition has suitably adapted in Python 2.3.

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement); in the latter, this field is set to a deallocator suitable to match PyType_GenericAlloc() and the value of the Py_TPFLAGS_HAVE_GC flag bit.

inquiry tp_is_gc
An optional pointer to a function called by the garbage collector.

The garbage collector needs to know whether a particular object is collectible or not. Normally, it is sufficient to look at the object's type's `tp_flags` field, and check the `Py_TPFLAGS_HAVE_GC` flag bit. But some types have a mixture of statically and dynamically allocated instances, and the statically allocated instances are not collectible. Such types should define this function; it should return 1 for a collectible instance, and 0 for a non-collectible instance. The signature is

```
int tp_is_gc(PyObject *self)
```

(The only example of this are types themselves. The metatype, `PyType_Type`, defines this function to distinguish between statically and dynamically allocated types.)

This field is inherited by subtypes. (VERSION NOTE: in Python 2.2, it was not inherited. It is inherited in 2.2.1 and later versions.)

**PyObject**

### `tp_flags`

Tuple of base types.

This is set for types created by a class statement. It should be NULL for statically defined types.

This field is not inherited.

**PyObject**

### `tp_mro`

Tuple containing the expanded set of base types, starting with the type itself and ending with `object`, in Method Resolution Order.

This field is not inherited; it is calculated fresh by `PyType_Ready()`.

**PyObject**

### `tp_cache`

Unused. Not inherited. Internal use only.

**PyObject**

### `tp_subclasses`

List of weak references to subclasses. Not inherited. Internal use only.

**PyObject**

### `tp_weaklist`
Weak reference list head, for weak references to this type object. Not inherited. Internal use only.

The remaining fields are only defined if the feature test macro `COUNT_ALLOCS` is defined, and are for internal use only. They are documented here for completeness. None of these fields are inherited by subtypes.

`Py_ssize_t tp_allocs`
Number of allocations.

`Py_ssize_t tp_frees`
Number of frees.

`Py_ssize_t tp_maxalloc`
Maximum simultaneously allocated objects.

`PyTypeObject* tp_next`
Pointer to the next type object with a non-zero `tp_allocs` field.

Also, note that, in a garbage collected Python, `tp_dealloc` may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a garbage collection on any thread). This is not a problem for Python API calls, since the thread on which `tp_dealloc` is called will own the Global Interpreter Lock (GIL). However, if the object being destroyed in turn destroys objects from some other C or C++ library, care should be taken to ensure that destroying those objects on the thread which called `tp_dealloc` will not violate any assumptions of the library.
10.4 Mapping Object Structures

PyMappingMethods
Structure used to hold pointers to the functions used to implement the mapping protocol for an extension type.
10.5 Number Object Structures

PyNumberMethods
Structure used to hold pointers to the functions an extension type uses to implement the number protocol.
10.6 Sequence Object Structures

**PySequenceMethods**

Structure used to hold pointers to the functions which an object uses to implement the sequence protocol.
10.7 Buffer Object Structures

The buffer interface exports a model where an object can expose its internal data as a set of chunks of data, where each chunk is specified as a pointer/length pair. These chunks are called segments and are presumed to be non-contiguous in memory.

If an object does not export the buffer interface, then its `tp_as_buffer` member in the `PyTypeObject` structure should be NULL. Otherwise, the `tp_as_buffer` will point to a `PyBufferProcs` structure.

**Note:** It is very important that your `PyTypeObject` structure uses `Py_TPFLAGS_DEFAULT` for the value of the `tp_flags` member rather than 0. This tells the Python runtime that your `PyBufferProcs` structure contains the `bf_getcharbuffer` slot. Older versions of Python did not have this member, so a new Python interpreter using an old extension needs to be able to test for its presence before using it.

**PyBufferProcs**

Structure used to hold the function pointers which define an implementation of the buffer protocol.

The first slot is `bf_getreadbuffer`, of type `getreadbufferproc`. If this slot is NULL, then the object does not support reading from the internal data. This is non-sensical, so implementors should fill this in, but callers should test that the slot contains a non-NULL value.

The next slot is `bf_getwritebuffer` having type `getwritebufferproc`. This slot may be NULL if the object does not allow writing into its returned buffers.

The third slot is `bf_getsegcount`, with type `getsegcountproc`. This slot must not be NULL and is used to inform the caller how many segments the object contains. Simple objects such as `PyString_Type` and `PyBuffer_Type` objects contain a single segment.

The last slot is `bf_getcharbuffer`, of type `getcharbufferproc`. 
This slot will only be present if the
Py_TPFLAGS_HAVE_GETCHARBUFFER flag is present in the
tp_flags field of the object's PyTypeObject. Before using this slot,
the caller should test whether it is present by using the
PyType_HasFeature() function. If the flag is present,
bf_getcharbuffer may be NULL, indicating that the object's contents
cannot be used as 8-bit characters. The slot function may also raise an error
if the object's contents cannot be interpreted as 8-bit characters. For
example, if the object is an array which is configured to hold floating point
values, an exception may be raised if a caller attempts to use
bf_getcharbuffer to fetch a sequence of 8-bit characters. This notion
of exporting the internal buffers as "text" is used to distinguish between
objects that are binary in nature, and those which have character-based
content.

Note: The current policy seems to state that these characters may be multi-
byte characters. This implies that a buffer size of $N$ does not mean there are
$N$ characters present.

**Py_TPFLAGS_HAVE_GETCHARBUFFER**
Flag bit set in the type structure to indicate that the bf_getcharbuffer
slot is known. This being set does not indicate that the object supports the
buffer interface or that the bf_getcharbuffer slot is non-NULL.

**Py_ssize_t (*readbufferproc) (PyObject *self,**
Py_ssize_t segment, void **ptrptr)**
Return a pointer to a readable segment of the buffer in *ptrptr. This
function is allowed to raise an exception, in which case it must return -1.
The segment which is specified must be zero or positive, and strictly less
than the number of segments returned by the bf_getsegcount slot
function. On success, it returns the length of the segment, and sets *ptrptr
to a pointer to that memory.

**Py_ssize_t (*writebufferproc) (PyObject *self,**
Py_ssize_t segment, void **ptrptr)**
Return a pointer to a writable memory buffer in *ptrptr, and the length of
that segment as the function return value. The memory buffer must
correspond to buffer segment segment. Must return -1 and set an exception
on error. TypeError should be raised if the object only supports read-
only buffers, and SystemError should be raised when `segment` specifies a segment that doesn't exist.

```python
Py_ssize_t (*segcountproc) (PyObject *self, Py_ssize_t *lenp)
    Return the number of memory segments which comprise the buffer. If `lenp` is not NULL, the implementation must report the sum of the sizes (in bytes) of all segments in `lenp`. The function cannot fail.
```

```python
Py_ssize_t (*charbufferproc) (PyObject *self, Py_ssize_t segment, const char **ptrptr)
    Return the size of the segment `segment` that `ptrptr` is set to. `*ptrptr` is set to the memory buffer. Returns -1 on error.
```
10.8 Supporting the Iterator Protocol
10.9 Supporting Cyclic Garbage Collection

Python's support for detecting and collecting garbage which involves circular references requires support from object types which are "containers" for other objects which may also be containers. Types which do not store references to other objects, or which only store references to atomic types (such as numbers or strings), do not need to provide any explicit support for garbage collection.

An example showing the use of these interfaces can be found in "Supporting the Cycle Collector" in *Extending and Embedding the Python Interpreter*.

To create a container type, the *tp_flags* field of the type object must include the *Py_TPFLAGS_HAVE_GC* and provide an implementation of the *tp_traverse* handler. If instances of the type are mutable, a *tp_clear* implementation must also be provided.

**Py_TPFLAGS_HAVE_GC**

Objects with a type with this flag set must conform with the rules documented here. For convenience these objects will be referred to as container objects.

Constructors for container types must conform to two rules:

1. The memory for the object must be allocated using
   `PyObject_GC_New()` or `PyObject_GC_VarNew()`.

2. Once all the fields which may contain references to other containers are initialized, it must call `PyObject_GC_Track()`.

```
TYPE* PyObject_GC_New( TYPE, PyTypeObject *type)
   Analogous to `PyObject_New()` but for container objects with the
   Py_TPFLAGS_HAVE_GC flag set.

TYPE* PyObject_GC_NewVar( TYPE, PyTypeObject *type, Py_ssize_t size)
```
Analogous to \texttt{PyObject\_NewVar()} but for container objects with the 
\texttt{Py\_TPFLAGS\_HAVE\_GC} flag set.

\begin{verbatim}
PyVarObject * \textbf{PyObject\_GC\_Resize}( PyVarObject *op, Py\_ssize_t)
   Resize an object allocated by \texttt{PyObject\_NewVar()}. Returns the resized
   object or NULL on failure.
\end{verbatim}

\begin{verbatim}
void \textbf{PyObject\_GC\_Track}( PyObject *op)
   Adds the object \texttt{op} to the set of container objects tracked by the collector.
   The collector can run at unexpected times so objects must be valid while
   being tracked. This should be called once all the fields followed by the
   \texttt{tp\_traverse} handler become valid, usually near the end of the
   constructor.
\end{verbatim}

\begin{verbatim}
void \_\textbf{PyObject\_GC\_TRACK}( PyObject *op)
   A macro version of \texttt{PyObject\_GC\_Track()}. It should not be used for
   extension modules.
\end{verbatim}

Similarly, the deallocator for the object must conform to a similar pair of rules:

1. Before fields which refer to other containers are invalidated, 
   \texttt{PyObject\_GC\_UnTrack()} must be called.

2. The object's memory must be deallocated using \texttt{PyObject\_GC\_Del()}. 

\begin{verbatim}
void \textbf{PyObject\_GC\_Del}( void *op)
   Releases memory allocated to an object using \texttt{PyObject\_GC\_New()} or 
   \texttt{PyObject\_GC\_NewVar()}. 
\end{verbatim}

\begin{verbatim}
void \textbf{PyObject\_GC\_UnTrack}( void *op)
   Remove the object \texttt{op} from the set of container objects tracked by the
   collector. Note that \texttt{PyObject\_GC\_Track()} can be called again on this 
   object to add it back to the set of tracked objects. The deallocator 
   (\texttt{tp\_dealloc} handler) should call this for the object before any of the
   fields used by the \texttt{tp\_traverse} handler become invalid.
\end{verbatim}

\begin{verbatim}
void \_\textbf{PyObject\_GC\_UNTRACK}( PyObject *op)
\end{verbatim}
A macro version of `PyObject_GC_UnTrack()`. It should not be used for extension modules.

The `tp_traverse` handler accepts a function parameter of this type:

```c
int (*visitproc)(PyObject *object, void *arg)
```

Type of the visitor function passed to the `tp_traverse` handler. The function should be called with an object to traverse as `object` and the third parameter to the `tp_traverse` handler as `arg`. The Python core uses several visitor functions to implement cyclic garbage detection; it's not expected that users will need to write their own visitor functions.

The `tp_traverse` handler must have the following type:

```c
int (*traverseproc)(PyObject *self, visitproc visit, void *arg)
```

Traversal function for a container object. Implementations must call the `visit` function for each object directly contained by `self`, with the parameters to `visit` being the contained object and the `arg` value passed to the handler. The `visit` function must not be called with a NULL object argument. If `visit` returns a non-zero value that value should be returned immediately.

To simplify writing `tp_traverse` handlers, a `Py_VISIT()` macro is provided. In order to use this macro, the `tp_traverse` implementation must name its arguments exactly `visit` and `arg`:

```c
void Py_VISIT( PyObject *o)
```

Call the `visit` callback, with arguments `o` and `arg`. If `visit` returns a non-zero value, then return it. Using this macro, `tp_traverse` handlers look like:

```c
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

New in version 2.4.

The `tp_clear` handler must be of the `inquiry` type, or NULL if the object is
immutable.

int (*inquiry)(PyObject *self)

Drop references that may have created reference cycles. Immutable objects do not have to define this method since they can never directly create reference cycles. Note that the object must still be valid after calling this method (don't just call Py_INCREF( ) on a reference). The collector will call this method if it detects that this object is involved in a reference cycle.
A. Reporting Bugs

Python is a mature programming language which has established a reputation for stability. In order to maintain this reputation, the developers would like to know of any deficiencies you find in Python or its documentation.

Before submitting a report, you will be required to log into SourceForge; this will make it possible for the developers to contact you for additional information if needed. It is not possible to submit a bug report anonymously.

All bug reports should be submitted via the Python Bug Tracker on SourceForge (http://sourceforge.net/bugs/?group_id=5470). The bug tracker offers a Web form which allows pertinent information to be entered and submitted to the developers.

The first step in filing a report is to determine whether the problem has already been reported. The advantage in doing so, aside from saving the developers time, is that you learn what has been done to fix it; it may be that the problem has already been fixed for the next release, or additional information is needed (in which case you are welcome to provide it if you can!). To do this, search the bug database using the search box on the left side of the page.

If the problem you're reporting is not already in the bug tracker, go back to the Python Bug Tracker (http://sourceforge.net/bugs/?group_id=5470). Select the "Submit a Bug" link at the top of the page to open the bug reporting form.

The submission form has a number of fields. The only fields that are required are the "Summary" and "Details" fields. For the summary, enter a very short description of the problem; less than ten words is good. In the Details field, describe the problem in detail, including what you expected to happen and what did happen. Be sure to include the version of Python you used, whether any extension modules were involved, and what hardware and software platform you were using (including version information as appropriate).

The only other field that you may want to set is the "Category" field, which allows you to place the bug report into a broad category (such as "Documentation" or "Library").
Each bug report will be assigned to a developer who will determine what needs to be done to correct the problem. You will receive an update each time action is taken on the bug.

See Also:

**How to Report Bugs Effectively**
Article which goes into some detail about how to create a useful bug report. This describes what kind of information is useful and why it is useful.

**Bug Writing Guidelines**
Information about writing a good bug report. Some of this is specific to the Mozilla project, but describes general good practices.
B. History and License

Subsections

- B.1 History of the software
- B.2 Terms and conditions for accessing or otherwise using Python
- B.3 Licenses and Acknowledgements for Incorporated Software
  - B.3.1 Mersenne Twister
  - B.3.2 Sockets
  - B.3.3 Floating point exception control
  - B.3.4 MD5 message digest algorithm
  - B.3.5 Asynchronous socket services
  - B.3.6 Cookie management
  - B.3.7 Profiling
  - B.3.8 Execution tracing
  - B.3.9 UUencode and UUdecode functions
  - B.3.10 XML Remote Procedure Calls

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
B.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

<table>
<thead>
<tr>
<th>Release</th>
<th>Derived from</th>
<th>Year</th>
<th>Owner</th>
<th>GPL compatible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.0 thru 1.2</td>
<td>n/a</td>
<td>1991-1995</td>
<td>CWI</td>
<td>yes</td>
</tr>
<tr>
<td>1.3 thru 1.5.2</td>
<td>1.2</td>
<td>1995-1999</td>
<td>CNRI</td>
<td>yes</td>
</tr>
<tr>
<td>1.6</td>
<td>1.5.2</td>
<td>2000</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.0</td>
<td>1.6</td>
<td>2000</td>
<td>BeOpen.com</td>
<td>no</td>
</tr>
<tr>
<td>1.6.1</td>
<td>1.6</td>
<td>2001</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>no</td>
</tr>
<tr>
<td>2.0.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.1</td>
<td>2.1+2.0.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2</td>
<td>2.1.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>Version</td>
<td>Previous</td>
<td>Year</td>
<td>License</td>
<td>Free?</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>------</td>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>2.1.2</td>
<td>2.1.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.3</td>
<td>2.1.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.1</td>
<td>2.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.2</td>
<td>2.2.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.3</td>
<td>2.2.2</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3</td>
<td>2.2.2</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.1</td>
<td>2.3</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.2</td>
<td>2.3.1</td>
<td>2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.3</td>
<td>2.3.2</td>
<td>2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.4</td>
<td>2.3.3</td>
<td>2004</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.5</td>
<td>2.3.4</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4</td>
<td>2.3</td>
<td>2004</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.1</td>
<td>2.4</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.2</td>
<td>2.4.1</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.3</td>
<td>2.4.2</td>
<td>2006</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.5</td>
<td>2.4</td>
<td>2006</td>
<td>PSF</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Note:** GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.
B.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.5

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.5 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.5 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2006 Python Software Foundation; All Rights Reserved" are retained in Python 2.5 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.5 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.5.

4. PSF is making Python 2.5 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.5 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.5, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com (``BeOpen''), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (``Licensee'') accessing and otherwise using this software in source or binary form and its associated documentation (``the Software'').

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an ``AS IS'' basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT
LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the ``BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (``CNRI"), and the Individual or Organization (``Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to
reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., ``Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved'' are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): ``Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an ``AS IS'' basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual
property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR
ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
B.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

Subsections

- B.3.1 Mersenne Twister
- B.3.2 Sockets
- B.3.3 Floating point exception control
- B.3.4 MD5 message digest algorithm
- B.3.5 Asynchronous socket services
- B.3.6 Cookie management
- B.3.7 Profiling
- B.3.8 Execution tracing
- B.3.9 UUencode and UUdecode functions
- B.3.10 XML Remote Procedure Calls

Previous: B.2 Terms and conditions  Up: B. History and License  Next: B.3.1 Mersenne Twister

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
B.3.1 Mersenne Twister

The _random module includes code based on a download from [http://www.math.keio.ac.jp/~matumoto/MT2002/emt19937ar.html](http://www.math.keio.ac.jp/~matumoto/MT2002/emt19937ar.html). The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26. Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed) or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura, All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.
email: matumoto@math.keio.ac.jp
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
B.3.2 Sockets

The socket module uses the functions, getaddrinfo, and getnameinfo, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/about/index.html.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS''
GAI ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR P
ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LI
FOR GAI ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CON
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GO
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION
HOWEVER CAUSED AND ON GAI ANY THEORY OF LIABILITY, WHETHER IN CONTRA
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN GA
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
SUCH DAMAGE.
B.3.3 Floating point exception control

The source for the fpectl module includes the following notice:

---------------------------------------------------------------------
/       Copyright (c) 1996.
|       The Regents of the University of California.
|       All rights reserved.
|       Permission to use, copy, modify, and distribute this software
|       any purpose without fee is hereby granted, provided that this
|       tire notice is included in all copies of any software which i
|       includes a copy or modification of this software and in
|       copies of the supporting documentation for such software.
|       This work was produced at the University of California, Lawr
|       Livermore National Laboratory under contract no. W-7405-EN
|       between the U.S. Department of Energy and The Regents of
|       University of California for the operation of UC LLNL.
|       DISCLAIMER
|       This software was prepared as an account of work sponsored b
|       agency of the United States Government. Neither the United St
|       Government nor the University of California nor any of their
|       ployees, makes any warranty, express or implied, or assumes
|       liability or responsibility for the accuracy, completeness
|       usefulness of any information, apparatus, product, or pro
|       disclosed, or represents that its use would not infr
|       privately-owned rights. Reference herein to any specific com
|       mercial products, process, or service by trade name, tradem
|       manufacturer, or otherwise, does not necessarily constitute
|       imply its endorsement, recommendation, or favoring by the Un
|       States Government or the University of California. The views
|       opinions of authors expressed herein do not necessarily stat
|       reflect those of the United States Government or the Univer
|       of California, and shall not be used for advertising or pro
|       endorsement purposes.
---------------------------------------------------------------------

Previous: B.3.2 Sockets  Up: B.3 Licenses and Acknowledgements  Next: B.3.4 MD5 message digest

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
B.3.4 MD5 message digest algorithm

The source code for the md5 module contains the following notice:

    Copyright (C) 1999, 2002 Aladdin Enterprises. All rights reserved

    This software is provided 'as-is', without any express or implied
    warranty. In no event will the authors be held liable for any dam
    arising from the use of this software.

    Permission is granted to anyone to use this software for any purpo
    including commercial applications, and to alter it and redistribut
    freely, subject to the following restrictions:

    1. The origin of this software must not be misrepresented; you mus
    claim that you wrote the original software. If you use this sof
    in a product, an acknowledgment in the product documentation wo
    appreciated but is not required.
    2. Altered source versions must be plainly marked as such, and mus
    misrepresented as being the original software.
    3. This notice may not be removed or altered from any source distr

    L. Peter Deutsch
    ghost@aladdin.com

    Independent implementation of MD5 (RFC 1321).

    This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
        http://www.ietf.org/rfc/rfc1321.txt
    The code is derived from the text of the RFC, including the test s
    (section A.5) but excluding the rest of Appendix A. It does not i
    any code or documentation that is identified in the RFC as being
    copyrighted.

    The original and principal author of md5.h is L. Peter Deutsch
    <ghost@aladdin.com>. Other authors are noted in the change histor
    that follows (in reverse chronological order):

    2002-04-13 lpd Removed support for non-ANSI compilers; removed
    references to Ghostscript; clarified derivation from RFC 132
    now handles byte order either statically or dynamically.
    1999-11-04 lpd Edited comments slightly for automatic TOC extracti
    1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than
    added conditionalization for C++ compilation from Martin
    Purschke <purschke@bnl.gov>.
    1999-05-03 lpd Original version.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
B.3.5 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
B.3.6 Cookie management

The Cookie module contains the following notice:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Timothy O'Malley not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
B.3.7 Profiling

The profile and pstats modules contain the following notice:

```
Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python softwar
and its associated documentation for any purpose (subject to the
restriction in the following sentence) without fee is hereby grante
provided that the above copyright notice appears in all copies, and
that both that copyright notice and this permission notice appear i
supporting documentation, and that the name of InfoSeek not be used
advertising or publicity pertaining to distribution of the software
without specific, written prior permission. This permission is
explicitly restricted to the copying and modification of the softwa
to remain in Python, compiled Python, or other languages (such as C
wherein the modified or derived code is exclusively imported into a
Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY
SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVE
RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION O
CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```
B.3.8 Execution tracing

The `trace` module contains the following notice:

```plaintext
portions copyright 2001, Autonomous Zones Industries, Inc., all rig
err... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights res
```

Permission to use, copy, modify, and distribute this Python softwar
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all co
and that both that copyright notice and this permission notice appe
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertai
distribution of the software without specific, written prior permis
B.3.9 UUencode and UUdecode functions

The uu module contains the following notice:

Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OU
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. Th
  version is still 5 times faster, though.
- Arguments more compliant with python standard
B.3.10 XML Remote Procedure Calls

The xmlrpclib module contains the following notice:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicit pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
Index

_ | a | b | c | d | e | f | g | h | i | k | l | m | n | o | p | r | s | t | u | v
abort()
abs() (built-in function)
apply() (built-in function), [Link], [Link]
argv (in module sys)
<table>
<thead>
<tr>
<th><strong>BaseException (built-in exception)</strong></th>
<th><strong>buffer object</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>buffer interface</strong></td>
<td><strong>BufferType (in module types)</strong></td>
</tr>
<tr>
<td>Calloc()</td>
<td>CObject object</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>classmethod()</td>
<td>coerce() (built-in function)</td>
</tr>
<tr>
<td>cleanup functions</td>
<td>compile() (built-in function)</td>
</tr>
<tr>
<td>close() (in module os)</td>
<td>complex number object</td>
</tr>
<tr>
<td>cmp() (built-in function), [Link]</td>
<td>copyright (in module sys)</td>
</tr>
<tr>
<td>dictionary object</td>
<td>DictType (in module types)</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>DictionaryType (in module types)</td>
<td>divmod() (built-in function)</td>
</tr>
</tbody>
</table>
environment variables

exec_prefix, [Link], [Link]
PATH, [Link]
prefix, [Link], [Link], [Link], [Link]
PYTHONDUMPPREFS
PYTHONHOME, [Link]
PYTHONPATH, [Link]
EOFError (built-in exception)
errno

exc_info() (in module sys)
exc_traceback (in module sys), [Link]
exc_type (in module sys), [Link]
exc_value (in module sys), [Link]
exceptions (built-in module)
exec_prefix (environment variable), [Link], [Link]
executable (in module sys)
exit()
<table>
<thead>
<tr>
<th>file object</th>
<th>fopen()</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileType (in module types)</td>
<td>free()</td>
</tr>
<tr>
<td>float() (built-in function)</td>
<td>freeze utility</td>
</tr>
<tr>
<td>floating point object</td>
<td>frozenset object</td>
</tr>
<tr>
<td>FloatType (in modules types)</td>
<td>function object</td>
</tr>
</tbody>
</table>
G

getcharbufferproc (C type)
getreadbufferproc (C type)
getsegcountproc (C type)

getwritebufferproc (C type)
global interpreter lock
hash() (built-in function), [Link]
Hooks (standard module)
incr_item(), [Link]
inquiry (C type)
instance object

int() (built-in function)
integer object
interpreter lock
IntType (in modules types)
KeyboardInterrupt (built-in exception), [Link]
len() (built-in function), [Link], [Link], [Link], [Link], list object
ListType (in module types)
lock, interpreter

long() (built-in function)
long integer object
LONG_MAX, [Link]
LongType (in modules types)
<table>
<thead>
<tr>
<th>Function</th>
<th>Method Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>METH_STATIC</td>
</tr>
<tr>
<td>malloc()</td>
<td>METH_VARARGS</td>
</tr>
<tr>
<td>mapping object</td>
<td>method object</td>
</tr>
<tr>
<td>METH_CLASS</td>
<td>MethodType (in module types), [Link]</td>
</tr>
<tr>
<td>METH_COEXIST</td>
<td>module</td>
</tr>
<tr>
<td>METH_KEYWORDS</td>
<td>search path, [Link], [Link]</td>
</tr>
<tr>
<td>METH_NOARGS</td>
<td>module object</td>
</tr>
<tr>
<td>METH_O</td>
<td>modules (in module sys), [Link]</td>
</tr>
<tr>
<td>METH_OLDARGS</td>
<td>ModuleType (in module types)</td>
</tr>
</tbody>
</table>
object
  buffer
  CObject
  complex number
  dictionary
  file
  floating point
  frozenset
  function
  instance
  integer

object (continued)
  list
  long integer
  mapping
  method
  module
  None
  numeric
  sequence
  set
  string
  tuple
  type, [Link]

OverflowError (built-in exception), [Link]
package variable

    all

path

    module search, [Link], [Link]
PATH (environment variable), [Link]
path (in module sys), [Link], [Link]
platform (in module sys)
pow() (built-in function), [Link]
prefix (environment variable), [Link], [Link], [Link], [Link]
Py_AtExit()
Py_BEGIN_ALLOW_THREADS, [Link]
Py_BLOCK_THREADS
Py_BuildValue()
Py_CLEAR()
Py_CompileString(), [Link], [Link], [Link]
Py_CompileStringFlags()
Py_complex (C type)
Py_DECREF(), [Link]
Py_END_ALLOW_THREADS, [Link]
Py_END_OF_BUFFER
Py_EndInterpreter()
Py_false
Py_Finalize(), [Link], [Link], [Link], [Link], [Link]
Py_FindMethod()
Py_GetBuildInfo()
Py_GetBuildNumber()
Py_GetCompiler()
PyLong_AsUnsignedLongMask()
PyLong_AsVoidPtr()
PyLong_Check()
PyLong_CheckExact()
PyLong_FromDouble()
PyLong_FromLong()
PyLong_FromLongLong()
PyLong_FromLongLong()
PyLong_FromLongLong()
PyLong_Type
PyLongObject (C type)
PyMapping_Check()
PyMapping_DelItem()
PyMapping_DelItemString()
PyMapping_GetItemString()
PyMapping_HasKey()
PyMapping_HasKeyString()
PyMapping_Items()
PyMapping_Keys()
PyMapping_Length()
PyMapping_SetItemString()
PyMapping_Values()
PyMappingMethods (C type)
PyMarshal_ReadLastObjectFromFile()
PyMarshal_ReadLongFromFile()
PyMarshal_ReadObjectFromFile()
PyMarshal_ReadObjectFromFileStr
PyMarshal_ReadShortFromFile()
PyMarshal_WriteLongToFile()
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Py_GetCopyright()</td>
<td>Get the copyright information</td>
</tr>
<tr>
<td>Py_GetExecPrefix(), [Link]</td>
<td>Get the execution prefix</td>
</tr>
<tr>
<td>Py_GetPath(), [Link], [Link]</td>
<td>Get the path</td>
</tr>
<tr>
<td>Py_GetPlatform()</td>
<td>Get the platform</td>
</tr>
<tr>
<td>Py_GetPrefix(), [Link]</td>
<td>Get the prefix</td>
</tr>
<tr>
<td>Py_GetProgramFullPath(), [Link]</td>
<td>Get the program full path</td>
</tr>
<tr>
<td>Py_GetProgramName()</td>
<td>Get the program name</td>
</tr>
<tr>
<td>Py_GetVersion()</td>
<td>Get the version</td>
</tr>
<tr>
<td>Py_INCREF()</td>
<td>Increase a reference count</td>
</tr>
<tr>
<td>Py_Initialize()</td>
<td>Initialize the interpreter</td>
</tr>
<tr>
<td>Py_InitializeEx()</td>
<td>Initialize the interpreter with additional arguments</td>
</tr>
<tr>
<td>Py_InitModule()</td>
<td>Initialize a module</td>
</tr>
<tr>
<td>Py_InitModule3()</td>
<td>Initialize a module with additional arguments</td>
</tr>
<tr>
<td>Py_InitModule4()</td>
<td>Initialize a module with additional arguments and a function pointer</td>
</tr>
<tr>
<td>Py_IsInitialized()</td>
<td>Check if the interpreter is initialized</td>
</tr>
<tr>
<td>Py_Main()</td>
<td>Main function</td>
</tr>
<tr>
<td>Py_NewInterpreter()</td>
<td>Create a new interpreter</td>
</tr>
<tr>
<td>Py_NONE</td>
<td>None</td>
</tr>
<tr>
<td>Py_PRINT_RAW</td>
<td>Print raw text</td>
</tr>
<tr>
<td>Py_RETURN_FALSE</td>
<td>Return false</td>
</tr>
<tr>
<td>Py_RETURN_NONE</td>
<td>Return none</td>
</tr>
<tr>
<td>Py_RETURN_TRUE</td>
<td>Return true</td>
</tr>
<tr>
<td>Py_SetProgramName(), [Link], [Link], [Link], [Link], [Link]</td>
<td>Set the program name</td>
</tr>
<tr>
<td>Py_single_input</td>
<td>Single input</td>
</tr>
<tr>
<td>Py_TPFLAGS_BASETYPE</td>
<td>Flags for different types</td>
</tr>
<tr>
<td>Py_TPFLAGS_CHECKTYPES</td>
<td>Flags for checking types</td>
</tr>
<tr>
<td>Py_TPFLAGS_DEFAULT</td>
<td>Flags for default behavior</td>
</tr>
<tr>
<td>Py_TPFLAGS_GC</td>
<td>Flags for garbage collection</td>
</tr>
<tr>
<td>Py_TPFLAGS_HAVE_CLASS</td>
<td>Flags for having a class</td>
</tr>
<tr>
<td>Py_TPFLAGS_HAVE_GC</td>
<td>Flags for having garbage collection</td>
</tr>
<tr>
<td>Py_TPFLAGS_HAVE_INPLACEOPS</td>
<td>Flags for having in-place operations</td>
</tr>
<tr>
<td>Py_TPFLAGS_HAVE_ITER</td>
<td>Flags for having iteration</td>
</tr>
<tr>
<td>Py_TPFLAGS_HAVE_RICHCOMPARE</td>
<td>Flags for having rich comparison</td>
</tr>
<tr>
<td>Py_TPFLAGS_HAVE_SEQUENCE_IN</td>
<td>Flags for having sequence in</td>
</tr>
<tr>
<td>Py_TPFLAGS_HAVE_WEAKREFS</td>
<td>Flags for having weak references</td>
</tr>
<tr>
<td>Py_TPFLAGS_HEAPTYPE</td>
<td>Flags for heap type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyMarshal_WriteObjectToFile()</td>
<td>Write object to a file</td>
</tr>
<tr>
<td>PyMarshal_WriteObjectToString()</td>
<td>Write object to a string</td>
</tr>
<tr>
<td>PyMem_Delete()</td>
<td>Delete memory</td>
</tr>
<tr>
<td>PyMem_Free()</td>
<td>Free memory</td>
</tr>
<tr>
<td>PyMem_Malloc()</td>
<td>Allocate memory</td>
</tr>
<tr>
<td>PyMem_New()</td>
<td>New memory</td>
</tr>
<tr>
<td>PyMem_Realloc()</td>
<td>Reallocate memory</td>
</tr>
<tr>
<td>PyMem_Resize()</td>
<td>Resize memory</td>
</tr>
<tr>
<td>PyMethod_Check()</td>
<td>Check if a method exists</td>
</tr>
<tr>
<td>PyMethod_Class()</td>
<td>Get the class of a method</td>
</tr>
<tr>
<td>PyMethod_Function()</td>
<td>Get the function of a method</td>
</tr>
<tr>
<td>PyMethod_GET_CLASS()</td>
<td>Get the class of a method</td>
</tr>
<tr>
<td>PyMethod_GET_FUNCTION()</td>
<td>Get the function of a method</td>
</tr>
<tr>
<td>PyMethod_GET_SELF()</td>
<td>Get the self of a method</td>
</tr>
<tr>
<td>PyMethod_New()</td>
<td>Create a new method</td>
</tr>
<tr>
<td>PyMethod_Type</td>
<td>Get the type of a method</td>
</tr>
<tr>
<td>PyMethodDef (C type)</td>
<td>Define a method</td>
</tr>
<tr>
<td>PyModule_AddIntConstant()</td>
<td>Add an integer constant to a module</td>
</tr>
<tr>
<td>PyModule_AddObject()</td>
<td>Add an object to a module</td>
</tr>
<tr>
<td>PyModule_AddStringConstant()</td>
<td>Add a string constant to a module</td>
</tr>
<tr>
<td>PyModule_Check()</td>
<td>Check if a module exists</td>
</tr>
<tr>
<td>PyModule_CheckExact()</td>
<td>Check if a module exists exactly</td>
</tr>
<tr>
<td>PyModule_GetDict()</td>
<td>Get the dictionary of a module</td>
</tr>
<tr>
<td>PyModule_GetFilename()</td>
<td>Get the filename of a module</td>
</tr>
<tr>
<td>PyModule_GetName()</td>
<td>Get the name of a module</td>
</tr>
<tr>
<td>PyModule_New()</td>
<td>Create a new module</td>
</tr>
<tr>
<td>PyModule_Type</td>
<td>Get the type of a module</td>
</tr>
<tr>
<td>PyNumber_Absolute()</td>
<td>Get the absolute value of a number</td>
</tr>
<tr>
<td>PyNumber_Add()</td>
<td>Add two numbers</td>
</tr>
<tr>
<td>PyNumber_And()</td>
<td>Logical AND two numbers</td>
</tr>
<tr>
<td>PyNumber_AsSsize_t()</td>
<td>Cast to a size_t</td>
</tr>
<tr>
<td>PyNumber_Check()</td>
<td>Check if a number is valid</td>
</tr>
<tr>
<td>PyNumber_Coerce()</td>
<td>Coerce a number</td>
</tr>
<tr>
<td>PyNumber_Divide()</td>
<td>Divide two numbers</td>
</tr>
<tr>
<td>PyNumber_Divmod()</td>
<td>Divide two numbers and get the remainder</td>
</tr>
<tr>
<td>PyNumber_Float()</td>
<td>Convert to a float</td>
</tr>
<tr>
<td>PyNumber_FloorDivide()</td>
<td>Floor divide two numbers</td>
</tr>
<tr>
<td>PyNumber_Index()</td>
<td>Index of a number</td>
</tr>
</tbody>
</table>
Py_TPFLAGS_READY
Py_TPFLAGS_READYING
Py_tracefunc (C type)
Py_True
Py_UNBLOCK_THREADS
Py_UNICODE (C type)
Py_UNICODE_ISALNUM()
Py_UNICODE_ISALPHA()
Py_UNICODE_ISDECIMAL()
Py_UNICODE_ISDIGIT()
Py_UNICODE_ISSPACE()
Py_UNICODE_ISTITLE()
Py_UNICODE_ISUPPER()
Py_UNICODE_TODECIMAL()
Py_UNICODE_TODIGIT()
Py_UNICODE_TOLOWER()
Py_UNICODE_TONUMERIC()
Py_VISIT()
Py_XDECREF(), [Link]
Py_XDECREF()
PyAnySet_Check()
PyAnySet_CheckExact()
PyArg_Parse()
PyArg_ParseTuple()
PyArg_ParseTupleAndKeywords()
PyArg_UnpackTuple()
PyArg_Parse()
PyArg_ParseTupleAndKeywords()
PyBool_Check()
PyBool_FromLong()
PyBuffer_Check()
PyBuffer_fromMemory()
PyBuffer_FromObject()
PyBuffer_fromReadWriteMemory()
PyNumber_InPlaceAdd()
PyNumber_InPlaceAnd()
PyNumber_InPlaceDivide()
PyNumber_InPlaceFloorDivide()
PyNumber_InPlaceLshift()
PyNumber_InPlaceMultiply()
PyNumber_InPlaceOr()
PyNumber_InPlacePower()
PyNumber_InPlaceRemainder()
PyNumber_InPlaceRshift()
PyNumber_InPlaceSubtract()
PyNumber_InPlaceTrueDivide()
PyNumber_InPlaceXor()
PyNumber_Int()
PyNumber_Invert()
PyNumber_Long()
PyNumber_Lshift()
PyNumber_Multiply()
PyNumber_Negative()
PyNumber_Or()
PyNumber_Positive()
PyNumber_Power()
PyNumber_Remainder()
PyNumber_Rshift()
PyNumber_Subtract()
PyNumber_TrueDivide()
PyNumber_Xor()
PyNumberMethods (C type)
PyObject (C type)
PyObject_AsCharBuffer()
PyObject_AsFileDescriptor()
PyObject_AsReadWriteBuffer()
PyObject_Call()
PyObject_CallFunction()
PyObject_CallFunctionObjArgs()
PyObject_CallMethod()
PyObject_CallMethodObjArgs()
PyObject_CallObject()
PyObject_Cmp()
PyObject_COMPARE()
PyObject_Del()
PyObject_DELATTR()
PyObject_DELATTRSTRING()
PyObject_DELITEM()
PyObject_DIR()
PyObject_GC_DEL()
PyObject_GC_NEW()
PyObject_GC_NEWVAR()
PyObject_GC_RESIZE()
PyObject_GC_TRACK()
PyObject_GC_UNTRACK()
PyObject_GETATTR()
PyObject_GETATTRSTRING()
PyObject_GETITEM()
PyObject_GETITER()
PyObject_HASATTR()
PyObject_HASATTRSTRING()
PyObject_HASH()
PyObject_HEAD
PyObject_INIT()
PyObject_INITVAR()
PyObject_ISINSTANCE()
PyObject_ISSUBCLASS()
PyObject_ISTRUE()
PyObject_LENGTH()
PyObject_NEW()
PyObject_NEWVAR()
PyObject_NOT()
PyObject_PRINT()
PyObject_REPR()
PyObject_RICHCOMPARE()
PyObject_RICHCOMPAREBOOL()
PyObject_SETATTR()
PyObject_SETATTRSTRING()
PyObject_SETITEM()
PyObject_SIZE()
PyEval_InitThreads(), PyEval_ReleaseLock(), PyEval_ReleaseThread(), PyEval_SaveThread(), PyEval_SetProfile(), PyEval_SetTrace(), PyEval_ThreadsInitialized(), PyExc_ArithmeticError, PyExc_AssertionError, PyExc_AttributeError, PyExc_BaseException, PyExc_EnvironmentError, PyExc_EOFError, PyExc_Exception, PyExc_FloatingPointError, PyExc_ImportError, PyExc_IndexError, PyExc_IOError, PyExc_KeyboardInterrupt, PyExc_KeyError, PyExc_LookupError, PyExc_MemoryError, PyExc_NameError, PyExc_NotImplementedError, PyExc_OSError, PyExc_OverflowError, PyExc_ReferenceError, PyExc_RuntimeError, PyExc_StandardError, PyExc_SyntaxError, PyExc_SystemError, PyExc_SystemExit, PyExc_TypeError, PyExc_ValueError, PyExc_WindowsError, PyExc_ZeroDivisionError, PyFile_AsFile(), PyFile_Check(), PyString_AS_STRING(), PyString_AsDecodedObject(), PyString_AsEncodedObject(), PyString_AsString(), PyString_AsStringAndSize(), PyString_Check(), PyString_CheckExact(), PyString_Concat(), PyString_ConcatAndDel(), PyString_Decode(), PyString.Encode(), PyString_Format(), PyString_FromFormat(), PyString_FromFormatV(), PyString_FromString(), PyString_FromStringAndSize(), PyString_GET_SIZE(), PyString_InternFromString(), PyString_InternInPlace(), PyString_Size(), PyString_Type, PyStringObject (C type), PySys_SetArgv(), PYTHONHOME (environment variable), PYTHONHOME (environment variable), PYTHONPATH (environment variable), PyThreadState, PyThreadState (C type), PyThreadState_Clear(), PyThreadState_Delete(), PyThreadState_Get(), PyThreadState_GetDict(), PyThreadState_New(), PyThreadState_Next(), PyThreadState_SetAsyncExc(), PyThreadState_Swap()
<table>
<thead>
<tr>
<th>Python Function</th>
<th>Python Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyGILState_Ensure()</td>
<td>PyUnicode_AsEncodedString()</td>
</tr>
<tr>
<td>PyGILState_Release()</td>
<td>PyUnicode_AsLatin1String()</td>
</tr>
<tr>
<td>PyImport_AddModule()</td>
<td>PyUnicode_AsMBCSString()</td>
</tr>
<tr>
<td>PyImport_AppendInittab()</td>
<td>PyUnicode_AsRawUnicodeEscape()</td>
</tr>
<tr>
<td>PyImport_Cleanup()</td>
<td>PyUnicode_AsUnicode()</td>
</tr>
<tr>
<td>PyImport_ExecCodeModule()</td>
<td>PyUnicode_AsUnicodeEscapeString()</td>
</tr>
<tr>
<td>PyImport_ExtendInittab()</td>
<td>PyUnicode_AsUTF16String()</td>
</tr>
<tr>
<td>PyImport_FrozenModules</td>
<td>PyUnicode_AsUTF8String()</td>
</tr>
<tr>
<td>PyImport_GetMagicNumber()</td>
<td>PyUnicode_AsWideChar()</td>
</tr>
<tr>
<td>PyImport_GetModuleDict()</td>
<td>PyUnicode_Check()</td>
</tr>
<tr>
<td>PyImport_Import()</td>
<td>PyUnicode_CheckExact()</td>
</tr>
<tr>
<td>PyImport_ImportFrozenModule()</td>
<td>PyUnicode_Compare()</td>
</tr>
<tr>
<td>PyImport_ImportModule()</td>
<td>PyUnicode_Concat()</td>
</tr>
<tr>
<td>PyImport_ImportModuleEx()</td>
<td>PyUnicode_Contains()</td>
</tr>
<tr>
<td>PyImport_ReloadModule()</td>
<td>PyUnicode_Count()</td>
</tr>
<tr>
<td>PyIndex_Check()</td>
<td>PyUnicode_Decode()</td>
</tr>
<tr>
<td>PyInstance_Check()</td>
<td>PyUnicode_DecodeASCII()</td>
</tr>
<tr>
<td>PyInstance_New()</td>
<td>PyUnicode_DecodeCharmap()</td>
</tr>
<tr>
<td>PyInstance_NewRaw()</td>
<td>PyUnicode_DecodeLatin1()</td>
</tr>
<tr>
<td>PyInstance_Type</td>
<td>PyUnicode_DecodeMBCS()</td>
</tr>
<tr>
<td>PyInt_AS_LONG()</td>
<td>PyUnicode_DecodeMBCSSingle()</td>
</tr>
<tr>
<td>PyInt_AsLong()</td>
<td>PyUnicode_DecodeRawUnicode()</td>
</tr>
<tr>
<td>PyInt_AsSsize_t()</td>
<td>PyUnicode_DecodeUTF16()</td>
</tr>
<tr>
<td>PyInt_AsUnsignedLongLongMask()</td>
<td>PyUnicode_DecodeUTF16Stateful()</td>
</tr>
<tr>
<td>PyInt_AsUnsignedLongMask()</td>
<td>PyUnicode_DecodeUTF8()</td>
</tr>
<tr>
<td>PyInt_Check()</td>
<td>PyUnicode_DecodeUTF8Stateful()</td>
</tr>
<tr>
<td>PyInt_CheckExact()</td>
<td>PyUnicode_Encode()</td>
</tr>
<tr>
<td>PyInt_FromLong()</td>
<td>PyUnicode_EncodeASCII()</td>
</tr>
<tr>
<td>PyInt_FromSsize_t()</td>
<td>PyUnicode_EncodeCharmap()</td>
</tr>
<tr>
<td>PyInt_FromString()</td>
<td>PyUnicode_EncodeLatin1()</td>
</tr>
<tr>
<td>PyInt_GetMax()</td>
<td>PyUnicode_EncodeMBCS()</td>
</tr>
<tr>
<td>PyInt_Type</td>
<td>PyUnicode_EncodeMBCSSingle()</td>
</tr>
<tr>
<td>PyInterpreterState (C type)</td>
<td>PyUnicode_EncodeRawUnicode()</td>
</tr>
<tr>
<td>PyInterpreterState_Clear()</td>
<td>PyUnicode_EncodeUnicode()</td>
</tr>
<tr>
<td>PyInterpreterState_Delete()</td>
<td>PyUnicode_EncodeUTF16()</td>
</tr>
<tr>
<td>PyInterpreterState_Head()</td>
<td>PyUnicode_EncodeUTF8()</td>
</tr>
<tr>
<td>PyInterpreterState_New()</td>
<td>PyUnicode_Find()</td>
</tr>
<tr>
<td>PyInterpreterState_Next()</td>
<td>PyUnicode_Format()</td>
</tr>
<tr>
<td>PyInterpreterState_ThreadHead()</td>
<td>PyUnicode_FromEncodedObject()</td>
</tr>
</tbody>
</table>
R

realloc()
reload() (built-in function)
repr() (built-in function), [Link]
reexec (standard module)
<table>
<thead>
<tr>
<th>text</th>
<th>link</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td></td>
</tr>
<tr>
<td>path, module</td>
<td>[Link],</td>
</tr>
<tr>
<td>sequence object</td>
<td></td>
</tr>
<tr>
<td>set object</td>
<td></td>
</tr>
<tr>
<td>set_all()</td>
<td></td>
</tr>
<tr>
<td>setcheckinterval()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(in module sys)</td>
</tr>
<tr>
<td>setvbuf()</td>
<td></td>
</tr>
<tr>
<td>SIGINT</td>
<td>[Link]</td>
</tr>
<tr>
<td>signal</td>
<td>(built-in module)</td>
</tr>
<tr>
<td>SliceType</td>
<td>(in module types)</td>
</tr>
<tr>
<td>softspace</td>
<td>(file attribute)</td>
</tr>
<tr>
<td>staticmethod()</td>
<td>(built-in function)</td>
</tr>
<tr>
<td>stderr</td>
<td>(in module sys)</td>
</tr>
<tr>
<td>stdin</td>
<td>(in module sys)</td>
</tr>
<tr>
<td>stdout</td>
<td>(in module sys)</td>
</tr>
<tr>
<td>str()</td>
<td>(built-in function)</td>
</tr>
<tr>
<td>strerror()</td>
<td></td>
</tr>
<tr>
<td>string object</td>
<td></td>
</tr>
<tr>
<td>StringType</td>
<td>(in module types)</td>
</tr>
<tr>
<td>sum_list()</td>
<td></td>
</tr>
<tr>
<td>sum_sequence()</td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td>(built-in module), [Link], [Link]</td>
</tr>
<tr>
<td>SystemError</td>
<td>(built-in exception), [Link]</td>
</tr>
</tbody>
</table>
thread (built-in module)
tp_alloc (PyTypeObject member)
tp_allocs (PyTypeObject member)
tp_as_buffer (PyTypeObject member)
tp_base (PyTypeObject member)
tp_bases (PyTypeObject member)
tp Basicsize (PyTypeObject member)
tp_call (PyTypeObject member)
tp_clear (PyTypeObject member)
tp_compare (PyTypeObject member)
tp dealloc (PyTypeObject member)
tp descr_get (PyTypeObject member)
tp descr_set (PyTypeObject member)
tp dict (PyTypeObject member)
tp dictoffset (PyTypeObject member)
tp_doc (PyTypeObject member)
tp_descflags (PyTypeObject member)
tp_free (PyTypeObject member)
tp frees (PyTypeObject member)
tp getattr (PyTypeObject member)
tp getattro (PyTypeObject member)
tp getset (PyTypeObject member)
tp getattro (PyTypeObject member)
tp_hash (PyTypeObject member)
tp_init (PyTypeObject member)
tp _is gc (PyTypeObject member)
tp _items (PyTypeObject member)
tp _iter (PyTypeObject member)
tp _iternext (PyTypeObject member)
tp _maxalloc (PyTypeObject member)
tp _members (PyTypeObject member)
tp _methods (PyTypeObject member)
tp _mro (PyTypeObject member)
tp_name (PyTypeObject member)
tp_new (PyTypeObject member)
tp_next (PyTypeObject member)
tp_print (PyTypeObject member)
tp _repr (PyTypeObject member)
tp _richcompare (PyTypeObject member)
tp_setattr (PyTypeObject member)
tp_setattro (PyTypeObject member)
tp_str (PyTypeObject member)
tp subclasses (PyTypeObject member)
tp traverse (PyTypeObject member)
tp weaklist (PyTypeObject member)
tp weaklist_offset (PyTypeObject member)
traverseproc (C type)
tuple () (built-in function), [Link]
tuple object
TupleType (in module types)
type () (built-in function)
type object, [Link]
TypeType (in module types)
ULONG_MAX

unicode() (built-in function)
version (in module sys), [Link], [Link]  visitproc (C type)
Abstract:

The Python language has a substantial body of documentation, much of it contributed by various authors. The markup used for the Python documentation is based on LaTeX and requires a significant set of macros written specifically for documenting Python. This document describes the macros introduced to support Python documentation and how they should be used to support a wide range of output formats.

This document describes the document classes and special markup used in the Python documentation. Authors may use this guide, in conjunction with the template files provided with the distribution, to create or maintain whole documents or sections.

If you're interested in contributing to Python's documentation, there's no need to learn LaTeX if you're not so inclined; plain text contributions are more than welcome as well.
4.2 Hierarchical Structure
4.3 Common Environments

5 Document Classes

6 Special Markup Constructs
6.1 Markup for the Preamble
6.2 Meta-information Markup
6.3 Information Units
6.4 Showing Code Examples
6.5 Inline Markup
6.6 Miscellaneous Text Markup
6.7 Module-specific Markup
6.8 Library-level Markup
6.9 Table Markup
6.10 Reference List Markup
6.11 Index-generating Markup
6.12 Grammar Production Displays
6.13 Graphical Interface Components

7 Processing Tools
7.1 External Tools
7.2 Internal Tools
7.3 Working on Cygwin

8 Including Graphics

9 Future Directions
9.1 Structured Documentation
9.2 Discussion Forums

About this document...
1 Introduction

Python's documentation has long been considered to be good for a free programming language. There are a number of reasons for this, the most important being the early commitment of Python's creator, Guido van Rossum, to providing documentation on the language and its libraries, and the continuing involvement of the user community in providing assistance for creating and maintaining documentation.

The involvement of the community takes many forms, from authoring to bug reports to just plain complaining when the documentation could be more complete or easier to use. All of these forms of input from the community have proved useful during the time I've been involved in maintaining the documentation.

This document is aimed at authors and potential authors of documentation for Python. More specifically, it is for people contributing to the standard documentation and developing additional documents using the same tools as the standard documents. This guide will be less useful for authors using the Python documentation tools for topics other than Python, and less useful still for authors not using the tools at all.

The material in this guide is intended to assist authors using the Python documentation tools. It includes information on the source distribution of the standard documentation, a discussion of the document types, reference material on the markup defined in the document classes, a list of the external tools needed for processing documents, and reference material on the tools provided with the documentation resources. At the end, there is also a section discussing future directions for the Python documentation and where to turn for more information.

If your interest is in contributing to the Python documentation, but you don't have the time or inclination to learn LaTeX and the markup structures documented here, there's a welcoming place for you among the Python contributors as well. Any time you feel that you can clarify existing documentation or provide documentation that's missing, the existing documentation team will gladly work with you to integrate your text, dealing with the markup for you. Please don't let the material in this document stand
between the documentation and your desire to help out!
2 Directory Structure

The source distribution for the standard Python documentation contains a large number of directories. While third-party documents do not need to be placed into this structure or need to be placed within a similar structure, it can be helpful to know where to look for examples and tools when developing new documents using the Python documentation tools. This section describes this directory structure.

The documentation sources are usually placed within the Python source distribution as the top-level directory Doc/, but are not dependent on the Python source distribution in any way.

The Doc/ directory contains a few files and several subdirectories. The files are mostly self-explanatory, including a README and a Makefile. The directories fall into three categories:

Document Sources
The LaTeX sources for each document are placed in a separate directory. These directories are given short names which vaguely indicate the document in each:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Document Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>api/</td>
<td>The Python/C API</td>
</tr>
<tr>
<td>dist/</td>
<td>Distributing Python Modules</td>
</tr>
<tr>
<td>doc/</td>
<td>Documenting Python</td>
</tr>
<tr>
<td>ext/</td>
<td>Extending and Embedding the Python Interpreter</td>
</tr>
<tr>
<td>inst/</td>
<td>Installing Python Modules</td>
</tr>
<tr>
<td>lib/</td>
<td>Python Library Reference</td>
</tr>
<tr>
<td>mac/</td>
<td>Macintosh Module Reference</td>
</tr>
<tr>
<td>ref/</td>
<td>Python Reference Manual</td>
</tr>
<tr>
<td>tut/</td>
<td>Python Tutorial</td>
</tr>
<tr>
<td>whatsnew/</td>
<td>What's New in Python 2.5</td>
</tr>
</tbody>
</table>

Format-Specific Output
Most output formats have a directory which contains a Makefile which
controls the generation of that format and provides storage for the formatted
documents. The only variations within this category are the Portable
Document Format (PDF) and PostScript versions are placed in the
directories paper-a4/ and paper-letter/ (this causes all the temporary files
created by LaTeX to be kept in the same place for each paper size, where
they can be more easily ignored).

<table>
<thead>
<tr>
<th>Directory</th>
<th>Output Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>html/</td>
<td>HTML output</td>
</tr>
<tr>
<td>info/</td>
<td>GNU info output</td>
</tr>
<tr>
<td>isilo/</td>
<td>iSilo documents (for Palm OS devices)</td>
</tr>
<tr>
<td>paper-a4/</td>
<td>PDF and PostScript, A4 paper</td>
</tr>
</tbody>
</table>

Supplemental Files

Some additional directories are used to store supplemental files used for the
various processes. Directories are included for the shared LaTeX document
classes, the LaTeX2HTML support, template files for various document
components, and the scripts used to perform various steps in the formatting
processes.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>commontex/</td>
<td>Document content shared among documents</td>
</tr>
<tr>
<td>perl/</td>
<td>Support for LaTeX2HTML processing</td>
</tr>
<tr>
<td>templates/</td>
<td>Example files for source documents</td>
</tr>
<tr>
<td>texinputs/</td>
<td>Style implementation for LaTeX</td>
</tr>
<tr>
<td>tools/</td>
<td>Custom processing scripts</td>
</tr>
</tbody>
</table>
3 Style Guide

The Python documentation should follow the *Apple Publications Style Guide* wherever possible. This particular style guide was selected mostly because it seems reasonable and is easy to get online.

Topics which are not covered in the Apple's style guide will be discussed in this document if necessary.

Footnotes are generally discouraged due to the pain of using footnotes in the HTML conversion of documents. Footnotes may be used when they are the best way to present specific information. When a footnote reference is added at the end of the sentence, it should follow the sentence-ending punctuation. The LaTeX markup should appear something like this:

```
This sentence has a footnote reference.\
\footnote{This is the footnote text.}
```

Footnotes may appear in the middle of sentences where appropriate.

Many special names are used in the Python documentation, including the names of operating systems, programming languages, standards bodies, and the like. Many of these were assigned LaTeX macros at some point in the distant past, and these macros lived on long past their usefulness. In the current markup, most of these entities are not assigned any special markup, but the preferred spellings are given here to aid authors in maintaining the consistency of presentation in the Python documentation.

Other terms and words deserve special mention as well; these conventions should be used to ensure consistency throughout the documentation:

**CPU**

For ``central processing unit.'' Many style guides say this should be spelled out on the first use (and if you must use it, do so!). For the Python documentation, this abbreviation should be avoided since there's no reasonable way to predict which occurrence will be the first seen by the reader. It is better to use the word ``processor'' instead.
POSIX
The name assigned to a particular group of standards. This is always uppercase. Use the macro \POSIX to represent this name.

Python
The name of our favorite programming language is always capitalized.

Unicode
The name of a character set and matching encoding. This is always written capitalized.

UNIX
The name of the operating system developed at AT&T Bell Labs in the early 1970s. Use the macro \UNIX to use this name.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4 LaTeX Primer

This section is a brief introduction to LaTeX concepts and syntax, to provide authors enough information to author documents productively without having to become ``TeXnicians.'' This does not teach everything needed to know about writing LaTeX for Python documentation; many of the standard ``environments'' are not described here (though you will learn how to mark something as an environment).

Perhaps the most important concept to keep in mind while marking up Python documentation is that while TeX is unstructured, LaTeX was designed as a layer on top of TeX which specifically supports structured markup. The Python-specific markup is intended to extend the structure provided by standard LaTeX document classes to support additional information specific to Python.

LaTeX documents contain two parts: the preamble and the body. The preamble is used to specify certain metadata about the document itself, such as the title, the list of authors, the date, and the class the document belongs to. Additional information used to control index generation and the use of bibliographic databases can also be placed in the preamble. For most authors, the preamble can be most easily created by copying it from an existing document and modifying a few key pieces of information.

The class of a document is used to place a document within a broad category of documents and set some fundamental formatting properties. For Python documentation, two classes are used: the manual class and the howto class. These classes also define the additional markup used to document Python concepts and structures. Specific information about these classes is provided in section §5. ``Document Classes,'' below. The first thing in the preamble is the declaration of the document's class.

After the class declaration, a number of macros are used to provide further information about the document and setup any additional markup that is needed. No output is generated from the preamble; it is an error to include free text in the preamble because it would cause output.

The document body follows the preamble. This contains all the printed
components of the document marked up structurally. Generic LaTeX structures include hierarchical sections, numbered and bulleted lists, and special structures for the document abstract and indexes.

### Subsections

- [4.1 Syntax](#)
- [4.2 Hierarchical Structure](#)
- [4.3 Common Environments](#)
4.1 Syntax

There are some things that an author of Python documentation needs to know about LaTeX syntax.

A comment is started by the `\percent` character ("\%") and continues through the end of the line and all leading whitespace on the following line. This is a little different from any programming language I know of, so an example is in order:

```
This is text. \% comment
   This is more text.  \% another comment
Still more text.
```

The first non-comment character following the first comment is the letter "T" on the second line; the leading whitespace on that line is consumed as part of the first comment. This means that there is no space between the first and second sentences, so the period and letter "T" will be directly adjacent in the typeset document.

Note also that though the first non-comment character after the second comment is the letter "S", there is whitespace preceding the comment, so the two sentences are separated as expected.

A group is an enclosure for a collection of text and commands which encloses the formatting context and constrains the scope of any changes to that context made by commands within the group. Groups can be nested hierarchically. The formatting context includes the font and the definition of additional macros (or overrides of macros defined in outer groups). Syntactically, groups are enclosed in braces:

```
{text in a group}
```

An alternate syntax for a group using brackets, [ . . . ], is used by macros and environment constructors which take optional parameters; brackets do not normally hold syntactic significance. A degenerate group, containing only one atomic bit of content, does not need to have an explicit group, unless it is required to avoid ambiguity. Since Python tends toward the explicit, groups are also made explicit in the documentation markup.
Groups are used only sparingly in the Python documentation, except for their use in marking parameters to macros and environments.

A *macro* is usually a simple construct which is identified by name and can take some number of parameters. In normal LaTeX usage, one of these can be optional. The markup is introduced using the backslash character ("\"), and the name is given by alphabetic characters (no digits, hyphens, or underscores). Required parameters should be marked as a group, and optional parameters should be marked using the alternate syntax for a group.

For example, a macro which takes a single parameter would appear like this:

\name{parameter}

A macro which takes an optional parameter would be typed like this when the optional parameter is given:

\name[optional]

If both optional and required parameters are to be required, it looks like this:

\name[optional]{required}

A macro name may be followed by a space or newline; a space between the macro name and any parameters will be consumed, but this usage is not practiced in the Python documentation. Such a space is still consumed if there are no parameters to the macro, in which case inserting an empty group (\{\}) or explicit word space ("\ " ) immediately after the macro name helps to avoid running the expansion of the macro into the following text. Macros which take no parameters but which should not be followed by a word space do not need special treatment if the following character in the document source if not a name character (such as punctuation).

Each line of this example shows an appropriate way to write text which includes a macro which takes no parameters:

This \UNIX{} is followed by a space.
This \UNIX\ is also followed by a space.
\UNIX, followed by a comma, needs no additional markup.

An *environment* is a larger construct than a macro, and can be used for things with more content than would conveniently fit in a macro parameter. They are
primarily used when formatting parameters need to be changed before and after a large chunk of content, but the content itself needs to be highly flexible. Code samples are presented using an environment, and descriptions of functions, methods, and classes are also marked using environments.

Since the content of an environment is free-form and can consist of several paragraphs, they are actually marked using a pair of macros: \texttt{\texttt{\begin{abstract}} and \texttt{\end{abstract}}. These macros both take the name of the environment as a parameter. An example is the environment used to mark the abstract of a document:

\begin{abstract}
This is the text of the abstract. It concisely explains what information is found in the document.

It can consist of multiple paragraphs.
\end{abstract}

An environment can also have required and optional parameters of its own. These follow the parameter of the \texttt{\texttt{\begin{abstract}}} macro. This example shows an environment which takes a single required parameter:

\begin{datadesc}{controlnames}
A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from \texttt{0} (NUL) to \texttt{0x1f} (US), in order, plus the mnemonic \texttt{\texttt{\samp{SP}}} for the space character.
\end{datadesc}

There are a number of less-used marks in LaTeX which are used to enter characters which are not found in ASCII or which a considered special, or active in TeX or LaTeX. Given that these are often used adjacent to other characters, the markup required to produce the proper character may need to be followed by a space or an empty group, or the markup can be enclosed in a group. Some which are found in Python documentation are:

<table>
<thead>
<tr>
<th>Character</th>
<th>Markup</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>\textasciicircum</td>
</tr>
<tr>
<td>~</td>
<td>\textasciitilde</td>
</tr>
<tr>
<td>&gt;</td>
<td>\textgreater</td>
</tr>
<tr>
<td>&lt;</td>
<td>\textless</td>
</tr>
<tr>
<td>ç</td>
<td>\texttt{\texttt{c c}}</td>
</tr>
</tbody>
</table>
| ö         | \textasciitilde
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.2 Hierarchical Structure

LaTeX expects documents to be arranged in a conventional, hierarchical way, with chapters, sections, sub-sections, appendixes, and the like. These are marked using macros rather than environments, probably because the end of a section can be safely inferred when a section of equal or higher level starts.

There are six "levels" of sectioning in the document classes used for Python documentation, and the deepest two levels\(^1\) are not used. The levels are:

<table>
<thead>
<tr>
<th>Level</th>
<th>Macro Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\chapter</td>
<td>(1)</td>
</tr>
<tr>
<td>2</td>
<td>\section</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>\subsection</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>\subsubsection</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>\paragraph</td>
<td>(2)</td>
</tr>
<tr>
<td>6</td>
<td>\subparagraph</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1) Only used for the manual documents, as described in section 5, "Document Classes."

(2) Not the same as a paragraph of text; nobody seems to use this.

Footnotes

... levels\(^1\)

The deepest levels have the highest numbers in the table.
4.3 Common Environments

LaTeX provides a variety of environments even without the additional markup provided by the Python-specific document classes introduced in the next section. The following environments are provided as part of standard LaTeX and are being used in the standard Python documentation; descriptions will be added here as time allows.

abstract
alltt
description
displaymath
document
enumerate
figure
flushleft
itemize
list
math
quotation
quote
sloppypar
verbatim
5 Document Classes

Two LaTeX document classes are defined specifically for use with the Python documentation. The \texttt{manual} class is for large documents which are sectioned into chapters, and the \texttt{howto} class is for smaller documents.

The \texttt{manual} documents are larger and are used for most of the standard documents. This document class is based on the standard LaTeX \texttt{report} class and is formatted very much like a long technical report. The \texttt{Python Reference Manual} is a good example of a \texttt{manual} document, and the \texttt{Python Library Reference} is a large example.

The \texttt{howto} documents are shorter, and don't have the large structure of the \texttt{manual} documents. This class is based on the standard LaTeX \texttt{article} class and is formatted somewhat like the Linux Documentation Project's ``HOWTO'' series as done originally using the LinuxDoc software. The original intent for the document class was that it serve a similar role as the LDP's HOWTO series, but the applicability of the class turns out to be somewhat broader. This class is used for ``how-to'' documents (this document is an example) and for shorter reference manuals for small, fairly cohesive module libraries. Examples of the later use include \texttt{Using Kerberos from Python}, which contains reference material for an extension package. These documents are roughly equivalent to a single chapter from a larger work.
6 Special Markup Constructs

The Python document classes define a lot of new environments and macros. This section contains the reference material for these facilities. Documentation for ``standard'' LaTeX constructs is not included here, though they are used in the Python documentation.

Subsections

- 6.1 Markup for the Preamble
- 6.2 Meta-information Markup
- 6.3 Information Units
- 6.4 Showing Code Examples
- 6.5 Inline Markup
- 6.6 Miscellaneous Text Markup
- 6.7 Module-specific Markup
- 6.8 Library-level Markup
- 6.9 Table Markup
- 6.10 Reference List Markup
- 6.11 Index-generating Markup
- 6.12 Grammar Production Displays
- 6.13 Graphical Interface Components

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
6.1 Markup for the Preamble

\release{ver}

Set the version number for the software described in the document.

\setshortversion{sver}

Specify the ``short'' version number of the documented software to be sver.
6.2 Meta-information Markup

\sectionauthor{author}{email}
Identifies the author of the current section. \textit{author} should be the author's name such that it can be used for presentation (though it isn't), and \textit{email} should be the author's email address. The domain name portion of the address should be lower case.

No presentation is generated from this markup, but it is used to help keep track of contributions.
6.3 Information Units

XXX Explain terminology, or come up with something more `lay.''

There are a number of environments used to describe specific features provided by modules. Each environment requires parameters needed to provide basic information about what is being described, and the environment content should be the description. Most of these environments make entries in the general index (if one is being produced for the document); if no index entry is desired, non-indexing variants are available for many of these environments. The environments have names of the form `featuredesc', and the non-indexing variants are named `featuredescni'. The available variants are explicitly included in the list below.

For each of these environments, the first parameter, name, provides the name by which the feature is accessed.

Environments which describe features of objects within a module, such as object methods or data attributes, allow an optional type name parameter. When the feature is an attribute of class instances, type name only needs to be given if the class was not the most recently described class in the module; the name value from the most recent `classdesc' is implied. For features of built-in or extension types, the type name value should always be provided. Another special case includes methods and members of general ``protocols," such as the formatter and writer protocols described for the formatter module: these may be documented without any specific implementation classes, and will always require the type name parameter to be provided.

\begin{cfuncdesc} \{type\}\{name\}\{args\} \\
\end{cfuncdesc}

Environment used to described a C function. The type should be specified as a typedef name, struct tag, or the name of a primitive type. If it is a pointer type, the trailing asterisk should not be preceded by a space. name should be the name of the function (or function-like pre-processor macro), and args should give the types and names of the parameters. The names need to be given so they may be used in the description.
\begin{cmemberdesc} \{container\}\{type\}\{name\} \\
\end{cmemberdesc}

Description for a structure member. \textit{container} should be the \texttt{typedef} name, if there is one, otherwise if should be \texttt{struct tag}'. The type of the member should given as \textit{type}, and the name should be given as \textit{name}. The text of the description should include the range of values allowed, how the value should be interpreted, and whether the value can be changed. References to structure members in text should use the \texttt{\member} macro.

\begin{csimplemacrodesc} \{name\} \\
\end{csimplemacrodesc}

Documentation for a ``simple'' macro. Simple macros are macros which are used for code expansion, but which do not take arguments so cannot be described as functions. This is not to be used for simple constant definitions. Examples of its use in the Python documentation include PyObject\_HEAD and Py\_BEGIN\_ALLOW\_THREADS.

\begin{ctypedesc} [\texttt{tag}]\{name\} \\
\end{ctypedesc}

Environment used to described a C type. The \textit{name} parameter should be the \texttt{typedef} name. If the type is defined as a \texttt{struct} without a \texttt{typedef}, \textit{name} should have the form \texttt{struct tag}. \textit{name} will be added to the index unless \texttt{tag} is provided, in which case \texttt{tag} will be used instead. \texttt{tag} should not be used for a \texttt{typedef} name.

\begin{cvardesc} \{type\}\{name\} \\
\end{cvardesc}

Description of a global C variable. \textit{type} should be the \texttt{typedef} name, \texttt{struct tag}, or the name of a primitive type. If variable has a pointer type, the trailing asterisk should not be preceded by a space.

\begin{datadesc} \{name\} \\
\end{datadesc}

This environment is used to document global data in a module, including both variables and values used as ``defined constants.'' Class and object attributes are not documented using this environment.

\begin{datadescni} \{name\} \\
\end{datadescni}
Like `{datadesc}`, but without creating any index entries.

\begin{excclassdesc} {name}{constructor parameters}\end{excclassdesc}
Describe an exception defined by a class. `constructor parameters` should not include the `self` parameter or the parentheses used in the call syntax. To describe an exception class without describing the parameters to its constructor, use the `{excdesc}` environment.

\begin{excdesc} {name}\end{excdesc}
Describe an exception. In the case of class exceptions, the constructor parameters are not described; use `{excclassdesc}` to describe an exception class and its constructor.

\begin{funcdesc} {name}{parameters}\end{funcdesc}
Describe a module-level function. `parameters` should not include the parentheses used in the call syntax. Object methods are not documented using this environment. Bound object methods placed in the module namespace as part of the public interface of the module are documented using this, as they are equivalent to normal functions for most purposes. The description should include information about the parameters required and how they are used (especially whether mutable objects passed as parameters are modified), side effects, and possible exceptions. A small example may be provided.

\begin{funcdescni} {name}{parameters}\end{funcdescni}
Like `{funcdesc}`, but without creating any index entries.

\begin{classdesc} {name}{constructor parameters}\end{classdesc}
Describe a class and its constructor. `constructor parameters` should not include the `self` parameter or the parentheses used in the call syntax.

\begin{classdesc*} {name}\end{classdesc*}
Describe a class without describing the constructor. This can be used to describe classes that are merely containers for attributes or which should never be instantiated or subclassed by user code.

\begin{memberdesc} [type name]{name}
\end{memberdesc}

Describe an object data attribute. The description should include information about the type of the data to be expected and whether it may be changed directly.

\begin{memberdescni} [type name]{name}
\end{memberdescni}

Like \memberdesc, but without creating any index entries.

\begin{methoddesc} [type name]{name}{parameters}
\end{methoddesc}

Describe an object method. \texttt{parameters} should not include the \texttt{self} parameter or the parentheses used in the call syntax. The description should include similar information to that described for \funcdesc.

\begin{methoddescni} [type name]{name}{parameters}
\end{methoddescni}

Like \methoddesc, but without creating any index entries.
6.4 Showing Code Examples

Examples of Python source code or interactive sessions are represented as `\verbatim` environments. This environment is a standard part of LaTeX. It is important to only use spaces for indentation in code examples since TeX drops tabs instead of converting them to spaces.

Representing an interactive session requires including the prompts and output along with the Python code. No special markup is required for interactive sessions. After the last line of input or output presented, there should not be an "unused" primary prompt; this is an example of what not to do:

```
>>> 1 + 1
2
>>> 
```

Within the `\verbatim` environment, characters special to LaTeX do not need to be specially marked in any way. The entire example will be presented in a monospaced font; no attempt at "pretty-printing" is made, as the environment must work for non-Python code and non-code displays. There should be no blank lines at the top or bottom of any `\verbatim` display.

Longer displays of verbatim text may be included by storing the example text in an external file containing only plain text. The file may be included using the standard `\verbatiminput` macro; this macro takes a single argument naming the file containing the text. For example, to include the Python source file `example.py`, use:

```
\verbatiminput{example.py}
```

Use of `\verbatiminput` allows easier use of special editing modes for the included file. The file should be placed in the same directory as the LaTeX files for the document.

The Python Documentation Special Interest Group has discussed a number of approaches to creating pretty-printed code displays and interactive sessions; see the Doc-SIG area on the Python Web site for more information on this topic.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
6.5 Inline Markup

The macros described in this section are used to mark just about anything interesting in the document text. They may be used in headings (though anything involving hyperlinks should be avoided there) as well as in the body text.

\bfcode {text}
Like \code, but also makes the font bold-face.

\cdata \{name\}
The name of a C-language variable.

\cfunction \{name\}
The name of a C-language function. name should include the function name and the trailing parentheses.

\character \{char\}
A character when discussing the character rather than a one-byte string value. The character will be typeset as with \samp.

\citetitle \[url\]\{title\}
A title for a referenced publication. If url is specified, the title will be made into a hyperlink when formatted as HTML.

\class \{name\}
A class name; a dotted name may be used.

\code \{text\}
A short code fragment or literal constant value. Typically, it should not include any spaces since no quotation marks are added.

\constant \{name\}
The name of a ``defined'' constant. This may be a C-language \#define or a Python variable that is not intended to be changed.

\csimplemacro \{name\}
The name of a ``simple'' macro. Simple macros are macros which are used for code expansion, but which do not take arguments so cannot be described
as functions. This is not to be used for simple constant definitions. Examples of its use in the Python documentation include PyObject\_HEAD and Py\_BEGIN\_ALLOW\_THREADS.

\ctype \{name\}
The name of a C typedef or structure. For structures defined without a typedef, use \ctype{struct struct_tag} to make it clear that the struct is required.

\deprecated \{version\}\{what to do\}
Declare whatever is being described as being deprecated starting with release version. The text given as what to do should recommend something to use instead. It should be complete sentences. The entire deprecation notice will be presented as a separate paragraph; it should either precede or succeed the description of the deprecated feature.

\dfn \{term\}
Mark the defining instance of term in the text. (No index entries are generated.)

\e
Produces a backslash. This is convenient in \code, \file, and similar macros, and the \alltt environment, and is only defined there. To create a backslash in ordinary text (such as the contents of the \citetitle macro), use the standard \textbackslash macro.

\email \{address\}
An email address. Note that this is not hyperlinked in any of the possible output formats. The domain name portion of the address should be lower case.

\emph \{text\}
Emphasized text; this will be presented in an italic font.

\envvar \{name\}
An environment variable. Index entries are generated.

\exception \{name\}
The name of an exception. A dotted name may be used.
\file \{file or dir\}
The name of a file or directory. In the PDF and PostScript outputs, single quotes and a font change are used to indicate the file name, but no quotes are used in the HTML output. **Warning:** The \file macro cannot be used in the content of a section title due to processing limitations.

\filenq \{file or dir\}
Like \file, but single quotes are never used. This can be used in conjunction with tables if a column will only contain file or directory names. **Warning:** The \filenq macro cannot be used in the content of a section title due to processing limitations.

\function \{name\}
The name of a Python function; dotted names may be used.

\infinity
The symbol for mathematical infinity: \( \infty \). Some Web browsers are not able to render the HTML representation of this symbol properly, but support is growing.

\kbd \{key sequence\}
Mark a sequence of keystrokes. What form *key sequence* takes may depend on platform- or application-specific conventions. When there are no relevant conventions, the names of modifier keys should be spelled out, to improve accessibility for new users and non-native speakers. For example, an *xemacs* key sequence may be marked like \kbd{C-x C-f}, but without reference to a specific application or platform, the same sequence should be marked as \kbd{Control-x Control-f}.

\keyword \{name\}
The name of a keyword in a programming language.

\mailheader \{name\}
The name of an RFC 822-style mail header. This markup does not imply that the header is being used in an email message, but can be used to refer to any header of the same ```style.' This is also used for headers defined by the various MIME specifications. The header name should be entered in the same way it would normally be found in practice, with the camel-casing conventions being preferred where there is more than one common usage.
The colon which follows the name of the header should not be included.
For example: \mailheader{Content-Type}.

\makevar {name}
The name of a make variable.

\manpage {name} {section}
A reference to a UNIX manual page.

\member {name}
The name of a data attribute of an object.

\method {name}
The name of a method of an object. name should include the method name
and the trailing parentheses. A dotted name may be used.

\mimetype {name}
The name of a MIME type, or a component of a MIME type (the major or
minor portion, taken alone).

\module {name}
The name of a module; a dotted name may be used. This should also be
used for package names.

\newsgroup {name}
The name of a Usenet newsgroup.

\note {text}
An especially important bit of information about an API that a user should
be aware of when using whatever bit of API the note pertains to. This
should be the last thing in the paragraph as the end of the note is not
visually marked in any way. The content of text should be written in
complete sentences and include all appropriate punctuation.

\pep {number}
A reference to a Python Enhancement Proposal. This generates appropriate
index entries. The text "PEP number" is generated; in the HTML output,
this text is a hyperlink to an online copy of the specified PEP.
The symbol for indicating a value that may take a positive or negative value of a specified magnitude, typically represented by a plus sign placed over a minus sign. For example: \plusminus 3%.

\program {name}
The name of an executable program. This may differ from the file name for the executable for some platforms. In particular, the .exe (or other) extension should be omitted for Windows programs.

\programopt {option}
A command-line option to an executable program. Use this only for ``short'' options, and include the leading hyphen.

\longprogramopt {option}
A long command-line option to an executable program. This should only be used for long option names which will be prefixed by two hyphens; the hyphens should not be provided as part of option.

\refmodule [key]{name}
Like \module, but create a hyperlink to the documentation for the named module. Note that the corresponding \declaremodule must be in the same document. If the \declaremodule defines a module key different from the module name, it must also be provided as key to the \refmodule macro.

\regexp {string}
Mark a regular expression.

\rfc {number}
A reference to an Internet Request for Comments. This generates appropriate index entries. The text "RFC number" is generated; in the HTML output, this text is a hyperlink to an online copy of the specified RFC.

\samp {text}
A short code sample, but possibly longer than would be given using \code. Since quotation marks are added, spaces are acceptable.
\shortversion
The ``short'' version number of the documented software, as specified using the \setshortversion macro in the preamble. For Python, the short version number for a release is the first three characters of the sys\texttt{.version} value. For example, versions 2.0b1 and 2.0.1 both have a short version of 2.0. This may not apply for all packages; if \setshortversion is not used, this produces an empty expansion. See also the \version macro.

\strong{\textit{text}}
Strongly emphasized text; this will be presented using a bold font.

\ulink{\textit{\texttt{url}}}{\textit{\texttt{url}}}
A hypertext link with a target specified by a URL, but for which the link text should not be the title of the resource. For resources being referenced by name, use the \texttt{\citetitle} macro. Not all formatted versions support arbitrary hypertext links. Note that many characters are special to LaTeX and this macro does not always do the right thing. In particular, the tilde character ("~") is mis-handled; encoding it as a hex-sequence does work, use "%7e" in place of the tilde character.

\url{\texttt{url}}
A URL (or URN). The URL will be presented as text. In the HTML and PDF formatted versions, the URL will also be a hyperlink. This can be used when referring to external resources without specific titles; references to resources which have titles should be marked using the \texttt{\citetitle} macro. See the comments about special characters in the description of the \ulink macro for special considerations.

\var{name}
The name of a variable or formal parameter in running text.

\version
The version number of the described software, as specified using \texttt{\release} in the preamble. See also the \shortversion macro.

\warning{\textit{\texttt{text}}}
An important bit of information about an API that a user should be very aware of when using whatever bit of API the warning pertains to. This
should be the last thing in the paragraph as the end of the warning is not visually marked in any way. The content of text should be written in complete sentences and include all appropriate punctuation. This differs from note in that it is recommended over note for information regarding security.

The following two macros are used to describe information that's associated with changes from one release to another. For features which are described by a single paragraph, these are typically added as separate source lines at the end of the paragraph. When adding these to features described by multiple paragraphs, they are usually collected in a single separate paragraph after the description. When both versionadded and versionchanged are used, versionadded should come first; the versions should be listed in chronological order. Both of these should come before availability statements. The location should be selected so the explanation makes sense and may vary as needed.

\versionadded [explanation]{version}
The version of Python which added the described feature to the library or C API. explanation should be a brief explanation of the change consisting of a capitalized sentence fragment; a period will be appended by the formatting process. When this applies to an entire module, it should be placed at the top of the module section before any prose.

\versionchanged [explanation]{version}
The version of Python in which the named feature was changed in some way (new parameters, changed side effects, etc.). explanation should be a brief explanation of the change consisting of a capitalized sentence fragment; a period will be appended by the formatting process. This should not generally be applied to modules.
6.6 Miscellaneous Text Markup

In addition to the inline markup, some additional ``block'' markup is defined to make it easier to bring attention to various bits of text. The markup described here serves this purpose, and is intended to be used when marking one or more paragraphs or other block constructs (such as \verbatim environments).

\begin{notice} [type] \end{notice}

Label some paragraphs as being worthy of additional attention from the reader. What sort of attention is warranted can be indicated by specifying the type of the notice. The only values defined for type are note and warning; these are equivalent in intent to the inline markup of the same name. If type is omitted, note is used. Additional values may be defined in the future.
6.7 Module-specific Markup

The markup described in this section is used to provide information about a module being documented. Each module should be documented in its own \section. A typical use of this markup appears at the top of that section and might look like this:

\section{\module{spam} ---
    Access to the SPAM facility}
\declaremodule{extension}{spam}
\platform{Unix}
\modulesynopsis{Access to the SPAM facility of \UNIX.}
\moduleauthor{Jane Doe}{jane.doe@frobnitz.org}

Python packages -- collections of modules that can be described as a unit -- are documented using the same markup as modules. The name for a module in a package should be typed in ``fully qualified'' form (it should include the package name). For example, a module ``foo'' in package ``bar'' should be marked as \module{bar.foo}, and the beginning of the reference section would appear as:

\section{\module{bar.foo} ---
    Module from the \module{bar} package}
\declaremodule{extension}{bar.foo}
\modulesynopsis{Nifty module from the \module{bar} package.}
\moduleauthor{Jane Doe}{jane.doe@frobnitz.org}

Note that the name of a package is also marked using \module.

\declaremodule \key \type \name
Requires two parameters: module type ("standard", "builtin", "extension", or "))", and the module name. An optional parameter should be given as the basis for the module's \``key'' used for linking to or referencing the section. The \``key'' should only be given if the module's name contains any underscores, and should be the name with the underscores stripped. Note that the \type\ parameter must be one of the values listed above or an error will be printed. For modules which are contained in packages, the fully-qualified name should be given as \name parameter. This should be the first thing after the \section used to
introduce the module.

\platform {specifier}
Specifies the portability of the module. specifier is a comma-separated list of keys that specify what platforms the module is available on. The keys are short identifiers; examples that are in use include "IRIX", "Mac", "Windows", and "Unix". It is important to use a key which has already been used when applicable. This is used to provide annotations in the Module Index and the HTML and GNU info output.

\modulesynopsis {text}
The text is a short, "one line" description of the module that can be used as part of the chapter introduction. This is must be placed after \declaremodule. The synopsis is used in building the contents of the table inserted as the \localmoduletable. No text is produced at the point of the markup.

\moduleauthor {name} {email}
This macro is used to encode information about who authored a module. This is currently not used to generate output, but can be used to help determine the origin of the module.
6.8 Library-level Markup

This markup is used when describing a selection of modules. For example, the Macintosh Library Modules document uses this to help provide an overview of the modules in the collection, and many chapters in the Python Library Reference use it for the same purpose.

\localmoduletable

If a .syn file exists for the current chapter (or for the entire document in howto documents), a \synopsistable is created with the contents loaded from the .syn file.
6.9 Table Markup

There are three general-purpose table environments defined which should be used whenever possible. These environments are defined to provide tables of specific widths and some convenience for formatting. These environments are not meant to be general replacements for the standard LaTeX table environments, but can be used for an advantage when the documents are processed using the tools for Python documentation processing. In particular, the generated HTML looks good! There is also an advantage for the eventual conversion of the documentation to XML (see section 9, "Future Directions").

Each environment is named \texttt{tablecols}, where \texttt{cols} is the number of columns in the table specified in lower-case Roman numerals. Within each of these environments, an additional macro, \texttt{linecols}, is defined, where \texttt{cols} matches the \texttt{cols} value of the corresponding table environment. These are supported for \texttt{cols} values of \texttt{ii}, \texttt{iii}, and \texttt{iv}. These environments are all built on top of the \texttt{tabular} environment. Variants based on the \texttt{longtable} environment are also provided.

Note that all tables in the standard Python documentation use vertical lines between columns, and this must be specified in the markup for each table. A general border around the outside of the table is not used, but would be the responsibility of the processor; the document markup should not include an exterior border.

The \texttt{longtable}-based variants of the table environments are formatted with extra space before and after, so should only be used on tables which are long enough that splitting over multiple pages is reasonable; tables with fewer than twenty rows should never be marked using the long flavors of the table environments. The header row is repeated across the top of each part of the table.

\begin{tableii} \{colspec\}\{col1font\}\{heading1\}\{heading2\} \\end{tableii}

Create a two-column table using the LaTeX column specifier \texttt{colspec}. The column specifier should indicate vertical bars between columns as appropriate for the specific table, but should not specify vertical bars on the outside of the table (that is considered a stylesheet issue). The \texttt{col1font}
parameter is used as a stylicic treatment of the first column of the table: the first column is presented as $\textsc{col1font}\{\text{column1}\}$. To avoid treating the first column specially, $\textsc{col1font}$ may be "\textit{textrm}". The column headings are taken from the values $heading1$ and $heading2$.

\begin{longtable}{l}
... \end{longtable}

Like \texttt{\textbackslash tableii}, but produces a table which may be broken across page boundaries. The parameters are the same as for \texttt{\textbackslash tableii}.

\begin{lineii}{column1}{column2}
Create a single table row within a \texttt{\textbackslash tableii} or \texttt{\textbackslash longtableii} environment. The text for the first column will be generated by applying the macro named by the $\textsc{col1font}$ value when the \texttt{\textbackslash tableii} was opened.

\begin{tableiii}{c}{c}{c}{c}
\texttt{\textbackslash colspec}{\textsc{col1font}}\{\textit{heading1}\}\{\textit{heading2}\}
\texttt{\textbackslash heading3}
\end{tableiii}

Like the \texttt{\textbackslash tableii} environment, but with a third column. The heading for the third column is given by $heading3$.

\begin{longtableiii} ...
\end{longtableiii}

Like \texttt{\textbackslash tableiii}, but produces a table which may be broken across page boundaries. The parameters are the same as for \texttt{\textbackslash tableiii}.

\begin{lineiii}{column1}{column2}{column3}
Like the \texttt{\textbackslash lineii} macro, but with a third column. The text for the third column is given by $column3$.

\begin{tableiv}{c}{c}{c}{c}
\texttt{\textbackslash colspec}{\textsc{col1font}}\{\textit{heading1}\}\{\textit{heading2}\}
\texttt{\textbackslash heading3}{\textit{heading4}}
\end{tableiv}

Like the \texttt{\textbackslash tableiii} environment, but with a fourth column. The heading for the fourth column is given by $heading4$.

\begin{longtableiv} ...
\end{longtableiv}

Like \texttt{\textbackslash tableiv}, but produces a table which may be broken across page
boundaries. The parameters are the same as for \texttt{tableiv}.

\texttt{\lineiv column1}\{column2}\{column3}\{column4}
Like the \texttt{\lineiii} macro, but with a fourth column. The text for the fourth column is given by \texttt{column4}.

\texttt{\begin{tablev} colspec\{col1font\}heading1\{heading2\}heading3\{heading4\}heading5\end{tablev}}
Like the \texttt{\tableiv} environment, but with a fifth column. The heading for the fifth column is given by \texttt{heading5}.

\texttt{\begin{longtablev} ...
\end{longtablev}}
Like \texttt{\tablev}, but produces a table which may be broken across page boundaries. The parameters are the same as for \texttt{\tablev}.

\texttt{\linev column1}\{column2}\{column3}\{column4}\{column5}
Like the \texttt{\lineiv} macro, but with a fifth column. The text for the fifth column is given by \texttt{column5}.

An additional table-like environment is \texttt{synopsistable}. The table generated by this environment contains two columns, and each row is defined by an alternate definition of \texttt{modulesynopsis}. This environment is not normally used by authors, but is created by the \texttt{\localmoduletable} macro.

Here is a small example of a table given in the documentation for the \texttt{warnings} module; markup inside the table cells is minimal so the markup for the table itself is readily discernable. Here is the markup for the table:

\texttt{\begin{tableii}{l|l}{exception}{Class}{Description}
\lineii{Warning}
{This is the base class of all warning category classes. It is a subclass of \texttt{Exception}.}
\lineii{UserWarning}
{The default category for \texttt{warn}().}
\lineii{DeprecationWarning}
{Base category for warnings about deprecated features.}
\lineii{SyntaxWarning}
{Base category for warnings about dubious syntactic features.}
\lineii{RuntimeWarning}
\end{tableii}}
Here is the resulting table:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warning</td>
<td>This is the base class of all warning category classes. It is a subclass of Exception.</td>
</tr>
<tr>
<td>UserWarning</td>
<td>The default category for warn().</td>
</tr>
<tr>
<td>DeprecationWarning</td>
<td>Base category for warnings about deprecated features.</td>
</tr>
<tr>
<td>SyntaxWarning</td>
<td>Base category for warnings about dubious syntactic features.</td>
</tr>
<tr>
<td>RuntimeWarning</td>
<td>Base category for warnings about dubious runtime features.</td>
</tr>
</tbody>
</table>

Note that the class names are implicitly marked using the \texttt{exception} macro, since that is given as the \texttt{col1font} value for the \texttt{tableii} environment. To create a table using different markup for the first column, use \texttt{textrm} for the \texttt{col1font} value and mark each entry individually.

To add a horizontal line between vertical sections of a table, use the standard \texttt{hline} macro between the rows which should be separated:

\begin{tableii}{l|l}{constant}{Language}{Audience}
\texttt{APL}\{Masochists.\}
\texttt{BASIC}\{First-time programmers on PC hardware.\}
\texttt{C}\{\texttt{UNIX}\} \&\ \texttt{Linux kernel developers.}\n\texttt{Python}\{Everyone!\}
\end{tableii}

Note that not all presentation formats are capable of displaying a horizontal rule in this position. This is how the table looks in the format you're reading now:

<table>
<thead>
<tr>
<th>Language</th>
<th>Audience</th>
</tr>
</thead>
<tbody>
<tr>
<td>APL</td>
<td>Masochists.</td>
</tr>
<tr>
<td>C</td>
<td>\texttt{UNIX} &amp; Linux kernel developers.</td>
</tr>
</tbody>
</table>
6.10 Reference List Markup

Many sections include a list of references to module documentation or external documents. These lists are created using the `\seealso` or `\seealso*` environments. These environments define some additional macros to support creating reference entries in a reasonable manner.

The `\seealso` environment is typically placed in a section just before any sub-sections. This is done to ensure that reference links related to the section are not hidden in a subsection in the hypertext renditions of the documentation. For the HTML output, it is shown as a ``side bar," boxed off from the main flow of the text. The `\seealso*` environment is different in that it should be used when a list of references is being presented as part of the primary content; it is not specially set off from the text.

\begin{seealso} 
\end{seealso} 

This environment creates a ``See also:" heading and defines the markup used to describe individual references.

\begin{seealso*} 
\end{seealso*} 

This environment is used to create a list of references which form part of the main content. It is not given a special header and is not set off from the main flow of the text. It provides the same additional markup used to describe individual references.

For each of the following macros, `why` should be one or more complete sentences, starting with a capital letter (unless it starts with an identifier, which should not be modified), and ending with the appropriate punctuation.

These macros are only defined within the content of the `\seealso` and `\seealso*` environments.

\seelink \{url\}\{linktext\}\{why\} 
References to specific on-line resources should be given using the `\seelink` macro if they don't have a meaningful title but there is some short description of what's at the end of the link. Online documents which
have identifiable titles should be referenced using the `\seetitle` macro, using the optional parameter to that macro to provide the URL.

\seemodule [key]{name}{why}
Refers to another module. *why* should be a brief explanation of why the reference may be interesting. The module name is given in *name*, with the link key given in *key* if necessary. In the HTML and PDF conversions, the module name will be a hyperlink to the referred-to module. **Note:** The module must be documented in the same document (the corresponding `\declaremodule` is required).

\seepep {number}{title}{why}
Refer to a Python Enhancement Proposal (PEP). *number* should be the official number assigned by the PEP Editor, *title* should be the human-readable title of the PEP as found in the official copy of the document, and *why* should explain what's interesting about the PEP. This should be used to refer the reader to PEPs which specify interfaces or language features relevant to the material in the annotated section of the documentation.

\seerfc {number}{title}{why}
Refer to an IETF Request for Comments (RFC). Otherwise very similar to `\seepep`. This should be used to refer the reader to PEPs which specify protocols or data formats relevant to the material in the annotated section of the documentation.

\seetext {text}
Add arbitrary text *text* to the "See also:" list. This can be used to refer to off-line materials or on-line materials using the `\url` macro. This should consist of one or more complete sentences.

\seetitle [url]{title}{why}
Add a reference to an external document named *title*. If *url* is given, the title is made a hyperlink in the HTML version of the documentation, and displayed below the title in the typeset versions of the documentation.

\seeurl {url}{why}
References to specific on-line resources should be given using the `\seeurl` macro if they don't have a meaningful title. Online documents which have identifiable titles should be referenced using the `\seetitle`
macro, using the optional parameter to that macro to provide the URL.
6.11 Index-generating Markup

Effective index generation for technical documents can be very difficult, especially for someone familiar with the topic but not the creation of indexes. Much of the difficulty arises in the area of terminology: including the terms an expert would use for a concept is not sufficient. Coming up with the terms that a novice would look up is fairly difficult for an author who, typically, is an expert in the area she is writing on.

The truly difficult aspects of index generation are not areas with which the documentation tools can help. However, ease of producing the index once content decisions are made is within the scope of the tools. Markup is provided which the processing software is able to use to generate a variety of kinds of index entry with minimal effort. Additionally, many of the environments described in section 6.3, ``Information Units,'' will generate appropriate entries into the general and module indexes.

The following macro can be used to control the generation of index data, and should be used in the document preamble:

\makemodindex

This should be used in the document preamble if a ``Module Index'' is desired for a document containing reference material on many modules. This causes a data file libjobname.idx to be created from the \declaremodule macros. This file can be processed by the makeindex program to generate a file which can be \input into the document at the desired location of the module index.

There are a number of macros that are useful for adding index entries for particular concepts, many of which are specific to programming languages or even Python.

\bifuncindex {name}

Add an index entry referring to a built-in function named name; parentheses should not be included after name.

\exindex {exception}

Add a reference to an exception named exception. The exception should be
class-based.

\kwindex{keyword}
Add a reference to a language keyword (not a keyword parameter in a function or method call).

\obindex{object type}
Add an index entry for a built-in object type.

\opindex{operator}
Add a reference to an operator, such as "+".

\refmodindex{key}{module}
Add an index entry for module module; if module contains an underscore, the optional parameter key should be provided as the same string with underscores removed. An index entry `module (module)` will be generated. This is intended for use with non-standard modules implemented in Python.

\refexmodindex{key}{module}
As for \refmodindex, but the index entry will be `module (extension module)'. This is intended for use with non-standard modules not implemented in Python.

\refbimodindex{key}{module}
As for \refmodindex, but the index entry will be `module (built-in module)'. This is intended for use with standard modules not implemented in Python.

\refstmodindex{key}{module}
As for \refmodindex, but the index entry will be `module (standard module)'. This is intended for use with standard modules implemented in Python.

\stindex{statement}
Add an index entry for a statement type, such as print or try/finally.

XXX Need better examples of difference from \kwindex.
Additional macros are provided which are useful for conveniently creating general index entries which should appear at many places in the index by rotating a list of words. These are simple macros that simply use `\index` to build some number of index entries. Index entries build using these macros contain both primary and secondary text.

\texttt{\textbf{\indexii}} \{word1\}\{word2\}

Build two index entries. This is exactly equivalent to using `\index{word1!word2}` and `\index{word2!word1}`.

\texttt{\textbf{\indexiii}} \{word1\}\{word2\}\{word3\}

Build three index entries. This is exactly equivalent to using `\index{word1!word2 word3}`, `\index{word2!word3, word1}`, and `\index{word3!word1 word2}`.

\texttt{\textbf{\indexiv}} \{word1\}\{word2\}\{word3\}\{word4\}

Build four index entries. This is exactly equivalent to using `\index{word1!word2 word3 word4}`, `\index{word2!word3 word4, word1}`, `\index{word3!word4, word1 word2}`, and `\index{word4!word1 word2 word3}`.
6.12 Grammar Production Displays

Special markup is available for displaying the productions of a formal grammar. The markup is simple and does not attempt to model all aspects of BNF (or any derived forms), but provides enough to allow context-free grammars to be displayed in a way that causes uses of a symbol to be rendered as hyperlinks to the definition of the symbol. There is one environment and a pair of macros:

\begin{productionlist} [language]
\end{productionlist}

This environment is used to enclose a group of productions. The two macros are only defined within this environment. If a document describes more than one language, the optional parameter *language* should be used to distinguish productions between languages. The value of the parameter should be a short name that can be used as part of a filename; colons or other characters that can't be used in filename across platforms should be included.

\production {name} {definition}

A production rule in the grammar. The rule defines the symbol *name* to be *definition*. *name* should not contain any markup, and the use of hyphens in a document which supports more than one grammar is undefined. *definition* may contain \token macros and any additional content needed to describe the grammatical model of *symbol*. Only one \production may be used to define a symbol -- multiple definitions are not allowed.

\token {name}

The name of a symbol defined by a \production macro, used in the definition of a symbol. Where possible, this will be rendered as a hyperlink to the definition of the symbol *name*.

Note that the entire grammar does not need to be defined in a single \productionlist environment; any number of groupings may be used to describe the grammar. Every use of the \token must correspond to a \production.

The following is an example taken from the *Python Reference Manual*:
\begin{productionlist}
\production{identifier}
  {{\token{letter}"_"} {\token{letter} | \token{digit} |}
\production{letter}
  {\token{lowercase} | \token{uppercase}}
\production{lowercase}
  {"a"..."z"}
\production{uppercase}
  {"A"..."Z"}
\production{digit}
  {"0"..."9"}
\end{productionlist}
6.13 Graphical Interface Components

The components of graphical interfaces will be assigned markup, but most of the specifics have not been determined.

\guilabel \{label\}
Labels presented as part of an interactive user interface should be marked using \guilabel. This includes labels from text-based interfaces such as those created using \curses or other text-based libraries. Any label used in the interface should be marked with this macro, including button labels, window titles, field names, menu and menu selection names, and even values in selection lists.

\menuselection \{menupath\}
Menu selections should be marked using a combination of \menuselection and \sub. This macro is used to mark a complete sequence of menu selections, including selecting submenus and choosing a specific operation, or any subsequence of such a sequence. The names of individual selections should be separated by occurrences of \sub.

For example, to mark the selection ``Start > Programs'', use this markup:

\menuselection{Start \sub Programs}

When including a selection that includes some trailing indicator, such as the ellipsis some operating systems use to indicate that the command opens a dialog, the indicator should be omitted from the selection name.

Individual selection names within the \menuselection should not be marked using \guilabel since that's implied by using \menuselection.

\sub
Separator for menu selections that include multiple levels. This macro is only defined within the context of the \menuselection macro.
7 Processing Tools

Subsections

- 7.1 External Tools
- 7.2 Internal Tools
- 7.3 Working on Cygwin

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.1 External Tools

Many tools are needed to be able to process the Python documentation if all supported formats are required. This section lists the tools used and when each is required. Consult the Doc/README file to see if there are specific version requirements for any of these.

**dvips**
This program is a typical part of TeX installations. It is used to generate PostScript from the ``device independent`` .dvi files. It is needed for the conversion to PostScript.

**emacs**
Emacs is the kitchen sink of programmers' editors, and a damn fine kitchen sink it is. It also comes with some of the processing needed to support the proper menu structures for Texinfo documents when an info conversion is desired. This is needed for the info conversion. Using xemacs instead of FSF emacs may lead to instability in the conversion, but that's because nobody seems to maintain the Emacs Texinfo code in a portable manner.

**latex**
LaTeX is a large and extensible macro package by Leslie Lamport, based on TeX, a world-class typesetter by Donald Knuth. It is used for the conversion to PostScript, and is needed for the HTML conversion as well (LaTeX2HTML requires one of the intermediate files it creates).

**latex2html**
Probably the longest Perl script anyone ever attempted to maintain. This converts LaTeX documents to HTML documents, and does a pretty reasonable job. It is required for the conversions to HTML and GNU info.

**lynx**
This is a text-mode Web browser which includes an HTML-to-plain text conversion. This is used to convert howto documents to text.

**make**
Just about any version should work for the standard documents, but GNU make is required for the experimental processes in Doc/tools/sgmlconv/,
at least while they're experimental. This is not required for running the
**mkhowto** script.

**makeindex**
This is a standard program for converting LaTeX index data to a formatted
index; it should be included with all LaTeX installations. It is needed for the
PDF and PostScript conversions.

**makeinfo**
GNU **makeinfo** is used to convert Texinfo documents to GNU info files.
Since Texinfo is used as an intermediate format in the info conversion, this
program is needed in that conversion.

**pdflatex**
pdfTeX is a relatively new variant of TeX, and is used to generate the PDF
version of the manuals. It is typically installed as part of most of the large
TeX distributions. **pdflatex** is pdfTeX using the LaTeX format.

**perl**
Perl is required for LaTeX2HTML and one of the scripts used to post-
process LaTeX2HTML output, as well as the HTML-to-Texinfo conversion.
This is required for the HTML and GNU info conversions.

**python**
Python is used for many of the scripts in the Doc/tools/ directory; it is
required for all conversions. This shouldn't be a problem if you're interested
in writing documentation for Python!
7.2 Internal Tools

This section describes the various scripts that are used to implement various stages of document processing or to orchestrate entire build sequences. Most of these tools are only useful in the context of building the standard documentation, but some are more general.

**mkhowto**

This is the primary script used to format third-party documents. It contains all the logic needed to ``get it right.'' The proper way to use this script is to make a symbolic link to it or run it in place; the actual script file must be stored as part of the documentation source tree, though it may be used to format documents outside the tree. Use `mkhowto --help` for a list of command line options.

`mkhowto` can be used for both `howto` and `manual` class documents. It is usually a good idea to always use the latest version of this tool rather than a version from an older source release of Python. It can be used to generate DVI, HTML, PDF, PostScript, and plain text documents. The GNU info and iSilo formats will be supported by this script in some future version.

Use the `--help` option on this script's command line to get a summary of options for this script.

XXX Need more here.
7.3 Working on Cygwin

Installing the required tools under Cygwin under Cygwin can be a little tedious. Most of the required packages can be installed using Cygwin's graphical installer, while netpbm and LaTeX2HTML must be installed from source.

Start with a reasonably modern version of Cygwin. If you haven't upgraded for a few years, now would be a good time.

Using the Cygwin installer, make sure your Cygwin installation includes Perl, Python, and the TeX packages. Perl and Python are located under the Interpreters heading. The TeX packages are located under the Text heading, and are named tetex-* . To ensure that all required packages are available, install every tetex package, except tetex-x11. (There may be a more minimal set, but I've not spent time trying to minimize the installation.)

The netpbm package is used by LaTeX2HTML, and must be installed before LaTeX2HTML can be successfully installed, even though its features will not be used for most Python documentation. References to download locations are located in the netpbm README. Install from the latest stable source distribution according to the instructions. (Note that binary packages of netpbm are sometimes available, but these may not work correctly with LaTeX2HTML.)

LaTeX2HTML can be installed from the source archive, but only after munging one of the files in the distribution. Download the source archive from the LaTeX2HTML website http://www.latex2html.org/ (or one of the many alternate sites) and unpack it to a build directory. In the top level of this build directory there will be a file named L2hos.pm. Open L2hos.pm in an editor, and near the bottom of the file replace the text $^O with the text 'unix'. Proceed using this command to build and install the software:

% ./configure && make install

You should now be able to build at least the DVI, HTML, PDF, and PostScript versions of the formatted documentation.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
Previous: 7.3 Working on Cygwin  Up: Documenting Python  Next: 9 Future Directions
8 Including Graphics

The standard documentation included with Python makes no use of diagrams or images; this is intentional. The outside tools used to format the documentation have not always been suited to working with graphics. As the tools have evolved and been improved by their maintainers, support for graphics has improved.

The internal tools, starting with the mkhowto script, do not provide any direct support for graphics. However, mkhowto will not interfere with graphics support in the external tools.

Experience using graphics together with these tools and the howto and manual document classes is not extensive, but has been known to work. The basic approach is this:

1. Create the image or graphic using your favorite application.

2. Convert the image to a format supported by the conversion to your desired output format. If you want to generate HTML or PostScript, you can convert the image or graphic to encapsulated PostScript (a .eps file); LaTeX2HTML can convert that to a .gif file; it may be possible to provide a .gif file directly. If you want to generate PDF, you need to provide an `"encapsulated" PDF file. This can be generated from encapsulated PostScript using the epstopdf tool provided with the tEx distribution on Linux and UNIX.

3. In your document, add this line to `"import" the general graphics support package graphicx:

   \usepackage{graphicx}

4. Where you want to include your graphic or image, include markup similar to this:

   \begin{figure}
     \centering
     \includegraphics[width=5in]{myimage}
     \caption{Description of my image}
   \end{figure}
In particular, note for the `\includegraphics` macro that no file extension is provided. If you're only interested in one target format, you can include the extension of the appropriate input file, but to allow support for multiple formats, omitting the extension makes life easier.

5. Run `mkhowto` normally.

If you're working on systems which support some sort of `make` facility, you can use that to ensure the intermediate graphic formats are kept up to date. This example shows a Makefile used to format a document containing a diagram created using the `dia` application:

```
default: pdf
all: html pdf ps

html: mydoc/mydoc.html
pdf: mydoc.pdf
ps: mydoc.ps

mydoc/mydoc.html: mydoc.tex mygraphic.eps
         mkhowto --html $<

mydoc.pdf: mydoc.tex mygraphic.pdf
         mkhowto --pdf $<

mydoc.ps: mydoc.tex mygraphic.eps
         mkhowto --postscript $<

.SUFFIXES: .dia .eps .pdf
.dia.eps:
         dia --nosplash --export @$<

.eps.pdf:
         epstopdf $<
```
9 Future Directions

The history of the Python documentation is full of changes, most of which have been fairly small and evolutionary. There has been a great deal of discussion about making large changes in the markup languages and tools used to process the documentation. This section deals with the nature of the changes and what appears to be the most likely path of future development.

Subsections

- 9.1 Structured Documentation
- 9.2 Discussion Forums

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
9.1 Structured Documentation

Most of the small changes to the LaTeX markup have been made with an eye to divorcing the markup from the presentation, making both a bit more maintainable. Over the course of 1998, a large number of changes were made with exactly this in mind; previously, changes had been made but in a less systematic manner and with more concern for not needing to update the existing content. The result has been a highly structured and semantically loaded markup language implemented in LaTeX. With almost no basic TeX or LaTeX markup in use, however, the markup syntax is about the only evidence of LaTeX in the actual document sources.

One side effect of this is that while we've been able to use standard `\``engines" for manipulating the documents, such as LaTeX and LaTeX2HTML, most of the actual transformations have been created specifically for Python. The LaTeX document classes and LaTeX2HTML support are both complete implementations of the specific markup designed for these documents.

Combining highly customized markup with the somewhat esoteric systems used to process the documents leads us to ask some questions: Can we do this more easily? and, Can we do this better? After a great deal of discussion with the community, we have determined that actively pursuing modern structured documentation systems is worth some investment of time.

There appear to be two real contenders in this arena: the Standard General Markup Language (SGML), and the Extensible Markup Language (XML). Both of these standards have advantages and disadvantages, and many advantages are shared.

SGML offers advantages which may appeal most to authors, especially those using ordinary text editors. There are also additional abilities to define content models. A number of high-quality tools with demonstrated maturity are available, but most are not free; for those which are, portability issues remain a problem.

The advantages of XML include the availability of a large number of evolving tools. Unfortunately, many of the associated standards are still evolving, and the tools will have to follow along. This means that developing a robust tool set that
uses more than the basic XML 1.0 recommendation is not possible in the short term. The promised availability of a wide variety of high-quality tools which support some of the most important related standards is not immediate. Many tools are likely to be free, and the portability issues of those which are, are not expected to be significant.

It turns out that converting to an XML or SGML system holds promise for translators as well; how much can be done to ease the burden on translators remains to be seen, and may have some impact on the schema and specific technologies used.

XXX Eventual migration to XML.

The documentation will be moved to XML in the future, and tools are being written which will convert the documentation from the current format to something close to a finished version, to the extent that the desired information is already present in the documentation. Some XSLT stylesheets have been started for presenting a preliminary XML version as HTML, but the results are fairly rough.

The timeframe for the conversion is not clear since there doesn't seem to be much time available to work on this, but the apparent benefits are growing more substantial at a moderately rapid pace.
9.2 Discussion Forums

Discussion of the future of the Python documentation and related topics takes place in the Documentation Special Interest Group, or `Doc-SIG." Information on the group, including mailing list archives and subscription information, is available at [http://www.python.org/sigs/doc-sig/](http://www.python.org/sigs/doc-sig/). The SIG is open to all interested parties.

Comments and bug reports on the standard documents should be sent to docs@python.org. This may include comments about formatting, content, grammatical and spelling errors, or this document. You can also send comments on this document directly to the author at fdrake@acm.org.
Abstract:
This document describes the Python Distribution Utilities (``Distutils'') from the
end-user's point-of-view, describing how to extend the capabilities of a standard
Python installation by building and installing third-party Python modules and
extensions.
6 Building Extensions: Tips and Tricks
   6.1 Tweaking compiler/linker flags
   6.2 Using non-Microsoft compilers on Windows
      ■ 6.2.1 Borland C++
      ■ 6.2.2 GNU C / Cygwin / MinGW

About this document...
Subsections

- 1.1 Best case: trivial installation
- 1.2 The new standard: Distutils
1 Introduction

Although Python's extensive standard library covers many programming needs, there often comes a time when you need to add some new functionality to your Python installation in the form of third-party modules. This might be necessary to support your own programming, or to support an application that you want to use and that happens to be written in Python.

In the past, there has been little support for adding third-party modules to an existing Python installation. With the introduction of the Python Distribution Utilities (Distutils for short) in Python 2.0, this changed.

This document is aimed primarily at the people who need to install third-party Python modules: end-users and system administrators who just need to get some Python application running, and existing Python programmers who want to add some new goodies to their toolbox. You don't need to know Python to read this document; there will be some brief forays into using Python's interactive mode to explore your installation, but that's it. If you're looking for information on how to distribute your own Python modules so that others may use them, see the \textit{Distributing Python Modules} manual.
1.1 Best case: trivial installation

In the best case, someone will have prepared a special version of the module distribution you want to install that is targeted specifically at your platform and is installed just like any other software on your platform. For example, the module developer might make an executable installer available for Windows users, an RPM package for users of RPM-based Linux systems (Red Hat, SuSE, Mandrake, and many others), a Debian package for users of Debian-based Linux systems, and so forth.

In that case, you would download the installer appropriate to your platform and do the obvious thing with it: run it if it's an executable installer, `rpm --install` it if it's an RPM, etc. You don't need to run Python or a setup script, you don't need to compile anything--you might not even need to read any instructions (although it's always a good idea to do so anyways).

Of course, things will not always be that easy. You might be interested in a module distribution that doesn't have an easy-to-use installer for your platform. In that case, you'll have to start with the source distribution released by the module's author/maintainer. Installing from a source distribution is not too hard, as long as the modules are packaged in the standard way. The bulk of this document is about building and installing modules from standard source distributions.
1.2 The new standard: Distutils

If you download a module source distribution, you can tell pretty quickly if it was packaged and distributed in the standard way, i.e. using the Distutils. First, the distribution's name and version number will be featured prominently in the name of the downloaded archive, e.g. foo-1.0.tar.gz or widget-0.9.7.zip. Next, the archive will unpack into a similarly-named directory: foo-1.0 or widget-0.9.7. Additionally, the distribution will contain a setup script setup.py, and a file named README.txt or possibly just README, which should explain that building and installing the module distribution is a simple matter of running

    python setup.py install

If all these things are true, then you already know how to build and install the modules you've just downloaded: Run the command above. Unless you need to install things in a non-standard way or customize the build process, you don't really need this manual. Or rather, the above command is everything you need to get out of this manual.
Subsections

- 2.1 Platform variations
- 2.2 Splitting the job up
- 2.3 How building works
- 2.4 How installation works
2 Standard Build and Install

As described in section 1.2, building and installing a module distribution using the Distutils is usually one simple command:

```python
python setup.py install
```

On UNIX, you'd run this command from a shell prompt; on Windows, you have to open a command prompt window (``DOS box'') and do it there; on Mac OS X, you open a Terminal window to get a shell prompt.
2.1 Platform variations

You should always run the setup command from the distribution root directory, i.e. the top-level subdirectory that the module source distribution unpacks into. For example, if you've just downloaded a module source distribution foo-1.0.tar.gz onto a UNIX system, the normal thing to do is:

```
    gunzip -c foo-1.0.tar.gz | tar xf -     # unpacks into directory foo-
    cd foo-1.0
    python setup.py install
```

On Windows, you'd probably download foo-1.0.zip. If you downloaded the archive file to C:\Temp, then it would unpack into C:\Temp\foo-1.0; you can use either a archive manipulator with a graphical user interface (such as WinZip) or a command-line tool (such as `unzip` or `pkunzip`) to unpack the archive. Then, open a command prompt window (``DOS box''), and run:

```
    cd c:\Temp\foo-1.0
    python setup.py install
```
2.2 Splitting the job up

Running `setup.py install` builds and installs all modules in one run. If you prefer to work incrementally--especially useful if you want to customize the build process, or if things are going wrong--you can use the setup script to do one thing at a time. This is particularly helpful when the build and install will be done by different users--for example, you might want to build a module distribution and hand it off to a system administrator for installation (or do it yourself, with super-user privileges).

For example, you can build everything in one step, and then install everything in a second step, by invoking the setup script twice:

```bash
  python setup.py build
  python setup.py install
```

If you do this, you will notice that running the `install` command first runs the `build` command, which--in this case--quickly notices that it has nothing to do, since everything in the `build` directory is up-to-date.

You may not need this ability to break things down often if all you do is install modules downloaded off the 'net, but it's very handy for more advanced tasks. If you get into distributing your own Python modules and extensions, you'll run lots of individual Distutils commands on their own.
2.3 How building works

As implied above, the build command is responsible for putting the files to install into a build directory. By default, this is build under the distribution root; if you're excessively concerned with speed, or want to keep the source tree pristine, you can change the build directory with the --build-base option. For example:

```bash
    python setup.py build --build-base=/tmp/pybuild/foo-1.0
```

(Or you could do this permanently with a directive in your system or personal Distutils configuration file; see section 5.) Normally, this isn't necessary.

The default layout for the build tree is as follows:

```plaintext
    --- build/ --- lib/
    or
    --- build/ --- lib.<plat>/
        temp.<plat>/
```

where `<plat>` expands to a brief description of the current OS/hardware platform and Python version. The first form, with just a lib directory, is used for "pure module distributions"--that is, module distributions that include only pure Python modules. If a module distribution contains any extensions (modules written in C/C++), then the second form, with two `<plat>` directories, is used. In that case, the temp.<plat> directory holds temporary files generated by the compile/link process that don't actually get installed. In either case, the lib (or lib.<plat>) directory contains all Python modules (pure Python and extensions) that will be installed.

In the future, more directories will be added to handle Python scripts, documentation, binary executables, and whatever else is needed to handle the job of installing Python modules and applications.
2.4 How installation works

After the build command runs (whether you run it explicitly, or the install command does it for you), the work of the install command is relatively simple: all it has to do is copy everything under build/lib (or build/lib.plat) to your chosen installation directory.

If you don't choose an installation directory--i.e., if you just run setup.py install--then the install command installs to the standard location for third-party Python modules. This location varies by platform and by how you built/installed Python itself. On UNIX (and Mac OS X, which is also UNIX-based), it also depends on whether the module distribution being installed is pure Python or contains extensions (``non-pure'')):

<table>
<thead>
<tr>
<th>Platform</th>
<th>Standard installation location</th>
<th>Default value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX (pure)</td>
<td>prefix/lib/python2.5/site-packages</td>
<td>/usr/local/lib/python2.5/site-packages</td>
<td>(1)</td>
</tr>
<tr>
<td>UNIX (non-pure)</td>
<td>exec-prefix/lib/python2.5/site-packages</td>
<td>/usr/local/lib/python2.5/site-packages</td>
<td>(1)</td>
</tr>
<tr>
<td>Windows</td>
<td>prefix</td>
<td>C:\Python</td>
<td>(2)</td>
</tr>
</tbody>
</table>

Notes:

(1) Most Linux distributions include Python as a standard part of the system, so prefix and exec-prefix are usually both /usr on Linux. If you build Python yourself on Linux (or any UNIX-like system), the default prefix and exec-prefix are /usr/local.

(2) The default installation directory on Windows was C:\Program Files\Python under Python 1.6a1, 1.5.2, and earlier.

prefix and exec-prefix stand for the directories that Python is installed to, and where it finds its libraries at run-time. They are always the same under Windows, and very often the same under UNIX and Mac OS X. You can find out
what your Python installation uses for *prefix* and *exec-prefix* by running Python in interactive mode and typing a few simple commands. Under *Unix*, just type `python` at the shell prompt. Under Windows, choose `Start > Programs > Python 2.5 > Python (command line)`. Once the interpreter is started, you type Python code at the prompt. For example, on my Linux system, I type the three Python statements shown below, and get the output as shown, to find out my *prefix* and *exec-prefix*:

```python
Python 2.4 (#26, Aug 7 2004, 17:19:02)
Type "help", "copyright", "credits" or "license" for more informatio
>>> import sys
>>> sys.prefix
'/usr'
>>> sys.exec_prefix
'/usr'
```

If you don't want to install modules to the standard location, or if you don't have permission to write there, then you need to read about alternate installations in section 3. If you want to customize your installation directories more heavily, see section 4 on custom installations.
Subsections

- 3.1 Alternate installation: the home scheme
- 3.2 Alternate installation: UNIX (the prefix scheme)
- 3.3 Alternate installation: Windows (the prefix scheme)
3 Alternate Installation

Often, it is necessary or desirable to install modules to a location other than the standard location for third-party Python modules. For example, on a UNIX system you might not have permission to write to the standard third-party module directory. Or you might wish to try out a module before making it a standard part of your local Python installation. This is especially true when upgrading a distribution already present: you want to make sure your existing base of scripts still works with the new version before actually upgrading.

The Distutils install command is designed to make installing module distributions to an alternate location simple and painless. The basic idea is that you supply a base directory for the installation, and the install command picks a set of directories (called an installation scheme) under this base directory in which to install files. The details differ across platforms, so read whichever of the following sections applies to you.
3.1 Alternate installation: the home scheme

The idea behind the ``home scheme'' is that you build and maintain a personal stash of Python modules. This scheme's name is derived from the idea of a ``home'' directory on UNIX, since it's not unusual for a UNIX user to make their home directory have a layout similar to /usr/ or /usr/local/. This scheme can be used by anyone, regardless of the operating system their installing for.

Installing a new module distribution is as simple as

```
python setup.py install --home=<dir>
```

where you can supply any directory you like for the --home option. On UNIX, lazy typists can just type a tilde (~); the install command will expand this to your home directory:

```
python setup.py install --home=~
```

The --home option defines the installation base directory. Files are installed to the following directories under the installation base as follows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation Directory</th>
<th>Override option</th>
</tr>
</thead>
<tbody>
<tr>
<td>pure module distribution</td>
<td>home/lib/python</td>
<td>--install-purelib</td>
</tr>
<tr>
<td>non-pure module distribution</td>
<td>home/lib/python</td>
<td>--install-platlib</td>
</tr>
<tr>
<td>scripts</td>
<td>home/bin</td>
<td>--install-scripts</td>
</tr>
<tr>
<td>data</td>
<td>home/share</td>
<td>--install-data</td>
</tr>
</tbody>
</table>

Changed in version 2.4: The --home option used to be supported only on UNIX.
3.2 Alternate installation: UNIX (the prefix scheme)

The ``prefix scheme'' is useful when you wish to use one Python installation to perform the build/install (i.e., to run the setup script), but install modules into the third-party module directory of a different Python installation (or something that looks like a different Python installation). If this sounds a trifle unusual, it is—that's why the ``home scheme'' comes first. However, there are at least two known cases where the prefix scheme will be useful.

First, consider that many Linux distributions put Python in /usr, rather than the more traditional /usr/local. This is entirely appropriate, since in those cases Python is part of ``the system'' rather than a local add-on. However, if you are installing Python modules from source, you probably want them to go in /usr/local/lib/python2.X rather than /usr/lib/python2.X. This can be done with

```
/usr/bin/python setup.py install --prefix=/usr/local
```

Another possibility is a network filesystem where the name used to write to a remote directory is different from the name used to read it: for example, the Python interpreter accessed as /usr/local/bin/python might search for modules in /usr/local/lib/python2.X, but those modules would have to be installed to, say, /mnt/@server/export/lib/python2.X. This could be done with

```
/usr/local/bin/python setup.py install --prefix=/mnt/@server/export
```

In either case, the --prefix option defines the installation base, and the --exec-prefix option defines the platform-specific installation base, which is used for platform-specific files. (Currently, this just means non-pure module distributions, but could be expanded to C libraries, binary executables, etc.) If --exec-prefix is not supplied, it defaults to --prefix. Files are installed as follows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation Directory</th>
<th>Override option</th>
</tr>
</thead>
<tbody>
<tr>
<td>pure module distribution</td>
<td>prefix/lib/python2.X/site-packages</td>
<td>--install-purelib</td>
</tr>
<tr>
<td>non-pure module distribution</td>
<td>exec-prefix/lib/python2.X/site-packages</td>
<td>--install-platlib</td>
</tr>
<tr>
<td>scripts</td>
<td>prefix/bin</td>
<td>--install-scripts</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>data</td>
<td>prefix/share</td>
<td>--install-data</td>
</tr>
</tbody>
</table>

There is no requirement that `--prefix` or `--exec-prefix` actually point to an alternate Python installation; if the directories listed above do not already exist, they are created at installation time.

Incidentally, the real reason the prefix scheme is important is simply that a standard UNIX installation uses the prefix scheme, but with `--prefix` and `--exec-prefix` supplied by Python itself as `sys.prefix` and `sys.exec_prefix`. Thus, you might think you'll never use the prefix scheme, but every time you run `python setup.py install` without any other options, you're using it.

Note that installing extensions to an alternate Python installation has no effect on how those extensions are built: in particular, the Python header files (`Python.h` and friends) installed with the Python interpreter used to run the setup script will be used in compiling extensions. It is your responsibility to ensure that the interpreter used to run extensions installed in this way is compatible with the interpreter used to build them. The best way to do this is to ensure that the two interpreters are the same version of Python (possibly different builds, or possibly copies of the same build). (Of course, if your `--prefix` and `--exec-prefix` don't even point to an alternate Python installation, this is immaterial.)
3.3 Alternate installation: Windows (the prefix scheme)

Windows has no concept of a user's home directory, and since the standard Python installation under Windows is simpler than under UNIX, the `--prefix` option has traditionally been used to install additional packages in separate locations on Windows.

```
python setup.py install --prefix="\Temp\Python"
```

to install modules to the `\Temp\Python` directory on the current drive.

The installation base is defined by the `--prefix` option; the `--exec-prefix` option is not supported under Windows. Files are installed as follows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation Directory</th>
<th>Override option</th>
</tr>
</thead>
<tbody>
<tr>
<td>pure module distribution</td>
<td><code>prefix</code></td>
<td><code>--install-purelib</code></td>
</tr>
<tr>
<td>non-pure module distribution</td>
<td><code>prefix</code></td>
<td><code>--install-platlib</code></td>
</tr>
<tr>
<td>scripts</td>
<td><code>prefix\Scripts</code></td>
<td><code>--install-scripts</code></td>
</tr>
<tr>
<td>data</td>
<td><code>prefix\Data</code></td>
<td><code>--install-data</code></td>
</tr>
</tbody>
</table>

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
Subsections

- 4.1 Modifying Python's Search Path
4 Custom Installation

Sometimes, the alternate installation schemes described in section 3 just don't do what you want. You might want to tweak just one or two directories while keeping everything under the same base directory, or you might want to completely redefine the installation scheme. In either case, you're creating a custom installation scheme.

You probably noticed the column of "override options" in the tables describing the alternate installation schemes above. Those options are how you define a custom installation scheme. These override options can be relative, absolute, or explicitly defined in terms of one of the installation base directories. (There are two installation base directories, and they are normally the same--they only differ when you use the UNIX "prefix scheme" and supply different --prefix and --exec-prefix options.)

For example, say you're installing a module distribution to your home directory under UNIX--but you want scripts to go in ~/scripts rather than ~/bin. As you might expect, you can override this directory with the --install-scripts option; in this case, it makes most sense to supply a relative path, which will be interpreted relative to the installation base directory (your home directory, in this case):

```
python setup.py install --home=~ --install-scripts=scripts
```

Another UNIX example: suppose your Python installation was built and installed with a prefix of /usr/local/python, so under a standard installation scripts will wind up in /usr/local/python/bin. If you want them in /usr/local/bin instead, you would supply this absolute directory for the --install-scripts option:

```
python setup.py install --install-scripts=/usr/local/bin
```

(This performs an installation using the "prefix scheme," where the prefix is whatever your Python interpreter was installed with-- /usr/local/python in this case.)

If you maintain Python on Windows, you might want third-party modules to live in a subdirectory of prefix, rather than right in prefix itself. This is almost as easy as customizing the script installation directory--you just have to remember
that there are two types of modules to worry about, pure modules and non-pure modules (i.e., modules from a non-pure distribution). For example:

```
python setup.py install --install-purelib=Site --install-platlib=Site
```

The specified installation directories are relative to `prefix`. Of course, you also have to ensure that these directories are in Python's module search path, such as by putting a `.pth` file in `prefix`. See section 4.1 to find out how to modify Python's search path.

If you want to define an entire installation scheme, you just have to supply all of the installation directory options. The recommended way to do this is to supply relative paths; for example, if you want to maintain all Python module-related files under `python` in your home directory, and you want a separate directory for each platform that you use your home directory from, you might define the following installation scheme:

```
python setup.py install --home=~
   --install-purelib=python/lib 
   --install-platlib=python/lib.$PLAT 
   --install-scripts=python/scripts
   --install-data=python/data
```

or, equivalently,

```
python setup.py install --home=~/python
   --install-purelib=lib 
   --install-platlib='lib.$PLAT' 
   --install-scripts=scripts
   --install-data=data
```

$PLAT is not (necessarily) an environment variable--it will be expanded by the Distutils as it parses your command line options, just as it does when parsing your configuration file(s).

Obviously, specifying the entire installation scheme every time you install a new module distribution would be very tedious. Thus, you can put these options into your Distutils config file (see section 5):

```
[install]
install-base=$HOME
install-purelib=python/lib
install-platlib=python/lib.$PLAT
install-scripts=python/scripts
```
install-data=python/data

or, equivalently,

[install]
install-base=$HOME/python
install-purelib=lib
install-platlib=lib.$PLAT
install-scripts=scripts
install-data=data

Note that these two are not equivalent if you supply a different installation base directory when you run the setup script. For example,

python setup.py --install-base=/tmp

would install pure modules to /tmp/python/lib in the first case, and to /tmp/lib in the second case. (For the second case, you probably want to supply an installation base of /tmp/python.)

You probably noticed the use of $HOME and $PLAT in the sample configuration file input. These are Distutils configuration variables, which bear a strong resemblance to environment variables. In fact, you can use environment variables in config files on platforms that have such a notion but the Distutils additionally define a few extra variables that may not be in your environment, such as $PLAT. (And of course, on systems that don't have environment variables, such as Mac OS 9, the configuration variables supplied by the Distutils are the only ones you can use.) See section 5 for details.
4.1 Modifying Python's Search Path

When the Python interpreter executes an import statement, it searches for both Python code and extension modules along a search path. A default value for the path is configured into the Python binary when the interpreter is built. You can determine the path by importing the sys module and printing the value of sys.path.

```
$ python
[GCC 2.96 20000731 (Red Hat Linux 7.3 2.96-112)] on linux2
Type `''help'', `''copyright'', `''credits'' or `''license'' for more in
>>> import sys
>>> sys.path
['', '/usr/local/lib/python2.3', '/usr/local/lib/python2.3/plat-linux2'
'/usr/local/lib/python2.3/lib-tk', '/usr/local/lib/python2.3/lib-dynload'
'/usr/local/lib/python2.3/site-packages']
>>> 
```

The null string in sys.path represents the current working directory.

The expected convention for locally installed packages is to put them in the .../site-packages/ directory, but you may want to install Python modules into some arbitrary directory. For example, your site may have a convention of keeping all software related to the web server under /www. Add-on Python modules might then belong in /www/python, and in order to import them, this directory must be added to sys.path. There are several different ways to add the directory.

The most convenient way is to add a path configuration file to a directory that's already on Python's path, usually to the .../site-packages/ directory. Path configuration files have an extension of .pth, and each line must contain a single path that will be appended to sys.path. (Because the new paths are appended to sys.path, modules in the added directories will not override standard modules. This means you can't use this mechanism for installing fixed versions of standard modules.)

Paths can be absolute or relative, in which case they're relative to the directory containing the .pth file. Any directories added to the search path will be scanned in turn for .pth files. See site module documentation for more information.
A slightly less convenient way is to edit the site.py file in Python's standard library, and modify sys.path. site.py is automatically imported when the Python interpreter is executed, unless the -S switch is supplied to suppress this behaviour. So you could simply edit site.py and add two lines to it:

```python
import sys
sys.path.append('/www/python/)
```

However, if you reinstall the same major version of Python (perhaps when upgrading from 2.2 to 2.2.2, for example) site.py will be overwritten by the stock version. You'd have to remember that it was modified and save a copy before doing the installation.

There are two environment variables that can modify sys.path. PYTHONHOME sets an alternate value for the prefix of the Python installation. For example, if PYTHONHOME is set to "/www/python", the search path will be set to ['', '/www/python/lib/python2.5/', '/www/python/lib/python2.5/plat-linux2', ...].

The PYTHONPATH variable can be set to a list of paths that will be added to the beginning of sys.path. For example, if PYTHONPATH is set to "/www/python:/opt/py", the search path will begin with ['/www/python', '/opt/py']. (Note that directories must exist in order to be added to sys.path; the site module removes paths that don't exist.)

Finally, sys.path is just a regular Python list, so any Python application can modify it by adding or removing entries.
Subsections

- 5.1 Location and names of config files
- 5.2 Syntax of config files
As mentioned above, you can use Distutils configuration files to record personal or site preferences for any Distutils options. That is, any option to any command can be stored in one of two or three (depending on your platform) configuration files, which will be consulted before the command-line is parsed. This means that configuration files will override default values, and the command-line will in turn override configuration files. Furthermore, if multiple configuration files apply, values from ``earlier'' files are overridden by ``later'' files.
5.1 Location and names of config files

The names and locations of the configuration files vary slightly across platforms. On UNIX and Mac OS X, the three configuration files (in the order they are processed) are:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Location and filename</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>prefix/lib/pythonver/distutils/distutils.cfg</td>
<td>(1)</td>
</tr>
<tr>
<td>personal</td>
<td>$HOME/.pydistutils.cfg</td>
<td>(2)</td>
</tr>
<tr>
<td>local</td>
<td>setup.cfg</td>
<td>(3)</td>
</tr>
</tbody>
</table>

And on Windows, the configuration files are:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Location and filename</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>prefix\Lib\distutils\distutils.cfg</td>
<td>(4)</td>
</tr>
<tr>
<td>personal</td>
<td>%HOME%\pydistutils.cfg</td>
<td>(5)</td>
</tr>
<tr>
<td>local</td>
<td>setup.cfg</td>
<td>(3)</td>
</tr>
</tbody>
</table>

Notes:

(1) Strictly speaking, the system-wide configuration file lives in the directory where the Distutils are installed; under Python 1.6 and later on UNIX, this is as shown. For Python 1.5.2, the Distutils will normally be installed to prefix/lib/python1.5/site-packages/distutils, so the system configuration file should be put there under Python 1.5.2.

(2) On UNIX, if the HOME environment variable is not defined, the user's home directory will be determined with the getpwuid() function from the standard pwd module.

(3) I.e., in the current directory (usually the location of the setup script).

(4) (See also note (1).) Under Python 1.6 and later, Python's default "installation prefix" is C:\Python, so the system configuration file is normally C:\Python\Lib\distutils\distutils.cfg. Under Python 1.5.2, the
default prefix was *C:\Program Files\Python*, and the Distutils were not part of the standard library--so the system configuration file would be *C:\Program Files\Python\distutils\distutils.cfg* in a standard Python 1.5.2 installation under Windows.

(5) On Windows, if the HOME environment variable is not defined, no personal configuration file will be found or used. (In other words, the Distutils make no attempt to guess your home directory on Windows.)
5.2 Syntax of config files

The Distutils configuration files all have the same syntax. The config files are grouped into sections. There is one section for each Distutils command, plus a global section for global options that affect every command. Each section consists of one option per line, specified as option=value.

For example, the following is a complete config file that just forces all commands to run quietly by default:

```
[global]
verbose=0
```

If this is installed as the system config file, it will affect all processing of any Python module distribution by any user on the current system. If it is installed as your personal config file (on systems that support them), it will affect only module distributions processed by you. And if it is used as the setup.cfg for a particular module distribution, it affects only that distribution.

You could override the default `"build base" directory and make the `build* commands always forcibly rebuild all files with the following:

```
[build]
build-base=blib
force=1
```

which corresponds to the command-line arguments

```
python setup.py build --build-base=blib --force
```

except that including the build command on the command-line means that command will be run. Including a particular command in config files has no such implication; it only means that if the command is run, the options in the config file will apply. (Or if other commands that derive values from it are run, they will use the values in the config file.)

You can find out the complete list of options for any command using the --help option, e.g.:

```
python setup.py build --help
```
and you can find out the complete list of global options by using --help without a command:

    python setup.py --help

See also the ``Reference'' section of the ``Distributing Python Modules'' manual.
Subsections

- **6.1 Tweaking compiler/linker flags**
- **6.2 Using non-Microsoft compilers on Windows**
  - 6.2.1 Borland C++
  - 6.2.2 GNU C / Cygwin / MinGW
6 Building Extensions: Tips and Tricks

Whenever possible, the Distutils try to use the configuration information made available by the Python interpreter used to run the setup.py script. For example, the same compiler and linker flags used to compile Python will also be used for compiling extensions. Usually this will work well, but in complicated situations this might be inappropriate. This section discusses how to override the usual Distutils behaviour.
### 6.1 Tweaking compiler/linker flags

Compiling a Python extension written in C or C++ will sometimes require specifying custom flags for the compiler and linker in order to use a particular library or produce a special kind of object code. This is especially true if the extension hasn't been tested on your platform, or if you're trying to cross-compile Python.

In the most general case, the extension author might have foreseen that compiling the extensions would be complicated, and provided a `Setup` file for you to edit. This will likely only be done if the module distribution contains many separate extension modules, or if they often require elaborate sets of compiler flags in order to work.

A `Setup` file, if present, is parsed in order to get a list of extensions to build. Each line in a `Setup` describes a single module. Lines have the following structure:

```
module ... [sourcefile ...] [cpparg ...] [library ...]
```

Let's examine each of the fields in turn.

- **module** is the name of the extension module to be built, and should be a valid Python identifier. You can't just change this in order to rename a module (edits to the source code would also be needed), so this should be left alone.

- **sourcefile** is anything that's likely to be a source code file, at least judging by the filename. Filenames ending in `.c` are assumed to be written in C, filenames ending in `.C`, `.cc`, and `.c++` are assumed to be C++, and filenames ending in `.m` or `.mm` are assumed to be in Objective C.

- **cpparg** is an argument for the C preprocessor, and is anything starting with `-I`, `-D`, `-U` or `-C`.

- **library** is anything ending in `.a` or beginning with `-l` or `-L`. 

If a particular platform requires a special library on your platform, you can add it by editing the Setup file and running `python setup.py build`. For example, if the module defined by the line

```python
foo foomodule.c
```

must be linked with the math library `libm.a` on your platform, simply add `-lm` to the line:

```bash
foo foomodule.c -lm
```

Arbitrary switches intended for the compiler or the linker can be supplied with the `-Xcompiler arg` and `-Xlinker arg` options:

```bash
foo foomodule.c -Xcompiler -o32 -Xlinker -shared -lm
```

The next option after `-Xcompiler` and `-Xlinker` will be appended to the proper command line, so in the above example the compiler will be passed the `-o32` option, and the linker will be passed `-shared`. If a compiler option requires an argument, you'll have to supply multiple `-Xcompiler` options; for example, to pass `-x c++` the Setup file would have to contain `-Xcompiler -x - Xcompiler c++`.

Compiler flags can also be supplied through setting the `CFLAGS` environment variable. If set, the contents of `CFLAGS` will be added to the compiler flags specified in the Setup file.
6.2 Using non-Microsoft compilers on Windows

6.2.1 Borland C++

This subsection describes the necessary steps to use Distutils with the Borland C++ compiler version 5.5.

First you have to know that Borland's object file format (OMF) is different from the format used by the Python version you can download from the Python or ActiveState Web site. (Python is built with Microsoft Visual C++, which uses COFF as the object file format.) For this reason you have to convert Python's library python25.lib into the Borland format. You can do this as follows:

```
coff2omf python25.lib python25_bcpp.lib
```

The `coff2omf` program comes with the Borland compiler. The file `python25.lib` is in the `Libs` directory of your Python installation. If your extension uses other libraries (zlib,...) you have to convert them too.

The converted files have to reside in the same directories as the normal libraries.

How does Distutils manage to use these libraries with their changed names? If the extension needs a library (eg. foo) Distutils checks first if it finds a library with suffix _bcpp (eg. foo_bcpp.lib) and then uses this library. In the case it doesn't find such a special library it uses the default name (foo.lib.).

To let Distutils compile your extension with Borland C++ you now have to type:

```
python setup.py build --compiler=bcpp
```

If you want to use the Borland C++ compiler as the default, you could specify this in your personal or system-wide configuration file for Distutils (see section 5.)

See Also:

[C++Builder Compiler]
6.2.2 GNU C / Cygwin / MinGW

These instructions only apply if you're using a version of Python prior to 2.4.1 with a MinGW prior to 3.0.0 (with binutils-2.13.90-20030111-1).

This section describes the necessary steps to use Distutils with the GNU C/C++ compilers in their Cygwin and MinGW distributions. For a Python interpreter that was built with Cygwin, everything should work without any of these following steps.

These compilers require some special libraries. This task is more complex than for Borland's C++, because there is no program to convert the library.

First you have to create a list of symbols which the Python DLL exports. (You can find a good program for this task at http://starship.python.net/crew/kernr/mingw32/Notes.html, see at PExports 0.42h there.)

```
pexports python25.dll >python25.def
```

The location of an installed python25.dll will depend on the installation options and the version and language of Windows. In a ``just for me'' installation, it will appear in the root of the installation directory. In a shared installation, it will be located in the system directory.

Then you can create from these information an import library for gcc.

```
/cygwin/bin/dlltool --dllname python25.dll --def python25.def --outp
```

The resulting library has to be placed in the same directory as python25.lib. (Should be the libs directory under your Python installation directory.)
If your extension uses other libraries (zlib,...) you might have to convert them too. The converted files have to reside in the same directories as the normal libraries do.

To let Distutils compile your extension with Cygwin you now have to type

```
python setup.py build --compiler=cygwin
```

and for Cygwin in no-cygwin mode\(^3\) or for MinGW type:

```
python setup.py build --compiler=mingw32
```

If you want to use any of these options/compilers as default, you should consider to write it in your personal or system-wide configuration file for Distutils (see section \(^5\)).

---

**See Also:**

*Building Python modules on MS Windows platform with MinGW*

Information about building the required libraries for the MinGW environment.

*http://pyopengl.sourceforge.net/ftp/win32-stuff/*

Converted import libraries in Cygwin/MinGW and Borland format, and a script to create the registry entries needed for Distutils to locate the built Python.

---

**Footnotes**

...foo.lib.)\(^1\)

This also means you could replace all existing COFF-libraries with OMF-libraries of the same name.

... distributions.\(^2\)

for more information

... mode

Then you have no POSIX emulation available, but you also don't need cygwin1.dll.
About the Python Documentation

The Python documentation was originally written by Guido van Rossum, but has increasingly become a community effort over the past several years. This growing collection of documents is available in several formats, including typeset versions in PDF and PostScript for printing, from the Python Web site.

A list of contributors is available.
Comments and Questions

General comments and questions regarding this document should be sent by email to docs@python.org. If you find specific errors in this document, please report the bug at the Python Bug Tracker at SourceForge. If you are able to provide suggested text, either to replace existing incorrect or unclear material, or additional text to supplement what's already available, we'd appreciate the contribution. There's no need to worry about text markup; our documentation team will gladly take care of that.

Questions regarding how to use the information in this document should be sent to the Python news group, comp.lang.python, or the Python mailing list (which is gated to the newsgroup and carries the same content).

For any of these channels, please be sure not to send HTML email. Thanks.
Abstract:

This document describes the Python Distribution Utilities (``Distutils'') from the module developer's point of view, describing how to use the Distutils to make Python modules and extensions easily available to a wider audience with very little overhead for build/release/install mechanics.
2. Writing the Setup Script
   2.1 Listing whole packages
   2.2 Listing individual modules
   2.3 Describing extension modules
      2.3.1 Extension names and packages
      2.3.2 Extension source files
      2.3.3 Preprocessor options
      2.3.4 Library options
      2.3.5 Other options
   2.4 Relationships between Distributions and Packages
   2.5 Installing Scripts
   2.6 Installing Package Data
   2.7 Installing Additional Files
   2.8 Additional meta-data
   2.9 Debugging the setup script

3. Writing the Setup Configuration File

4. Creating a Source Distribution
   4.1 Specifying the files to distribute
   4.2 Manifest-related options

5. Creating Built Distributions
   5.1 Creating dumb built distributions
   5.2 Creating RPM packages
   5.3 Creating Windows Installers
      5.3.1 The Postinstallation script

6. Registering with the Package Index
   6.1 The .pypirc file

7. Uploading Packages to the Package Index

8. Examples
   8.1 Pure Python distribution (by module)
   8.2 Pure Python distribution (by package)
   8.3 Single extension module

9. Extending Distutils
   9.1 Integrating new commands
   9.2 Adding new distribution types

10. Command Reference
    10.1 Installing modules: the install command family
        10.1.1 install_data
        10.1.2 install_scripts
    10.2 Creating a source distribution: the sdist command
11. API Reference

- 11.1 distutils.core -- Core Distutils functionality
- 11.2 distutils.ccompiler -- CCompiler base class
- 11.3 distutils.unixccompiler -- Unix C Compiler
- 11.4 distutils.msvccompiler -- Microsoft Compiler
- 11.5 distutils.bcppcompiler -- Borland Compiler
- 11.6 distutils.cygwincompiler -- Cygwin Compiler
- 11.7 distutils.emxccompiler -- OS/2 EMX Compiler
- 11.8 distutils.mwerkscompiler -- Metrowerks CodeWarrior support
- 11.9 distutils.archive_util -- Archiving utilities
- 11.10 distutils.dep_util -- Dependency checking
- 11.11 distutils.dir_util -- Directory tree operations
- 11.12 distutils.file_util -- Single file operations
- 11.13 distutils.util -- Miscellaneous other utility functions
- 11.14 distutils.dist -- The Distribution class
- 11.15 distutils.extension -- The Extension class
- 11.16 distutils.debug -- Distutils debug mode
- 11.17 distutils.errors -- Distutils exceptions
- 11.18 distutils.fancy_getopt -- Wrapper around the standard getopt module
- 11.19 distutils.filelist -- The FileList class
- 11.20 distutils.log -- Simple PEP 282-style logging
- 11.21 distutils.spawn -- Spawn a sub-process
- 11.22 distutils.sysconfig -- System configuration information
- 11.23 distutils.text_file -- The TextFile class
- 11.24 distutils.version -- Version number classes
- 11.25 distutils.cmd -- Abstract base class for Distutils commands
- 11.26 distutils.command -- Individual Distutils commands
- 11.27 distutils.command.bdist -- Build a binary installer
- 11.28 distutils.command.bdist_packager -- Abstract base class for packagers
- 11.29 distutils.command.bdist_dumb -- Build a "dumb" installer
- 11.30 distutils.command.bdist_rpm -- Build a binary
distribution as a Redhat RPM and SRPM

- **11.31 distutils.command.bdist_wininst** -- Build a Windows installer
- **11.32 distutils.command.sdist** -- Build a source distribution
- **11.33 distutils.command.build** -- Build all files of a package
- **11.34 distutils.command.build_clib** -- Build any C libraries in a package
- **11.35 distutils.command.build_ext** -- Build any extensions in a package
- **11.36 distutils.command.build_py** -- Build the .py/.pyc files of a package
- **11.37 distutils.command.build_scripts** -- Build the scripts of a package
- **11.38 distutils.command.clean** -- Clean a package build area
- **11.39 distutils.command.config** -- Perform package configuration
- **11.40 distutils.command.install** -- Install a package
- **11.41 distutils.command.install_data** -- Install data files from a package
- **11.42 distutils.command.install_headers** -- Install C/C++ header files from a package
- **11.43 distutils.command.install_lib** -- Install library files from a package
- **11.44 distutils.command.install_scripts** -- Install script files from a package
- **11.45 distutils.command.register** -- Register a module with the Python Package Index
- **11.46 Creating a new Distutils command**

- Module Index
- Index
- About this document...

---

**Distributing Python Modules**

Up: Python Documentation Index  Next: 1. An Introduction to

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1. An Introduction to Distutils

This document covers using the Distutils to distribute your Python modules, concentrating on the role of developer/distributor: if you're looking for information on installing Python modules, you should refer to the Installing Python Modules manual.

Subsections

- 1.1 Concepts & Terminology
- 1.2 A Simple Example
- 1.3 General Python terminology
- 1.4 Distutils-specific terminology

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
1.1 Concepts & Terminology

Using the Distutils is quite simple, both for module developers and for users/administrators installing third-party modules. As a developer, your responsibilities (apart from writing solid, well-documented and well-tested code, of course!) are:

- write a setup script (setup.py by convention)
- (optional) write a setup configuration file
- create a source distribution
- (optional) create one or more built (binary) distributions

Each of these tasks is covered in this document.

Not all module developers have access to a multitude of platforms, so it's not always feasible to expect them to create a multitude of built distributions. It is hoped that a class of intermediaries, called *packagers*, will arise to address this need. Packagers will take source distributions released by module developers, build them on one or more platforms, and release the resulting built distributions. Thus, users on the most popular platforms will be able to install most popular Python module distributions in the most natural way for their platform, without having to run a single setup script or compile a line of code.
1.2 A Simple Example

The setup script is usually quite simple, although since it's written in Python, there are no arbitrary limits to what you can do with it, though you should be careful about putting arbitrarily expensive operations in your setup script. Unlike, say, Autoconf-style configure scripts, the setup script may be run multiple times in the course of building and installing your module distribution.

If all you want to do is distribute a module called `foo`, contained in a file `foo.py`, then your setup script can be as simple as this:

```python
from distutils.core import setup
setup(name='foo',
      version='1.0',
      py_modules=['foo'],
)
```

Some observations:

- most information that you supply to the Distutils is supplied as keyword arguments to the `setup()` function
- those keyword arguments fall into two categories: package metadata (name, version number) and information about what's in the package (a list of pure Python modules, in this case)
- modules are specified by module name, not filename (the same will hold true for packages and extensions)
- it's recommended that you supply a little more metadata, in particular your name, email address and a URL for the project (see section 2 for an example)

To create a source distribution for this module, you would create a setup script, `setup.py`, containing the above code, and run:

```bash
python setup.py sdist
```

which will create an archive file (e.g., tarball on UNIX, ZIP file on Windows) containing your setup script `setup.py`, and your module `foo.py`. The archive file will be named `foo-1.0.tar.gz` (or `.zip`), and will unpack into a directory `foo-1.0`. 
If an end-user wishes to install your foo module, all she has to do is download foo-1.0.tar.gz (or .zip), unpack it, and--from the foo-1.0 directory--run

```
python setup.py install
```

which will ultimately copy foo.py to the appropriate directory for third-party modules in their Python installation.

This simple example demonstrates some fundamental concepts of the Distutils. First, both developers and installers have the same basic user interface, i.e. the setup script. The difference is which Distutils commands they use: the sdist command is almost exclusively for module developers, while install is more often for installers (although most developers will want to install their own code occasionally).

If you want to make things really easy for your users, you can create one or more built distributions for them. For instance, if you are running on a Windows machine, and want to make things easy for other Windows users, you can create an executable installer (the most appropriate type of built distribution for this platform) with the bdist_wininst command. For example:

```
python setup.py bdist_wininst
```

will create an executable installer, foo-1.0.win32.exe, in the current directory.

Other useful built distribution formats are RPM, implemented by the bdist_rpm command, Solaris pkgtool (bdist_pkgtool), and HP-UX swinstall (bdist_sdux). For example, the following command will create an RPM file called foo-1.0.noarch.rpm:

```
python setup.py bdist_rpm
```

(The bdist_rpm command uses the rpm executable, therefore this has to be run on an RPM-based system such as Red Hat Linux, SuSE Linux, or Mandrake Linux.)

You can find out what distribution formats are available at any time by running

```
python setup.py bdist --help-formats
```
Distributing Python Modules

Previous: 1.1 Concepts & Terminology  Up: 1. An Introduction to  Next: 1.3
General Python terminology

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
1.3 General Python terminology

If you're reading this document, you probably have a good idea of what modules, extensions, and so forth are. Nevertheless, just to be sure that everyone is operating from a common starting point, we offer the following glossary of common Python terms:

**module**
the basic unit of code reusability in Python: a block of code imported by some other code. Three types of modules concern us here: pure Python modules, extension modules, and packages.

**pure Python module**
a module written in Python and contained in a single .py file (and possibly associated .pyc and/or .pyo files). Sometimes referred to as a ``pure module.''

**extension module**
a module written in the low-level language of the Python implementation: C/C++ for Python, Java for Jython. Typically contained in a single dynamically loadable pre-compiled file, e.g. a shared object (.so) file for Python extensions on UNIX, a DLL (given the .pyd extension) for Python extensions on Windows, or a Java class file for Jython extensions. (Note that currently, the Distutils only handles C/C++ extensions for Python.)

**package**
a module that contains other modules; typically contained in a directory in the filesystem and distinguished from other directories by the presence of a file __init__.py.

**root package**
the root of the hierarchy of packages. (This isn't really a package, since it doesn't have an __init__.py file. But we have to call it something.) The vast majority of the standard library is in the root package, as are many small, standalone third-party modules that don't belong to a larger module collection. Unlike regular packages, modules in the root package can be found in many directories: in fact, every directory listed in sys.path
contributes modules to the root package.
1.4 Distutils-specific terminology

The following terms apply more specifically to the domain of distributing Python modules using the Distutils:

**module distribution**

A collection of Python modules distributed together as a single downloadable resource and meant to be installed *en masse*. Examples of some well-known module distributions are Numeric Python, PyXML, PIL (the Python Imaging Library), or mxBase. (This would be called a *package*, except that term is already taken in the Python context: a single module distribution may contain zero, one, or many Python packages.)

**pure module distribution**

A module distribution that contains only pure Python modules and packages. Sometimes referred to as a ``pure distribution."

**non-pure module distribution**

A module distribution that contains at least one extension module. Sometimes referred to as a ``non-pure distribution."

**distribution root**

The top-level directory of your source tree (or source distribution); the directory where *setup.py* exists. Generally *setup.py* will be run from this directory.

---

*Previous:* 1.3 General Python terminology  *Up:* 1. An Introduction to  *Next:* 2. Writing the Setup

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
2. Writing the Setup Script

The setup script is the centre of all activity in building, distributing, and installing modules using the Distutils. The main purpose of the setup script is to describe your module distribution to the Distutils, so that the various commands that operate on your modules do the right thing. As we saw in section 1.2 above, the setup script consists mainly of a call to setup(), and most information supplied to the Distutils by the module developer is supplied as keyword arguments to setup().

Here's a slightly more involved example, which we'll follow for the next couple of sections: the Distutils' own setup script. (Keep in mind that although the Distutils are included with Python 1.6 and later, they also have an independent existence so that Python 1.5.2 users can use them to install other module distributions. The Distutils' own setup script, shown here, is used to install the package into Python 1.5.2.)

```python
#!/usr/bin/env python

from distutils.core import setup

setup(name='Distutils',
      version='1.0',
      description='Python Distribution Utilities',
      author='Greg Ward',
      author_email='gward@python.net',
      url='http://www.python.org/sigs/distutils-sig/',
      packages=['distutils', 'distutils.command'],
)
```

There are only two differences between this and the trivial one-file distribution presented in section 1.2: more metadata, and the specification of pure Python modules by package, rather than by module. This is important since the Distutils consist of a couple of dozen modules split into (so far) two packages; an explicit list of every module would be tedious to generate and difficult to maintain. For more information on the additional meta-data, see section 2.8.

Note that any pathnames (files or directories) supplied in the setup script should be written using the UNIX convention, i.e. slash-separated. The Distutils will take care of converting this platform-neutral representation into whatever is
appropriate on your current platform before actually using the pathname. This makes your setup script portable across operating systems, which of course is one of the major goals of the Distutils. In this spirit, all pathnames in this document are slash-separated. (Mac OS 9 programmers should keep in mind that the absence of a leading slash indicates a relative path, the opposite of the Mac OS convention with colons.)

This, of course, only applies to pathnames given to Distutils functions. If you, for example, use standard Python functions such as `glob.glob()` or `os.listdir()` to specify files, you should be careful to write portable code instead of hardcoding path separators:

```python
glob.glob(os.path.join('mydir', 'subdir', '*.html'))
o.listdir(os.path.join('mydir', 'subdir'))
```

### Subsections

- [2.1 Listing whole packages](#)
- [2.2 Listing individual modules](#)
- [2.3 Describing extension modules](#)
  - [2.3.1 Extension names and packages](#)
  - [2.3.2 Extension source files](#)
  - [2.3.3 Preprocessor options](#)
  - [2.3.4 Library options](#)
  - [2.3.5 Other options](#)
- [2.4 Relationships between Distributions and Packages](#)
- [2.5 Installing Scripts](#)
- [2.6 Installing Package Data](#)
- [2.7 Installing Additional Files](#)
- [2.8 Additional meta-data](#)
- [2.9 Debugging the setup script](#)
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
2.1 Listing whole packages

The packages option tells the Distutils to process (build, distribute, install, etc.) all pure Python modules found in each package mentioned in the packages list. In order to do this, of course, there has to be a correspondence between package names and directories in the filesystem. The default correspondence is the most obvious one, i.e. package distutils is found in the directory distutils relative to the distribution root. Thus, when you say packages = ['foo'] in your setup script, you are promising that the Distutils will find a file foo/__init__.py (which might be spelled differently on your system, but you get the idea) relative to the directory where your setup script lives. If you break this promise, the Distutils will issue a warning but still process the broken package anyways.

If you use a different convention to lay out your source directory, that's no problem: you just have to supply the package_dir option to tell the Distutils about your convention. For example, say you keep all Python source under lib, so that modules in the "root package" (i.e., not in any package at all) are in lib, modules in the foo package are in lib/foo, and so forth. Then you would put

```
package_dir = {'': 'lib'}
```

in your setup script. The keys to this dictionary are package names, and an empty package name stands for the root package. The values are directory names relative to your distribution root. In this case, when you say packages = ['foo'], you are promising that the file lib/foo/__init__.py exists.

Another possible convention is to put the foo package right in lib, the foo.bar package in lib/bar, etc. This would be written in the setup script as

```
package_dir = {'foo': 'lib'}
```

A package: dir entry in the package_dir dictionary implicitly applies to all packages below package, so the foo.bar case is automatically handled here. In this example, having packages = ['foo', 'foo.bar'] tells the Distutils to look for lib/__init__.py and lib/bar/__init__.py. (Keep in mind that although package_dir applies recursively, you must explicitly list all packages in packages: the Distutils will not recursively scan your source tree looking for
any directory with an __init__.py file.)
2.2 Listing individual modules

For a small module distribution, you might prefer to list all modules rather than listing packages--especially the case of a single module that goes in the ``root package'' (i.e., no package at all). This simplest case was shown in section 1.2; here is a slightly more involved example:

   py_modules = ['mod1', 'pkg.mod2']

This describes two modules, one of them in the ``root'' package, the other in the pkg package. Again, the default package/directory layout implies that these two modules can be found in mod1.py and pkg/mod2.py, and that pkg/__init__.py exists as well. And again, you can override the package/directory correspondence using the package_dir option.
Subsections

- 2.3.1 Extension names and packages
- 2.3.2 Extension source files
- 2.3.3 Preprocessor options
- 2.3.4 Library options
- 2.3.5 Other options
2.3 Describing extension modules

Just as writing Python extension modules is a bit more complicated than writing pure Python modules, describing them to the Distutils is a bit more complicated. Unlike pure modules, it's not enough just to list modules or packages and expect the Distutils to go out and find the right files; you have to specify the extension name, source file(s), and any compile/link requirements (include directories, libraries to link with, etc.).

All of this is done through another keyword argument to setup(), the ext_modules option. ext_modules is just a list of Extension instances, each of which describes a single extension module. Suppose your distribution includes a single extension, called foo and implemented by foo.c. If no additional instructions to the compiler/linker are needed, describing this extension is quite simple:

```
Extension('foo', ['foo.c'])
```

The Extension class can be imported from distutils.core along with setup(). Thus, the setup script for a module distribution that contains only this one extension and nothing else might be:

```
from distutils.core import setup, Extension
setup(name='foo',
       version='1.0',
       ext_modules=[Extension('foo', ['foo.c'])],
)
```

The Extension class (actually, the underlying extension-building machinery implemented by the build_ext command) supports a great deal of flexibility in describing Python extensions, which is explained in the following sections.
2.3.1 Extension names and packages

The first argument to the \texttt{Extension} constructor is always the name of the extension, including any package names. For example,

\begin{verbatim}
    Extension('foo', ['src/foo1.c', 'src/foo2.c'])
\end{verbatim}

describes an extension that lives in the root package, while

\begin{verbatim}
    Extension('pkg.foo', ['src/foo1.c', 'src/foo2.c'])
\end{verbatim}

describes the same extension in the \texttt{pkg} package. The source files and resulting object code are identical in both cases; the only difference is where in the filesystem (and therefore where in Python's namespace hierarchy) the resulting extension lives.

If you have a number of extensions all in the same package (or all under the same base package), use the \texttt{ext\_package} keyword argument to \texttt{setup(\)}. For example,

\begin{verbatim}
    setup(...
        ext\_package='pkg',
        ext\_modules=[Extension('foo', ['foo.c']),
                       Extension('subpkg.bar', ['bar.c'])],
    )
\end{verbatim}

will compile \texttt{foo.c} to the extension \texttt{pkg.foo}, and \texttt{bar.c} to \texttt{pkg.subpkg.bar}.
2.3.2 Extension source files

The second argument to the `Extension` constructor is a list of source files. Since the Distutils currently only support C, C++, and Objective-C extensions, these are normally C/C++/Objective-C source files. (Be sure to use appropriate extensions to distinguish C++ source files: `.cc` and `.cpp` seem to be recognized by both UNIX and Windows compilers.)

However, you can also include SWIG interface (.i) files in the list; the `build_ext` command knows how to deal with SWIG extensions: it will run SWIG on the interface file and compile the resulting C/C++ file into your extension.

**SWIG support is rough around the edges and largely untested; especially SWIG support for C++ extensions! Explain in more detail here when the interface firms up.**

On some platforms, you can include non-source files that are processed by the compiler and included in your extension. Currently, this just means Windows message text (.mc) files and resource definition (.rc) files for Visual C++. These will be compiled to binary resource (.res) files and linked into the executable.
2.3.3 Preprocessor options

Three optional arguments to `Extension` will help if you need to specify include directories to search or preprocessor macros to define/undefine: `include_dirs`, `define_macros`, and `undef_macros`.

For example, if your extension requires header files in the `include` directory under your distribution root, use the `include_dirs` option:

   Extension('foo', ['foo.c'], include_dirs=['include'])

You can specify absolute directories there; if you know that your extension will only be built on UNIX systems with X11R6 installed to `/usr`, you can get away with

   Extension('foo', ['foo.c'], include_dirs=['/usr/include/X11'])

You should avoid this sort of non-portable usage if you plan to distribute your code: it's probably better to write C code like

   #include <X11/Xlib.h>

If you need to include header files from some other Python extension, you can take advantage of the fact that header files are installed in a consistent way by the Distutils `install_header` command. For example, the Numerical Python header files are installed (on a standard UNIX installation) to `/usr/local/include/python1.5/Numerical`. (The exact location will differ according to your platform and Python installation.) Since the Python include directory--`/usr/local/include/python1.5` in this case--is always included in the search path when building Python extensions, the best approach is to write C code like

   #include <Numerical/arrayobject.h>

If you must put the Numerical include directory right into your header search path, though, you can find that directory using the Distutils `distutils.sysconfig` module:

   from distutils.sysconfig import get_python_inc
   incdir = os.path.join(get_python_inc(plat_specific=1), 'Numerical')
Even though this is quite portable--it will work on any Python installation, regardless of platform--it's probably easier to just write your C code in the sensible way.

You can define and undefine pre-processor macros with the `define_macros` and `undef_macros` options. `define_macros` takes a list of (name, value) tuples, where name is the name of the macro to define (a string) and value is its value: either a string or `None`. (Defining a macro `FOO` to `None` is the equivalent of a bare `#define FOO` in your C source: with most compilers, this sets `FOO` to the string 1.) `undef_macros` is just a list of macros to undefine.

For example:

```python
setup(...,
    Extension(..., include_dirs=[incdir]),
)

 Extension(...,
             define_macros=[('NDEBUG', '1'),
                            ('HAVE_STRFTIME', None)],
             undef_macros=['HAVE_FOO', 'HAVE_BAR'])
```

is the equivalent of having this at the top of every C source file:

```
#define NDEBUG 1
#define HAVE_STRFTIME
#undef HAVE_FOO
#undef HAVE_BAR
```
2.3.4 Library options

You can also specify the libraries to link against when building your extension, and the directories to search for those libraries. The libraries option is a list of libraries to link against, library_dirs is a list of directories to search for libraries at link-time, and runtime_library_dirs is a list of directories to search for shared (dynamically loaded) libraries at run-time.

For example, if you need to link against libraries known to be in the standard library search path on target systems

```python
Extension(...,
    libraries=['gdbm', 'readline'])
```

If you need to link with libraries in a non-standard location, you'll have to include the location in library_dirs:

```python
Extension(...,
    library_dirs=['/usr/X11R6/lib'],
    libraries=['X11', 'Xt'])
```

(Again, this sort of non-portable construct should be avoided if you intend to distribute your code.)

** Should mention clib libraries here or somewhere else! **
2.3.5 Other options

There are still some other options which can be used to handle special cases.

The extra_objects option is a list of object files to be passed to the linker. These files must not have extensions, as the default extension for the compiler is used.

extra_compile_args and extra_link_args can be used to specify additional command line options for the respective compiler and linker command lines.

export_symbols is only useful on Windows. It can contain a list of symbols (functions or variables) to be exported. This option is not needed when building compiled extensions: Distutils will automatically add initmodule to the list of exported symbols.
2.4 Relationships between Distributions and Packages

A distribution may relate to packages in three specific ways:

1. It can require packages or modules.
2. It can provide packages or modules.
3. It can obsolete packages or modules.

These relationships can be specified using keyword arguments to the `distutils.core.setup()` function.

Dependencies on other Python modules and packages can be specified by supplying the `requires` keyword argument to `setup()`. The value must be a list of strings. Each string specifies a package that is required, and optionally what versions are sufficient.

To specify that any version of a module or package is required, the string should consist entirely of the module or package name. Examples include 'mymodule' and 'xml.parsers.expat'.

If specific versions are required, a sequence of qualifiers can be supplied in parentheses. Each qualifier may consist of a comparison operator and a version number. The accepted comparison operators are:

```
<  >  ==
<=  >=  !=
```

These can be combined by using multiple qualifiers separated by commas (and optional whitespace). In this case, all of the qualifiers must be matched; a logical AND is used to combine the evaluations.

Let's look at a bunch of examples:

<table>
<thead>
<tr>
<th>Requires</th>
<th>Explanation</th>
</tr>
</thead>
</table>
### Expression

<table>
<thead>
<tr>
<th>Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \equiv 1.0 )</td>
<td>Only version 1.0 is compatible</td>
</tr>
<tr>
<td>( &gt;1.0, \neq 1.5.1, &lt;2.0 )</td>
<td>Any version after 1.0 and before 2.0 is compatible, except 1.5.1</td>
</tr>
</tbody>
</table>

Now that we can specify dependencies, we also need to be able to specify what we provide that other distributions can require. This is done using the *provides* keyword argument to `setup()`. The value for this keyword is a list of strings, each of which names a Python module or package, and optionally identifies the version. If the version is not specified, it is assumed to match that of the distribution.

Some examples:

<table>
<thead>
<tr>
<th>Provides</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mymod</code></td>
<td>Provide <code>mymod</code>, using the distribution version</td>
</tr>
<tr>
<td><code>mymod (1.1</code></td>
<td>Provide <code>mymod</code> version 1.1, regardless of the distribution version</td>
</tr>
</tbody>
</table>

A package can declare that it obsoletes other packages using the *obsoletes* keyword argument. The value for this is similar to that of the *requires* keyword: a list of strings giving module or package specifiers. Each specifier consists of a module or package name optionally followed by one or more version qualifiers. Version qualifiers are given in parentheses after the module or package name.

The versions identified by the qualifiers are those that are obsoleted by the distribution being described. If no qualifiers are given, all versions of the named module or package are understood to be obsoleted.
2.5 Installing Scripts

So far we have been dealing with pure and non-pure Python modules, which are usually not run by themselves but imported by scripts.

Scripts are files containing Python source code, intended to be started from the command line. Scripts don't require Distutils to do anything very complicated. The only clever feature is that if the first line of the script starts with `#!` and contains the word `python`, the Distutils will adjust the first line to refer to the current interpreter location. By default, it is replaced with the current interpreter location. The `--executable` (or `-e`) option will allow the interpreter path to be explicitly overridden.

The `scripts` option simply is a list of files to be handled in this way. From the PyXML setup script:

```python
setup(...
    scripts=['scripts/xmlproc_parse', 'scripts/xmlproc_val']
)
```

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
2.6 Installing Package Data

Often, additional files need to be installed into a package. These files are often data that's closely related to the package's implementation, or text files containing documentation that might be of interest to programmers using the package. These files are called *package data*.

Package data can be added to packages using the `package_data` keyword argument to the `setup()` function. The value must be a mapping from package name to a list of relative path names that should be copied into the package. The paths are interpreted as relative to the directory containing the package (information from the `package_dir` mapping is used if appropriate); that is, the files are expected to be part of the package in the source directories. They may contain glob patterns as well.

The path names may contain directory portions; any necessary directories will be created in the installation.

For example, if a package should contain a subdirectory with several data files, the files can be arranged like this in the source tree:

```plaintext
setup.py
src/
    mypkg/
        __init__.py
        module.py
        data/
            tables.dat
            spoons.dat
            forks.dat
```

The corresponding call to `setup()` might be:

```python
setup(...,
    packages=['mypkg'],
    package_dir={'mypkg': 'src/mypkg'},
    package_data={'mypkg': ['data/*.dat']},
)
```

New in version 2.4.
2.7 Installing Additional Files

The data_files option can be used to specify additional files needed by the module distribution: configuration files, message catalogs, data files, anything which doesn't fit in the previous categories.

data_files specifies a sequence of (directory, files) pairs in the following way:

```python
setup(...
    data_files=[('bitmaps', ['bm/b1.gif', 'bm/b2.gif']),
                ('config', ['cfg/data.cfg']),
                ('/etc/init.d', ['init-script'])]
)
```

Note that you can specify the directory names where the data files will be installed, but you cannot rename the data files themselves.

Each (directory, files) pair in the sequence specifies the installation directory and the files to install there. If directory is a relative path, it is interpreted relative to the installation prefix (Python's `sys.prefix` for pure-Python packages, `sys.exec_prefix` for packages that contain extension modules). Each file name in files is interpreted relative to the `setup.py` script at the top of the package source distribution. No directory information from files is used to determine the final location of the installed file; only the name of the file is used.

You can specify the data_files options as a simple sequence of files without specifying a target directory, but this is not recommended, and the install command will print a warning in this case. To install data files directly in the target directory, an empty string should be given as the directory.
## 2.8 Additional meta-data

The setup script may include additional meta-data beyond the name and version. This information includes:

<table>
<thead>
<tr>
<th>Meta-Data</th>
<th>Description</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name of the package</td>
<td>short string</td>
<td>(1)</td>
</tr>
<tr>
<td>version</td>
<td>version of this release</td>
<td>short string</td>
<td>(1)(2)</td>
</tr>
<tr>
<td>author</td>
<td>package author's name</td>
<td>short string</td>
<td>(3)</td>
</tr>
<tr>
<td>author_email</td>
<td>email address of the package author</td>
<td>email address</td>
<td>(3)</td>
</tr>
<tr>
<td>maintainer</td>
<td>package maintainer's name</td>
<td>short string</td>
<td>(3)</td>
</tr>
<tr>
<td>maintainer_email</td>
<td>email address of the package maintainer</td>
<td>email address</td>
<td>(3)</td>
</tr>
<tr>
<td>url</td>
<td>home page for the package</td>
<td>URL</td>
<td>(1)</td>
</tr>
<tr>
<td>description</td>
<td>short, summary description of the package</td>
<td>short string</td>
<td></td>
</tr>
<tr>
<td>long_description</td>
<td>longer description of the package</td>
<td>long string</td>
<td></td>
</tr>
<tr>
<td>download_url</td>
<td>location where the package may be downloaded</td>
<td>URL</td>
<td>(4)</td>
</tr>
<tr>
<td>classifiers</td>
<td>a list of classifiers</td>
<td>list of strings</td>
<td>(4)</td>
</tr>
</tbody>
</table>

Notes:

(1) These fields are required.
(2) It is recommended that versions take the form major.minor[.patch[.sub]].
(3)
Either the author or the maintainer must be identified.

These fields should not be used if your package is to be compatible with Python versions prior to 2.2.3 or 2.3. The list is available from the PyPI website.

'short string'
A single line of text, not more than 200 characters.

'long string'
Multiple lines of plain text in reStructuredText format (see http://docutils.sf.net/).

'list of strings'
See below.

None of the string values may be Unicode.

Encoding the version information is an art in itself. Python packages generally adhere to the version format major.minor[.patch][sub]. The major number is 0 for initial, experimental releases of software. It is incremented for releases that represent major milestones in a package. The minor number is incremented when important new features are added to the package. The patch number increments when bug-fix releases are made. Additional trailing version information is sometimes used to indicate sub-releases. These are "a1,a2,...,aN" (for alpha releases, where functionality and API may change), "b1,b2,...,bN" (for beta releases, which only fix bugs) and "pr1,pr2,...,prN" (for final pre-release release testing). Some examples:

0.1.0
the first, experimental release of a package

1.0.1a2
the second alpha release of the first patch version of 1.0

classifiers are specified in a python list:

```python
setup(
    classifiers=[
        'Development Status :: 4 - Beta',
        'Environment :: Console',
        'Environment :: Web Environment',
        'Intended Audience :: End Users/Desktop',
        'Intended Audience :: Developers',
    ]
```
If you wish to include classifiers in your setup.py file and also wish to remain backwards-compatible with Python releases prior to 2.2.3, then you can include the following code fragment in your setup.py before the setup() call.

```
# patch distutils if it can't cope with the "classifiers" or
# "download_url" keywords
from sys import version
if version < '2.2.3':
    from distutils.dist import DistributionMetadata
    DistributionMetadata.classifiers = None
    DistributionMetadata.download_url = None
```

Distributing Python Modules

Previous: 2.7 Installing Additional Files  Up: 2. Writing the Setup  Next: 2.9 Debugging the setup

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
2.9 Debugging the setup script

Sometimes things go wrong, and the setup script doesn't do what the developer wants.

Distutils catches any exceptions when running the setup script, and print a simple error message before the script is terminated. The motivation for this behaviour is to not confuse administrators who don't know much about Python and are trying to install a package. If they get a big long traceback from deep inside the guts of Distutils, they may think the package or the Python installation is broken because they don't read all the way down to the bottom and see that it's a permission problem.

On the other hand, this doesn't help the developer to find the cause of the failure. For this purpose, the DISTUTILS_DEBUG environment variable can be set to anything except an empty string, and distutils will now print detailed information what it is doing, and prints the full traceback in case an exception occurs.
3. Writing the Setup Configuration File

Often, it's not possible to write down everything needed to build a distribution \textit{a priori}: you may need to get some information from the user, or from the user's system, in order to proceed. As long as that information is fairly simple--a list of directories to search for C header files or libraries, for example--then providing a configuration file, \texttt{setup.cfg}, for users to edit is a cheap and easy way to solicit it. Configuration files also let you provide default values for any command option, which the installer can then override either on the command-line or by editing the config file.

The setup configuration file is a useful middle-ground between the setup script--which, ideally, would be opaque to installers\textsuperscript{31}--and the command-line to the setup script, which is outside of your control and entirely up to the installer. In fact, \texttt{setup.cfg} (and any other Distutils configuration files present on the target system) are processed after the contents of the setup script, but before the command-line. This has several useful consequences:

- installers can override some of what you put in \texttt{setup.py} by editing \texttt{setup.cfg}
- you can provide non-standard defaults for options that are not easily set in \texttt{setup.py}
- installers can override anything in \texttt{setup.cfg} using the command-line options to \texttt{setup.py}

The basic syntax of the configuration file is simple:

```
[command]
option=value
...
```

where \textit{command} is one of the Distutils commands (e.g. \texttt{build\_py}, \texttt{install}), and \textit{option} is one of the options that command supports. Any number of options can be supplied for each command, and any number of command sections can be included in the file. Blank lines are ignored, as are comments, which run from a "#" character until the end of the line. Long option values can be split across
multiple lines simply by indenting the continuation lines.

You can find out the list of options supported by a particular command with the universal \texttt{--help} option, e.g.

\begin{verbatim}
> python setup.py --help build_ext
[...]
Options for 'build_ext' command:
  --build-lib (-b) directory for compiled extension modules
  --build-temp (-t) directory for temporary files (build by-produ
  --inplace (-i) ignore build-lib and put compiled extensions
    source directory alongside your pure Python m
  --include-dirs (-I) list of directories to search for header file
  --define (-D) C preprocessor macros to define
  --undef (-U) C preprocessor macros to undefine
[...]
\end{verbatim}

Note that an option spelled \texttt{--foo-bar} on the command-line is spelled \texttt{foo_bar} in configuration files.

For example, say you want your extensions to be built ``in-place''--that is, you have an extension \texttt{pkg.ext}, and you want the compiled extension file (\texttt{ext.so} on UNIX, say) to be put in the same source directory as your pure Python modules \texttt{pkg.mod1} and \texttt{pkg.mod2}. You can always use the \texttt{--inplace} option on the command-line to ensure this:

\begin{verbatim}
python setup.py build_ext --inplace
\end{verbatim}

But this requires that you always specify the \texttt{build_ext} command explicitly, and remember to provide \texttt{--inplace}. An easier way is to ``set and forget'' this option, by encoding it in \texttt{setup.cfg}, the configuration file for this distribution:

\begin{verbatim}
[build_ext]
inplace=1
\end{verbatim}

This will affect all builds of this module distribution, whether or not you explicitly specify \texttt{build_ext}. If you include \texttt{setup.cfg} in your source distribution, it will also affect end-user builds--which is probably a bad idea for this option, since always building extensions in-place would break installation of the module distribution. In certain peculiar cases, though, modules are built right in their installation directory, so this is conceivably a useful ability. (Distributing extensions that expect to be built in their installation directory is almost always a bad idea, though.)
Another example: certain commands take a lot of options that don't change from run to run; for example, `bdist_rpm` needs to know everything required to generate a ``spec'' file for creating an RPM distribution. Some of this information comes from the setup script, and some is automatically generated by the Distutils (such as the list of files installed). But some of it has to be supplied as options to `bdist_rpm`, which would be very tedious to do on the command-line for every run. Hence, here is a snippet from the Distutils' own `setup.cfg`:

```ini
[bdist_rpm]
release = 1
packager = Greg Ward <gward@python.net>
doc_files = CHANGES.txt
            README.txt
            USAGE.txt
            doc/
            examples/
```

Note that the `doc_files` option is simply a whitespace-separated string split across multiple lines for readability.

### See Also:

*Installing Python Modules*

More information on the configuration files is available in the manual for system administrators.

### Footnotes

... installers\footnote{3.1}{This ideal probably won't be achieved until auto-configuration is fully supported by the Distutils.}
Creating a Source

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4. Creating a Source Distribution

As shown in section 1.2, you use the sdist command to create a source distribution. In the simplest case,

```
python setup.py sdist
```

(assuming you haven't specified any sdist options in the setup script or config file), sdist creates the archive of the default format for the current platform. The default format is a gzip'ed tar file (.tar.gz) on UNIX, and ZIP file on Windows.

You can specify as many formats as you like using the `--formats` option, for example:

```
python setup.py sdist --formats=gztar,zip
```

to create a gzipped tarball and a zip file. The available formats are:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>zip</td>
<td>zip file (.zip)</td>
<td>(1),(3)</td>
</tr>
<tr>
<td>gztar</td>
<td>gzip'ed tar file (.tar.gz)</td>
<td>(2),(4)</td>
</tr>
<tr>
<td>bztar</td>
<td>bzip2'ed tar file (.tar.bz2)</td>
<td>(4)</td>
</tr>
<tr>
<td>ztar</td>
<td>compressed tar file (.tar.Z)</td>
<td>(4)</td>
</tr>
<tr>
<td>tar</td>
<td>tar file (.tar)</td>
<td>(4)</td>
</tr>
</tbody>
</table>

Notes:

(1) default on Windows
(2) default on UNIX
(3) requires either external `zip` utility or `zipfile` module (part of the standard Python library since Python 1.6)
(4) requires external utilities: `tar` and possibly one of `gzip`, `bzip2`, or `compress`
Subsections

- 4.1 Specifying the files to distribute
- 4.2 Manifest-related options

Distributing Python Modules

Previous: 3. Writing the Setup Up: Distributing Python Modules Next: 4.1 Specifying the files

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
4.1 Specifying the files to distribute

If you don't supply an explicit list of files (or instructions on how to generate one), the sdist command puts a minimal default set into the source distribution:

- all Python source files implied by the py_modules and packages options
- all C source files mentioned in the ext_modules or libraries options (**
getting C library sources currently broken--no
get_source_files() method in build_clib.py! **)  
- scripts identified by the scripts option
- anything that looks like a test script: test/test*.py (currently, the Distutils don't do anything with test scripts except include them in source distributions, but in the future there will be a standard for testing Python module distributions)
- README.txt (or README), setup.py (or whatever you called your setup script), and setup.cfg

Sometimes this is enough, but usually you will want to specify additional files to distribute. The typical way to do this is to write a manifest template, called MANIFEST.in by default. The manifest template is just a list of instructions for how to generate your manifest file, MANIFEST, which is the exact list of files to include in your source distribution. The sdist command processes this template and generates a manifest based on its instructions and what it finds in the filesystem.

If you prefer to roll your own manifest file, the format is simple: one filename per line, regular files (or symlinks to them) only. If you do supply your own MANIFEST, you must specify everything: the default set of files described above does not apply in this case.

The manifest template has one command per line, where each command specifies a set of files to include or exclude from the source distribution. For an example, again we turn to the Distutils' own manifest template:

```
include *.txt
recursive-include examples *.txt *.py
prune examples/sample?/build
```
The meanings should be fairly clear: include all files in the distribution root matching *.txt, all files anywhere under the examples directory matching *.txt or *.py, and exclude all directories matching examples/sample?/build. All of this is done after the standard include set, so you can exclude files from the standard set with explicit instructions in the manifest template. (Or, you can use the --no-defaults option to disable the standard set entirely.) There are several other commands available in the manifest template mini-language; see section 10.2.

The order of commands in the manifest template matters: initially, we have the list of default files as described above, and each command in the template adds to or removes from that list of files. Once we have fully processed the manifest template, we remove files that should not be included in the source distribution:

- all files in the Distutils ``build'' tree (default build/)
- all files in directories named RCS, CVS or .svn

Now we have our complete list of files, which is written to the manifest for future reference, and then used to build the source distribution archive(s).

You can disable the default set of included files with the --no-defaults option, and you can disable the standard exclude set with --no-prune.

Following the Distutils' own manifest template, let's trace how the sdist command builds the list of files to include in the Distutils source distribution:

1. include all Python source files in the distutils and distutils/command subdirectories (because packages corresponding to those two directories were mentioned in the packages option in the setup script--see section 2)
2. include README.txt, setup.py, and setup.cfg (standard files)
3. include test/test*.py (standard files)
4. include *.txt in the distribution root (this will find README.txt a second time, but such redundancies are weeded out later)
5. include anything matching *.txt or *.py in the sub-tree under examples,
6. exclude all files in the sub-trees starting at directories matching examples/sample?/build--this may exclude files included by the previous two steps, so it's important that the prune command in the manifest template comes after the recursive-include command
7. exclude the entire build tree, and any RCS, CVS and .svn directories
Just like in the setup script, file and directory names in the manifest template should always be slash-separated; the Distutils will take care of converting them to the standard representation on your platform. That way, the manifest template is portable across operating systems.
4.2 Manifest-related options

The normal course of operations for the sdist command is as follows:

- if the manifest file, MANIFEST doesn't exist, read MANIFEST.in and create the manifest
- if neither MANIFEST nor MANIFEST.in exist, create a manifest with just the default file set
- if either MANIFEST.in or the setup script (setup.py) are more recent than MANIFEST, recreate MANIFEST by reading MANIFEST.in
- use the list of files now in MANIFEST (either just generated or read in) to create the source distribution archive(s)

There are a couple of options that modify this behaviour. First, use the --no-defaults and --no-prune to disable the standard ``include'' and ``exclude'' sets.

Second, you might want to force the manifest to be regenerated--for example, if you have added or removed files or directories that match an existing pattern in the manifest template, you should regenerate the manifest:

```
python setup.py sdist --force-manifest
```

Or, you might just want to (re)generate the manifest, but not create a source distribution:

```
python setup.py sdist --manifest-only
```

`--manifest-only` implies `--force-manifest`. `-o` is a shortcut for `--manifest-only`, and `-f` for `--force-manifest`.

---

Distributing Python Modules

Previous: 4.1 Specifying the files  Up: 4. Creating a Source  Next: 5. Creating Built Distributions

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5. Creating Built Distributions

A "built distribution" is what you're probably used to thinking of either as a "binary package" or an "installer" (depending on your background). It's not necessarily binary, though, because it might contain only Python source code and/or byte-code; and we don't call it a package, because that word is already spoken for in Python. (And "installer" is a term specific to the world of mainstream desktop systems.)

A built distribution is how you make life as easy as possible for installers of your module distribution: for users of RPM-based Linux systems, it's a binary RPM; for Windows users, it's an executable installer; for Debian-based Linux users, it's a Debian package; and so forth. Obviously, no one person will be able to create built distributions for every platform under the sun, so the Distutils are designed to enable module developers to concentrate on their specialty--writing code and creating source distributions--while an intermediary species called packagers springs up to turn source distributions into built distributions for as many platforms as there are packagers.

Of course, the module developer could be his own packager; or the packager could be a volunteer "out there" somewhere who has access to a platform which the original developer does not; or it could be software periodically grabbing new source distributions and turning them into built distributions for as many platforms as the software has access to. Regardless of who they are, a packager uses the setup script and the bdist command family to generate built distributions.

As a simple example, if I run the following command in the Distutils source tree:

```
python setup.py bdist
```

then the Distutils builds my module distribution (the Distutils itself in this case), does a "fake" installation (also in the build directory), and creates the default type of built distribution for my platform. The default format for built distributions is a "dumb" tar file on UNIX, and a simple executable installer on Windows. (That tar file is considered "dumb" because it has to be unpacked in a specific location to work.)
Thus, the above command on a UNIX system creates Distutils-1.0.plat.tar.gz; unpacking this tarball from the right place installs the Distutils just as though you had downloaded the source distribution and run `python setup.py install`. (The "right place" is either the root of the filesystem or Python's `prefix` directory, depending on the options given to the `bdist_dumb` command; the default is to make dumb distributions relative to `prefix`.)

Obviously, for pure Python distributions, this isn't any simpler than just running `python setup.py install`—but for non-pure distributions, which include extensions that would need to be compiled, it can mean the difference between someone being able to use your extensions or not. And creating "smart" built distributions, such as an RPM package or an executable installer for Windows, is far more convenient for users even if your distribution doesn't include any extensions.

The `bdist` command has a `--formats` option, similar to the `sdist` command, which you can use to select the types of built distribution to generate: for example,

```
python setup.py bdist --format=zip
```

would, when run on a UNIX system, create Distutils-1.0.plat.zip—again, this archive would be unpacked from the root directory to install the Distutils.

The available formats for built distributions are:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>gztar</td>
<td>gzipped tar file (.tar.gz)</td>
<td>(1),(3)</td>
</tr>
<tr>
<td>ztar</td>
<td>compressed tar file (.tar.Z)</td>
<td>(3)</td>
</tr>
<tr>
<td>tar</td>
<td>tar file (.tar)</td>
<td>(3)</td>
</tr>
<tr>
<td>zip</td>
<td>zip file (.zip)</td>
<td>(4)</td>
</tr>
<tr>
<td>rpm</td>
<td>RPM</td>
<td>(5)</td>
</tr>
<tr>
<td>pkgtool</td>
<td>Solaris pkgtool</td>
<td></td>
</tr>
<tr>
<td>sdux</td>
<td>HP-UX <code>swinstall</code></td>
<td></td>
</tr>
<tr>
<td>rpm</td>
<td>RPM</td>
<td>(5)</td>
</tr>
<tr>
<td>wininst</td>
<td>self-extracting ZIP file for Windows</td>
<td>(2),(4)</td>
</tr>
</tbody>
</table>

Notes:
default on **Unix**

default on Windows **to-do!**

requires external utilities: **tar** and possibly one of **gzip**, **bzip2**, or **compress**

requires either external **zip** utility or **zipfile** module (part of the standard Python library since Python 1.6)

requires external **rpm** utility, version 3.0.4 or better (use **rpm --version** to find out which version you have)

You don't have to use the **bdist** command with the **--formats** option; you can also use the command that directly implements the format you're interested in. Some of these **bdist** ``sub-commands'' actually generate several similar formats; for instance, the **bdist_dumb** command generates all the ``dumb'' archive formats (**tar**, **ztar**, **gztar**, and **zip**), and **bdist_rpm** generates both binary and source RPMs. The **bdist** sub-commands, and the formats generated by each, are:

<table>
<thead>
<tr>
<th>Command</th>
<th>Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bdist_dumb</strong></td>
<td><strong>tar</strong>, <strong>ztar</strong>, <strong>gztar</strong>, <strong>zip</strong></td>
</tr>
<tr>
<td><strong>bdist_rpm</strong></td>
<td><strong>rpm</strong>, <strong>srpm</strong></td>
</tr>
<tr>
<td><strong>bdist_wininst</strong></td>
<td><strong>wininst</strong></td>
</tr>
</tbody>
</table>

The following sections give details on the individual **bdist_*** commands.

---

**Subsections**

- [5.1 Creating dumb built distributions](#)
- [5.2 Creating RPM packages](#)
- [5.3 Creating Windows Installers](#)
  - [5.3.1 The Postinstallation script](#)
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.1 Creating dumb built distributions

** Need to document absolute vs. prefix-relative packages here, but first I have to implement it! **

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.2 Creating RPM packages

The RPM format is used by many popular Linux distributions, including Red Hat, SuSE, and Mandrake. If one of these (or any of the other RPM-based Linux distributions) is your usual environment, creating RPM packages for other users of that same distribution is trivial. Depending on the complexity of your module distribution and differences between Linux distributions, you may also be able to create RPMs that work on different RPM-based distributions.

The usual way to create an RPM of your module distribution is to run the `bdist_rpm` command:

```
python setup.py bdist_rpm
```

or the `bdist` command with the `--format` option:

```
python setup.py bdist --formats=rpm
```

The former allows you to specify RPM-specific options; the latter allows you to easily specify multiple formats in one run. If you need to do both, you can explicitly specify multiple `bdist_*` commands and their options:

```
python setup.py bdist_rpm --packager="John Doe <jdoe@example.org>" \
bdist_wininst --target_version="2.0"
```

Creating RPM packages is driven by a `.spec` file, much as using the Distutils is driven by the setup script. To make your life easier, the `bdist_rpm` command normally creates a `.spec` file based on the information you supply in the setup script, on the command line, and in any Distutils configuration files. Various options and sections in the `.spec` file are derived from options in the setup script as follows:

<table>
<thead>
<tr>
<th>RPM <code>.spec</code> file option or section</th>
<th>Distutils setup script option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>name</td>
</tr>
<tr>
<td>Summary (in preamble)</td>
<td>description</td>
</tr>
<tr>
<td>Version</td>
<td>version</td>
</tr>
<tr>
<td>Vendor</td>
<td>author and author_email, or &amp; maintainer and maintainer_email</td>
</tr>
</tbody>
</table>
Additionally, there are many options in `.spec` files that don't have corresponding options in the setup script. Most of these are handled through options to the `bdist_rpm` command as follows:

<table>
<thead>
<tr>
<th>RPM .spec file option or section</th>
<th>bdist_rpm option</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release</td>
<td>release</td>
<td><code>1</code></td>
</tr>
<tr>
<td>Group</td>
<td>group</td>
<td><code>Development/Libraries</code></td>
</tr>
<tr>
<td>Vendor</td>
<td>vendor</td>
<td>(see above)</td>
</tr>
<tr>
<td>Packager</td>
<td>packager</td>
<td>(none)</td>
</tr>
<tr>
<td>Provides</td>
<td>provides</td>
<td>(none)</td>
</tr>
<tr>
<td>Requires</td>
<td>requires</td>
<td>(none)</td>
</tr>
<tr>
<td>Conflicts</td>
<td>conflicts</td>
<td>(none)</td>
</tr>
<tr>
<td>Obsoletes</td>
<td>obsoletes</td>
<td>(none)</td>
</tr>
<tr>
<td>Distribution</td>
<td>distribution_name</td>
<td>(none)</td>
</tr>
<tr>
<td>BuildRequires</td>
<td>buildRequires</td>
<td>(none)</td>
</tr>
<tr>
<td>Icon</td>
<td>icon</td>
<td>(none)</td>
</tr>
</tbody>
</table>

Obviously, supplying even a few of these options on the command-line would be tedious and error-prone, so it's usually best to put them in the setup configuration file, `setup.cfg`—see section 3. If you distribute or package many Python module distributions, you might want to put options that apply to all of them in your personal Distutils configuration file (~/.pydistutils.cfg).

There are three steps to building a binary RPM package, all of which are handled automatically by the Distutils:

1. create a `.spec` file, which describes the package (analogous to the Distutils setup script; in fact, much of the information in the setup script winds up in the `.spec` file)
2. create the source RPM
3. create the ```binary``` RPM (which may or may not contain binary code, depending on whether your module distribution contains Python
Normally, RPM bundles the last two steps together; when you use the Distutils, all three steps are typically bundled together.

If you wish, you can separate these three steps. You can use the \texttt{--spec-only} option to make \texttt{bdist\_rpm} just create the \texttt{.spec} file and exit; in this case, the \texttt{.spec} file will be written to the \texttt{``distribution directory''} -- normally \texttt{dist/}, but customizable with the \texttt{--dist-dir} option. (Normally, the \texttt{.spec} file winds up deep in the \texttt{``build tree,''} in a temporary directory created by \texttt{bdist\_rpm}.)

\begin{verbatim}
Previous: 5.1 Creating dumb built Up: 5. Creating Built Distributions
Next: 5.3 Creating Windows Installers
\end{verbatim}

\textit{Release 2.5, documentation updated on 19th September, 2006.}
\textit{See \texttt{About this document...} for information on suggesting changes.}
Subsections

- 5.3.1 The Postinstallation script
5.3 Creating Windows Installers

Executable installers are the natural format for binary distributions on Windows. They display a nice graphical user interface, display some information about the module distribution to be installed taken from the metadata in the setup script, let the user select a few options, and start or cancel the installation.

Since the metadata is taken from the setup script, creating Windows installers is usually as easy as running:

    python setup.py bdist_wininst

or the bdist command with the --formats option:

    python setup.py bdist --formats=wininst

If you have a pure module distribution (only containing pure Python modules and packages), the resulting installer will be version independent and have a name like foo-1.0.win32.exe. These installers can even be created on UNIX or Mac OS platforms.

If you have a non-pure distribution, the extensions can only be created on a Windows platform, and will be Python version dependent. The installer filename will reflect this and now has the form foo-1.0.win32-py2.0.exe. You have to create a separate installer for every Python version you want to support.

The installer will try to compile pure modules into bytecode after installation on the target system in normal and optimizing mode. If you don't want this to happen for some reason, you can run the bdist_wininst command with the --no-target-compile and/or the --no-target-optimize option.

By default the installer will display the cool "Python Powered" logo when it is run, but you can also supply your own bitmap which must be a Windows .bmp file with the --bitmap option.

The installer will also display a large title on the desktop background window when it is run, which is constructed from the name of your distribution and the version number. This can be changed to another text by using the --title option.
The installer file will be written to the ``distribution directory'' -- normally `dist/`, but customizable with the `--dist-dir` option.
5.3.1 The Postinstallation script

Starting with Python 2.3, a postinstallation script can be specified which the --install-script option. The basename of the script must be specified, and the script filename must also be listed in the scripts argument to the setup function.

This script will be run at installation time on the target system after all the files have been copied, with argv[1] set to -install, and again at uninstallation time before the files are removed with argv[1] set to -remove.

The installation script runs embedded in the windows installer, every output (sys.stdout, sys.stderr) is redirected into a buffer and will be displayed in the GUI after the script has finished.

Some functions especially useful in this context are available as additional built-in functions in the installation script.

**directory_created**(path)

**file_created**(path)

These functions should be called when a directory or file is created by the postinstall script at installation time. It will register path with the uninstaller, so that it will be removed when the distribution is uninstalled. To be safe, directories are only removed if they are empty.

**get_special_folder_path**(csidl_string)

This function can be used to retrieve special folder locations on Windows like the Start Menu or the Desktop. It returns the full path to the folder. csidl_string must be one of the following strings:

"CSIDL_APPDATA"

"CSIDL_COMMON_STARTMENU"
"CSIDL_STARTMENU"

"CSIDL_COMMON_DESKTOPDIRECTORY"
"CSIDL_DESKTOPDIRECTORY"

"CSIDL_COMMON_STARTUP"
"CSIDL_STARTUP"
"CSIDL_COMMON_PROGRAMS"
"CSIDL_PROGRAMS"
"CSIDL_FONTS"

If the folder cannot be retrieved, 

\texttt{OSError} is raised.

Which folders are available depends on the exact Windows version, and probably also the configuration. For details refer to Microsoft's documentation of the \texttt{SHGetSpecialFolderPath()} function.

\texttt{create\_shortcut( target, description, filename[, arguments[, workdir[, iconpath[, iconindex]]]]])}

This function creates a shortcut. \texttt{target} is the path to the program to be started by the shortcut. \texttt{description} is the description of the shortcut. \texttt{filename} is the title of the shortcut that the user will see. \texttt{arguments} specifies the command line arguments, if any. \texttt{workdir} is the working directory for the program. \texttt{iconpath} is the file containing the icon for the shortcut, and \texttt{iconindex} is the index of the icon in the file \texttt{iconpath}. Again, for details consult the Microsoft documentation for the \texttt{IShellLink} interface.
6. Registering with the Package Index

The Python Package Index (PyPI) holds meta-data describing distributions packaged with distutils. The distutils command register is used to submit your distribution's meta-data to the index. It is invoked as follows:

```python
python setup.py register
```

Distutils will respond with the following prompt:

```
running register
We need to know who you are, so please choose either:
1. use your existing login,
2. register as a new user,
3. have the server generate a new password for you (and email it to you),
4. quit
Your selection [default 1]:
```

Note: if your username and password are saved locally, you will not see this menu.

If you have not registered with PyPI, then you will need to do so now. You should choose option 2, and enter your details as required. Soon after submitting your details, you will receive an email which will be used to confirm your registration.

Once you are registered, you may choose option 1 from the menu. You will be prompted for your PyPI username and password, and register will then submit your meta-data to the index.

You may submit any number of versions of your distribution to the index. If you alter the meta-data for a particular version, you may submit it again and the index will be updated.

PyPI holds a record for each (name, version) combination submitted. The first user to submit information for a given name is designated the Owner of that name. They may submit changes through the register command or through the web interface. They may also designate other users as Owners or Maintainers.
Maintainers may edit the package information, but not designate other Owners or Maintainers.

By default PyPI will list all versions of a given package. To hide certain versions, the Hidden property should be set to yes. This must be edited through the web interface.

---

Subsections

- [6.1 The .pypirc file](#)
6.1 The .pypirc file

The format of the .pypirc file is formatted as follows:

```
[server-login]
repository: <repository-url>
username: <username>
password: <password>
```

`repository` can be omitted and defaults to http://www.python.org/pypi.
7. Uploading Packages to the Package Index

New in version 2.5.

The Python Package Index (PyPI) not only stores the package info, but also the package data if the author of the package wishes to. The distutils command upload pushes the distribution files to PyPI.

The command is invoked immediately after building one or more distribution files. For example, the command

```
python setup.py sdist bdist_wininst upload
```

will cause the source distribution and the Windows installer to be uploaded to PyPI. Note that these will be uploaded even if they are built using an earlier invocation of setup.py, but that only distributions named on the command line for the invocation including the upload command are uploaded.

The upload command uses the username, password, and repository URL from the $HOME/.pypirc file (see section 6.1 for more on this file).

You can use the --sign option to tell upload to sign each uploaded file using GPG (GNU Privacy Guard). The gpg program must be available for execution on the system PATH. You can also specify which key to use for signing using the --identity=name option.

Other upload options include --repository=url (which lets you override the repository setting from $HOME/.pypirc), and --show-response (which displays the full response text from the PyPI server for help in debugging upload problems).
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
8. Examples

This chapter provides a number of basic examples to help get started with distutils. Additional information about using distutils can be found in the Distutils Cookbook.

See Also:

Distutils Cookbook
Collection of recipes showing how to achieve more control over distutils.

Subsections

- 8.1 Pure Python distribution (by module)
- 8.2 Pure Python distribution (by package)
- 8.3 Single extension module
8.1 Pure Python distribution (by module)

If you're just distributing a couple of modules, especially if they don't live in a particular package, you can specify them individually using the \texttt{py\_modules} option in the setup script.

In the simplest case, you'll have two files to worry about: a setup script and the single module you're distributing, \texttt{foo.py} in this example:

\begin{verbatim}
<root>/
    setup.py
    foo.py
\end{verbatim}

(In all diagrams in this section, \texttt{<root>} will refer to the distribution root directory.) A minimal setup script to describe this situation would be:

\begin{verbatim}
from distutils.core import setup
setup(name='foo',
      version='1.0',
      py_modules=['foo'],
)
\end{verbatim}

Note that the name of the distribution is specified independently with the \texttt{name} option, and there's no rule that says it has to be the same as the name of the sole module in the distribution (although that's probably a good convention to follow). However, the distribution name is used to generate filenames, so you should stick to letters, digits, underscores, and hyphens.

Since \texttt{py\_modules} is a list, you can of course specify multiple modules, eg. if you're distributing modules \texttt{foo} and \texttt{bar}, your setup might look like this:

\begin{verbatim}
<root>/
    setup.py
    foo.py
    bar.py
\end{verbatim}

and the setup script might be

\begin{verbatim}
from distutils.core import setup
setup(name='foobaz',
      version='1.0',
)
\end{verbatim}
You can put module source files into another directory, but if you have enough modules to do that, it's probably easier to specify modules by package rather than listing them individually.
8.2 Pure Python distribution (by package)

If you have more than a couple of modules to distribute, especially if they are in multiple packages, it's probably easier to specify whole packages rather than individual modules. This works even if your modules are not in a package; you can just tell the Distutils to process modules from the root package, and that works the same as any other package (except that you don't have to have an __init__.py file).

The setup script from the last example could also be written as

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      packages=[''],
      )
```

(The empty string stands for the root package.)

If those two files are moved into a subdirectory, but remain in the root package, e.g.:

```
<root>/
  setup.py
  src/
    foo.py
    bar.py
```

then you would still specify the root package, but you have to tell the Distutils where source files in the root package live:

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      package_dir={'': 'src'},
      packages=[''],
      )
```

More typically, though, you will want to distribute multiple modules in the same package (or in sub-packages). For example, if the foo and bar modules belong in package foobar, one way to layout your source tree is
This is in fact the default layout expected by the Distutils, and the one that requires the least work to describe in your setup script:

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      packages=['foobar'],
)
```

If you want to put modules in directories not named for their package, then you need to use the `package_dir` option again. For example, if the `src` directory holds modules in the `foobar` package:

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      package_dir={'foobar': 'src'},
      packages=['foobar'],
)
```

Or, you might put modules from your main package right in the distribution root:

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      package_dir={'foobar': ''},
      packages=['foobar'],
)
```
packages=['foobar'],
)

(The empty string also stands for the current directory.)

If you have sub-packages, they must be explicitly listed in packages, but any entries in package_dir automatically extend to sub-packages. (In other words, the Distutils does not scan your source tree, trying to figure out which directories correspond to Python packages by looking for __init__.py files.) Thus, if the default layout grows a sub-package:

```
<root>/
    setup.py
    foobar/
        __init__.py
        foo.py
        bar.py
    subfoo/
        __init__.py
        blah.py
```

then the corresponding setup script would be

```
from distutils.core import setup
setup(name='foobar',
    version='1.0',
    packages=['foobar', 'foobar.subfoo'],
)
```

(Again, the empty string in package_dir stands for the current directory.)
8.3 Single extension module

Extension modules are specified using the `ext_modules` option. `package_dir` has no effect on where extension source files are found; it only affects the source for pure Python modules. The simplest case, a single extension module in a single C source file, is:

```python
<root>/
    setup.py
    foo.c
```

If the `foo` extension belongs in the root package, the setup script for this could be

```python
from distutils.core import setup
from distutils.extension import Extension
setup(name='foobar',
      version='1.0',
      ext_modules=[Extension('foo', ['foo.c'])],
)
```

If the extension actually belongs in a package, say `foopkg`, then

With exactly the same source tree layout, this extension can be put in the `foopkg` package simply by changing the name of the extension:

```python
from distutils.core import setup
from distutils.extension import Extension
setup(name='foobar',
      version='1.0',
      ext_modules=[Extension('foopkg.foo', ['foo.c'])],
)
```
9. Extending Distutils

Distutils can be extended in various ways. Most extensions take the form of new commands or replacements for existing commands. New commands may be written to support new types of platform-specific packaging, for example, while replacements for existing commands may be made to modify details of how the command operates on a package.

Most extensions of the distutils are made within setup.py scripts that want to modify existing commands; many simply add a few file extensions that should be copied into packages in addition to .py files as a convenience.

Most distutils command implementations are subclasses of the Command class from distutils.cmd. New commands may directly inherit from Command, while replacements often derive from Command indirectly, directly subclassing the command they are replacing. Commands are required to derive from Command.

---

Subsections

- [9.1 Integrating new commands](#)
- [9.2 Adding new distribution types](#)

---

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](#) for information on suggesting changes.
9.1 Integrating new commands

There are different ways to integrate new command implementations into distutils. The most difficult is to lobby for the inclusion of the new features in distutils itself, and wait for (and require) a version of Python that provides that support. This is really hard for many reasons.

The most common, and possibly the most reasonable for most needs, is to include the new implementations with your setup.py script, and cause the distutils.core.setup() function use them:

```python
from distutils.command.build_py import build_py as _build_py
from distutils.core import setup

class build_py(_build_py):
    """Specialized Python source builder."""

        # implement whatever needs to be different...

setup(cmdclass={'build_py': build_py},
     ...
```

This approach is most valuable if the new implementations must be used to use a particular package, as everyone interested in the package will need to have the new command implementation.

Beginning with Python 2.4, a third option is available, intended to allow new commands to be added which can support existing setup.py scripts without requiring modifications to the Python installation. This is expected to allow third-party extensions to provide support for additional packaging systems, but the commands can be used for anything distutils commands can be used for. A new configuration option, command_packages (command-line option --command-packages), can be used to specify additional packages to be searched for modules implementing commands. Like all distutils options, this can be specified on the command line or in a configuration file. This option can only be set in the [global] section of a configuration file, or before any commands on the command line. If set in a configuration file, it can be overridden from the command line; setting it to an empty string on the command line causes the default to be used. This should never be set in a configuration file provided with a package.
This new option can be used to add any number of packages to the list of packages searched for command implementations; multiple package names should be separated by commas. When not specified, the search is only performed in the distutils.command package. When setup.py is run with the option --command-packages distcmds,buildcmds, however, the packages distutils.command, distcmds, and buildcmds will be searched in that order. New commands are expected to be implemented in modules of the same name as the command by classes sharing the same name. Given the example command line option above, the command bdist_openpkg could be implemented by the class distcmds.bdist_openpkg.bdist_openpkg or buildcmds.bdist_openpkg.bdist_openpkg.
9.2 Adding new distribution types

Commands that create distributions (files in the dist/ directory) need to add
(command, filename) pairs to self.distribution.dist_files so
that upload can upload it to PyPI. The filename in the pair contains no path
information, only the name of the file itself. In dry-run mode, pairs should still
be added to represent what would have been created.
10. Command Reference

Subsections

- 10.1 Installing modules: the `install` command family
  - 10.1.1 `install_data`
  - 10.1.2 `install_scripts`
- 10.2 Creating a source distribution: the `sdist` command

Distributing Python Modules

Previous: 9.2 Adding new distribution  Up: Distributing Python Modules  Next: 10.1 Installing modules: the

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
Subsections

- 10.1.1 install data
- 10.1.2 install scripts
10.1 Installing modules: the `install` command family

The `install` command ensures that the build commands have been run and then runs the subcommands `install_lib`, `install_data` and `install_scripts`. 
10.1.1 install_data

This command installs all data files provided with the distribution.
10.1.2 install_scripts

This command installs all (Python) scripts in the distribution.
10.2 Creating a source distribution: the \texttt{sdist} command

** fragment moved down from above: needs context! **

The manifest template commands are:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{include} pat1 pat2 ...</td>
<td>include all files matching any of the listed patterns</td>
</tr>
<tr>
<td>\texttt{exclude} pat1 pat2 ...</td>
<td>exclude all files matching any of the listed patterns</td>
</tr>
<tr>
<td>\texttt{recursive-include} dir pat1 pat2 ...</td>
<td>include all files under \textit{dir} matching any of the listed patterns</td>
</tr>
<tr>
<td>\texttt{recursive-exclude} dir pat1 pat2 ...</td>
<td>exclude all files under \textit{dir} matching any of the listed patterns</td>
</tr>
<tr>
<td>\texttt{global-include} pat1 pat2 ...</td>
<td>include all files anywhere in the source tree matching &amp; any of the listed patterns</td>
</tr>
<tr>
<td>\texttt{global-exclude} pat1 pat2 ...</td>
<td>exclude all files anywhere in the source tree matching &amp; any of the listed patterns</td>
</tr>
<tr>
<td>\texttt{prune} dir</td>
<td>exclude all files under \textit{dir}</td>
</tr>
<tr>
<td>\texttt{graft} dir</td>
<td>include all files under \textit{dir}</td>
</tr>
</tbody>
</table>

The patterns here are \texttt{UNIX}-style ``glob'' patterns: * matches any sequence of regular filename characters, ? matches any single regular filename character, and \texttt{[range]} matches any of the characters in \textit{range} (e.g., a-z, a-zA-Z, a-f0-9_.). The definition of ``regular filename character'' is platform-specific: on \texttt{UNIX} it is anything except slash; on Windows anything except backslash or colon; on Mac OS 9 anything except colon.

** Windows support not there yet **
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11. API Reference

Subsections

- 11.1 distutils.core -- Core Distutils functionality
- 11.2 distutils.ccompiler -- CCompiler base class
- 11.3 distutils.unixccompiler -- Unix C Compiler
- 11.4 distutils.msvccompiler -- Microsoft Compiler
- 11.5 distutils.bcppcompiler -- Borland Compiler
- 11.6 distutils.cygwincompiler -- Cygwin Compiler
- 11.7 distutils.emxccompiler -- OS/2 EMX Compiler
- 11.8 distutils.mwerkscompiler -- Metrowerks CodeWarrior support
- 11.9 distutils.archive_util -- Archiving utilities
- 11.10 distutils.dep_util -- Dependency checking
- 11.11 distutils.dir_util -- Directory tree operations
- 11.12 distutils.file_util -- Single file operations
- 11.13 distutils.util -- Miscellaneous other utility functions
- 11.14 distutils.dist -- The Distribution class
- 11.15 distutils.extension -- The Extension class
- 11.16 distutils.debug -- Distutils debug mode
- 11.17 distutils.errors -- Distutils exceptions
- 11.18 distutils.fancy_getopt -- Wrapper around the standard getopt module
- 11.19 distutils.filelist -- The FileList class
- 11.20 distutils.log -- Simple PEP 282-style logging
- 11.21 distutils.spawn -- Spawn a sub-process
- 11.22 distutils.sysconfig -- System configuration information
- 11.23 distutils.text_file -- The TextFile class
- 11.24 distutils.version -- Version number classes
- 11.25 distutils.cmd -- Abstract base class for Distutils commands
- 11.26 distutils.command -- Individual Distutils commands
- 11.27 distutils.command.bdist -- Build a binary installer
• **11.28 distutils.command.bdist_packager** -- Abstract base class for packagers
• **11.29 distutils.command.bdist_dumb** -- Build a "dumb" installer
• **11.30 distutils.command.bdist_rpm** -- Build a binary distribution as a Redhat RPM and SRPM
• **11.31 distutils.command.bdist_wininst** -- Build a Windows installer
• **11.32 distutils.command.sdist** -- Build a source distribution
• **11.33 distutils.command.build** -- Build all files of a package
• **11.34 distutils.command.build_clib** -- Build any C libraries in a package
• **11.35 distutils.command.build_ext** -- Build any extensions in a package
• **11.36 distutils.command.build_py** -- Build the .py/.pyc files of a package
• **11.37 distutils.command.build_scripts** -- Build the scripts of a package
• **11.38 distutils.command.clean** -- Clean a package build area
• **11.39 distutils.command.config** -- Perform package configuration
• **11.40 distutils.command.install** -- Install a package
• **11.41 distutils.command.install_data** -- Install data files from a package
• **11.42 distutils.command.install_headers** -- Install C/C++ header files from a package
• **11.43 distutils.command.install_lib** -- Install library files from a package
• **11.44 distutils.command.install_scripts** -- Install script files from a package
• **11.45 distutils.command.register** -- Register a module with the Python Package Index
• **11.46 Creating a new Distutils command**

---

**Distributing Python Modules**

Previous: **10.2 Creating a source**  Up: Distributing Python Modules  Next: **11.1 distutils.core**

*Release 2.5, documentation updated on 19th September, 2006.*
See About this document... for information on suggesting changes.
11.1 distutils.core -- Core Distutils functionality

The distutils.core module is the only module that needs to be installed to use the Distutils. It provides the \texttt{setup()} (which is called from the setup script). Indirectly provides the \texttt{distutils.dist.Distribution} and \texttt{distutils.cmd.Command} class.

\texttt{setup( arguments)}

The basic do-everything function that does most everything you could ever ask for from a Distutils method. See XXXXX

The setup function takes a large number of arguments. These are laid out in the following table.

<table>
<thead>
<tr>
<th>argument name</th>
<th>value</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the package</td>
<td>a string</td>
</tr>
<tr>
<td>version</td>
<td>The version number of the package</td>
<td>See \texttt{distutils.version}</td>
</tr>
<tr>
<td>description</td>
<td>A single line describing the package</td>
<td>a string</td>
</tr>
<tr>
<td>long_description</td>
<td>Longer description of the package</td>
<td>a string</td>
</tr>
<tr>
<td>author</td>
<td>The name of the package author</td>
<td>a string</td>
</tr>
<tr>
<td>author_email</td>
<td>The email address of the package author</td>
<td>a string</td>
</tr>
<tr>
<td>maintainer</td>
<td>The name of the package author</td>
<td>a string</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
<td>Type</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>maintainer_email</td>
<td>The email address of the current maintainer, if different from the author.</td>
<td>The email address of the current maintainer, if different from the author.</td>
</tr>
<tr>
<td>url</td>
<td>A URL for the package (homepage)</td>
<td>A URL</td>
</tr>
<tr>
<td>download_url</td>
<td>A URL to download the package</td>
<td>A URL</td>
</tr>
<tr>
<td>packages</td>
<td>A list of Python packages that distutils will manipulate</td>
<td>A list of strings</td>
</tr>
<tr>
<td>py_modules</td>
<td>A list of Python modules that distutils will manipulate</td>
<td>A list of strings</td>
</tr>
<tr>
<td>scripts</td>
<td>A list of standalone script files to be built and installed</td>
<td>A list of strings</td>
</tr>
<tr>
<td>ext_modules</td>
<td>A list of Python extensions to be built</td>
<td>A list of instances of distutils.</td>
</tr>
<tr>
<td>classifiers</td>
<td>A list of categories for the package</td>
<td>The list of available categorizations</td>
</tr>
<tr>
<td>distclass</td>
<td>the Distribution class to use</td>
<td>A subclass of distutils.core.</td>
</tr>
</tbody>
</table>
**script_name** | The name of the setup.py script - defaults to `sys.argv[0]` | a string
---|---|---
**script_args** | Arguments to supply to the setup script | a list of strings
**options** | default options for the setup script | a string
**license** | The license for the package |  
**keywords** | Descriptive metadata. See PEP 314 |  
**platforms** |  
**cmdclass** | A mapping of command names to Command subclasses | a dictionary

**run_setup**(script_name[, script_args=None, stop_after='run'])

Run a setup script in a somewhat controlled environment, and return the `distutils.dist.Distribution` instance that drives things. This is useful if you need to find out the distribution meta-data (passed as keyword args from `script` to `setup()`), or the contents of the config files or command-line.

**script_name** is a file that will be run with `execfile()` `sys.argv[0]` will be replaced with `script` for the duration of the call. **script_args** is a list of strings; if supplied, `sys.argv[1:]` will be replaced by **script_args** for the duration of the call.

**stop_after** tells `setup()` when to stop processing; possible values:

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>Stop after the <code>Distribution</code> instance has been created</td>
</tr>
</tbody>
</table>
and populated with the keyword arguments to setup()

<table>
<thead>
<tr>
<th>config</th>
<th>Stop after config files have been parsed (and their data stored in the Distribution instance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmdline</td>
<td>Stop after the command-line (sys.argv[1:] or script_args) have been parsed (and the data stored in the Distribution instance.)</td>
</tr>
<tr>
<td>run</td>
<td>Stop after all commands have been run (the same as if setup() had been called in the usual way). This is the default value.</td>
</tr>
</tbody>
</table>

In addition, the distutils.core module exposed a number of classes that live elsewhere.

- Extension from distutils.extension
- Command from distutils.cmd
- Distribution from distutils.dist

A short description of each of these follows, but see the relevant module for the full reference.

**class Extension**

The Extension class describes a single C or C++ extension module in a setup script. It accepts the following keyword arguments in its constructor

<table>
<thead>
<tr>
<th>argument name</th>
<th>value</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>the full name of the extension, including any packages -- ie. not a filename or pathname, but Python dotted name</td>
<td>string</td>
</tr>
<tr>
<td>sources</td>
<td>list of source filenames, relative to the distribution root (where the setup script lives), in Unix form (slash-separated) for portability. Source files may be C, C++, SWIG (.i), platform-specific resource files, or whatever else is recognized by the build_ext</td>
<td>string</td>
</tr>
</tbody>
</table>
command as source for a Python extension.

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>include_dirs</td>
<td>list of directories to search for C/C++ header files (in UNIX form for portability)</td>
<td>string</td>
</tr>
<tr>
<td>define_macros</td>
<td>list of macros to define; each macro is defined using a 2-tuple, where 'value' is either the string to define it to or None to define it without a particular value (equivalent of <code>#define FOO</code> in source or <code>-DFOO</code> on UNIX C compiler command line)</td>
<td>(string,string) tuple or (name,None)</td>
</tr>
<tr>
<td>undef_macros</td>
<td>list of macros to undefine explicitly</td>
<td>string</td>
</tr>
<tr>
<td>library_dirs</td>
<td>list of directories to search for C/C++ libraries at link time</td>
<td>string</td>
</tr>
<tr>
<td>libraries</td>
<td>list of library names (not filenames or paths) to link against</td>
<td>string</td>
</tr>
<tr>
<td>runtime_library_dirs</td>
<td>list of directories to search for C/C++ libraries at run time (for shared extensions, this is when the extension is loaded)</td>
<td>string</td>
</tr>
<tr>
<td>extra_objects</td>
<td>list of extra files to link with (eg. object files not implied by 'sources', static library that must be explicitly specified, binary resource files, etc.)</td>
<td>string</td>
</tr>
<tr>
<td>extra_compile_args</td>
<td>any extra platform- and compiler-specific information to use when compiling the source files in 'sources'. For platforms and compilers where a command line makes sense, this is typically a list of command-line arguments, but for other platforms it could be anything.</td>
<td>string</td>
</tr>
</tbody>
</table>
extra_link_args | any extra platform- and compiler-specific information to use when linking object files together to create the extension (or to create a new static Python interpreter). Similar interpretation as for 'extra_compile_args'. | string
---|---|---
export_symbols | list of symbols to be exported from a shared extension. Not used on all platforms, and not generally necessary for Python extensions, which typically export exactly one symbol: init + extension_name. | string
depends | list of files that the extension depends on | string
language | extension language (i.e. 'c', 'c++', 'objc'). Will be detected from the source extensions if not provided. | string

**class Distribution**

A Distribution describes how to build, install and package up a Python software package.

See the setup() function for a list of keyword arguments accepted by the Distribution constructor. setup() creates a Distribution instance.

**class Command**

A Command class (or rather, an instance of one of its subclasses) implement a single distutils command.
11.2 distutils.ccompiler -- CCompiler base class

This module provides the abstract base class for the CCompiler classes. A CCompiler instance can be used for all the compile and link steps needed to build a single project. Methods are provided to set options for the compiler -- macro definitions, include directories, link path, libraries and the like.

This module provides the following functions.

**gen_lib_options**(*compiler*, *library_dirs*, *runtime_library_dirs*, *libraries*)
Generate linker options for searching library directories and linking with specific libraries. *libraries* and *library_dirs* are, respectively, lists of library names (not filenames!) and search directories. Returns a list of command-line options suitable for use with some compiler (depending on the two format strings passed in).

**gen_preprocess_options**(*macros*, *include_dirs*)
Generate C pre-processor options (-D, -U, -I) as used by at least two types of compilers: the typical UNIX compiler and Visual C++. *macros* is the usual thing, a list of 1- or 2-tuples, where (*name*, ) means undefine (-U) macro *name*, and (*name*, *value*) means define (-D) macro *name* to *value*. *include_dirs* is just a list of directory names to be added to the header file search path (-I). Returns a list of command-line options suitable for either UNIX compilers or Visual C++.

**get_default_compiler**(*osname*, *platform*)
Determine the default compiler to use for the given platform.

*osname* should be one of the standard Python OS names (i.e. the ones returned by *os.name*) and *platform* the common value returned by *sys.platform* for the platform in question.

The default values are *os.name* and *sys.platform* in case the parameters are not given.
new_compiler( plat=None, compiler=None, verbose=0, dry_run=0, force=0)

Factory function to generate an instance of some CCompiler subclass for
the supplied platform/compiler combination. plat defaults to os.name (eg.
'posix', 'nt'), and compiler defaults to the default compiler for that
platform. Currently only 'posix' and 'nt' are supported, and the
default compilers are "traditional UNIX interface" (UnixCCompiler
class) and Visual C++(MSVCCompiler class). Note that it's perfectly
possible to ask for a UNIX compiler object under Windows, and a Microsoft
compiler object under UNIX--if you supply a value for compiler, plat is
ignored.

show_compilers()

Print list of available compilers (used by the --help-compiler options to
build, build_ext, build_clib).

class CCompiler([verbose=0, dry_run=0, force=0])

The abstract base class CCompiler defines the interface that must be
implemented by real compiler classes. The class also has some utility
methods used by several compiler classes.

The basic idea behind a compiler abstraction class is that each instance can
be used for all the compile/link steps in building a single project. Thus,
attributes common to all of those compile and link steps -- include
directories, macros to define, libraries to link against, etc. -- are attributes of
the compiler instance. To allow for variability in how individual files are
treated, most of those attributes may be varied on a per-compilation or per-
link basis.

The constructor for each subclass creates an instance of the Compiler
object. Flags are verbose (show verbose output), dry_run (don't actually
execute the steps) and force (rebuild everything, regardless of
dependencies). All of these flags default to 0 (off). Note that you probably
don't want to instantiate CCompiler or one of its subclasses directly - use
the distutils.CCompiler.new_compiler() factory function
instead.
The following methods allow you to manually alter compiler options for the instance of the Compiler class.

**add_include_dir** (*dir*)
Add *dir* to the list of directories that will be searched for header files. The compiler is instructed to search directories in the order in which they are supplied by successive calls to `add_include_dir()`.

**set_include_dirs** (*dirs*)
Set the list of directories that will be searched to *dirs* (a list of strings). Overrides any preceding calls to `add_include_dir()`; subsequent calls to `add_include_dir()` add to the list passed to `set_include_dirs()`. This does not affect any list of standard include directories that the compiler may search by default.

**add_library** (*libname*)
Add *libname* to the list of libraries that will be included in all links driven by this compiler object. Note that *libname* should *not* be the name of a file containing a library, but the name of the library itself: the actual filename will be inferred by the linker, the compiler, or the compiler class (depending on the platform).

The linker will be instructed to link against libraries in the order they were supplied to `add_library()` and/or `set_libraries()`. It is perfectly valid to duplicate library names; the linker will be instructed to link against libraries as many times as they are mentioned.

**set_libraries** (*libnames*)
Set the list of libraries to be included in all links driven by this compiler object to *libnames* (a list of strings). This does not affect any standard system libraries that the linker may include by default.

**add_library_dir** (*dir*)
Add *dir* to the list of directories that will be searched for libraries specified to `add_library()` and `set_libraries()`. The linker will be instructed to search for libraries in the order they are supplied
to add_library_dir() and/or set_library_dirs().

**set_library_dirs(dirs)**
Set the list of library search directories to *dirs* (a list of strings). This does not affect any standard library search path that the linker may search by default.

**add_runtime_library_dir(dir)**
Add *dir* to the list of directories that will be searched for shared libraries at runtime.

**set_runtime_library_dirs(dirs)**
Set the list of directories to search for shared libraries at runtime to *dirs* (a list of strings). This does not affect any standard search path that the runtime linker may search by default.

**define_macro(name[, value=None])**
Define a preprocessor macro for all compilations driven by this compiler object. The optional parameter *value* should be a string; if it is not supplied, then the macro will be defined without an explicit value and the exact outcome depends on the compiler used (XXX true? does ANSI say anything about this?)

**undefine_macro(name)**
Undefine a preprocessor macro for all compilations driven by this compiler object. If the same macro is defined by define_macro() and undefined by undefine_macro() the last call takes precedence (including multiple redefinitions or undefinitions). If the macro is redefined/undefined on a per-compilation basis (ie. in the call to compile()), then that takes precedence.

**add_link_object(object)**
Add *object* to the list of object files (or analogues, such as explicitly named library files or the output of \"resource compilers\") to be included in every link driven by this compiler object.

**set_link_objects(objects)**
Set the list of object files (or analogues) to be included in every link to objects. This does not affect any standard object files that the linker may include by default (such as system libraries).

The following methods implement methods for autodetection of compiler options, providing some functionality similar to GNU autoconf.

**detect_language** *(sources)*
Detect the language of a given file, or list of files. Uses the instance attributes `language_map` (a dictionary), and `language_order` (a list) to do the job.

**find_library_file** *(dirs, lib[, debug=\theta]*)
Search the specified list of directories for a static or shared library file `lib` and return the full path to that file. If `debug` is true, look for a debugging version (if that makes sense on the current platform). Return None if `lib` wasn't found in any of the specified directories.

**has_function** *(funcname [, includes=\texttt{None}, include_dirs=\texttt{None}, libraries=\texttt{None}, library_dirs=\texttt{None}])*  
Return a boolean indicating whether `funcname` is supported on the current platform. The optional arguments can be used to augment the compilation environment by providing additional include files and paths and libraries and paths.

**library_dir_option** *(dir)*
Return the compiler option to add `dir` to the list of directories searched for libraries.

**library_option** *(lib)*
Return the compiler option to add `dir` to the list of libraries linked into the shared library or executable.

**runtime_library_dir_option** *(dir)*
Return the compiler option to add `dir` to the list of directories searched for runtime libraries.
**set_executables( **args**)

Define the executables (and options for them) that will be run to perform the various stages of compilation. The exact set of executables that may be specified here depends on the compiler class (via the 'executables' class attribute), but most will have:

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiler</td>
<td>the C/C++ compiler</td>
</tr>
<tr>
<td>linker_so</td>
<td>linker used to create shared objects and libraries</td>
</tr>
<tr>
<td>linker_exe</td>
<td>linker used to create binary executables</td>
</tr>
<tr>
<td>archiver</td>
<td>static library creator</td>
</tr>
</tbody>
</table>

On platforms with a command-line (UNIX, DOS/Windows), each of these is a string that will be split into executable name and (optional) list of arguments. (Splitting the string is done similarly to how UNIX shells operate: words are delimited by spaces, but quotes and backslashes can override this. See distutils.util.split_quoted().)

The following methods invoke stages in the build process.

```python
compile( sources[, output_dir=None, macros=None, include_dirs=None, debug=0, extra_preargs=None, extra_postargs=None, depends=None])
```

Compile one or more source files. Generates object files (e.g. transforms a .c file to a .o file.)

*sources* must be a list of filenames, most likely C/C++ files, but in reality anything that can be handled by a particular compiler and compiler class (eg. MSVCCompiler can handle resource files in *sources*). Return a list of object filenames, one per source filename in *sources*. Depending on the implementation, not all source files will necessarily be compiled, but all corresponding object filenames will be returned.

If *output_dir* is given, object files will be put under it, while retaining their original path component. That is, foo/bar.c normally compiles to foo/bar.o (for a UNIX implementation); if *output_dir* is *build*, then it
would compile to `build/foo/bar.o`.

*macros*, if given, must be a list of macro definitions. A macro definition is either a `(name, value)` 2-tuple or a `(name,)` 1-tuple. The former defines a macro; if the value is `None`, the macro is defined without an explicit value. The 1-tuple case undefines a macro. Later definitions/redefinitions/undefinitions take precedence.

*include_dirs*, if given, must be a list of strings, the directories to add to the default include file search path for this compilation only.

*debug* is a boolean; if true, the compiler will be instructed to output debug symbols in (or alongside) the object file(s).

*extra_pareargs* and *extra_postargs* are implementation-dependent. On platforms that have the notion of a command-line (e.g. UNIX, DOS/Windows), they are most likely lists of strings: extra command-line arguments to prepend/append to the compiler command line. On other platforms, consult the implementation class documentation. In any event, they are intended as an escape hatch for those occasions when the abstract compiler framework doesn't cut the mustard.

*depends*, if given, is a list of filenames that all targets depend on. If a source file is older than any file in depends, then the source file will be recompiled. This supports dependency tracking, but only at a coarse granularity.

Raises `CompileError` on failure.

```
create_static_lib(objects, output_libname[, output_dir=None, debug=0, target_lang=None])
```

Link a bunch of stuff together to create a static library file. The "bunch of stuff" consists of the list of object files supplied as `objects`, the extra object files supplied to `add_link_object()` and/or `set_link_objects()`, the libraries supplied to `add_library()` and/or `set_libraries()`, and the libraries supplied as `libraries` (if any).

`output_libname` should be a library name, not a filename; the filename
will be inferred from the library name. *output_dir* is the directory where the library file will be put. XXX defaults to what?

*debug* is a boolean; if true, debugging information will be included in the library (note that on most platforms, it is the compile step where this matters: the *debug* flag is included here just for consistency).

*target_lang* is the target language for which the given objects are being compiled. This allows specific linkage time treatment of certain languages.

Raises *LibError* on failure.

```python
link(target_desc, objects, output_filename[, output_dir=None, libraries=None, library_dirs=None, runtime_library_dirs=None, export_symbols=None, debug=0, extra_preargs=None, extra_postargs=None, build_temp=None, target_lang=None])
```

Link a bunch of stuff together to create an executable or shared library file.

The ``bunch of stuff`` consists of the list of object files supplied as *objects*. *output_filename* should be a filename. If *output_dir* is supplied, *output_filename* is relative to it (i.e. *output_filename* can provide directory components if needed).

*libraries* is a list of libraries to link against. These are library names, not filenames, since they’re translated into filenames in a platform-specific way (eg. *foo* becomes *libfoo.a* on *UNIX* and *foo.lib* on DOS/Windows). However, they can include a directory component, which means the linker will look in that specific directory rather than searching all the normal locations.

*library_dirs*, if supplied, should be a list of directories to search for libraries that were specified as bare library names (ie. no directory component). These are on top of the system default and those supplied to *add_library_dir() and/or set_library_dirs()*.

*runtime_library_dirs* is a list of directories that will be embedded into the shared library and used to search for other shared libraries that *it* depends on at run-time. (This may only be relevant on *UNIX.*).
export_symbols is a list of symbols that the shared library will export. (This appears to be relevant only on Windows.)

debug is as for compile() and create_static_lib(), with the slight distinction that it actually matters on most platforms (as opposed to create_static_lib(), which includes a debug flag mostly for form's sake).

extra_preargs and extra_postargs are as for compile() (except of course that they supply command-line arguments for the particular linker being used).

target_lang is the target language for which the given objects are being compiled. This allows specific linkage time treatment of certain languages.

Raises LinkError on failure.

```
link_executable(objects, output_progname[, output_dir=’None’,
libraries=’None’, library_dirs=’None’,
runtime_library_dirs=’None’, debug=0,
extra_preargs=’None’, extra_postargs=’None’,
target_lang=’None’])
```

Link an executable. output_progname is the name of the file executable, while objects are a list of object filenames to link in. Other arguments are as for the link method.

```
link_shared_lib(objects, output_libname[, output_dir=’None’,
libraries=’None’, library_dirs=’None’,
runtime_library_dirs=’None’,
export_symbols=’None’, debug=0,
extra_preargs=’None’, extra_postargs=’None’,
build_temp=’None’, target_lang=’None’])
```

Link a shared library. output_libname is the name of the output library, while objects is a list of object filenames to link in. Other arguments are as for the link method.
Link a shared object. `output_filename` is the name of the shared object that will be created, while `objects` is a list of object filenames to link in. Other arguments are as for the `link` method.

Preprocess a single C/C++ source file, named in `source`. Output will be written to file named `output_file`, or `stdout` if `output_file` not supplied. `macros` is a list of macro definitions as for `compile()`, which will augment the macros set with `define_macro()` and `undefine_macro()`. `include_dirs` is a list of directory names that will be added to the default list, in the same way as `add_include_dir()`.

Raises `PreprocessError` on failure.

The following utility methods are defined by the `CCompiler` class, for use by the various concrete subclasses.

Returns the filename of the executable for the given `basename`. Typically for non-Windows platforms this is the same as the basename, while Windows will get a `.exe` added.

Returns the filename for the given library name on the current platform. On UNIX a library with `lib_type` of 'static' will typically be of the form `liblibname.a`, while a `lib_type` of 'dynamic' will be of the form `liblibname.so`. 
**object_filenames** *source_filenames*[ , *strip_dir=0, output_dir=' ']*)

Returns the name of the object files for the given source files.  
*source_filenames* should be a list of filenames.

**shared_object_filename** *

Returns the name of a shared object file for the given file name *basename*.

**execute** *(func, args[, msg=None, level=1])*

Invokes *distutils.util.execute()* This method invokes a Python function *func* with the given arguments *args*, after logging and taking into account the *dry_run* flag.  XXX see also.

**spawn** *(cmd)*

Invokes *distutils.util.spawn()* This invokes an external process to run the given command.  XXX see also.

**mkpath** *(name[, mode=511])*

Invokes *distutils.dir_util.mkpath()* This creates a directory and any missing ancestor directories.  XXX see also.

**move_file** *(src, dst)*

Invokes *distutils.file_util.move_file()* Renames *src* to *dst*.  XXX see also.

**announce** *(msg[, level=1])*

Write a message using *distutils.log.debug()* XXX see also.

**warn** *(msg)*

Write a warning message *msg* to standard error.

**debug_print** *(msg)*

If the *debug* flag is set on this CCompiler instance, print *msg* to standard output, otherwise do nothing.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.3 distutils.unixccompiler -- Unix C Compiler

This module provides the UnixCCompiler class, a subclass of CCompiler that handles the typical UNIX-style command-line C compiler:

- macros defined with `-Dname[=value]`
- macros undefined with `-Uname`
- include search directories specified with `-Idir`
- libraries specified with `-l` `lib`
- library search directories specified with `-L` `dir`
- compile handled by `cc` (or similar) executable with `-c` option: compiles `.c` to `.o`
- link static library handled by `ar` command (possibly with `ranlib`)
- link shared library handled by `cc -shared`

---

**Distributing Python Modules**

Previous: 11.2 distutils.ccompiler Up: 11. API Reference Next: 11.4 distutils.msvccompiler

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
11.4 distutils.msvccompiler -- Microsoft Compiler

This module provides MsvcCompiler, an implementation of the abstract CCompiler class for Microsoft Visual Studio. Typically, extension modules need to be compiled with the same compiler that was used to compile Python. For Python 2.3 and earlier, the compiler was Visual Studio 6. For Python 2.4 and 2.5, the compiler is Visual Studio .NET 2003. The AMD64 and Itanium binaries are created using the Platform SDK.

MsvcCompiler will normally choose the right compiler, linker etc. on its own. To override this choice, the environment variables DISTUTILS_USE_SDK and MSSdk must be both set. MSSdk indicates that the current environment has been setup by the SDK's SetEnv.Cmd script, or that the environment variables had been registered when the SDK was installed; DISTUTILS_USE_SDK indicates that the distutils user has made an explicit choice to override the compiler selection by MsvcCompiler.

Distributing Python Modules

Previous: 11.3 distutils.unixccompiler Up: 11. API Reference Next: 11.5 distutils.bcppcompiler

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
11.5 distutils.bcppcompiler -- Borland Compiler

This module provides BorlandCCompiler, a subclass of the abstract CCompiler class for the Borland C++ compiler.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.6 distutils.cygwincompiler -- Cygwin Compiler

This module provides the CygwinCCompiler class, a subclass of UnixCCompiler that handles the Cygwin port of the GNU C compiler to Windows. It also contains the Mingw32CCompiler class which handles the mingw32 port of GCC (same as cygwin in no-cygwin mode).
11.7 distutils.emxccompiler -- OS/2 EMX Compiler

This module provides the EMXCCompiler class, a subclass of UnixCCompiler that handles the EMX port of the GNU C compiler to OS/2.

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
11.8 distutils.mwerkscompiler -- Metrowerks CodeWarrior support

Contains MWerksCompiler, an implementation of the abstract CCompiler class for MetroWerks CodeWarrior on the pre-Mac OS X Macintosh. Needs work to support CW on Windows or Mac OS X.
11.9 distutils.archive_util -- Archiving utilities

This module provides a few functions for creating archive files, such as tarballs or zipfiles.

```
make_archive(base_name, format[, root_dir=None, base_dir=None, verbose=0, dry_run=0])
```

Create an archive file (eg. zip or tar). base_name is the name of the file to create, minus any format-specific extension; format is the archive format: one of zip, tar, ztar, or gztar. root_dir is a directory that will be the root directory of the archive; ie. we typically chdir into root_dir before creating the archive. base_dir is the directory where we start archiving from; ie. base_dir will be the common prefix of all files and directories in the archive. root_dir and base_dir both default to the current directory. Returns the name of the archive file.

**Warning:** This should be changed to support bz2 files

```
make_tarball(base_name, base_dir[, compress='gzip', verbose=0, dry_run=0])
```

'Create an (optional compressed) archive as a tar file from all files in and under base_dir. compress must be 'gzip' (the default), 'compress', 'bzip2', or None. Both tar and the compression utility named by compress must be on the default program search path, so this is probably UNIX-specific. The output tar file will be named base_dir.tar, possibly plus the appropriate compression extension (.gz, .bz2 or .Z). Return the output filename.

**Warning:** This should be replaced with calls to the tarfile module.

```
make_zipfile(base_name, base_dir[, verbose=0, dry_run=0])
```

Create a zip file from all files in and under base_dir. The output zip file will be named base_dir + .zip. Uses either the zipfile Python module (if
available) or the InfoZIP zip utility (if installed and found on the default search path). If neither tool is available, raises DistutilsExecError. Returns the name of the output zip file.
11.10 distutils.dep_util --
Dependency checking

This module provides functions for performing simple, timestamp-based
dependency of files and groups of files; also, functions based entirely on such
timestamp dependency analysis.

**newer**(*source*, *target*)

Return true if *source* exists and is more recently modified than *target*, or if
*source* exists and *target* doesn't. Return false if both exist and *target* is the
same age or newer than *source*. Raise *DistutilsFileError* if *source*
does not exist.

**newer_pairwise**(*sources*, *targets*)

Walk two filename lists in parallel, testing if each source is newer than its
 corresponding target. Return a pair of lists (*sources*, *targets*) where source
is newer than target, according to the semantics of *newer()*

**newer_group**(*sources*, *target*, *missing*='error')

Return true if *target* is out-of-date with respect to any file listed in *sources*
In other words, if *target* exists and is newer than every file in *sources*,
return false; otherwise return true. *missing* controls what we do when a
source file is missing; the default ('error') is to blow up with an
*OSError* from inside *os.stat()*. if it is 'ignore', we silently drop
any missing source files; if it is 'newer', any missing source files make
us assume that *target* is out-of-date (this is handy in ``dry-run" mode: it'll
make you pretend to carry out commands that wouldn't work because inputs
are missing, but that doesn't matter because you're not actually going to run
the commands).
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.11 distutils.dir_util --
Directory tree operations

This module provides functions for operating on directories and trees of
directories.

**mkpath**( `name[, mode=0777, verbose=0, dry_run=0]`)  
Create a directory and any missing ancestor directories. If the directory already exists (or if `name` is the empty string, which means the current directory, which of course exists), then do nothing. Raise `DistutilsFileError` if unable to create some directory along the way (e.g. some sub-path exists, but is a file rather than a directory). If `verbose` is true, print a one-line summary of each mkdir to stdout. Return the list of
directories actually created.

**create_tree**( `base_dir, files[, mode=0777, verbose=0, dry_run=0]`)  
Create all the empty directories under `base_dir` needed to put `files` there. 
`base_dir` is just the a name of a directory which doesn't necessarily exist yet; `files` is a list of filenames to be interpreted relative to `base_dir`. 
`base_dir` + the directory portion of every file in `files` will be created if it doesn't already exist. `mode`, `verbose` and `dry_run` flags are as for `mkpath()`.

**copy_tree**( `src, dst[ preserve_mode=1, preserve_times=1, 
    preserve_symlinks=0, update=0, verbose=0, dry_run=0 ]`)  
Copy an entire directory tree `src` to a new location `dst`. Both `src` and `dst` must be directory names. If `src` is not a directory, raise `DistutilsFileError`. If `dst` does not exist, it is created with `mkpath()`. The end result of the copy is that every file in `src` is copied to 
`dst`, and directories under `src` are recursively copied to `dst`. Return the list of
files that were copied or might have been copied, using their output name. 
The return value is unaffected by `update` or `dry_run`: it is simply the list of
all files under `src`, with the names changed to be under `dst`. 

preserve_mode and preserve_times are the same as for copy_file in distutils.file_util; note that they only apply to regular files, not to directories. If preserve_symlinks is true, symlinks will be copied as symlinks (on platforms that support them!); otherwise (the default), the destination of the symlink will be copied. update and verbose are the same as for copy_file.

** Some of this could be replaced with the shutil module? **

remove_tree(directory[verbose=0, dry_run=0])
Recursively remove directory and all files and directories underneath it. Any errors are ignored (apart from being reported to sys.stdout if verbose is true).

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.12 distutils.file_util -- Single file operations

This module contains some utility functions for operating on individual files.

```
copy_file( src, dst[preserve_mode=1, preserve_times=1, update=0,  
    link=None, verbose=0, dry_run=0])
```

Copy file `src` to `dst`. If `dst` is a directory, then `src` is copied there with the same name; otherwise, it must be a filename. (If the file exists, it will be ruthlessly clobbered.) If `preserve_mode` is true (the default), the file's mode (type and permission bits, or whatever is analogous on the current platform) is copied. If `preserve_times` is true (the default), the last-modified and last-access times are copied as well. If `update` is true, `src` will only be copied if `dst` does not exist, or if `dst` does exist but is older than `src`.

`link` allows you to make hard links (using `os.link`) or symbolic links (using `os.symlink`) instead of copying: set it to 'hard' or 'sym'; if it is `None` (the default), files are copied. Don't set `link` on systems that don't support it: `copy_file()` doesn't check if hard or symbolic linking is available. It uses `_copy_file_contents()` to copy file contents.

Return a tuple "(dest_name, copied)": `dest_name` is the actual name of the output file, and `copied` is true if the file was copied (or would have been copied, if `dry_run` true).

```
move_file( src, dst[verbose, dry_run])
```

Move file `src` to `dst`. If `dst` is a directory, the file will be moved into it with the same name; otherwise, `src` is just renamed to `dst`. Returns the new full name of the file. **Warning:** Handles cross-device moves on UNIX using `copy_file()`. What about other systems???

```
write_file( filename, contents)
```

Create a file called `filename` and write `contents` (a sequence of strings without line terminators) to it.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.13 distutils.util -- Miscellaneous other utility functions

This module contains other assorted bits and pieces that don't fit into any other utility module.

get_platform()

Return a string that identifies the current platform. This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by 'os.uname()'), although the exact information included depends on the OS; eg. for IRIX the architecture isn't particularly important (IRIX only runs on SGI hardware), but for Linux the kernel version isn't particularly important.

Examples of returned values:

- linux-i586
- linux-alpha
- solaris-2.6-sun4u
- irix-5.3
- irix64-6.2

For non-POSIX platforms, currently just returns sys.platform.

convert_path(pathname)

Return 'pathname' as a name that will work on the native filesystem, i.e. split it on '/' and put it back together again using the current directory separator. Needed because filenames in the setup script are always supplied in UNIX style, and have to be converted to the local convention before we can actually use them in the filesystem. Raises ValueError on non-UNIX-ish systems if pathname either starts or ends with a slash.

change_root(new_root, pathname)

Return pathname with new_root prepended. If pathname is relative, this is
equivalent to "os.path.join(new_root, pathname)"
Otherwise, it requires making pathname relative and then joining the two, which is tricky on DOS/Windows.

**check_environ()**
Ensure that 'os.environ' has all the environment variables we guarantee that users can use in config files, command-line options, etc. Currently this includes:
- **HOME** - user's home directory (UNIX only)
- **PLAT** - description of the current platform, including hardware and OS
  (see get_platform())

**subst_vars(s, local_vars)**
Perform shell/Perl-style variable substitution on s. Every occurrence of $ followed by a name is considered a variable, and variable is substituted by the value found in the local_vars dictionary, or in os.environ if it's not in local_vars. os.environ is first checked/augmented to guarantee that it contains certain values: see check_environ(). Raise ValueError for any variables not found in either local_vars or os.environ.

Note that this is not a fully-fledged string interpolation function. A valid $variable can consist only of upper and lower case letters, numbers and an underscore. No { } or style quoting is available.

**grok_environment_error(exc[, prefix=\texttt{tex2html_deferred}{'error:}]]
Generate a useful error message from an EnvironmentError (IOError or OSError) exception object. Handles Python 1.5.1 and later styles, and does what it can to deal with exception objects that don't have a filename (which happens when the error is due to a two-file operation, such as rename() or link()). Returns the error message as a string prefixed with prefix.

**split_quoted(s)**
Split a string up according to UNIX shell-like rules for quotes and backslashes. In short: words are delimited by spaces, as long as those spaces are not escaped by a backslash, or inside a quoted string. Single and
double quotes are equivalent, and the quote characters can be backslash-escaped. The backslash is stripped from any two-character escape sequence, leaving only the escaped character. The quote characters are stripped from any quoted string. Returns a list of words.

**execute**(*func*, *args*[*, *msg=None*, *verbose=0*, *dry_run=0]*)

Perform some action that affects the outside world (for instance, writing to the filesystem). Such actions are special because they are disabled by the `dry_run` flag. This method takes care of all that bureaucracy for you; all you have to do is supply the function to call and an argument tuple for it (to embody the "external action" being performed), and an optional message to print.

**strtobool**(*val*)

Convert a string representation of truth to true (1) or false (0).

True values are *y*, *yes*, *t*, *true*, *on* and 1; false values are *n*, *no*, *f*, *false*, *off* and 0. Raises *ValueError* if *val* is anything else.

**byte_compile**(*py_files*[*, *optimize=0*, *force=0*, *prefix=None*, *base_dir=None*, *verbose=1*, *dry_run=0*, *direct=None]*)

Byte-compile a collection of Python source files to either .pyc or .pyo files in the same directory. *py_files* is a list of files to compile; any files that don't end in .py are silently skipped. optimize must be one of the following:

- 0 - don't optimize (generate .pyc)
- 1 - normal optimization (like "python -O")
- 2 - extra optimization (like "python -OO")

If force is true, all files are recompiled regardless of timestamps.

The source filename encoded in each bytecode file defaults to the filenames listed in *py_files*; you can modify these with prefix and basedir. prefix is a string that will be stripped off of each source filename, and basedir is a directory name that will be prepended (after prefix is stripped). You can supply either or both (or neither) of prefix and basedir, as you wish.

If dry_run is true, doesn't actually do anything that would affect the filesystem.
Byte-compilation is either done directly in this interpreter process with the standard `py_compile` module, or indirectly by writing a temporary script and executing it. Normally, you should let `byte_compile()` figure out to use direct compilation or not (see the source for details). The `direct` flag is used by the script generated in indirect mode; unless you know what you're doing, leave it set to `None`.

**rfc822_escape( header)**

Return a version of `header` escaped for inclusion in an RFC 822 header, by ensuring there are 8 spaces space after each newline. Note that it does no other modification of the string.
11.14 distutils.dist -- The Distribution class

This module provides the Distribution class, which represents the module distribution being built/installed/distributed.
11.15 distutils.extension -- The Extension class

This module provides the Extension class, used to describe C/C++ extension modules in setup scripts.
11.16 distutils.debug -- Distutils debug mode

This module provides the DEBUG flag.
11.17 distutils.errors -- Distutils exceptions

Provides exceptions used by the Distutils modules. Note that Distutils modules may raise standard exceptions; in particular, SystemExit is usually raised for errors that are obviously the end-user's fault (eg. bad command-line arguments).

This module is safe to use in "from ... import *" mode; it only exports symbols whose names start with Distutils and end with Error.

Distributing Python Modules

Previous: 11.16 distutils.debug  Up: 11. API Reference  Next: 11.18 distutils.fancy_getopt

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.18 distutils.fancy_getopt --
Wrapper around the standard getopt module

This module provides a wrapper around the standard getopt module that provides the following additional features:

- short and long options are tied together
- options have help strings, so `fancy_getopt` could potentially create a complete usage summary
- options set attributes of a passed-in object
- boolean options can have ``negative aliases'' -- eg. if `--quiet` is the ``negative alias'' of `--verbose`, then `--quiet` on the command line sets `verbose` to false.

** Should be replaced with optik (which is also now known as optrparase in Python 2.3 and later). **

`fancy_getopt(options, negative_opt, object, args)`
Wrapper function. `options` is a list of ``(long_option, short_option, help_string)`` 3-tuples as described in the constructor for FancyGetopt. `negative_opt` should be a dictionary mapping option names to option names, both the key and value should be in the `options` list. `object` is an object which will be used to store values (see the getopt() method of the FancyGetopt class). `args` is the argument list. Will use `sys.argv[1:]` if you pass `None` as `args`.

`wrap_text(text, width)`
Wraps `text` to less than `width` wide.

**Warning:** Should be replaced with textwrap (which is available in Python 2.3 and later).

`class FancyGetopt([option_table=None])`
The option_table is a list of 3-tuples: "(long_option, short_option, help_string)"

If an option takes an argument, its long_option should have '=' appended; short_option should just be a single character, no ':' in any case. short_option should be None if a long_option doesn't have a corresponding short_option. All option tuples must have long options.

The FancyGetopt class provides the following methods:

getopt([args=None, object=None])
Parse command-line options in args. Store as attributes on object.

If args is None or not supplied, uses sys.argv[1:]. If object is None or not supplied, creates a new OptionDummy instance, stores option values there, and returns a tuple "(args, object)". If object is supplied, it is modified in place and getopt() just returns args; in both cases, the returned args is a modified copy of the passed-in args list, which is left untouched.

get_option_order()
Returns the list of "(option, value)" tuples processed by the previous run of getopt() Raises RuntimeError if getopt() hasn't been called yet.

generate_help([header=None])
Generate help text (a list of strings, one per suggested line of output) from the option table for this FancyGetopt object.

If supplied, prints the supplied header at the top of the help.
11.19 distutils.filelist -- The FileList class

This module provides the FileList class, used for poking about the filesystem and building lists of files.
11.20 distutils.log -- Simple PEP 282-style logging

**Warning:** Should be replaced with standard **logging** module.

Release 2.5, documentation updated on 19th September, 2006.
See [About this document...](...) for information on suggesting changes.
11.21 distutils.spawn -- Spawn a sub-process

This module provides the `spawn()` function, a front-end to various platform-specific functions for launching another program in a sub-process. Also provides `find_executable()` to search the path for a given executable name.
11.22 distutils.sysconfig -- System configuration information

The distutils.sysconfig module provides access to Python's low-level configuration information. The specific configuration variables available depend heavily on the platform and configuration. The specific variables depend on the build process for the specific version of Python being run; the variables are those found in the Makefile and configuration header that are installed with Python on UNIX systems. The configuration header is called pyconfig.h for Python versions starting with 2.2, and config.h for earlier versions of Python.

Some additional functions are provided which perform some useful manipulations for other parts of the distutils package.

**PREFIX**

The result of os.path.normpath(sys.prefix).

**EXEC_PREFIX**

The result of os.path.normpath(sys.exec_prefix).

**get_config_var(name)**

Return the value of a single variable. This is equivalent to get_config_vars().get(name).

**get_config_vars(…)**

Return a set of variable definitions. If there are no arguments, this returns a dictionary mapping names of configuration variables to values. If arguments are provided, they should be strings, and the return value will be a sequence giving the associated values. If a given name does not have a corresponding value, None will be included for that variable.

**get_config_h_filename()**

Return the full path name of the configuration header. For UNIX, this will be the header generated by the configure script; for other platforms the header will have been supplied directly by the Python source distribution. The file
is a platform-specific text file.

**get_makefile_filename()**

Return the full path name of the Makefile used to build Python. For UNIX, this will be a file generated by the `configure` script; the meaning for other platforms will vary. The file is a platform-specific text file, if it exists. This function is only useful on POSIX platforms.

**get_python_inc([plat_specific[, prefix]])**

Return the directory for either the general or platform-dependent C include files. If `plat_specific` is true, the platform-dependent include directory is returned; if false or omitted, the platform-independent directory is returned. If `prefix` is given, it is used as either the prefix instead of `PREFIX`, or as the exec-prefix instead of `EXEC_PREFIX` if `plat_specific` is true.

**get_python_lib([plat_specific[, standard_lib[, prefix]]])**

Return the directory for either the general or platform-dependent library installation. If `plat_specific` is true, the platform-dependent include directory is returned; if false or omitted, the platform-independent directory is returned. If `prefix` is given, it is used as either the prefix instead of `PREFIX`, or as the exec-prefix instead of `EXEC_PREFIX` if `plat_specific` is true. If `standard_lib` is true, the directory for the standard library is returned rather than the directory for the installation of third-party extensions.

The following function is only intended for use within the `distutils` package.

**customize_compiler(compiler)**

Do any platform-specific customization of a `distutils.ccompiler.CCompiler` instance.

This function is only needed on UNIX at this time, but should be called consistently to support forward-compatibility. It inserts the information that varies across UNIX flavors and is stored in Python's `Makefile`. This information includes the selected compiler, compiler and linker options, and the extension used by the linker for shared objects.

This function is even more special-purpose, and should only be used from
Python's own build procedures.

**set_python_build()**

Inform the `distutils.sysconfig` module that it is being used as part of the build process for Python. This changes a lot of relative locations for files, allowing them to be located in the build area rather than in an installed Python.
11.23 distutils.text_file -- The TextFile class

This module provides the TextFile class, which gives an interface to text files that (optionally) takes care of stripping comments, ignoring blank lines, and joining lines with backslashes.

**class TextFile([filename=None, file=None, **options])**

This class provides a file-like object that takes care of all the things you commonly want to do when processing a text file that has some line-by-line syntax: strip comments (as long as # is your comment character), skip blank lines, join adjacent lines by escaping the newline (ie. backslash at end of line), strip leading and/or trailing whitespace. All of these are optional and independently controllable.

The class provides a `warn()` method so you can generate warning messages that report physical line number, even if the logical line in question spans multiple physical lines. Also provides `unreadline()` for implementing line-at-a-time lookahead.

TextFile instances are create with either `filename`, `file`, or both. `RuntimeError` is raised if both are `None`. `filename` should be a string, and `file` a file object (or something that provides `readline()` and `close()` methods). It is recommended that you supply at least `filename`, so that TextFile can include it in warning messages. If `file` is not supplied, TextFile creates its own using the `open()` built-in function.

The options are all boolean, and affect the values returned by `readline()`

<table>
<thead>
<tr>
<th>option name</th>
<th>description</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>strip from # to end-of-line, as well as any whitespace leading up to the #--unless it is escaped by a backslash</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>strip leading whitespace from each line before returning it</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>strip trailing whitespace (including line terminator!) from each line before returning it.</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>skip lines that are empty <em>after</em> stripping comments and whitespace. (If both lstrip_ws and rstrip_ws are false, then some lines may consist of solely whitespace: these will <em>not</em> be skipped, even if skip_blanks is true.)</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>if a backslash is the last non-newline character on a line after stripping comments and whitespace, join the following line to it to form one logical line; if N consecutive lines end with a backslash, then N+1 physical lines will be joined to form one logical line.</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>strip leading whitespace from lines that are joined to their predecessor; only matters if &quot;(join_lines and not lstrip_ws)&quot;</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

Note that since rstrip_ws can strip the trailing newline, the semantics of readline() must differ from those of the built-in file object's readline() method! In particular, readline() returns None for end-of-file: an empty string might just be a blank line (or an all-whitespace line), if rstrip_ws is true but skip_blanks is not.

**open( filename )**

Open a new file filename. This overrides any file or filename constructor arguments.

**close()**

Close the current file and forget everything we know about it (including the filename and the current line number).

**warn( msg[, line=None] )**

Print (to stderr) a warning message tied to the current logical line in the current file. If the current logical line in the file spans multiple physical lines, the warning refers to the whole range, such as ":lines 3-5". If line is supplied, it overrides the current line
number; it may be a list or tuple to indicate a range of physical lines, or an integer for a single physical line.

**readline()**

Read and return a single logical line from the current file (or from an internal buffer if lines have previously been ``unread'' with unreadline()). If the join_lines option is true, this may involve reading multiple physical lines concatenated into a single string. Updates the current line number, so calling warn() after readline() emits a warning about the physical line(s) just read. Returns None on end-of-file, since the empty string can occur if rstrip_ws is true but strip_blanks is not.

**readlines()**

Read and return the list of all logical lines remaining in the current file. This updates the current line number to the last line of the file.

**unreadline( line)**

Push line (a string) onto an internal buffer that will be checked by future readline() calls. Handy for implementing a parser with line-at-a-time lookahead. Note that lines that are ``unread'' with unreadline are not subsequently re-cleansed (whitespace stripped, or whatever) when read with readline. If multiple calls are made to unreadline before a call to readline, the lines will be returned most in most recent first order.
11.24 distutils.version --
Version number classes

Distributing Python Modules

Previous: 11.23 distutils.text_file  Up: 11. API Reference  Next: 11.25 distutils.cmd

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.25 distutils.cmd -- Abstract base class for Distutils commands

This module supplies the abstract base class Command.

```python
class Command(dist)
    Abstract base class for defining command classes, the ``worker bees'' of the Distutils. A useful analogy for command classes is to think of them as subroutines with local variables called options. The options are declared in initialize_options() and defined (given their final values) in finalize_options(), both of which must be defined by every command class. The distinction between the two is necessary because option values might come from the outside world (command line, config file, ...), and any options dependent on other options must be computed after these outside influences have been processed -- hence finalize_options(). The body of the subroutine, where it does all its work based on the values of its options, is the run() method, which must also be implemented by every command class.

The class constructor takes a single argument dist, a Distribution instance.
```
11.26 distutils.command --
Individual Distutils commands

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.27 distutils.command.bdist --
Build a binary installer

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.28

distutils.command.bdist_packager

-- Abstract base class for packagers

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.29

distutils.command.bdist_dumb - - Build a `dumb'' installer

Distributing Python Modules

Previous: 11.28 distutils.command.bdist_packager  Up: 11. API Reference
Next: 11.30 distutils.command.bdist_rpm

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.30
distutils.command.bdist_rpm --
Build a binary distribution as a
Redhat RPM and SRPM

Distributing Python Modules

Previous: 11.29 distutils.command.bdist_dumb  Up: 11. API Reference  Next: 11.31 distutils.command.bdist_wininst

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.31
distutils.command.bdist_wininst
-- Build a Windows installer

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.32 distutils.command.sdist -- Build a source distribution

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.33 distutils.command.build --
Build all files of a package

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.34

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

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.35
distutils.command.build_ext --
Build any extensions in a package

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.36
distutils.command.build_py --
Build the .py/.pyc files of a package

Distributing Python Modules
Previous: 11.35 distutils.command.build_ext Up: 11. API Reference Next:
11.37 distutils.command.build_scripts

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.37
distutils.command.build_scripts
-- Build the scripts of a package

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.38 distutils.command.clean -- Clean a package build area

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.39 distutils.command.config
-- Perform package configuration

---

Distributing Python Modules

Previous: 11.38 distutils.command.clean  Up: 11. API Reference  Next: 11.40 distutils.command.install

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.40
distutils.command.install -- Install a package

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.41
distutils.command.install_data
-- Install data files from a package

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.42
distutils.command.install_headers
-- Install C/C++ header files from a package

Distributing Python Modules
Previous: 11.41 distutils.command.install_data Up: 11. API Reference Next: 11.43 distutils.command.install_lib

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.43
```
distutils.command.install_lib
-- Install library files from a package
```

Distributing Python Modules

Previous: 11.42 distutils.command.install_headers  Up: 11. API Reference
Next: 11.44 distutils.command.install_scripts

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.44

distutils.command.install_scr:
-- Install script files from a package

Distributing Python Modules

Previous: 11.43 distutils.command.install_lib Up: 11. API Reference Next:
11.45 distutils.command.register

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.45
distutils.command.register --
Register a module with the Python Package Index

The register command registers the package with the Python Package Index. This is described in more detail in PEP 301.

Distributing Python Modules

Previous: 11.44 distutils.command.install_scripts  Up: 11. API Reference  Next: 11.46 Creating a new

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
11.46 Creating a new Distutils command

This section outlines the steps to create a new Distutils command.

A new command lives in a module in the distutils.command package. There is a sample template in that directory called command_template. Copy this file to a new module with the same name as the new command you're implementing. This module should implement a class with the same name as the module (and the command). So, for instance, to create the command peel_banana (so that users can run "setup.py peel_banana"), you'd copy command_template to distutils/command/peel_banana.py, then edit it so that it's implementing the class peel_banana, a subclass of distutils.cmd.Command.

Subclasses of Command must define the following methods.

initialize_options()

Set default values for all the options that this command supports. Note that these defaults may be overridden by other commands, by the setup script, by config files, or by the command-line. Thus, this is not the place to code dependencies between options; generally, initialize_options() implementations are just a bunch of "self.foo = None" assignments.

finalize_options()

Set final values for all the options that this command supports. This is always called as late as possible, i.e. after any option assignments from the command-line or from other commands have been done. Thus, this is the place to to code option dependencies: if foo depends on bar, then it is safe to set foo from bar as long as foo still has the same value it was assigned in initialize_options().

run()

A command's raison d'être: carry out the action it exists to perform, controlled by the options initialized in initialize_options(),
customized by other commands, the setup script, the command-line, and config files, and finalized in `finalize_options()`. All terminal output and filesystem interaction should be done by `run()`.

`sub_commands` formalizes the notion of a ``family'' of commands, eg. `install` as the parent with sub-commands `install_lib`, `install_headers`, etc. The parent of a family of commands defines `sub_commands` as a class attribute; it's a list of 2-tuples "(command_name, predicate)", with `command_name` a string and `predicate` an unbound method, a string or None. `predicate` is a method of the parent command that determines whether the corresponding command is applicable in the current situation. (Eg. we `install_headers` is only applicable if we have any C header files to install.) If `predicate` is None, that command is always applicable.

`sub_commands` is usually defined at the *end* of a class, because predicates can be unbound methods, so they must already have been defined. The canonical example is the `install` command.
Index

abcdefghijklmnopqrstuvwxyz
add_include_dir() (CCompiler method)
add_library() (CCompiler method)
add_library_dir() (CCompiler method)
add_link_object() (CCompiler method)
add_runtime_library_dir() (CCompiler method)
announce() (CCompiler method)
byte_compile() (in module distutils.util)
C

CCompiler (class in distutils.ccompiler)
change_root() (in module distutils.util)
check_environ() (in module distutils.util)
close() (TextFile method)
Command (class in distutils.cmd)
Command (class in distutils.core)
compile() (CCompiler method)

convert_path() (in module distutils.util)
copy_file() (in module distutils.file_util)
copy_tree() (in module distutils.dir_util)
customize_compiler() (in module distutils.sysconfig)
create_shortcut()
create_static_lib() (CCompiler method)
create_tree() (in module distutils.dir_util)
debug_print() (CCompiler method)
declare_macro() (CCompiler method)
detect_language() (CCompiler method)
directory_created()
Distutils (class in distutils.core)
distutils.archive_util (standard module)
distutils.bcppcompiler (standard module)
distutils.ccompiler (standard module)
distutils.cmd (standard module)
distutils.command (standard module)
distutils.command.bdist (standard module)
distutils.command.bdist_dumb (standard module)
distutils.command.bdist_packager (standard module)
distutils.command.bdist_rpm (standard module)
distutils.command.bdist_wininst (standard module)
distutils.command.build (standard module)
distutils.command.build_clib (standard module)
distutils.command.build_ext (standard module)
distutils.command.build_py (standard module)
distutils.command.build_scripts (standard module)
distutils.command.clean (standard module)
distutils.command.config (standard module)
distutils.command.install_lib (standard module)
distutils.command.install_scripts (standard module)
distutils.command.register (standard module)
distutils.command.sdist (standard module)
distutils.core (standard module)
distutils.cygwinccompiler (standard module)
distutils.debug (standard module)
distutils.dep_util (standard module)
distutils.dir_util (standard module)
distutils.dist (standard module)
distutils.emxccompiler (standard module)
distutils.errors (standard module)
distutils.extension (standard module)
distutils.fancy_getopt (standard module)
distutils.file_util (standard module)
distutils.filelist (standard module)
distutils.log (standard module)
distutils.msvccompiler (standard module)
distutils.mwerkscompiler (standard module)
distutils.spawn (standard module)
distutils.sysconfig (standard module)
distutils.text_file (standard module)
module)
distutils.command.install (standard module)
distutils.command.install_data (standard module)
distutils.command.install_headers (standard module)
distutils.unixccompiler (standard module)
distutils.util (standard module)
distutils.version (standard module)
**E**

Environment variables

- HOME
- PATH
- PLAT
- EXEC_PREFIX (in module distutils.sysconfig)

Methods:

- executable_filename() (CCompiler method)
- execute() (CCompiler method)
- execute() (in module distutils.util)
- Extension (class in distutils.core)
fancy_getopt() (in module distutils.fancy_getopt)
FancyGetopt (class in distutils.fancy_getopt)
file_created()

finalize_options() (Command method)
find_library_file() (CCompiler method)
gen_lib_options() (in module distutils.ccompiler)
gen_preprocess_options() (in module distutils.ccompiler)
generate_help() (FancyGetopt method)
get_config_h_filename() (in module distutils.sysconfig)
get_config_var() (in module distutils.sysconfig)
get_config_vars() (in module distutils.sysconfig)
get_default_compiler() (in module distutils.ccompiler)
get_makefile_filename() (in module distutils.sysconfig)

get_option_order() (FancyGetopt method)
get_platform() (in module distutils.util)
get_python_inc() (in module distutils.sysconfig)
get_python_lib() (in module distutils.sysconfig)
get_special_folder_path()
getopt() (FancyGetopt method)
grok_environment_error() (in module distutils.util)
H

has_function() (CCompiler method)    HOME (environment variable)
initialize_options() (Command method)
make_archive() (in module distutils.archive_util)
make_tarball() (in module distutils.archive_util)
make_zipfile() (in module distutils.archive_util)
mkpath() (CCompiler method)
mkpath() (in module distutils.dir_util)
move_file() (CCompiler method)
mv_file() (in module distutils.file_util)
new_compiler() (in module distutils.ccompiler)
newer() (in module distutils.dep_util)
newer_group() (in module distutils.dep_util)
newer_pairwise() (in module distutils.dep_util)
object_filenames() (CCompiler method)     open() (TextFile method)
PATH (environment variable)  PREFIX (in module distutils.sysconfig)
PLAT (environment variable)  preprocess() (CCompiler method)
R

readline() (TextFile method)
readlines() (TextFile method)
remove_tree() (in module distutils.dir_util)
rfc822_escape() (in module distutils.util)
run() (Command method)
run_setup() (in module distutils.core)
runtime_library_dir_option() (CCompiler method)
set_executables() (CCompiler method)
set_include_dirs() (CCompiler method)
set_libraries() (CCompiler method)
set_library_dirs() (CCompiler method)
set_link_objects() (CCompiler method)
set_python_build() (in module distutils.sysconfig)
set_runtime_library_dirs() (CCompiler method)
setup() (in module distutils.core)
shared_object_filename() (CCompiler method)
show_compilers() (in module distutils.ccompiler)
spawn() (CCompiler method)
split_quoted() (in module distutils.util)
strtobool() (in module distutils.util)
subst_vars() (in module distutils.util)
TextFile (class in distutils.text_file)
U

**undefined_macro() (CCompiler method)**  **unreadline() (TextFile method)**
WARN

warn() (CCompiler method)
warn() (TextFile method)

WRAP TEXT() (in module
distutils.fancy_getopt)
write_file() (in module
distutils.file_util)

Distributing Python Modules

Previous: Module Index  Up: Distributing Python Modules  Next: About this
document ...
What's New in Python 2.5

A.M. Kuchling

Contents

1 PEP 308: Conditional Expressions
2 PEP 309: Partial Function Application
3 PEP 314: Metadata for Python Software Packages v1.1
4 PEP 328: Absolute and Relative Imports
5 PEP 338: Executing Modules as Scripts
6 PEP 341: Unified try/except/finally
7 PEP 342: New Generator Features
8 PEP 343: The 'with' statement
   8.1 Writing Context Managers
   8.2 The contextlib module
9 PEP 352: Exceptions as New-Style Classes
10 PEP 353: Using ssize_t as the index type
11 PEP 357: The '__index__' method
12 Other Language Changes
   12.1 Interactive Interpreter Changes
   12.2 Optimizations
13 New, Improved, and Removed Modules
   13.1 The ctypes package
   13.2 The ElementTree package
   13.3 The hashlib package
   13.4 The sqlite3 package
   13.5 The wsgiref package
14 Build and C API Changes
   14.1 Port-Specific Changes
15 Porting to Python 2.5
16 Acknowledgements
About this document...
Release 1.0.
See About this document... for information on suggesting changes.
This article explains the new features in Python 2.5. The final release of Python 2.5 is scheduled for August 2006; PEP 356 describes the planned release schedule.

The changes in Python 2.5 are an interesting mix of language and library improvements. The library enhancements will be more important to Python's user community, I think, because several widely-useful packages were added. New modules include ElementTree for XML processing (section 13.2), the
SQLite database module (section 13.4), and the ctypes module for calling C functions (section 13.1).

The language changes are of middling significance. Some pleasant new features were added, but most of them aren't features that you'll use every day. Conditional expressions were finally added to the language using a novel syntax; see section 1. The new 'with' statement will make writing cleanup code easier (section 8). Values can now be passed into generators (section 7). Imports are now visible as either absolute or relative (section 4). Some corner cases of exception handling are handled better (section 6). All these improvements are worthwhile, but they're improvements to one specific language feature or another; none of them are broad modifications to Python's semantics.

As well as the language and library additions, other improvements and bugfixes were made throughout the source tree. A search through the SVN change logs finds there were 353 patches applied and 458 bugs fixed between Python 2.4 and 2.5. (Both figures are likely to be underestimates.)

This article doesn't try to be a complete specification of the new features; instead changes are briefly introduced using helpful examples. For full details, you should always refer to the documentation for Python 2.5 at http://docs.python.org. If you want to understand the complete implementation and design rationale, refer to the PEP for a particular new feature.

Comments, suggestions, and error reports for this document are welcome; please e-mail them to the author or open a bug in the Python bug tracker.

What's New in Python 2.5

Release 1.0.
See About this document... for information on suggesting changes.
About this document ...

What's New in Python 2.5

This document was generated using the LaTeX2HTML translator.


The application of LaTeX2HTML to the Python documentation has been heavily tailored by Fred L. Drake, Jr. Original navigation icons were contributed by Christopher Petrilli.
Comments and Questions

General comments and questions regarding this document should be sent by email to docs@python.org. If you find specific errors in this document, either in the content or the presentation, please report the bug at the Python Bug Tracker at SourceForge. If you are able to provide suggested text, either to replace existing incorrect or unclear material, or additional text to supplement what's already available, we'd appreciate the contribution. There's no need to worry about text markup; our documentation team will gladly take care of that.

Questions regarding how to use the information in this document should be sent to the Python news group, comp.lang.python, or the Python mailing list (which is gated to the newsgroup and carries the same content).

For any of these channels, please be sure not to send HTML email. Thanks.
16 Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Georg Brandl, Nick Coghlan, Phillip J. Eby, Lars Gustäbel, Raymond Hettinger, Ralf W. Grosse-Kunstleve, Kent Johnson, Iain Lowe, Martin von Löwis, Fredrik Lundh, Andrew McNamara, Skip Montanaro, Gustavo Niemeyer, Paul Prescod, James Pryor, Mike Rovner, Scott Weikart, Barry Warsaw, Thomas Wouters.
Abstract:

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, http://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.

This tutorial introduces the reader informally to the basic concepts and features of the Python language and system. It helps to have a Python interpreter handy for hands-on experience, but all examples are self-contained, so the tutorial can be read off-line as well.

For a description of standard objects and modules, see the Python Library.
Reference document. The Python Reference Manual gives a more formal definition of the language. To write extensions in C or C++, read Extending and Embedding the Python Interpreter and Python/C API Reference. There are also several books covering Python in depth.

This tutorial does not attempt to be comprehensive and cover every single feature, or even every commonly used feature. Instead, it introduces many of Python's most noteworthy features, and will give you a good idea of the language's flavor and style. After reading it, you will be able to read and write Python modules and programs, and you will be ready to learn more about the various Python library modules described in the Python Library Reference.
Contents

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

6. Modules
6.1 More on Modules
   6.1.1 The Module Search Path
   6.1.2 `Compiled" Python files
6.2 Standard Modules
6.3 The dir() Function
6.4 Packages
   6.4.1 Importing * From a Package
   6.4.2 Intra-package References
   6.4.3 Packages in Multiple Directories

7. Input and Output
7.1 Fancier Output Formatting
7.2 Reading and Writing Files
   7.2.1 Methods of File Objects
   7.2.2 The pickle Module

8. Errors and Exceptions
8.1 Syntax Errors
8.2 Exceptions
8.3 Handling Exceptions
8.4 Raising Exceptions
8.5 User-defined Exceptions
8.6 Defining Clean-up Actions
8.7 Predefined Clean-up Actions

9. Classes
9.1 A Word About Terminology
9.2 Python Scopes and Name Spaces
9.3 A First Look at Classes
   9.3.1 Class Definition Syntax
   9.3.2 Class Objects
   9.3.3 Instance Objects
9.3.4 Method Objects
9.4 Random Remarks
9.5 Inheritance
  9.5.1 Multiple Inheritance
9.6 Private Variables
9.7 Odds and Ends
9.8 Exceptions Are Classes Too
9.9 Iterators
9.10 Generators
9.11 Generator Expressions

10. Brief Tour of the Standard Library
  10.1 Operating System Interface
  10.2 File Wildcards
  10.3 Command Line Arguments
  10.4 Error Output Redirection and Program Termination
  10.5 String Pattern Matching
  10.6 Mathematics
  10.7 Internet Access
  10.8 Dates and Times
  10.9 Data Compression
  10.10 Performance Measurement
  10.11 Quality Control
  10.12 Batteries Included

11. Brief Tour of the Standard Library - Part II
  11.1 Output Formatting
  11.2 Templating
  11.3 Working with Binary Data Record Layouts
  11.4 Multi-threading
  11.5 Logging
  11.6 Weak References
  11.7 Tools for Working with Lists
  11.8 Decimal Floating Point Arithmetic

12. What Now?
  A. Interactive Input Editing and History Substitution
    A.1 Line Editing
    A.2 History Substitution
    A.3 Key Bindings
    A.4 Commentary
  B. Floating Point Arithmetic: Issues and Limitations
B.1 Representation Error

C. History and License
- C.1 History of the software
- C.2 Terms and conditions for accessing or otherwise using Python
- C.3 Licenses and Acknowledgements for Incorporated Software
  - C.3.1 Mersenne Twister
  - C.3.2 Sockets
  - C.3.3 Floating point exception control
  - C.3.4 MD5 message digest algorithm
  - C.3.5 Asynchronous socket services
  - C.3.6 Cookie management
  - C.3.7 Profiling
  - C.3.8 Execution tracing
  - C.3.9 UUencode and UUdecode functions
  - C.3.10 XML Remote Procedure Calls

- D. Glossary
- Index

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
About this document ...

Python Tutorial, 19th September, 2006, Release 2.5

This document was generated using the LaTeX2HTML translator.


The application of LaTeX2HTML to the Python documentation has been heavily tailored by Fred L. Drake, Jr. Original navigation icons were contributed by Christopher Petrilli.
Comments and Questions

General comments and questions regarding this document should be sent by email to docs@python.org. If you find specific errors in this document, either in the content or the presentation, please report the bug at the Python Bug Tracker at SourceForge. If you are able to provide suggested text, either to replace existing incorrect or unclear material, or additional text to supplement what's already available, we'd appreciate the contribution. There's no need to worry about text markup; our documentation team will gladly take care of that.

Questions regarding how to use the information in this document should be sent to the Python news group, comp.lang.python, or the Python mailing list (which is gated to the newsgroup and carries the same content).

For any of these channels, please be sure not to send HTML email. Thanks.
About this document ...

Python Tutorial, 19th September, 2006, Release 2.5

This document was generated using the LaTeX2HTML translator.


The application of LaTeX2HTML to the Python documentation has been heavily tailored by Fred L. Drake, Jr. Original navigation icons were contributed by Christopher Petrilli.
Comments and Questions

General comments and questions regarding this document should be sent by email to docs@python.org. If you find specific errors in this document, either in the content or the presentation, please report the bug at the Python Bug Tracker at SourceForge. If you are able to provide suggested text, either to replace existing incorrect or unclear material, or additional text to supplement what's already available, we'd appreciate the contribution. There's no need to worry about text markup; our documentation team will gladly take care of that.

Questions regarding how to use the information in this document should be sent to the Python news group, comp.lang.python, or the Python mailing list (which is gated to the newsgroup and carries the same content).

For any of these channels, please be sure not to send HTML email. Thanks.
Abstract:

Python is an extensible, interpreted, object-oriented programming language. It supports a wide range of applications, from simple text processing scripts to interactive Web browsers.

While the Python Reference Manual describes the exact syntax and semantics of the language, it does not describe the standard library that is distributed with the language, and which greatly enhances its immediate usability. This library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs.

This library reference manual documents Python's standard library, as well as many optional library modules (which may or may not be available, depending on whether the underlying platform supports them and on the configuration choices made at compile time). It also documents the standard types of the language and its built-in functions and exceptions, many of which are not or incompletely documented in the Reference Manual.

This manual assumes basic knowledge about the Python language. For an
informal introduction to Python, see the Python Tutorial; the Python Reference Manual remains the highest authority on syntactic and semantic questions. Finally, the manual entitled Extending and Embedding the Python Interpreter describes how to add new extensions to Python and how to embed it in other applications.
Contents

• 1. Introduction
• 2. Built-in Objects
  ○ 2.1 Built-in Functions
  ○ 2.2 Non-essential Built-in Functions
  ○ 2.3 Built-in Exceptions
  ○ 2.4 Built-in Constants
• 3. Built-in Types
  ○ 3.1 Truth Value Testing
  ○ 3.2 Boolean Operations -- and, or, not
  ○ 3.3 Comparisons
  ○ 3.4 Numeric Types -- int, float, long, complex
    ■ 3.4.1 Bit-string Operations on Integer Types
  ○ 3.5 Iterator Types
  ○ 3.6 Sequence Types -- str, unicode, list, tuple, buffer, xrange
    ■ 3.6.1 String Methods
    ■ 3.6.2 String Formatting Operations
    ■ 3.6.3 XRange Type
    ■ 3.6.4 Mutable Sequence Types
  ○ 3.7 Set Types -- set, frozenset
  ○ 3.8 Mapping Types -- dict
  ○ 3.9 File Objects
  ○ 3.10 Context Manager Types
  ○ 3.11 Other Built-in Types
    ■ 3.11.1 Modules
    ■ 3.11.2 Classes and Class Instances
    ■ 3.11.3 Functions
    ■ 3.11.4 Methods
    ■ 3.11.5 Code Objects
    ■ 3.11.6 Type Objects
    ■ 3.11.7 The Null Object
    ■ 3.11.8 The Ellipsis Object
    ■ 3.11.9 Boolean Values
    ■ 3.11.10 Internal Objects
  ○ 3.12 Special Attributes
• 4. String Services
4.1 string -- Common string operations
   4.1.1 String constants
   4.1.2 Template strings
   4.1.3 String functions
   4.1.4 Deprecated string functions
4.2 re -- Regular expression operations
   4.2.1 Regular Expression Syntax
   4.2.2 Matching vs Searching
   4.2.3 Module Contents
   4.2.4 Regular Expression Objects
   4.2.5 Match Objects
   4.2.6 Examples
4.3 struct -- Interpret strings as packed binary data
4.4 difflib -- Helpers for computing deltas
   4.4.1 SequenceMatcher Objects
   4.4.2 SequenceMatcher Examples
   4.4.3 Differ Objects
   4.4.4 Differ Example
4.5 StringIO -- Read and write strings as files
4.6 cStringIO -- Faster version of StringIO
4.7 textwrap -- Text wrapping and filling
4.8 codecs -- Codec registry and base classes
   4.8.1 Codec Base Classes
   4.8.2 Encodings and Unicode
   4.8.3 Standard Encodings
   4.8.4 encodings.idna -- Internationalized Domain Names in Applications
   4.8.5 encodings.utf_8_sig -- UTF-8 codec with BOM signature
4.9 unicodedata -- Unicode Database
4.10 stringprep -- Internet String Preparation
4.11 ffpformat -- Floating point conversions

5. Data Types
5.1 datetime -- Basic date and time types
   5.1.1 Available Types
   5.1.2 timedelta Objects
   5.1.3 date Objects
   5.1.4 datetime Objects
   5.1.5 time Objects
   5.1.6 tzinfo Objects
- 5.1.7 strftime() Behavior
- 5.1.8 Examples

- 5.2 calendar -- General calendar-related functions
- 5.3 collections -- High-performance container datatypes
  - 5.3.1 deque objects
  - 5.3.2 defaultdict objects
- 5.4 heapq -- Heap queue algorithm
  - 5.4.1 Theory
- 5.5 bisect -- Array bisection algorithm
  - 5.5.1 Examples
- 5.6 array -- Efficient arrays of numeric values
- 5.7 sets -- Unordered collections of unique elements
  - 5.7.1 Set Objects
  - 5.7.2 Example
  - 5.7.3 Protocol for automatic conversion to immutable
  - 5.7.4 Comparison to the built-in set types
- 5.8 sched -- Event scheduler
  - 5.8.1 Scheduler Objects
- 5.9 mutex -- Mutual exclusion support
  - 5.9.1 Mutex Objects
- 5.10 Queue -- A synchronized queue class
  - 5.10.1 Queue Objects
- 5.11 weakref -- Weak references
  - 5.11.1 Weak Reference Objects
  - 5.11.2 Example
- 5.12 UserDict -- Class wrapper for dictionary objects
- 5.13 UserList -- Class wrapper for list objects
- 5.14 UserString -- Class wrapper for string objects
- 5.15 types -- Names for built-in types
- 5.16 new -- Creation of runtime internal objects
- 5.17 copy -- Shallow and deep copy operations
- 5.18 pprint -- Data pretty printer
  - 5.18.1 PrettyPrinter Objects
- 5.19 repr -- Alternate repr() implementation
  - 5.19.1 Repr Objects
  - 5.19.2 Subclassing Repr Objects

- 6. Numeric and Mathematical Modules
  - 6.1 math -- Mathematical functions
  - 6.2 cmath -- Mathematical functions for complex numbers
6.3 decimal -- Decimal floating point arithmetic
   - 6.3.1 Quick-start Tutorial
   - 6.3.2 Decimal objects
   - 6.3.3 Context objects
   - 6.3.4 Signals
   - 6.3.5 Floating Point Notes
   - 6.3.6 Working with threads
   - 6.3.7 Recipes
   - 6.3.8 Decimal FAQ

6.4 random -- Generate pseudo-random numbers

6.5 itertools -- Functions creating iterators for efficient looping
   - 6.5.1 Itertool functions
   - 6.5.2 Examples
   - 6.5.3 Recipes

6.6 functools -- Higher order functions and operations on callable objects.
   - 6.6.1 partial Objects

6.7 operator -- Standard operators as functions.
   - 6.7.1 Mapping Operators to Functions

7. Internet Data Handling
   - 7.1 email -- An email and MIME handling package
     - 7.1.1 Representing an email message
     - 7.1.2 Parsing email messages
     - 7.1.3 Generating MIME documents
     - 7.1.4 Creating email and MIME objects from scratch
     - 7.1.5 Internationalized headers
     - 7.1.6 Representing character sets
     - 7.1.7 Encoders
     - 7.1.8 Exception and Defect classes
     - 7.1.9 Miscellaneous utilities
     - 7.1.10 Iterators
     - 7.1.11 Package History
     - 7.1.12 Differences from mimelib
     - 7.1.13 Examples
   - 7.2 mailcap -- Mailcap file handling.
   - 7.3 mailbox -- Manipulate mailboxes in various formats
     - 7.3.1 Mailbox objects
     - 7.3.2 Message objects
     - 7.3.3 Exceptions
7.3.4 Deprecated classes and methods
7.3.5 Examples

7.4 mhlib -- Access to MH mailboxes
  7.4.1 MH Objects
  7.4.2 Folder Objects
  7.4.3 Message Objects

7.5 mimeTools -- Tools for parsing MIME messages
  7.5.1 Additional Methods of Message Objects

7.6 mimeTypes -- Map filenames to MIME types
  7.6.1 MimeTypes Objects

7.7 MimeWriter -- Generic MIME file writer
  7.7.1 MimeWriter Objects

7.8 mimify -- MIME processing of mail messages

7.9 multiFile -- Support for files containing distinct parts
  7.9.1 MultiFile Objects
  7.9.2 MultiFile Example

7.10 rfc822 -- Parse RFC 2822 mail headers
  7.10.1 Message Objects
  7.10.2 AddressList Objects

7.11 base64 -- RFC 3548: Base16, Base32, Base64 Data Encodings

7.12 binhex -- Encode and decode binhex4 files
  7.12.1 Notes

7.13 binascii -- Convert between binary and ASCII

7.14 quopri -- Encode and decode MIME quoted-printable data

7.15 uu -- Encode and decode uuencode files

8. Structured Markup Processing Tools
  8.1 HTMLParser -- Simple HTML and XHTML parser
    8.1.1 Example HTML Parser Application
  8.2 sgmllib -- Simple SGML parser
  8.3 htmllib -- A parser for HTML documents
    8.3.1 HTMLParser Objects
  8.4 htmlentitydefs -- Definitions of HTML general entities
  8.5 xml.parsers.expat -- Fast XML parsing using Expat
    8.5.1 XMLParser Objects
    8.5.2 ExpatError Exceptions
    8.5.3 Example
    8.5.4 Content Model Descriptions
    8.5.5 Expat error constants
  8.6 xml.dom -- The Document Object Model API
8.6.1 Module Contents
8.6.2 Objects in the DOM
8.6.3 Conformance
8.7 xml.dom.minidom -- Lightweight DOM implementation
  8.7.1 DOM Objects
  8.7.2 DOM Example
  8.7.3 minidom and the DOM standard
8.8 xml.dom.pulldom -- Support for building partial DOM trees
  8.8.1 DOMEventStream Objects
8.9 xml.sax -- Support for SAX2 parsers
  8.9.1 SAXException Objects
8.10 xml.sax.handler -- Base classes for SAX handlers
  8.10.1 ContentHandler Objects
  8.10.2 DTDHandler Objects
  8.10.3 EntityResolver Objects
  8.10.4 ErrorHandler Objects
8.11 xml.sax.saxutils -- SAX Utilities
8.12 xml.sax.xmlreader -- Interface for XML parsers
  8.12.1 XMLReader Objects
  8.12.2 IncrementalParser Objects
  8.12.3 Locator Objects
  8.12.4 InputSource Objects
  8.12.5 The Attributes Interface
  8.12.6 The AttributesNS Interface
8.13 xml.etree.ElementTree -- The ElementTree XML API
  8.13.1 Functions
  8.13.2 ElementTree Objects
  8.13.3 QName Objects
  8.13.4 TreeBuilder Objects
  8.13.5 XMLTreeBuilder Objects
9. File Formats
  9.1 csv -- CSV File Reading and Writing
    9.1.1 Module Contents
    9.1.2 Dialects and Formatting Parameters
    9.1.3 Reader Objects
    9.1.4 Writer Objects
    9.1.5 Examples
  9.2 ConfigParser -- Configuration file parser
    9.2.1 RawConfigParser Objects
9.2.2 ConfigParser Objects
9.2.3 SafeConfigParser Objects
9.3 robotparser -- Parser for robots.txt
9.4 netrc -- netrc file processing
  9.4.1 netrc Objects
9.5 xdrlib -- Encode and decode XDR data
  9.5.1 Packer Objects
  9.5.2 Unpacker Objects
  9.5.3 Exceptions
10. Cryptographic Services
  10.1 hashlib -- Secure hashes and message digests
  10.2 hmac -- Keyed-Hashing for Message Authentication
  10.3 md5 -- MD5 message digest algorithm
  10.4 sha -- SHA-1 message digest algorithm
11. File and Directory Access
  11.1 os.path -- Common pathname manipulations
  11.2 fileinput -- Iterate over lines from multiple input streams
  11.3 stat -- Interpreting stat() results
  11.4 statvfs -- Constants used with os.statvfs()
  11.5 filecmp -- File and Directory Comparisons
    11.5.1 The dircmp class
  11.6 tempfile -- Generate temporary files and directories
  11.7 glob -- Unix style pathname pattern expansion
  11.8 fnmatch -- Unix filename pattern matching
  11.9 linecache -- Random access to text lines
  11.10 shutil -- High-level file operations
    11.10.1 Example
  11.11 dircache -- Cached directory listings
12. Data Compression and Archiving
  12.1 zlib -- Compression compatible with gzip
  12.2 gzip -- Support for gzip files
  12.3 bz2 -- Compression compatible with bzip2
    12.3.1 (De)compression of files
    12.3.2 Sequential (de)compression
    12.3.3 One-shot (de)compression
  12.4 zipfile -- Work with ZIP archives
    12.4.1 ZipFile Objects
    12.4.2 PyZipFile Objects
    12.4.3 ZipInfo Objects
12.5 tarfile -- Read and write tar archive files
   - 12.5.1 TarFile Objects
   - 12.5.2 TarInfo Objects
   - 12.5.3 Examples

13. Data Persistence
   - 13.1 pickle -- Python object serialization
     - 13.1.1 Relationship to other Python modules
     - 13.1.2 Data stream format
     - 13.1.3 Usage
     - 13.1.4 What can be pickled and unpickled?
     - 13.1.5 The pickle protocol
     - 13.1.6 Subclassing Unpicklers
     - 13.1.7 Example
   - 13.2 cPickle -- A faster pickle
   - 13.3 copy_reg -- Register pickle support functions
   - 13.4 shelve -- Python object persistence
     - 13.4.1 Restrictions
     - 13.4.2 Example
   - 13.5 marshal -- Internal Python object serialization
   - 13.6 anydbm -- Generic access to DBM-style databases
   - 13.7 whichdb -- Guess which DBM module created a database
   - 13.8 dbm -- Simple ``database'' interface
   - 13.9 gdbm -- GNU's reinterpretation of dbm
   - 13.10 dbhash -- DBM-style interface to the BSD database library
     - 13.10.1 Database Objects
   - 13.11 bsddb -- Interface to Berkeley DB library
     - 13.11.1 Hash, BTree and Record Objects
   - 13.12 dumbdbm -- Portable DBM implementation
     - 13.12.1 Dumbdbm Objects
   - 13.13 sqlite3 -- DB-API 2.0 interface for SQLite databases
     - 13.13.1 Module functions and constants
     - 13.13.2 Connection Objects
     - 13.13.3 Cursor Objects
     - 13.13.4 SQLite and Python types
     - 13.13.5 Controlling Transactions
     - 13.13.6 Using pysqlite efficiently

14. Generic Operating System Services
   - 14.1 os -- Miscellaneous operating system interfaces
     - 14.1.1 Process Parameters
14.1.2 File Object Creation
14.1.3 File Descriptor Operations
14.1.4 Files and Directories
14.1.5 Process Management
14.1.6 Miscellaneous System Information
14.1.7 Miscellaneous Functions

14.2 time -- Time access and conversions
14.3 optparse -- More powerful command line option parser
  14.3.1 Background
  14.3.2 Tutorial
  14.3.3 Reference Guide
  14.3.4 Option Callbacks
  14.3.5 Extending optparse

14.4 getopt -- Parser for command line options
14.5 logging -- Logging facility for Python
  14.5.1 Logger Objects
  14.5.2 Basic example
  14.5.3 Logging to multiple destinations
  14.5.4 Sending and receiving logging events across a network
  14.5.5 Handler Objects
  14.5.6 Formatter Objects
  14.5.7 Filter Objects
  14.5.8 LogRecord Objects
  14.5.9 Thread Safety
  14.5.10 Configuration

14.6 getpass -- Portable password input
14.7 curses -- Terminal handling for character-cell displays
  14.7.1 Functions
  14.7.2 Window Objects
  14.7.3 Constants

14.8 curses.textpad -- Text input widget for curses programs
  14.8.1 Textbox objects
14.9 curses.wrapper -- Terminal handler for curses programs
14.10 curses.ascii -- Utilities for ASCII characters
14.11 curses.panel -- A panel stack extension for curses.
  14.11.1 Functions
  14.11.2 Panel Objects

14.12 platform -- Access to underlying platform's identifying data
  14.12.1 Cross Platform
14.12.2 Java Platform
14.12.3 Windows Platform
14.12.4 Mac OS Platform
14.12.5 Unix Platforms

14.13 errno -- Standard errno system symbols
14.14 ctypes -- A foreign function library for Python.
  14.14.1 ctypes tutorial
  14.14.2 ctypes reference

15. Optional Operating System Services
  15.1 select -- Waiting for I/O completion
    15.1.1 Polling Objects
  15.2 thread -- Multiple threads of control
  15.3 threading -- Higher-level threading interface
    15.3.1 Lock Objects
    15.3.2 RLock Objects
    15.3.3 Condition Objects
    15.3.4 Semaphore Objects
    15.3.5 Event Objects
    15.3.6 Thread Objects
    15.3.7 Timer Objects
    15.3.8 Using locks, conditions, and semaphores in the with statement
  15.4 dummy_thread -- Drop-in replacement for the thread module
  15.5 dummy_threading -- Drop-in replacement for the threading module
  15.6 mmap -- Memory-mapped file support
  15.7 readline -- GNU readline interface
    15.7.1 Example
  15.8 rlcompleter -- Completion function for GNU readline
    15.8.1 Completer Objects

16. Unix Specific Services
  16.1 posix -- The most common POSIX system calls
    16.1.1 Large File Support
    16.1.2 Module Contents
  16.2 pwd -- The password database
  16.3 spwd -- The shadow password database
  16.4 grp -- The group database
  16.5 crypt -- Function to check Unix passwords
  16.6 dl -- Call C functions in shared objects
16.6.1 D1 Objects

16.7 termios -- POSIX style tty control
   16.7.1 Example

16.8 tty -- Terminal control functions

16.9 pty -- Pseudo-terminal utilities

16.10 fcntl -- The fcntl() and ioctl() system calls

16.11 pipes -- Interface to shell pipelines
   16.11.1 Template Objects

16.12 posixfile -- File-like objects with locking support

16.13 resource -- Resource usage information
   16.13.1 Resource Limits
   16.13.2 Resource Usage

16.14 nis -- Interface to Sun's NIS (Yellow Pages)

16.15 syslog -- Unix syslog library routines

16.16 commands -- Utilities for running commands

17. Interprocess Communication and Networking

17.1 subprocess -- Subprocess management
   17.1.1 Using the subprocess Module
   17.1.2 Popen Objects
   17.1.3 Replacing Older Functions with the subprocess Module

17.2 socket -- Low-level networking interface
   17.2.1 Socket Objects
   17.2.2 SSL Objects
   17.2.3 Example

17.3 signal -- Set handlers for asynchronous events
   17.3.1 Example

17.4 popen2 -- Subprocesses with accessible I/O streams
   17.4.1 Popen3 and Popen4 Objects
   17.4.2 Flow Control Issues

17.5 asyncore -- Asynchronous socket handler
   17.5.1 asyncore Example basic HTTP client

17.6 asynchat -- Asynchronous socket command/response handler
   17.6.1 asynchat - Auxiliary Classes and Functions
   17.6.2 asynchat Example

18. Internet Protocols and Support

18.1 webbrowser -- Convenient Web-browser controller
   18.1.1 Browser Controller Objects

18.2 cgi -- Common Gateway Interface support
   18.2.1 Introduction
18.2.2 Using the cgi module
18.2.3 Higher Level Interface
18.2.4 Old classes
18.2.5 Functions
18.2.6 Caring about security
18.2.7 Installing your CGI script on a Unix system
18.2.8 Testing your CGI script
18.2.9 Debugging CGI scripts
18.2.10 Common problems and solutions

18.3 cgitb -- Traceback manager for CGI scripts

18.4 wsgiref -- WSGI Utilities and Reference Implementation
18.4.1 wsgiref.util - WSGI environment utilities
18.4.2 wsgiref.headers - WSGI response header tools
18.4.3 wsgiref.simple_server - a simple WSGI HTTP server
18.4.4 wsgiref.validate - WSGI conformance checker
18.4.5 wsgiref.handlers - server/gateway base classes

18.5 urllib -- Open arbitrary resources by URL
18.5.1 URLopener Objects
18.5.2 Examples

18.6 urllib2 -- extensible library for opening URLs
18.6.1 Request Objects
18.6.2 OpenerDirector Objects
18.6.3 BaseHandler Objects
18.6.4 HTTPRedirectHandler Objects
18.6.5 HTTPCookieProcessor Objects
18.6.6 ProxyHandler Objects
18.6.7 HTTPPasswordMgr Objects
18.6.8 AbstractBasicAuthHandler Objects
18.6.9 HTTPBasicAuthHandler Objects
18.6.10 ProxyBasicAuthHandler Objects
18.6.11 AbstractDigestAuthHandler Objects
18.6.12 HTTPDigestAuthHandler Objects
18.6.13 ProxyDigestAuthHandler Objects
18.6.14 HTTPHandler Objects
18.6.15 HTTPSHandler Objects
18.6.16 FileHandler Objects
18.6.17 FTPHandler Objects
18.6.18 CacheFTPHandler Objects
18.6.19 GopherHandler Objects
18.6.20 UnknownHandler Objects
18.6.21 HTTPErrorProcessor Objects
18.6.22 Examples

18.7 httplib -- HTTP protocol client
18.7.1 HTTPConnection Objects
18.7.2 HTTPResponse Objects
18.7.3 Examples

18.8 ftplib -- FTP protocol client
18.8.1 FTP Objects

18.9 gopherlib -- Gopher protocol client

18.10 poplib -- POP3 protocol client
18.10.1 POP3 Objects
18.10.2 POP3 Example

18.11 imaplib -- IMAP4 protocol client
18.11.1 IMAP4 Objects
18.11.2 IMAP4 Example

18.12 nntplib -- NNTP protocol client
18.12.1 NNTP Objects

18.13 smtplib -- SMTP protocol client
18.13.1 SMTP Objects
18.13.2 SMTP Example

18.14 smtpd -- SMTP Server
18.14.1 SMTPServer Objects
18.14.2 DebuggingServer Objects
18.14.3 PureProxy Objects
18.14.4 MailmanProxy Objects

18.15 telnetlib -- Telnet client
18.15.1 Telnet Objects
18.15.2 Telnet Example

18.16 uuid -- UUID objects according to RFC 4122
18.16.1 Example

18.17 urlparse -- Parse URLs into components
18.17.1 Results of urlparse() and urlsplit()

18.18 SocketServer -- A framework for network servers
18.18.1 Server Creation Notes
18.18.2 Server Objects
18.18.3 RequestHandler Objects

18.19 BaseHTTPServer -- Basic HTTP server

18.20 SimpleHTTPServer -- Simple HTTP request handler
18.21 CGIHTTPServer -- CGI-capable HTTP request handler
18.22 cookielib -- Cookie handling for HTTP clients
  18.22.1 CookieJar and FileCookieJar Objects
  18.22.2 FileCookieJar subclasses and co-operation with web browsers
  18.22.3 CookiePolicy Objects
  18.22.4 DefaultCookiePolicy Objects
  18.22.5 Cookie Objects
  18.22.6 Examples
18.23 Cookie -- HTTP state management
  18.23.1 Cookie Objects
  18.23.2 Morsel Objects
  18.23.3 Example
18.24 xmlrpclib -- XML-RPC client access
  18.24.1 ServerProxy Objects
  18.24.2 Boolean Objects
  18.24.3 DateTime Objects
  18.24.4 Binary Objects
  18.24.5 Fault Objects
  18.24.6 ProtocolError Objects
  18.24.7 MultiCall Objects
  18.24.8 Convenience Functions
  18.24.9 Example of Client Usage
18.25 SimpleXMLRPCServer -- Basic XML-RPC server
  18.25.1 SimpleXMLRPCServer Objects
  18.25.2 CGIXMLRPCRequestHandler
18.26 DocXMLRPCServer -- Self-documenting XML-RPC server
  18.26.1 DocXMLRPCServer Objects
  18.26.2 DocCGIXMLRPCRequestHandler

19. Multimedia Services
19.1 audioop -- Manipulate raw audio data
19.2 imageop -- Manipulate raw image data
19.3 aifc -- Read and write AIFF and AIFC files
19.4 sunau -- Read and write Sun AU files
  19.4.1 AU_read Objects
  19.4.2 AU_write Objects
19.5 wave -- Read and write WAV files
  19.5.1 Wave_read Objects
  19.5.2 Wave_write Objects
• **19.6** chunk -- Read IFF chunked data
• **19.7** colorsys -- Conversions between color systems
• **19.8** rgbimg -- Read and write ``SGI RGB'' files
• **19.9** imghdr -- Determine the type of an image
• **19.10** sndhdr -- Determine type of sound file
• **19.11** ossaudiodev -- Access to OSS-compatible audio devices
  - 19.11.1 Audio Device Objects
  - 19.11.2 Mixer Device Objects
• **20.** Graphical User Interfaces with Tk
  - **20.1** Tkinter -- Python interface to Tcl/Tk
    - 20.1.1 Tkinter Modules
    - 20.1.2 Tkinter Life Preserver
    - 20.1.3 A (Very) Quick Look at Tcl/Tk
    - 20.1.4 Mapping Basic Tk into Tkinter
    - 20.1.5 How Tk and Tkinter are Related
    - 20.1.6 Handy Reference
  - **20.2** Tix -- Extension widgets for Tk
    - 20.2.1 Using Tix
    - 20.2.2 Tix Widgets
    - 20.2.3 Tix Commands
  - **20.3** ScrolledText -- Scrolled Text Widget
  - **20.4** turtle -- Turtle graphics for Tk
    - 20.4.1 Turtle, Pen and RawPen Objects
  - **20.5** Idle
    - 20.5.1 Menus
    - 20.5.2 Basic editing and navigation
    - 20.5.3 Syntax colors
  - **20.6** Other Graphical User Interface Packages
• **21.** Internationalization
  - **21.1** gettext -- Multilingual internationalization services
    - 21.1.1 GNU gettext API
    - 21.1.2 Class-based API
    - 21.1.3 Internationalizing your programs and modules
    - 21.1.4 Acknowledgements
  - **21.2** locale -- Internationalization services
    - 21.2.1 Background, details, hints, tips and caveats
    - 21.2.2 For extension writers and programs that embed Python
    - 21.2.3 Access to message catalogs
• **22.** Program Frameworks
22.1 cmd -- Support for line-oriented command interpreters
  - 22.1.1 Cmd Objects
22.2 shlex -- Simple lexical analysis
  - 22.2.1 shlex Objects
  - 22.2.2 Parsing Rules

23. Development Tools

23.1 pydoc -- Documentation generator and online help system
23.2 doctest -- Test interactive Python examples
  - 23.2.1 Simple Usage: Checking Examples in Docstrings
  - 23.2.2 Simple Usage: Checking Examples in a Text File
  - 23.2.3 How It Works
  - 23.2.4 Basic API
  - 23.2.5 Unittest API
  - 23.2.6 Advanced API
  - 23.2.7 Debugging
  - 23.2.8 Soapbox
23.3 unittest -- Unit testing framework
  - 23.3.1 Basic example
  - 23.3.2 Organizing test code
  - 23.3.3 Re-using old test code
  - 23.3.4 Classes and functions
  - 23.3.5 TestCase Objects
  - 23.3.6 TestSuite Objects
  - 23.3.7 TestResult Objects
  - 23.3.8 TestLoader Objects
23.4 test -- Regression tests package for Python
  - 23.4.1 Writing Unit Tests for the test package
  - 23.4.2 Running tests using test.regrtest
23.5 test.test_support -- Utility functions for tests

24. The Python Debugger

24.1 Debugger Commands
24.2 How It Works

25. The Python Profilers

25.1 Introduction to the profilers
25.2 Instant User's Manual
25.3 What Is Deterministic Profiling?
25.4 Reference Manual - profile and cProfile
  - 25.4.1 The Stats Class
25.5 Limitations
- 25.6 Calibration
- 25.7 Extensions -- Deriving Better Profilers
- 25.8 hotshot -- High performance logging profiler
  - 25.8.1 Profile Objects
  - 25.8.2 Using hotshot data
  - 25.8.3 Example Usage
- 25.9 timeit -- Measure execution time of small code snippets
  - 25.9.1 Command Line Interface
  - 25.9.2 Examples
- 25.10 trace -- Trace or track Python statement execution
  - 25.10.1 Command Line Usage
  - 25.10.2 Programming Interface

• 26. Python Runtime Services
  - 26.1 sys -- System-specific parameters and functions
  - 26.2 __builtin__ -- Built-in objects
  - 26.3 __main__ -- Top-level script environment
  - 26.4 warnings -- Warning control
    - 26.4.1 Warning Categories
    - 26.4.2 The Warnings Filter
    - 26.4.3 Available Functions
  - 26.5 contextlib -- Utilities for with-statement contexts.
  - 26.6 atexit -- Exit handlers
    - 26.6.1 atexit Example
  - 26.7 traceback -- Print or retrieve a stack traceback
    - 26.7.1 Traceback Example
  - 26.8 __future__ -- Future statement definitions
  - 26.9 gc -- Garbage Collector interface
  - 26.10 inspect -- Inspect live objects
    - 26.10.1 Types and members
    - 26.10.2 Retrieving source code
    - 26.10.3 Classes and functions
    - 26.10.4 The interpreter stack
  - 26.11 site -- Site-specific configuration hook
  - 26.12 user -- User-specific configuration hook
  - 26.13 fpectl -- Floating point exception control
    - 26.13.1 Example
    - 26.13.2 Limitations and other considerations

• 27. Custom Python Interpreters
  - 27.1 code -- Interpreter base classes
27.1.1 Interactive Interpreter Objects
27.1.2 Interactive Console Objects

27.2 codeop -- Compile Python code

28. Restricted Execution
28.1 rexec -- Restricted execution framework
  28.1.1 RExec Objects
  28.1.2 Defining restricted environments
  28.1.3 An example
28.2 Bastion -- Restricting access to objects

29. Importing Modules
29.1 imp -- Access the import internals
  29.1.1 Examples
29.2 zipimport -- Import modules from Zip archives
  29.2.1 zipimporter Objects
  29.2.2 Examples
29.3 pkgutil -- Package extension utility
29.4 modulefinder -- Find modules used by a script
29.5 runpy -- Locating and executing Python modules.

30. Python Language Services
30.1 parser -- Access Python parse trees
  30.1.1 Creating AST Objects
  30.1.2 Converting AST Objects
  30.1.3 Queries on AST Objects
  30.1.4 Exceptions and Error Handling
  30.1.5 AST Objects
  30.1.6 Examples
30.2 symbol -- Constants used with Python parse trees
30.3 token -- Constants used with Python parse trees
30.4 keyword -- Testing for Python keywords
30.5 tokenize -- Tokenizer for Python source
30.6 tabnanny -- Detection of ambiguous indentation
30.7 pyclbr -- Python class browser support
  30.7.1 Class Descriptor Objects
  30.7.2 Function Descriptor Objects
30.8 py_compile -- Compile Python source files
30.9 compileall -- Byte-compile Python libraries
30.10 dis -- Disassembler for Python byte code
  30.10.1 Python Byte Code Instructions
30.11 pickletools -- Tools for pickle developers.
30.12 distutils -- Building and installing Python modules

31. Python compiler package
   - 31.1 The basic interface
   - 31.2 Limitations
   - 31.3 Python Abstract Syntax
     - 31.3.1 AST Nodes
     - 31.3.2 Assignment nodes
     - 31.3.3 Examples
   - 31.4 Using Visitors to Walk ASTs
   - 31.5 Bytecode Generation

32. Abstract Syntax Trees
   - 32.1 Abstract Grammar

33. Miscellaneous Services
   - 33.1 formatter -- Generic output formatting
     - 33.1.1 The Formatter Interface
     - 33.1.2 Formatter Implementations
     - 33.1.3 The Writer Interface
     - 33.1.4 Writer Implementations

34. SGI IRIX Specific Services
   - 34.1 al -- Audio functions on the SGI
     - 34.1.1 Configuration Objects
     - 34.1.2 Port Objects
   - 34.2 AL -- Constants used with the al module
   - 34.3 cd -- CD-ROM access on SGI systems
     - 34.3.1 Player Objects
     - 34.3.2 Parser Objects
   - 34.4 fl -- FORMS library for graphical user interfaces
     - 34.4.1 Functions Defined in Module fl
     - 34.4.2 Form Objects
     - 34.4.3 FORMS Objects
   - 34.5 FL -- Constants used with the fl module
   - 34.6 flp -- Functions for loading stored FORMS designs
   - 34.7 fm -- Font Manager interface
   - 34.8 gl -- Graphics Library interface
   - 34.9 DEVICE -- Constants used with the gl module
   - 34.10 GL -- Constants used with the gl module
   - 34.11 imgfile -- Support for SGI imglib files
   - 34.12 jpeg -- Read and write JPEG files

35. SunOS Specific Services
- 35.1 sunaudiodev -- Access to Sun audio hardware
  - 35.1.1 Audio Device Objects
- 35.2 SUNAUDIODEV -- Constants used with sunaudiodev

- 36. MS Windows Specific Services
  - 36.1 msilib -- Read and write Microsoft Installer files
    - 36.1.1 Database Objects
    - 36.1.2 View Objects
    - 36.1.3 Summary Information Objects
    - 36.1.4 Record Objects
    - 36.1.5 Errors
    - 36.1.6 CAB Objects
    - 36.1.7 Directory Objects
    - 36.1.8 Features
    - 36.1.9 GUI classes
    - 36.1.10 Precomputed tables
  - 36.2 msvcrt - Useful routines from the MS VC++ runtime
    - 36.2.1 File Operations
    - 36.2.2 Console I/O
    - 36.2.3 Other Functions
  - 36.3 winreg - Windows registry access
    - 36.3.1 Registry Handle Objects
  - 36.4 winsound -- Sound-playing interface for Windows

- A. Undocumented Modules
  - A.1 Frameworks
  - A.2 Miscellaneous useful utilities
  - A.3 Platform specific modules
  - A.4 Multimedia
  - A.5 Obsolete
  - A.6 SGI-specific Extension modules

- B. Reporting Bugs

- C. History and License
  - C.1 History of the software
  - C.2 Terms and conditions for accessing or otherwise using Python
  - C.3 Licenses and Acknowledgements for Incorporated Software
    - C.3.1 Mersenne Twister
    - C.3.2 Sockets
    - C.3.3 Floating point exception control
    - C.3.4 MD5 message digest algorithm
    - C.3.5 Asynchronous socket services
- C.3.6 Cookie management
- C.3.7 Profiling
- C.3.8 Execution tracing
- C.3.9 UUencode and UUdecode functions
- C.3.10 XML Remote Procedure Calls
- Module Index
- Index

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
# Module Index

This index only lists modules documented in this manual. The *Global Module Index* lists all modules that are documented in this set of manuals.

Some module names are followed by an annotation indicating what platform they are available on.

<table>
<thead>
<tr>
<th>Module</th>
<th>Platform</th>
<th>Module</th>
<th>Platform</th>
<th>Module</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>builtin</strong></td>
<td></td>
<td>email.encoders</td>
<td></td>
<td>mimetools</td>
<td></td>
</tr>
<tr>
<td><strong>future</strong></td>
<td></td>
<td>email.errors</td>
<td></td>
<td>mimetypes</td>
<td></td>
</tr>
<tr>
<td><strong>main</strong></td>
<td></td>
<td>email.generator</td>
<td></td>
<td>MimeWriter</td>
<td></td>
</tr>
<tr>
<td>_winreg (Windows)</td>
<td></td>
<td>email.header</td>
<td></td>
<td>minify</td>
<td></td>
</tr>
<tr>
<td>aifc</td>
<td></td>
<td>email.iterators</td>
<td></td>
<td>mmap</td>
<td></td>
</tr>
<tr>
<td>AL (IRIX)</td>
<td></td>
<td>email.message</td>
<td></td>
<td>modulefinder</td>
<td></td>
</tr>
<tr>
<td>al (IRIX)</td>
<td></td>
<td>email.mime.audio</td>
<td></td>
<td>msilib (Windows)</td>
<td></td>
</tr>
<tr>
<td>anydbm</td>
<td></td>
<td>email.mime.base</td>
<td></td>
<td>msvcrt (Windows)</td>
<td></td>
</tr>
<tr>
<td>array</td>
<td></td>
<td>email.mime.image</td>
<td></td>
<td>mutilfile</td>
<td></td>
</tr>
<tr>
<td>asynchat</td>
<td></td>
<td>email.mime.message</td>
<td></td>
<td>mutex</td>
<td></td>
</tr>
<tr>
<td>asyncore</td>
<td></td>
<td>email.mime.multipart</td>
<td></td>
<td>netrc</td>
<td></td>
</tr>
<tr>
<td>atexit</td>
<td></td>
<td>email.mime.nonmultipart</td>
<td></td>
<td>new</td>
<td></td>
</tr>
<tr>
<td>audioop</td>
<td></td>
<td>email.mime.text</td>
<td></td>
<td>nis (UNIX)</td>
<td></td>
</tr>
<tr>
<td>base64</td>
<td></td>
<td>email.mime</td>
<td></td>
<td>nntplib</td>
<td></td>
</tr>
<tr>
<td>BaseHTTPServer</td>
<td></td>
<td>email.parser</td>
<td></td>
<td>operator</td>
<td></td>
</tr>
<tr>
<td>Bastion</td>
<td></td>
<td>email.utils</td>
<td></td>
<td>optparse</td>
<td></td>
</tr>
<tr>
<td>binascii</td>
<td></td>
<td>email</td>
<td></td>
<td>os.path</td>
<td></td>
</tr>
<tr>
<td>binhex</td>
<td></td>
<td>encodings.idna</td>
<td></td>
<td>os</td>
<td></td>
</tr>
<tr>
<td>bisect</td>
<td></td>
<td>encodings.utf-8-sig</td>
<td></td>
<td>ossaudiodev (LinFreeBSD)</td>
<td></td>
</tr>
<tr>
<td>bsddb (Unix, Windows)</td>
<td></td>
<td>exceptions</td>
<td></td>
<td>parser</td>
<td></td>
</tr>
<tr>
<td>bz2</td>
<td></td>
<td>fcntl (Unix)</td>
<td></td>
<td>pdb</td>
<td></td>
</tr>
<tr>
<td>calendar</td>
<td></td>
<td>filecmp</td>
<td></td>
<td>pickle</td>
<td></td>
</tr>
<tr>
<td>cd (IRIX)</td>
<td></td>
<td>fileinput</td>
<td></td>
<td>pickletools</td>
<td></td>
</tr>
<tr>
<td>cgi</td>
<td></td>
<td>FL (IRIX)</td>
<td></td>
<td>pipes (Unix)</td>
<td></td>
</tr>
<tr>
<td>CGIHTTPServer</td>
<td></td>
<td>f1 (IRIX)</td>
<td></td>
<td>pkgutil</td>
<td></td>
</tr>
<tr>
<td>cgitb</td>
<td></td>
<td>flp (IRIX)</td>
<td></td>
<td>platform</td>
<td></td>
</tr>
</tbody>
</table>
About this document ...

Python Library Reference, 19th September, 2006, Release 2.5

This document was generated using the \LaTeX2HTML translator.


The application of \LaTeX2HTML to the Python documentation has been heavily tailored by Fred L. Drake, Jr. Original navigation icons were contributed by Christopher Petrilli.
Comments and Questions

General comments and questions regarding this document should be sent by email to docs@python.org. If you find specific errors in this document, either in the content or the presentation, please report the bug at the Python Bug Tracker at SourceForge. If you are able to provide suggested text, either to replace existing incorrect or unclear material, or additional text to supplement what's already available, we'd appreciate the contribution. There's no need to worry about text markup; our documentation team will gladly take care of that.

Questions regarding how to use the information in this document should be sent to the Python news group, comp.lang.python, or the Python mailing list (which is gated to the newsgroup and carries the same content).

For any of these channels, please be sure not to send HTML email. Thanks.

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
4.8.1.1 Codec Objects

The Codec class defines these methods which also define the function interfaces of the stateless encoder and decoder:

**encode**(*input*, *errors*)

Encodes the object *input* and returns a tuple (output object, length consumed). While codecs are not restricted to use with Unicode, in a Unicode context, encoding converts a Unicode object to a plain string using a particular character set encoding (e.g., cp1252 or iso-8859-1).

*errors* defines the error handling to apply. It defaults to 'strict' handling.

The method may not store state in the Codec instance. Use StreamCodec for codecs which have to keep state in order to make encoding/decoding efficient.

The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

**decode**(*input*, *errors*)

Decodes the object *input* and returns a tuple (output object, length consumed). In a Unicode context, decoding converts a plain string encoded using a particular character set encoding to a Unicode object.

*input* must be an object which provides the **bf_getreadbuf** buffer slot. Python strings, buffer objects and memory mapped files are examples of objects providing this slot.

*errors* defines the error handling to apply. It defaults to 'strict' handling.

The method may not store state in the Codec instance. Use StreamCodec for codecs which have to keep state in order to make
encoding/decoding efficient.

The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

The IncrementalEncoder and IncrementalDecoder classes provide the basic interface for incremental encoding and decoding. Encoding/decoding the input isn't done with one call to the stateless encoder/decoder function, but with multiple calls to the encode/decode method of the incremental encoder/decoder. The incremental encoder/decoder keeps track of the encoding/decoding process during method calls.

The joined output of calls to the encode/decode method is the same as if all the single inputs were joined into one, and this input was encoded/decoded with the stateless encoder/decoder.
4.8.1.2 IncrementalEncoder Objects

New in version 2.5.

The IncrementalEncoder class is used for encoding an input in multiple steps. It defines the following methods which every incremental encoder must define in order to be compatible with the Python codec registry.

```python
class IncrementalEncoder([errors])
```

Constructor for an IncrementalEncoder instance.

All incremental encoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The IncrementalEncoder may implement different error handling schemes by providing the errors keyword argument. These parameters are predefined:

- 'strict' Raise ValueError (or a subclass); this is the default.
- 'ignore' Ignore the character and continue with the next.
- 'replace' Replace with a suitable replacement character
- 'xmlcharrefreplace' Replace with the appropriate XML character reference
- 'backslashreplace' Replace with backslashed escape sequences.

The errors argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the IncrementalEncoder object.

The set of allowed values for the errors argument can be extended with register_error().
encode(object[, final])

Encodes object (taking the current state of the encoder into account) and returns the resulting encoded object. If this is the last call to encode final must be true (the default is false).

reset()

Reset the encoder to the initial state.
4.8.1.3 IncrementalDecoder Objects

The Incremental Decoder class is used for decoding an input in multiple steps. It defines the following methods which every incremental decoder must define in order to be compatible with the Python codec registry.

```python
class IncrementalDecoder([errors])
    Constructor for an IncrementalDecoder instance.
```

All incremental decoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The IncrementalDecoder may implement different error handling schemes by providing the errors keyword argument. These parameters are predefined:

- 'strict' Raise ValueError (or a subclass); this is the default.
- 'ignore' Ignore the character and continue with the next.
- 'replace' Replace with a suitable replacement character.

The errors argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the IncrementalEncoder object.

The set of allowed values for the errors argument can be extended with register_error().

```python
decode(object[, final])
    Decodes object (taking the current state of the decoder into account) and returns the resulting decoded object. If this is the last call to decode final must be true (the default is false). If final is true the decoder must decode the input completely and must flush all buffers. If this isn't possible (e.g. because of incomplete byte sequences at the end of the input) it must
initiate error handling just like in the stateless case (which might raise an exception).

**reset()**
Reset the decoder to the initial state.

The StreamWriter and StreamReader classes provide generic working interfaces which can be used to implement new encoding submodules very easily. See `encodings.utf_8` for an example of how this is done.
4.8.1.4 StreamWriter Objects

The StreamWriter class is a subclass of Codec and defines the following methods which every stream writer must define in order to be compatible with the Python codec registry.

```python
class StreamWriter(stream[, errors])
    Constructor for a StreamWriter instance.
```

All stream writers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

*stream* must be a file-like object open for writing binary data.

The StreamWriter may implement different error handling schemes by providing the *errors* keyword argument. These parameters are predefined:

- 'strict' Raise ValueError (or a subclass); this is the default.
- 'ignore' Ignore the character and continue with the next.
- 'replace' Replace with a suitable replacement character
- 'xmlcharrefreplace' Replace with the appropriate XML character reference
- 'backslashreplace' Replace with backslashed escape sequences.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the StreamWriter object.

The set of allowed values for the *errors* argument can be extended with `register_error()`.

```python
write(object)
    Writes the object's contents encoded to the stream.
```
**writelines**( *list*)

Writes the concatenated list of strings to the stream (possibly by reusing the `write()` method).

**reset()**

Flushes and resets the codec buffers used for keeping state.

Calling this method should ensure that the data on the output is put into a clean state that allows appending of new fresh data without having to rescan the whole stream to recover state.

In addition to the above methods, the `StreamWriter` must also inherit all other methods and attributes from the underlying stream.
### 4.8.1.5 StreamReader Objects

The `StreamReader` class is a subclass of `Codec` and defines the following methods which every stream reader must define in order to be compatible with the Python codec registry.

```python
class StreamReader(stream[, errors])
```

Constructor for a `StreamReader` instance.

All stream readers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

*stream* must be a file-like object open for reading (binary) data.

The `StreamReader` may implement different error handling schemes by providing the `errors` keyword argument. These parameters are defined:

- `'strict'` Raise `ValueError` (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
- `'replace'` Replace with a suitable replacement character.

The `errors` argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamReader` object.

The set of allowed values for the `errors` argument can be extended with `register_error()`.

```python
read([size[, chars[, firstline]]])
```

Decodes data from the stream and returns the resulting object.

*chars* indicates the number of characters to read from the stream. `read()` will never return more than `chars` characters, but it might return less, if there are not enough characters available.
size indicates the approximate maximum number of bytes to read from the stream for decoding purposes. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. size is intended to prevent having to decode huge files in one step.

firstline indicates that it would be sufficient to only return the first line, if there are decoding errors on later lines.

The method should use a greedy read strategy meaning that it should read as much data as is allowed within the definition of the encoding and the given size, e.g. if optional encoding endings or state markers are available on the stream, these should be read too.

Changed in version 2.4: chars argument added. Changed in version 2.4.2: firstline argument added.

**readline**([size[, keepends]])

Read one line from the input stream and return the decoded data.

size, if given, is passed as size argument to the stream's readline() method.

If keepends is false line-endings will be stripped from the lines returned.

Changed in version 2.4: keepends argument added.

**readlines**([sizehint[, keepends]])

Read all lines available on the input stream and return them as a list of lines.

Line-endings are implemented using the codec's decoder method and are included in the list entries if keepends is true.

sizehint, if given, is passed as the size argument to the stream's read() method.

**reset**()

Resets the codec buffers used for keeping state.
Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

In addition to the above methods, the `StreamReader` must also inherit all other methods and attributes from the underlying stream.

The next two base classes are included for convenience. They are not needed by the codec registry, but may provide useful in practice.
4.8.1.6 StreamReaderWriter Objects

The StreamReaderWriter allows wrapping streams which work in both read and write modes.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

```python
class StreamReaderWriter(stream, Reader, Writer, errors)
```

Creates a `StreamReaderWriter` instance. `stream` must be a file-like object. `Reader` and `Writer` must be factory functions or classes providing the `StreamReader` and `StreamWriter` interface resp. Error handling is done in the same way as defined for the stream readers and writers.

`StreamReaderWriter` instances define the combined interfaces of `StreamReader` and `StreamWriter` classes. They inherit all other methods and attributes from the underlying stream.
4.8.1.7 StreamRecoder Objects

The StreamRecoder provide a frontend - backend view of encoding data which is sometimes useful when dealing with different encoding environments.

The design is such that one can use the factory functions returned by the lookup() function to construct the instance.

class StreamRecoder(stream, encode, decode, Reader, Writer, errors)

Creates a StreamRecoder instance which implements a two-way conversion: encode and decode work on the frontend (the input to read() and output of write()) while Reader and Writer work on the backend (reading and writing to the stream).

You can use these objects to do transparent direct recodings from e.g. Latin-1 to UTF-8 and back.

stream must be a file-like object.

encode, decode must adhere to the Codec interface. Reader, Writer must be factory functions or classes providing objects of the StreamReader and StreamWriter interface respectively.

encode and decode are needed for the frontend translation, Reader and Writer for the backend translation. The intermediate format used is determined by the two sets of codecs, e.g. the Unicode codecs will use Unicode as the intermediate encoding.

Error handling is done in the same way as defined for the stream readers and writers.

StreamRecoder instances define the combined interfaces of StreamReader and StreamWriter classes. They inherit all other methods and attributes from the underlying stream.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
5.1.8.1 Creating Datetime Objects from Formatted Strings

The `datetime` class does not directly support parsing formatted time strings. You can use `time.strptime` to do the parsing and create a `datetime` object from the tuple it returns:

```python
>>> s = "2005-12-06T12:13:14"
>>> from datetime import datetime
>>> from time import strptime
>>> datetime(*strptime(s, "%Y-%m-%dT%H:%M:%S"))[0:6])
datetime.datetime(2005, 12, 6, 12, 13, 14)
```
5.3.1.1 Recipes

This section shows various approaches to working with deques.

The `rotate()` method provides a way to implement deque slicing and deletion. For example, a pure python implementation of `del d[n]` relies on the `rotate()` method to position elements to be popped:

```python
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

To implement deque slicing, use a similar approach applying `rotate()` to bring a target element to the left side of the deque. Remove old entries with `popleft()`, add new entries with `extend()`, and then reverse the rotation.

With minor variations on that approach, it is easy to implement Forth style stack manipulations such as `dup`, `drop`, `swap`, `over`, `pick`, `rot`, and `roll`.

A roundrobin task server can be built from a `deque` using `popleft()` to select the current task and `append()` to add it back to the tasklist if the input stream is not exhausted:

```python
def roundrobin(*iterables):
    pending = deque(iter(i) for i in iterables)
    while pending:
        task = pending.popleft()
        try:
            yield task.next()
        except StopIteration:
            continue
        pending.append(task)
```

```bash
>>> for value in roundrobin('abc', 'd', 'efgh'):
...    print value

a
d
```
Multi-pass data reduction algorithms can be succinctly expressed and efficiently coded by extracting elements with multiple calls to `popleft()`, applying the reduction function, and calling `append()` to add the result back to the queue.

For example, building a balanced binary tree of nested lists entails reducing two adjacent nodes into one by grouping them in a list:

```python
def maketree(iterable):
    d = deque(iterable)
    while len(d) > 1:
        pair = [d.popleft(), d.popleft()]
        d.append(pair)
    return list(d)
```

```python
>>> print maketree('abcdefgh')
[[[['a', 'b'], ['c', 'd']], [['e', 'f'], ['g', 'h']]]]
```
5.3.2.1 defaultdict Examples

Using list as the default_factory, it is easy to group a sequence of key-value pairs into a dictionary of lists:

```python
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
    d[k].append(v)

>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the default_factory function which returns an empty list. The list.append() operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the list.append() operation adds another value to the list. This technique is simpler and faster than an equivalent technique using dict.setdefault():

```python
>>> d = {}
>>> for k, v in s:
    d.setdefault(k, []).append(v)

>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Setting the default_factory to int makes the defaultdict useful for counting (like a bag or multiset in other languages):

```python
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
    d[k] += 1

>>> d.items()
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

When a letter is first encountered, it is missing from the mapping, so the default_factory function calls int() to supply a default count of zero.
The increment operation then builds up the count for each letter. This technique makes counting simpler and faster than an equivalent technique using `dict.get()`:

```python
>>> d = {}
>>> for k in s:
    d[k] = d.get(k, 0) + 1

>>> d.items()
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

Setting the `default_factory` to `set` makes the `defaultdict` useful for building a dictionary of sets:

```python
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1
>>> d = defaultdict(set)
>>> for k, v in s:
    d[k].add(v)

>>> d.items()
[('blue', set([2, 4])), ('red', set([1, 3]))]
```
6.3.5.1 Mitigating round-off error with increased precision

The use of decimal floating point eliminates decimal representation error (making it possible to represent 0.1 exactly); however, some operations can still incur round-off error when non-zero digits exceed the fixed precision.

The effects of round-off error can be amplified by the addition or subtraction of nearly offsetting quantities resulting in loss of significance. Knuth provides two instructive examples where rounded floating point arithmetic with insufficient precision causes the breakdown of the associative and distributive properties of addition:

```python
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal("9.5111111")
>>> u + (v + w)
Decimal("10")

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal("0.01")
>>> u * (v+w)
Decimal("0.0060000")
```

The `decimal` module makes it possible to restore the identities by expanding the precision sufficiently to avoid loss of significance:

```python
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal("9.51111111")
>>> u + (v + w)
Decimal("9.51111111")

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.00000003')
>>> (u*v) + (u*w)
Decimal("0.0060000")
>>> u * (v+w)
```
Decimal("0.0060000")

Previous: 6.3.5 Floating Point Notes  Up: 6.3.5 Floating Point Notes  Next: 6.3.5.2 Special values

Release 2.5, documentation updated on 19th September, 2006.  See About this document... for information on suggesting changes.
6.3.5.2 Special values

The number system for the `decimal` module provides special values including `NaN`, `sNaN`, `-Infinity`, `Infinity`, and two zeroes, `+0` and `-0`.

Infinities can be constructed directly with: `Decimal('Infinity')`. Also, they can arise from dividing by zero when the `DivisionByZero` signal is not trapped. Likewise, when the `Overflow` signal is not trapped, infinity can result from rounding beyond the limits of the largest representable number.

The infinities are signed (affine) and can be used in arithmetic operations where they get treated as very large, indeterminate numbers. For instance, adding a constant to infinity gives another infinite result.

Some operations are indeterminate and return `NaN`, or if the `InvalidOperation` signal is trapped, raise an exception. For example, `0/0` returns `NaN` which means ``not a number``. This variety of `NaN` is quiet and, once created, will flow through other computations always resulting in another `NaN`. This behavior can be useful for a series of computations that occasionally have missing inputs -- it allows the calculation to proceed while flagging specific results as invalid.

A variant is `sNaN` which signals rather than remaining quiet after every operation. This is a useful return value when an invalid result needs to interrupt a calculation for special handling.

The signed zeros can result from calculations that underflow. They keep the sign that would have resulted if the calculation had been carried out to greater precision. Since their magnitude is zero, both positive and negative zeros are treated as equal and their sign is informational.

In addition to the two signed zeros which are distinct yet equal, there are various representations of zero with differing precisions yet equivalent in value. This takes a bit of getting used to. For an eye accustomed to normalized floating point representations, it is not immediately obvious that the following calculation
returns a value equal to zero:

```python
>>> 1 / Decimal('Infinity')
Decimal("0E-1000000026")
```
7.1.2.1 FeedParser API

New in version 2.4.

The FeedParser, imported from the email.feedparser module, provides an API that is conducive to incremental parsing of email messages, such as would be necessary when reading the text of an email message from a source that can block (e.g. a socket). The FeedParser can of course be used to parse an email message fully contained in a string or a file, but the classic Parser API may be more convenient for such use cases. The semantics and results of the two parser APIs are identical.

The FeedParser's API is simple; you create an instance, feed it a bunch of text until there's no more to feed it, then close the parser to retrieve the root message object. The FeedParser is extremely accurate when parsing standards-compliant messages, and it does a very good job of parsing non-compliant messages, providing information about how a message was deemed broken. It will populate a message object's defects attribute with a list of any problems it found in a message. See the email.errors module for the list of defects that it can find.

Here is the API for the FeedParser:

```python
class FeedParser([_factory])

Create a FeedParser instance. Optional _factory is a no-argument callable that will be called whenever a new message object is needed. It defaults to the email.message.Message class.

feed(data)

Feed the FeedParser some more data. data should be a string containing one or more lines. The lines can be partial and the FeedParser will stitch such partial lines together properly. The lines in the string can have any of the common three line endings, carriage return, newline, or carriage return and newline (they can even be mixed).
```
close()

Closing a FeedParser completes the parsing of all previously fed data, and returns the root message object. It is undefined what happens if you feed more data to a closed FeedParser.
The `Parser` class, imported from the `email.parser` module, provides an API that can be used to parse a message when the complete contents of the message are available in a string or file. The `email.parser` module also provides a second class, called `HeaderParser` which can be used if you're only interested in the headers of the message. `HeaderParser` can be much faster in these situations, since it does not attempt to parse the message body, instead setting the payload to the raw body as a string. `HeaderParser` has the same API as the `Parser` class.

**class Parser([_class])**

The constructor for the `Parser` class takes an optional argument `_class`. This must be a callable factory (such as a function or a class), and it is used whenever a sub-message object needs to be created. It defaults to `Message` (see `email.message`). The factory will be called without arguments.

The optional `strict` flag is ignored.

**Deprecated since release 2.4.** Because the `Parser` class is a backward compatible API wrapper around the new-in-Python 2.4 `FeedParser`, _all_ parsing is effectively non-strict. You should simply stop passing a `strict` flag to the `Parser` constructor.

Changed in version 2.2.2: The `strict` flag was added. Changed in version 2.4: The `strict` flag was deprecated.

The other public `Parser` methods are:

**parse(fp[, headersonly])**

Read all the data from the file-like object `fp`, parse the resulting text, and return the root message object. `fp` must support both the `readline()` and the `read()` methods on file-like objects.
The text contained in \texttt{fp} must be formatted as a block of RFC\,2822 style headers and header continuation lines, optionally preceded by a envelope header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts).

Optional \texttt{headersonly} is as with the \texttt{parse()} method.

Changed in version 2.2.2: The \texttt{headersonly} flag was added.

\texttt{parsestr(\text{text}[\text{, headersonly}])}

Similar to the \texttt{parse()} method, except it takes a string object instead of a file-like object. Calling this method on a string is exactly equivalent to wrapping \texttt{text} in a \texttt{StringIO} instance first and calling \texttt{parse()}.

Optional \texttt{headersonly} is a flag specifying whether to stop parsing after reading the headers or not. The default is \texttt{False}, meaning it parses the entire contents of the file.

Changed in version 2.2.2: The \texttt{headersonly} flag was added.

Since creating a message object structure from a string or a file object is such a common task, two functions are provided as a convenience. They are available in the top-level \texttt{email} package namespace.

\texttt{message\_from\_string(s[, \_class[, strict]])}

Return a message object structure from a string. This is exactly equivalent to \texttt{Parser().parsestr(s)}. Optional \texttt{\_class} and \texttt{strict} are interpreted as with the \texttt{Parser} class constructor.

Changed in version 2.2.2: The \texttt{strict} flag was added.

\texttt{message\_from\_file(fp[, \_class[, strict]])}

Return a message object structure tree from an open file object. This is exactly equivalent to \texttt{Parser().parse(fp)}. Optional \texttt{\_class} and \texttt{strict} are interpreted as with the \texttt{Parser} class constructor.

Changed in version 2.2.2: The \texttt{strict} flag was added.
Here's an example of how you might use this at an interactive Python prompt:

```python
>>> import email
>>> msg = email.message_from_string(myString)
```
7.1.2.3 Additional notes

Here are some notes on the parsing semantics:

- Most non-multipart type messages are parsed as a single message object with a string payload. These objects will return `False` for `is_multipart()`. Their `get_payload()` method will return a string object.

- All multipart type messages will be parsed as a container message object with a list of sub-message objects for their payload. The outer container message will return `True` for `is_multipart()` and their `get_payload()` method will return the list of Message subparts.

- Most messages with a content type of `message/*` (e.g. `message/delivery-status` and `message/rfc822`) will also be parsed as container object containing a list payload of length 1. Their `is_multipart()` method will return `True`. The single element in the list payload will be a sub-message object.

- Some non-standards compliant messages may not be internally consistent about their multipart-edness. Such messages may have a Content-Type: header of type `multipart`, but their `is_multipart()` method may return `False`. If such messages were parsed with the `FeedParser`, they will have an instance of the `MultipartInvariantViolationDefect` class in their `defects` attribute list. See `email.errors` for details.
7.3.1.1 Maildir

class Maildir(dirname[, factory=rfc822.Message[, create=True]])

A subclass of Mailbox for mailboxes in Maildir format. Parameter factory is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If factory is None, MaildirMessage is used as the default message representation. If create is True, the mailbox is created if it does not exist.

It is for historical reasons that factory defaults to rfc822.Message and that dirname is named as such rather than path. For a Maildir instance that behaves like instances of other Mailbox subclasses, set factory to None.

Maildir is a directory-based mailbox format invented for the qmail mail transfer agent and now widely supported by other programs. Messages in a Maildir mailbox are stored in separate files within a common directory structure. This design allows Maildir mailboxes to be accessed and modified by multiple unrelated programs without data corruption, so file locking is unnecessary.

Maildir mailboxes contain three subdirectories, namely: tmp, new, and cur. Messages are created momentarily in the tmp subdirectory and then moved to the new subdirectory to finalize delivery. A mail user agent may subsequently move the message to the cur subdirectory and store information about the state of the message in a special "info" section appended to its file name.

Folders of the style introduced by the Courier mail transfer agent are also supported. Any subdirectory of the main mailbox is considered a folder if "." is the first character in its name. Folder names are represented by Maildir without the leading ".". Each folder is itself a Maildir mailbox but should not contain other folders. Instead, a logical nesting is indicated using "." to delimit levels, e.g., "Archived.2005.07".

Note: The Maildir specification requires the use of a colon ("::") in certain message file names. However, some operating
systems do not permit this character in file names, If you wish to
use a Maildir-like format on such an operating system, you
should specify another character to use instead. The exclamation
point ("!") is a popular choice. For example:

```
import mailbox
mailbox.Maildir.colon = '!
```

The colon attribute may also be set on a per-instance basis.

Maildir instances have all of the methods of Mailbox in addition to the
following:

**list_folders()**
Return a list of the names of all folders.

**get_folder(folder)**
Return a Maildir instance representing the folder whose name is `folder`.
A NoSuchMailboxError exception is raised if the folder does not exist.

**add_folder(folder)**
Create a folder whose name is `folder` and return a Maildir instance
representing it.

**remove_folder(folder)**
Delete the folder whose name is `folder`. If the folder contains any messages,
aNotEmptyError exception will be raised and the folder will not be
deleted.

**clean()**
Delete temporary files from the mailbox that have not been accessed in the
last 36 hours. The Maildir specification says that mail-reading programs
should do this occasionally.

Some Mailbox methods implemented by Maildir deserve special remarks:

**add(message)**

**__setitem__(key, message)**
update( arg)

Warning: These methods generate unique file names based upon the current process ID. When using multiple threads, undetected name clashes may occur and cause corruption of the mailbox unless threads are coordinated to avoid using these methods to manipulate the same mailbox simultaneously.

flush()

All changes to Maildir mailboxes are immediately applied, so this method does nothing.

lock()

unlock()

Maildir mailboxes do not support (or require) locking, so these methods do nothing.

close()

Maildir instances do not keep any open files and the underlying mailboxes do not support locking, so this method does nothing.

get_file( key)

Depending upon the host platform, it may not be possible to modify or remove the underlying message while the returned file remains open.

See Also:

maildir man page from qmail

The original specification of the format.

Using maildir format

Notes on Maildir by its inventor. Includes an updated name-creation scheme and details on "info" semantics.

maildir man page from Courier

Another specification of the format. Describes a common extension for supporting folders.
### 7.3.1.2 mbox

**class mbox**( *path*, *factory=None*, *create=True*)

A subclass of Mailbox for mailboxes in mbox format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is *None*, *mboxMessage* is used as the default message representation. If *create* is *True*, the mailbox is created if it does not exist.

The mbox format is the classic format for storing mail on UNIX systems. All messages in an mbox mailbox are stored in a single file with the beginning of each message indicated by a line whose first five characters are "From ".

Several variations of the mbox format exist to address perceived shortcomings in the original. In the interest of compatibility, *mbox* implements the original format, which is sometimes referred to as *mboxo*. This means that the Content-Length: header, if present, is ignored and that any occurrences of "From " at the beginning of a line in a message body are transformed to ">From " when storing the message, although occurrences of ">From " are not transformed to "From " when reading the message.

Some Mailbox methods implemented by *mbox* deserve special remarks:

**get_file**( *key*)

Using the file after calling flush() or close() on the *mbox* instance may yield unpredictable results or raise an exception.

**lock()**

**unlock()**

Three locking mechanisms are used--dot locking and, if available, the flock() and lockf() system calls.

---

**See Also:**
mbox man page from qmail
A specification of the format and its variations.

mbox man page from tin
Another specification of the format, with details on locking.

Configuring Netscape Mail on UNIX: Why The Content-Length Format is Bad
An argument for using the original mbox format rather than a variation.

"mbox" is a family of several mutually incompatible mailbox formats
A history of mbox variations.
7.3.1.3 MH

class MH( path[, factory=None[, create=True]])

A subclass of Mailbox for mailboxes in MH format. Parameter factory is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If factory is None, MHMessage is used as the default message representation. If create is True, the mailbox is created if it does not exist.

MH is a directory-based mailbox format invented for the MH Message Handling System, a mail user agent. Each message in an MH mailbox resides in its own file. An MH mailbox may contain other MH mailboxes (called folders) in addition to messages. Folders may be nested indefinitely. MH mailboxes also support sequences, which are named lists used to logically group messages without moving them to sub-folders. Sequences are defined in a file called .mh_sequences in each folder.

The MH class manipulates MH mailboxes, but it does not attempt to emulate all of mh's behaviors. In particular, it does not modify and is not affected by the context or .mh_profile files that are used by mh to store its state and configuration.

MH instances have all of the methods of Mailbox in addition to the following:

list_folders()

Return a list of the names of all folders.

get_folder( folder)

Return an MH instance representing the folder whose name is folder. A NoSuchMailboxError exception is raised if the folder does not exist.

add_folder( folder)

Create a folder whose name is folder and return an MH instance representing it.
**remove_folder**( *folder*)
Delete the folder whose name is *folder*. If the folder contains any messages, a **NotEmptyError** exception will be raised and the folder will not be deleted.

**get_sequences()**
Return a dictionary of sequence names mapped to key lists. If there are no sequences, the empty dictionary is returned.

**set_sequences**( *sequences*)
Re-define the sequences that exist in the mailbox based upon *sequences*, a dictionary of names mapped to key lists, like returned by **get_sequences()**.

**pack()**
Rename messages in the mailbox as necessary to eliminate gaps in numbering. Entries in the sequences list are updated correspondingly. **Note:** Already-issued keys are invalidated by this operation and should not be subsequently used.

Some **Mailbox** methods implemented by **MH** deserve special remarks:

**remove**( *key*)

**__delitem__**( *key*)

**discard**( *key*)
These methods immediately delete the message. The **MH** convention of marking a message for deletion by prepending a comma to its name is not used.

**lock()**

**unlock()**
Three locking mechanisms are used--dot locking and, if available, the **flock()** and **lockf()** system calls. For **MH** mailboxes, locking the mailbox means locking the **.mh_sequences** file and, only for the duration of any operations that affect them, locking individual message files.

**get_file**( *key*)
Depending upon the host platform, it may not be possible to remove the underlying message while the returned file remains open.

**flush()**
All changes to MH mailboxes are immediately applied, so this method does nothing.

**close()**
MH instances do not keep any open files, so this method is equivalent to `unlock()`.

---

**See Also:**

- nmh - Message Handling System
  Home page of nmh, an updated version of the original mh.

- MH & nmh: Email for Users & Programmers
  A GPL-licensed book on mh and nmh, with some information on the mailbox format.

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.3.1.4 Babyl

class Babyl(path[, factory=None, create=True]):

A subclass of Mailbox for mailboxes in Babyl format. Parameter factory
is a callable object that accepts a file-like message representation (which
behaves as if opened in binary mode) and returns a custom representation.
If factory is None, BabylMessage is used as the default message
representation. If create is True, the mailbox is created if it does not exist.

Babyl is a single-file mailbox format used by the Rmail mail user agent included
with Emacs. The beginning of a message is indicated by a line containing the
two characters Control-Underscore ("\037") and Control-L ("\014"). The end
of a message is indicated by the start of the next message or, in the case of the
last message, a line containing a Control-Underscore ("\037") character.

Messages in a Babyl mailbox have two sets of headers, original headers and so-
called visible headers. Visible headers are typically a subset of the original
headers that have been reformatted or abridged to be more attractive. Each
message in a Babyl mailbox also has an accompanying list of labels, or short
strings that record extra information about the message, and a list of all user-
defined labels found in the mailbox is kept in the Babyl options section.

Babyl instances have all of the methods of Mailbox in addition to the
following:

get_labels()

Return a list of the names of all user-defined labels used in the mailbox.
Note: The actual messages are inspected to determine which labels exist in
the mailbox rather than consulting the list of labels in the Babyl options
section, but the Babyl section is updated whenever the mailbox is modified.

Some Mailbox methods implemented by Babyl deserve special remarks:

get_file(key)

In Babyl mailboxes, the headers of a message are not stored contiguously
with the body of the message. To generate a file-like representation, the headers and body are copied together into a `StringIO` instance (from the `StringIO` module), which has an API identical to that of a file. As a result, the file-like object is truly independent of the underlying mailbox but does not save memory compared to a string representation.

```python
lock()
unlock()
```

Three locking mechanisms are used--dot locking and, if available, the `flock()` and `lockf()` system calls.

**See Also:**

- [Format of Version 5 Babyl Files](#)
  A specification of the Babyl format.

- [Reading Mail with Rmail](#)
  The Rmail manual, with some information on Babyl semantics.
7.3.1.5 MMDF

```python
class MMDF(path[, factory=None[, create=True]])
```
A subclass of `Mailbox` for mailboxes in MMDF format. Parameter `factory` is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If `factory` is `None`, `MMDFMessage` is used as the default message representation. If `create` is `True`, the mailbox is created if it does not exist.

MMDF is a single-file mailbox format invented for the Multichannel Memorandum Distribution Facility, a mail transfer agent. Each message is in the same form as an mbox message but is bracketed before and after by lines containing four Control-A ("\001") characters. As with the mbox format, the beginning of each message is indicated by a line whose first five characters are "From ", but additional occurrences of "From " are not transformed to ">From " when storing messages because the extra message separator lines prevent mistaking such occurrences for the starts of subsequent messages.

Some `Mailbox` methods implemented by `MMDF` deserve special remarks:

- `get_file(key)`
  Using the file after calling `flush()` or `close()` on the `MMDF` instance may yield unpredictable results or raise an exception.

- `lock()`
- `unlock()`

Three locking mechanisms are used--dot locking and, if available, the `flock()` and `lockf()` system calls.

### See Also:

- [mmdf man page from tin](#)
  A specification of MMDF format from the documentation of tin, a newsreader.
A Wikipedia article describing the Multichannel Memorandum Distribution Facility.
7.3.2.1 MaildirMessage

class MaildirMessage([message])

A message with Maildir-specific behaviors. Parameter message has the same meaning as with the Message constructor.

Typically, a mail user agent application moves all of the messages in the new subdirectory to the cur subdirectory after the first time the user opens and closes the mailbox, recording that the messages are old whether or not they've actually been read. Each message in cur has an "info" section added to its file name to store information about its state. (Some mail readers may also add an "info" section to messages in new.) The "info" section may take one of two forms: it may contain "2," followed by a list of standardized flags (e.g., "2,FR") or it may contain "1," followed by so-called experimental information. Standard flags for Maildir messages are as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Draft</td>
<td>Under composition</td>
</tr>
<tr>
<td>F</td>
<td>Flagged</td>
<td>Marked as important</td>
</tr>
<tr>
<td>P</td>
<td>Passed</td>
<td>Forwarded, resent, or bounced</td>
</tr>
<tr>
<td>R</td>
<td>Replied</td>
<td>Replied to</td>
</tr>
<tr>
<td>S</td>
<td>Seen</td>
<td>Read</td>
</tr>
<tr>
<td>T</td>
<td>Trashed</td>
<td>Marked for subsequent deletion</td>
</tr>
</tbody>
</table>

MaildirMessage instances offer the following methods:

get_subdir()

Return either "new" (if the message should be stored in the new subdirectory) or "cur" (if the message should be stored in the cur subdirectory). **Note:** A message is typically moved from new to cur after its mailbox has been accessed, whether or not the message is has been read. A message msg has been read if "S" not in msg.get_flags() is True.
set_subdir(subdir)

Set the subdirectory the message should be stored in. Parameter subdir must be either "new" or "cur".

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the standard Maildir format, the result is the concatenation in alphabetical order of zero or one occurrence of each of "D", "F", "P", "R", "S", and "T". The empty string is returned if no flags are set or if "info" contains experimental semantics.

set_flags(flags)

Set the flags specified by flags and unset all others.

add_flag(flag)

Set the flag(s) specified by flag without changing other flags. To add more than one flag at a time, flag may be a string of more than one character. The current "info" is overwritten whether or not it contains experimental information rather than flags.

remove_flag(flag)

Unset the flag(s) specified by flag without changing other flags. To remove more than one flag at a time, flag may be a string of more than one character. If "info" contains experimental information rather than flags, the current "info" is not modified.

get_date()

Return the delivery date of the message as a floating-point number representing seconds since the epoch.

set_date(date)

Set the delivery date of the message to date, a floating-point number representing seconds since the epoch.

get_info()

Return a string containing the "info" for a message. This is useful for accessing and modifying "info" that is experimental (i.e., not a list of flags).
**set_info**( `info` )

Set "info" to `info`, which should be a string.

When a `MaildirMessage` instance is created based upon an `mboxMessage` or `MMDFMessage` instance, the `Status:` and `X-Status:` headers are omitted and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th><code>mboxMessage</code> or <code>MMDFMessage</code> state</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;cur&quot; subdirectory</td>
<td>O flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>R flag</td>
<td>A flag</td>
</tr>
<tr>
<td>S flag</td>
<td>R flag</td>
</tr>
<tr>
<td>T flag</td>
<td>D flag</td>
</tr>
</tbody>
</table>

When a `MaildirMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th><code>MHMessage</code> state</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;cur&quot; subdirectory</td>
<td>&quot;unseen&quot; sequence</td>
</tr>
<tr>
<td>&quot;cur&quot; subdirectory and S flag</td>
<td>no &quot;unseen&quot; sequence</td>
</tr>
<tr>
<td>F flag</td>
<td>&quot;flagged&quot; sequence</td>
</tr>
<tr>
<td>R flag</td>
<td>&quot;replied&quot; sequence</td>
</tr>
</tbody>
</table>

When a `MaildirMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th><code>BabylMessage</code> state</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;cur&quot; subdirectory</td>
<td>&quot;unseen&quot; label</td>
</tr>
<tr>
<td>&quot;cur&quot; subdirectory and S flag</td>
<td>no &quot;unseen&quot; label</td>
</tr>
<tr>
<td>P flag</td>
<td>&quot;forwarded&quot; or &quot;resent&quot; label</td>
</tr>
<tr>
<td>R flag</td>
<td>&quot;answered&quot; label</td>
</tr>
<tr>
<td>T flag</td>
<td>&quot;deleted&quot; label</td>
</tr>
</tbody>
</table>
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.3.2.2 mboxMessage

```
class mboxMessage([message])
    A message with mbox-specific behaviors. Parameter message has the same meaning as with the Message constructor.
```

Messages in an mbox mailbox are stored together in a single file. The sender's envelope address and the time of delivery are typically stored in a line beginning with "From " that is used to indicate the start of a message, though there is considerable variation in the exact format of this data among mbox implementations. Flags that indicate the state of the message, such as whether it has been read or marked as important, are typically stored in Status: and X-Status: headers.

Conventional flags for mbox messages are as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Read</td>
<td>Read</td>
</tr>
<tr>
<td>O</td>
<td>Old</td>
<td>Previously detected by MUA</td>
</tr>
<tr>
<td>D</td>
<td>Deleted</td>
<td>Marked for subsequent deletion</td>
</tr>
<tr>
<td>F</td>
<td>Flagged</td>
<td>Marked as important</td>
</tr>
<tr>
<td>A</td>
<td>Answered</td>
<td>Replied to</td>
</tr>
</tbody>
</table>

The "R" and "O" flags are stored in the Status: header, and the "D", "F", and "A" flags are stored in the X-Status: header. The flags and headers typically appear in the order mentioned.

mboxMessage instances offer the following methods:

```python
get_from()
    Return a string representing the "From " line that marks the start of the message in an mbox mailbox. The leading "From " and the trailing newline are excluded.
```
set_from(from_, time_=None)
Set the "From " line to from_, which should be specified without a leading
"From " or trailing newline. For convenience, time_ may be specified and
will be formatted appropriately and appended to from_. If time_ is
specified, it should be a struct_time instance, a tuple suitable for
passing to time.strftime(), or True (to use time.gmtime()).

get_flags()
Return a string specifying the flags that are currently set. If the message
complies with the conventional format, the result is the concatenation in the
following order of zero or one occurrence of each of "R", "O", "D", "F", and
"A".

set_flags(flags)
Set the flags specified by flags and unset all others. Parameter flags should
be the concatenation in any order of zero or more occurrences of each of
"R", "O", "D", "F", and "A".

add_flag(flag)
Set the flag(s) specified by flag without changing other flags. To add more
than one flag at a time, flag may be a string of more than one character.

remove_flag(flag)
Unset the flag(s) specified by flag without changing other flags. To remove
more than one flag at a time, flag maybe a string of more than one
character.

When an mboxMessage instance is created based upon a MaildirMessage
instance, a "From " line is generated based upon the MaildirMessage
instance's delivery date, and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>S flag</td>
</tr>
<tr>
<td>O flag</td>
<td>&quot;cur&quot; subdirectory</td>
</tr>
<tr>
<td>D flag</td>
<td>T flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>A flag</td>
<td>R flag</td>
</tr>
</tbody>
</table>
When an `mboxMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MHMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no &quot;unseen&quot; sequence</td>
</tr>
<tr>
<td>O flag</td>
<td>&quot;unseen&quot; sequence</td>
</tr>
<tr>
<td>F flag</td>
<td>&quot;flagged&quot; sequence</td>
</tr>
<tr>
<td>A flag</td>
<td>&quot;replied&quot; sequence</td>
</tr>
</tbody>
</table>

When an `mboxMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>BabylMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no &quot;unseen&quot; label</td>
</tr>
<tr>
<td>O flag</td>
<td>&quot;unseen&quot; label</td>
</tr>
<tr>
<td>D flag</td>
<td>&quot;deleted&quot; label</td>
</tr>
<tr>
<td>A flag</td>
<td>&quot;answered&quot; label</td>
</tr>
</tbody>
</table>

When a `Message` instance is created based upon an `MMDFMessage` instance, the "From " line is copied and all flags directly correspond:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MMDFMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>R flag</td>
</tr>
<tr>
<td>O flag</td>
<td>O flag</td>
</tr>
<tr>
<td>D flag</td>
<td>D flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>A flag</td>
<td>A flag</td>
</tr>
</tbody>
</table>
7.3.2.3 MHMessage

**class MHMessage([message])**

A message with MH-specific behaviors. Parameter *message* has the same meaning as with the Message constructor.

MH messages do not support marks or flags in the traditional sense, but they do support sequences, which are logical groupings of arbitrary messages. Some mail reading programs (although not the standard `mh` and `nmh`) use sequences in much the same way flags are used with other formats, as follows:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unseen</td>
<td>Not read, but previously detected by MUA</td>
</tr>
<tr>
<td>replied</td>
<td>Replied to</td>
</tr>
<tr>
<td>flagged</td>
<td>Marked as important</td>
</tr>
</tbody>
</table>

MHMessage instances offer the following methods:

**get_sequences()**

Return a list of the names of sequences that include this message.

**set_sequences(sequences)**

Set the list of sequences that include this message.

**add_sequence(sequence)**

Add *sequence* to the list of sequences that include this message.

**remove_sequence(sequence)**

Remove *sequence* from the list of sequences that include this message.

When an MHMessage instance is created based upon a MaildirMessage instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
</table>

Previous: 7.3.2.2 mboxMessage Up: 7.3.2 Message objects Next: 7.3.2.4 BabylMessage
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;unseen&quot; sequence</td>
<td>no S flag</td>
</tr>
<tr>
<td>&quot;replied&quot; sequence</td>
<td>R flag</td>
</tr>
<tr>
<td>&quot;flagged&quot; sequence</td>
<td>F flag</td>
</tr>
</tbody>
</table>

When an `MHMessage` instance is created based upon an `mboxMessage` or `MMDFMessage` instance, the `Status:` and `X-Status:` headers are omitted and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th><code>mboxMessage</code> or <code>MMDFMessage</code> state</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;unseen&quot; sequence</td>
<td>no R flag</td>
</tr>
<tr>
<td>&quot;replied&quot; sequence</td>
<td>A flag</td>
</tr>
<tr>
<td>&quot;flagged&quot; sequence</td>
<td>F flag</td>
</tr>
</tbody>
</table>

When an `MHMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th><code>BabylMessage</code> state</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;unseen&quot; sequence</td>
<td>&quot;unseen&quot; label</td>
</tr>
<tr>
<td>&quot;replied&quot; sequence</td>
<td>&quot;answered&quot; label</td>
</tr>
</tbody>
</table>

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
7.3.2.4 BabylMessage

**class BabylMessage**([*message*])

A message with Babyl-specific behaviors. Parameter *message* has the same meaning as with the `Message` constructor.

Certain message labels, called *attributes*, are defined by convention to have special meanings. The attributes are as follows:

<table>
<thead>
<tr>
<th>Label</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unseen</td>
<td>Not read, but previously detected by MUA</td>
</tr>
<tr>
<td>deleted</td>
<td>Marked for subsequent deletion</td>
</tr>
<tr>
<td>filed</td>
<td>Copied to another file or mailbox</td>
</tr>
<tr>
<td>answered</td>
<td>Replied to</td>
</tr>
<tr>
<td>forwarded</td>
<td>Forwarded</td>
</tr>
<tr>
<td>edited</td>
<td>Modified by the user</td>
</tr>
<tr>
<td>resent</td>
<td>Resent</td>
</tr>
</tbody>
</table>

By default, Rmail displays only visible headers. The `BabylMessage` class, though, uses the original headers because they are more complete. Visible headers may be accessed explicitly if desired.

`BabylMessage` instances offer the following methods:

**get_labels()**

Return a list of labels on the message.

**set_labels**( *labels*)

Set the list of labels on the message to *labels*.

**add_label**( *label*)

Add *label* to the list of labels on the message.
**remove_label(label)**
Remove *label* from the list of labels on the message.

**get_visible()**
Return an *Message* instance whose headers are the message's visible headers and whose body is empty.

**set_visible(visible)**
Set the message's visible headers to be the same as the headers in *message*. Parameter *visible* should be a *Message* instance, an *email.Message.Message* instance, a string, or a file-like object (which should be open in text mode).

**update_visible()**
When a *BabylMessage* instance's original headers are modified, the visible headers are not automatically modified to correspond. This method updates the visible headers as follows: each visible header with a corresponding original header is set to the value of the original header, each visible header without a corresponding original header is removed, and any of *Date:*, *From:*, *Reply-To:*, *To:*, *CC:*, and *Subject:* that are present in the original headers but not the visible headers are added to the visible headers.

When a *BabylMessage* instance is created based upon a *MaildirMessage* instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;unseen&quot; label</td>
<td>no S flag</td>
</tr>
<tr>
<td>&quot;deleted&quot; label</td>
<td>T flag</td>
</tr>
<tr>
<td>&quot;answered&quot; label</td>
<td>R flag</td>
</tr>
<tr>
<td>&quot;forwarded&quot; label</td>
<td>P flag</td>
</tr>
</tbody>
</table>

When a *BabylMessage* instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status:* and *X-Status:* headers are omitted and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>mboxMessage or MMDFMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;unseen&quot; label</td>
<td>no R flag</td>
</tr>
</tbody>
</table>
When a `BabylMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MHMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;unseen&quot; label</td>
<td>&quot;unseen&quot; sequence</td>
</tr>
<tr>
<td>&quot;answered&quot; label</td>
<td>&quot;replied&quot; sequence</td>
</tr>
</tbody>
</table>

Python Library Reference

Previous: 7.3.2.3 `MHMessage` Up: 7.3.2 `Message objects` Next: 7.3.2.5 `MMDFMessage`

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
7.3.2.5 MMDFMessage

**class MMDFMessage([message])**

A message with MMDF-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

As with message in an mbox mailbox, MMDF messages are stored with the sender's address and the delivery date in an initial line beginning with "From ". Likewise, flags that indicate the state of the message are typically stored in Status: and X-Status: headers.

Conventional flags for MMDF messages are identical to those of mbox message and are as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Read</td>
<td>Read</td>
</tr>
<tr>
<td>O</td>
<td>Old</td>
<td>Previously detected by MUA</td>
</tr>
<tr>
<td>D</td>
<td>Deleted</td>
<td>Marked for subsequent deletion</td>
</tr>
<tr>
<td>F</td>
<td>Flagged</td>
<td>Marked as important</td>
</tr>
<tr>
<td>A</td>
<td>Answered</td>
<td>Replied to</td>
</tr>
</tbody>
</table>

The "R" and "O" flags are stored in the Status: header, and the "D", "F", and "A" flags are stored in the X-Status: header. The flags and headers typically appear in the order mentioned.

**MMDFMessage** instances offer the following methods, which are identical to those offered by *mboxMessage*:

**get_from()**

Return a string representing the "From " line that marks the start of the message in an mbox mailbox. The leading "From " and the trailing newline are excluded.

**set_from(from[, time_=None])**
Set the "From " line to \texttt{from}_, which should be specified without a leading "From " or trailing newline. For convenience, \texttt{time}_ may be specified and will be formatted appropriately and appended to \texttt{from}_. If \texttt{time}_ is specified, it should be a \texttt{struct\_time} instance, a tuple suitable for passing to \texttt{time.strftime()}, or \texttt{True} (to use \texttt{time.gmtime()}).

\texttt{get\_flags()} 
Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of "R", "O", "D", "F", and "A".

\texttt{set\_flags(\texttt{flags})} 
Set the flags specified by \texttt{flags} and unset all others. Parameter \texttt{flags} should be the concatenation in any order of zero or more occurrences of each of "R", "O", "D", "F", and "A".

\texttt{add\_flag(\texttt{flag})} 
Set the flag(s) specified by \texttt{flag} without changing other flags. To add more than one flag at a time, \texttt{flag} may be a string of more than one character.

\texttt{remove\_flag(\texttt{flag})} 
Unset the flag(s) specified by \texttt{flag} without changing other flags. To remove more than one flag at a time, \texttt{flag} maybe a string of more than one character.

When an \texttt{MMDFMessage} instance is created based upon a \texttt{MaildirMessage} instance, a "From " line is generated based upon the \texttt{MaildirMessage} instance's delivery date, and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>S flag</td>
</tr>
<tr>
<td>O flag</td>
<td>&quot;cur&quot; subdirectory</td>
</tr>
<tr>
<td>D flag</td>
<td>T flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>A flag</td>
<td>R flag</td>
</tr>
</tbody>
</table>
When an `MMDFMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MHMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no &quot;unseen&quot; sequence</td>
</tr>
<tr>
<td>O flag</td>
<td>&quot;unseen&quot; sequence</td>
</tr>
<tr>
<td>F flag</td>
<td>&quot;flagged&quot; sequence</td>
</tr>
<tr>
<td>A flag</td>
<td>&quot;replied&quot; sequence</td>
</tr>
</tbody>
</table>

When an `MMDFMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>BabylMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no &quot;unseen&quot; label</td>
</tr>
<tr>
<td>O flag</td>
<td>&quot;unseen&quot; label</td>
</tr>
<tr>
<td>D flag</td>
<td>&quot;deleted&quot; label</td>
</tr>
<tr>
<td>A flag</td>
<td>&quot;answered&quot; label</td>
</tr>
</tbody>
</table>

When an `MMDFMessage` instance is created based upon an `mboxMessage` instance, the "From " line is copied and all flags directly correspond:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>mboxMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>R flag</td>
</tr>
<tr>
<td>O flag</td>
<td>O flag</td>
</tr>
<tr>
<td>D flag</td>
<td>D flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>A flag</td>
<td>A flag</td>
</tr>
</tbody>
</table>
8.6.2.1 DOMImplementation Objects

The DOMImplementation interface provides a way for applications to determine the availability of particular features in the DOM they are using. DOM Level 2 added the ability to create new Document and DocumentType objects using the DOMImplementation as well.

hasFeature(feature, version)
  Return true if the feature identified by the pair of strings feature and version is implemented.

createDocument(namespaceUri, qualifiedName, doctype)
  Return a new Document object (the root of the DOM), with a child Element object having the given namespaceUri and qualifiedName. The doctype must be a DocumentType object created by createDocumentType(), or None. In the Python DOM API, the first two arguments can also be None in order to indicate that no Element child is to be created.

cREATEDocumentType(qualifiedName, publicId, systemId)
  Return a new DocumentType object that encapsulates the given qualifiedName, publicId, and systemId strings, representing the information contained in an XML document type declaration.
8.6.2.2 Node Objects

All of the components of an XML document are subclasses of Node.

**nodeType**
An integer representing the node type. Symbolic constants for the types are on the Node object: ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE, CDATA_SECTION_NODE, ENTITY_NODE, PROCESSING_INSTRUCTION_NODE, COMMENT_NODE, DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE. This is a read-only attribute.

**parentNode**
The parent of the current node, or None for the document node. The value is always a Node object or None. For Element nodes, this will be the parent element, except for the root element, in which case it will be the Document object. For Attr nodes, this is always None. This is a read-only attribute.

**attributes**
A NamedNodeMap of attribute objects. Only elements have actual values for this; others provide None for this attribute. This is a read-only attribute.

**previousSibling**
The node that immediately precedes this one with the same parent. For instance the element with an end-tag that comes just before the self element's start-tag. Of course, XML documents are made up of more than just elements so the previous sibling could be text, a comment, or something else. If this node is the first child of the parent, this attribute will be None. This is a read-only attribute.

**nextSibling**
The node that immediately follows this one with the same parent. See also previousSibling. If this is the last child of the parent, this attribute will be None. This is a read-only attribute.
childNodes
   A list of nodes contained within this node. This is a read-only attribute.

firstChild
   The first child of the node, if there are any, or None. This is a read-only attribute.

lastChild
   The last child of the node, if there are any, or None. This is a read-only attribute.

localName
   The part of the tagName following the colon if there is one, else the entire tagName. The value is a string.

prefix
   The part of the tagName preceding the colon if there is one, else the empty string. The value is a string, or None

namespaceURI
   The namespace associated with the element name. This will be a string or None. This is a read-only attribute.

nodeName
   This has a different meaning for each node type; see the DOM specification for details. You can always get the information you would get here from another property such as the tagName property for elements or the name property for attributes. For all node types, the value of this attribute will be either a string or None. This is a read-only attribute.

nodeValue
   This has a different meaning for each node type; see the DOM specification for details. The situation is similar to that with nodeName. The value is a string or None.

hasAttributes()
   Returns true if the node has any attributes.

hasChildNodes()
Returns true if the node has any child nodes.

**isSameNode**( other)

Returns true if `other` refers to the same node as this node. This is especially useful for DOM implementations which use any sort of proxy architecture (because more than one object can refer to the same node).

**Note:** This is based on a proposed DOM Level 3 API which is still in the “working draft” stage, but this particular interface appears uncontroversial. Changes from the W3C will not necessarily affect this method in the Python DOM interface (though any new W3C API for this would also be supported).

**appendChild**( newChild)

Add a new child node to this node at the end of the list of children, returning `newChild`.

**insertBefore**( newChild, refChild)

Insert a new child node before an existing child. It must be the case that `refChild` is a child of this node; if not, `ValueError` is raised. `newChild` is returned. If `refChild` is `None`, it inserts `newChild` at the end of the children's list.

**removeChild**( oldChild)

Remove a child node. `oldChild` must be a child of this node; if not, `ValueError` is raised. `oldChild` is returned on success. If `oldChild` will not be used further, its `unlink()` method should be called.

**replaceChild**( newChild, oldChild)

Replace an existing node with a new node. It must be the case that `oldChild` is a child of this node; if not, `ValueError` is raised.

**normalize**( )

Join adjacent text nodes so that all stretches of text are stored as single `Text` instances. This simplifies processing text from a DOM tree for many
applications. New in version 2.1.

**cloneNode** *(deep)*

Clone this node. Setting *deep* means to clone all child nodes as well. This returns the clone.
8.6.2.3 NodeList Objects

A NodeList represents a sequence of nodes. These objects are used in two ways in the DOM Core recommendation: the Element objects provides one as its list of child nodes, and the getElementsByTagName() and getElementsByTagNameNS() methods of Node return objects with this interface to represent query results.

The DOM Level 2 recommendation defines one method and one attribute for these objects:

item(i)
Return the i'th item from the sequence, if there is one, or None. The index i is not allowed to be less then zero or greater than or equal to the length of the sequence.

length
The number of nodes in the sequence.

In addition, the Python DOM interface requires that some additional support is provided to allow NodeList objects to be used as Python sequences. All NodeList implementations must include support for __len__() and __getitem__(); this allows iteration over the NodeList in for statements and proper support for the len() built-in function.

If a DOM implementation supports modification of the document, the NodeList implementation must also support the __setitem__() and __delitem__() methods.
See About this document... for information on suggesting changes.
8.6.2.4 DocumentType Objects

Information about the notations and entities declared by a document (including the external subset if the parser uses it and can provide the information) is available from a DocumentType object. The DocumentType for a document is available from the Document object's doctype attribute; if there is no DOCTYPE declaration for the document, the document's doctype attribute will be set to None instead of an instance of this interface.

DocumentType is a specialization of Node, and adds the following attributes:

publicId
The public identifier for the external subset of the document type definition. This will be a string or None.

systemId
The system identifier for the external subset of the document type definition. This will be a URI as a string, or None.

internalSubset
A string giving the complete internal subset from the document. This does not include the brackets which enclose the subset. If the document has no internal subset, this should be None.

name
The name of the root element as given in the DOCTYPE declaration, if present.

entities
This is a NamedNodeMap giving the definitions of external entities. For entity names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no entities are defined.
notations

This is a NamedNodeMap giving the definitions of notations. For notation names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no notations are defined.
8.6.2.5 Document Objects

A Document represents an entire XML document, including its constituent elements, attributes, processing instructions, comments etc. Remember that it inherits properties from Node.

documentElement
  The one and only root element of the document.

createElement( tagName)
  Create and return a new element node. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as insertBefore() or appendChild().

createElementNS( namespaceURI, tagName)
  Create and return a new element with a namespace. The tagName may have a prefix. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as insertBefore() or appendChild().

createTextNode( data)
  Create and return a text node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

createComment( data)
  Create and return a comment node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

createProcessingInstruction( target, data)
  Create and return a processing instruction node containing the target and data passed as parameters. As with the other creation methods, this one does not insert the node into the tree.
**createAttribute**(*name*)
Create and return an attribute node. This method does not associate the attribute node with any particular element. You must use `setAttributeNode()` on the appropriate `Element` object to use the newly created attribute instance.

**createAttributeNS**(*namespaceURI, qualifiedName*)
Create and return an attribute node with a namespace. The `tagName` may have a prefix. This method does not associate the attribute node with any particular element. You must use `setAttributeNode()` on the appropriate `Element` object to use the newly created attribute instance.

**getElementsByTagName**(*tagName*)
Search for all descendants (direct children, children's children, etc.) with a particular element type name.

**getElementsByTagNameNS**(*namespaceURI, localName*)
Search for all descendants (direct children, children's children, etc.) with a particular namespace URI and localname. The localname is the part of the namespace after the prefix.
8.6.2.6 Element Objects

Element is a subclass of Node, so inherits all the attributes of that class.

tagName
The element type name. In a namespace-using document it may have colons in it. The value is a string.

getElementsByTagName(tagName)
Same as equivalent method in the Document class.

getElementsByTagNameNS(tagName)
Same as equivalent method in the Document class.

hasAttribute(name)
Returns true if the element has an attribute named by name.

hasAttributeNS(namespaceURI, localName)
Returns true if the element has an attribute named by namespaceURI and localName.

getAttribute(name)
Return the value of the attribute named by name as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

getAttributeNode(attrname)
Return the Attr node for the attribute named by attrname.

getAttributeNS(namespaceURI, localName)
Return the value of the attribute named by namespaceURI and localName as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

getAttributeNodeNS(namespaceURI, localName)
Return an attribute value as a node, given a namespaceURI and localName.

**removeAttribute**(name)
Remove an attribute by name. No exception is raised if there is no matching attribute.

**removeAttributeNode**(oldAttr)
Remove and return oldAttr from the attribute list, if present. If oldAttr is not present, NotFoundErr is raised.

**removeAttributeNS**(namespaceURI, localName)
Remove an attribute by name. Note that it uses a localName, not a qname. No exception is raised if there is no matching attribute.

**setAttribute**(name, value)
Set an attribute value from a string.

**setAttributeNode**(newAttr)
Add a new attribute node to the element, replacing an existing attribute if necessary if the name attribute matches. If a replacement occurs, the old attribute node will be returned. If newAttr is already in use, InuseAttributeErr will be raised.

**setAttributeNodeNS**(newAttr)
Add a new attribute node to the element, replacing an existing attribute if necessary if the namespaceURI and localName attributes match. If a replacement occurs, the old attribute node will be returned. If newAttr is already in use, InuseAttributeErr will be raised.

**setAttributeNS**(namespaceURI, qname, value)
Set an attribute value from a string, given a namespaceURI and a qname. Note that a qname is the whole attribute name. This is different than above.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
8.6.2.7 Attr Objects

Attr inherits from Node, so inherits all its attributes.

name
The attribute name. In a namespace-using document it may have colons in it.

localName
The part of the name following the colon if there is one, else the entire name. This is a read-only attribute.

prefix
The part of the name preceding the colon if there is one, else the empty string.
8.6.2.8 NamedNodeMap Objects

NamedNodeMap does not inherit from Node.

length
The length of the attribute list.

item(index)
Return an attribute with a particular index. The order you get the attributes in is arbitrary but will be consistent for the life of a DOM. Each item is an attribute node. Get its value with the value attribute.

There are also experimental methods that give this class more mapping behavior. You can use them or you can use the standardized getAttribute*() family of methods on the Element objects.
8.6.2.9 Comment Objects

Comment represents a comment in the XML document. It is a subclass of Node, but cannot have child nodes.

data
The content of the comment as a string. The attribute contains all characters between the leading <!- - - and trailing - - >, but does not include them.

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
8.6.2.10 Text and CDATASection Objects

The Text interface represents text in the XML document. If the parser and DOM implementation support the DOM's XML extension, portions of the text enclosed in CDATA marked sections are stored in CDATASection objects. These two interfaces are identical, but provide different values for the nodeType attribute.

These interfaces extend the Node interface. They cannot have child nodes.

**data**

The content of the text node as a string.

**Note:** The use of a CDATASection node does not indicate that the node represents a complete CDATA marked section, only that the content of the node was part of a CDATA section. A single CDATA section may be represented by more than one node in the document tree. There is no way to determine whether two adjacent CDATASection nodes represent different CDATA marked sections.
8.6.2.11 ProcessingInstruction Objects

Represents a processing instruction in the XML document; this inherits from the Node interface and cannot have child nodes.

**target**
The content of the processing instruction up to the first whitespace character. This is a read-only attribute.

**data**
The content of the processing instruction following the first whitespace character.
8.6.2.12 Exceptions

New in version 2.1.

The DOM Level 2 recommendation defines a single exception, DOMException, and a number of constants that allow applications to determine what sort of error occurred. DOMException instances carry a code attribute that provides the appropriate value for the specific exception.

The Python DOM interface provides the constants, but also expands the set of exceptions so that a specific exception exists for each of the exception codes defined by the DOM. The implementations must raise the appropriate specific exception, each of which carries the appropriate value for the code attribute.

**exception DOMException**

Base exception class used for all specific DOM exceptions. This exception class cannot be directly instantiated.

**exception DomstringSizeErr**

Raised when a specified range of text does not fit into a string. This is not known to be used in the Python DOM implementations, but may be received from DOM implementations not written in Python.

**exception HierarchyRequestErr**

Raised when an attempt is made to insert a node where the node type is not allowed.

**exception IndexSizeErr**

Raised when an index or size parameter to a method is negative or exceeds the allowed values.

**exception InuseAttributeErr**

Raised when an attempt is made to insert an Attr node that is already present elsewhere in the document.

**exception InvalidAccessErr**
Raised if a parameter or an operation is not supported on the underlying object.

**exception InvalidCharacterErr**
This exception is raised when a string parameter contains a character that is not permitted in the context it's being used in by the XML 1.0 recommendation. For example, attempting to create an `Element` node with a space in the element type name will cause this error to be raised.

**exception InvalidModificationErr**
Raised when an attempt is made to modify the type of a node.

**exception InvalidStateErr**
Raised when an attempt is made to use an object that is not defined or is no longer usable.

**exception NamespaceErr**
If an attempt is made to change any object in a way that is not permitted with regard to the *Namespaces in XML* recommendation, this exception is raised.

**exception NotFoundErr**
Exception when a node does not exist in the referenced context. For example, `NamedNodeMap.removeNamedItem()` will raise this if the node passed in does not exist in the map.

**exception NotSupportedErr**
Raised when the implementation does not support the requested type of object or operation.

**exception NoDataAllowedErr**
This is raised if data is specified for a node which does not support data.

**exception NoModificationAllowedErr**
Raised on attempts to modify an object where modifications are not allowed (such as for read-only nodes).

**exception SyntaxErr**
Raised when an invalid or illegal string is specified.
**exception WrongDocumentErr**

Raised when a node is inserted in a different document than it currently belongs to, and the implementation does not support migrating the node from one document to the other.

The exception codes defined in the DOM recommendation map to the exceptions described above according to this table:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMSTRING_SIZE_ERR</td>
<td>DomstringSizeErr</td>
</tr>
<tr>
<td>HIERARCHY_REQUEST_ERR</td>
<td>HierarchyRequestErr</td>
</tr>
<tr>
<td>INDEX_SIZE_ERR</td>
<td>IndexSizeErr</td>
</tr>
<tr>
<td>INUSE_ATTRIBUTE_ERR</td>
<td>InuseAttributeErr</td>
</tr>
<tr>
<td>INVALID_ACCESS_ERR</td>
<td>InvalidAccessErr</td>
</tr>
<tr>
<td>INVALID_CHARACTER_ERR</td>
<td>InvalidCharacterErr</td>
</tr>
<tr>
<td>INVALID_MODIFICATION_ERR</td>
<td>InvalidModificationErr</td>
</tr>
<tr>
<td>INVALID_STATE_ERR</td>
<td>InvalidStateErr</td>
</tr>
<tr>
<td>NAMESPACE_ERR</td>
<td>NamespaceErr</td>
</tr>
<tr>
<td>NOT_FOUND_ERR</td>
<td>NotFoundErr</td>
</tr>
<tr>
<td>NOT_SUPPORTED_ERR</td>
<td>NotSupportedErr</td>
</tr>
<tr>
<td>NO_DATA_ALLOWED_ERR</td>
<td>NoDataAllowedErr</td>
</tr>
<tr>
<td>NO_MODIFICATION_ALLOWED_ERR</td>
<td>NoModificationAllowedErr</td>
</tr>
<tr>
<td>SYNTAX_ERR</td>
<td>SyntaxErr</td>
</tr>
<tr>
<td>WRONG_DOCUMENT_ERR</td>
<td>WrongDocumentErr</td>
</tr>
</tbody>
</table>

---

**Python Library Reference**

Previous: [8.6.2.11 ProcessingInstruction Objects](#)  
Up: [8.6.2 Objects in the](#)  
Next: [8.6.3 Conformance](#)  

Release 2.5, documentation updated on 19th September, 2006.  
See [About this document...](#) for information on suggesting changes.
8.6.3.1 Type Mapping

The primitive IDL types used in the DOM specification are mapped to Python types according to the following table.

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Python Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>IntegerType (with a value of 0 or 1)</td>
</tr>
<tr>
<td>int</td>
<td>IntegerType</td>
</tr>
<tr>
<td>long int</td>
<td>IntegerType</td>
</tr>
<tr>
<td>unsigned int</td>
<td>IntegerType</td>
</tr>
</tbody>
</table>

Additionally, the DOMString defined in the recommendation is mapped to a Python string or Unicode string. Applications should be able to handle Unicode whenever a string is returned from the DOM.

The IDL null value is mapped to None, which may be accepted or provided by the implementation whenever null is allowed by the API.
The mapping from OMG IDL to Python defines accessor functions for IDL attribute declarations in much the way the Java mapping does. Mapping the IDL declarations

```idl
readonly attribute string someValue;
attribute string anotherValue;
```

yields three accessor functions: a `"get" method for `someValue` (_get_someValue()), and `"get" and `"set" methods for `anotherValue` (_get_anotherValue() and _set_anotherValue()). The mapping, in particular, does not require that the IDL attributes are accessible as normal Python attributes: `object.someValue` is not required to work, and may raise an `AttributeError`.

The Python DOM API, however, does require that normal attribute access work. This means that the typical surrogates generated by Python IDL compilers are not likely to work, and wrapper objects may be needed on the client if the DOM objects are accessed via CORBA. While this does require some additional consideration for CORBA DOM clients, the implementers with experience using DOM over CORBA from Python do not consider this a problem. Attributes that are declared `readonly` may not restrict write access in all DOM implementations.

In the Python DOM API, accessor functions are not required. If provided, they should take the form defined by the Python IDL mapping, but these methods are considered unnecessary since the attributes are accessible directly from Python. `"Set" accessors should never be provided for `readonly` attributes.

The IDL definitions do not fully embody the requirements of the W3C DOM API, such as the notion of certain objects, such as the return value of `getElementsByTagName()`, being `"live". The Python DOM API does not require implementations to enforce such requirements.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
def handleSlideshowTitle(slideshowTitle)
slides = slideshow.getElementsByTagName("title")
handleToc(slides)
handleSlides(slides) print "" def handleSlides(slides): for slide in slides:
handleSlide(slide) def handleSlide(slide):
handleSlideTitle(slide.getElementsByTagName("slide"))
handlePoints(slide.getElementsByTagName("point")) def
handleSlideshowTitle(title): print "" % getText(title.childNodes)
def handleSlideTitle(title): print ""
" % getText(title.childNodes) def handlePoints(points): print "
    " for point in points: handlePoint(point) print "

" def handlePoint(point): print "

" %s

" % getText(point.childNodes) def handleToc(slides): for slide in slides: title = slide.getElementsByTagName("title")[0] print "

%s

" % getText(title.childNodes) handleSlideshow(dom)
13.1.5.1 Pickling and unpickling normal class instances

When a pickled class instance is unpickled, its __init__() method is normally *not* invoked. If it is desirable that the __init__() method be called on unpickling, an old-style class can define a method __getinitargs__(), which should return a *tuple* containing the arguments to be passed to the class constructor (__init__() for example). The __getinitargs__() method is called at pickle time; the tuple it returns is incorporated in the pickle for the instance.

New-style types can provide a __getnewargs__() method that is used for protocol 2. Implementing this method is needed if the type establishes some internal invariants when the instance is created, or if the memory allocation is affected by the values passed to the __new__() method for the type (as it is for tuples and strings). Instances of a new-style type C are created using

```
obj = c.__new__(c, *args)
```

where *args* is the result of calling __getnewargs__() on the original object; if there is no __getnewargs__() method, an empty tuple is assumed.

Classes can further influence how their instances are pickled; if the class defines the method __getstate__(), it is called and the return state is pickled as the contents for the instance, instead of the contents of the instance's dictionary. If there is no __getstate__() method, the instance's __dict__ is pickled.

Upon unpickling, if the class also defines the method __setstate__(), it is called with the unpickled state. If there is no __setstate__() method, the pickled state must be a dictionary and its items are assigned to the new instance's dictionary. If a class defines both __getstate__() and __setstate__(), the state object needn't be a dictionary and these methods can do what they want.
**Warning:** For new-style classes, if `__getstate__()` returns a false value, the `__setstate__()` method will not be called.

---

**Footnotes**

... state.\(^{13.5}\)
These methods can also be used to implement copying class instances.

... want.\(^{13.6}\)
This protocol is also used by the shallow and deep copying operations defined in the `copy` module.

---

*Previous:* 13.1.5 The pickle protocol  *Up:* 13.1.5 The pickle protocol  *Next:* 13.1.5.2 Pickling and unpickling

Release 2.5, documentation updated on 19th September, 2006.
See *About this document...* for information on suggesting changes.
13.1.5.2 Pickling and unpickling extension types

When the Pickler encounters an object of a type it knows nothing about -- such as an extension type -- it looks in two places for a hint of how to pickle it. One alternative is for the object to implement a \_reduce\_ method. If provided, at pickling time \_reduce\_ will be called with no arguments, and it must return either a string or a tuple.

If a string is returned, it names a global variable whose contents are pickled as normal. The string returned by \_reduce\_ should be the object's local name relative to its module; the pickle module searches the module namespace to determine the object's module.

When a tuple is returned, it must be between two and five elements long. Optional elements can either be omitted, or None can be provided as their value. The semantics of each element are:

- A callable object that will be called to create the initial version of the object. The next element of the tuple will provide arguments for this callable, and later elements provide additional state information that will subsequently be used to fully reconstruct the pickled date.

  In the unpickling environment this object must be either a class, a callable registered as a "safe constructor" (see below), or it must have an attribute \_safe_for unpickling\_ with a true value. Otherwise, an UnpicklingError will be raised in the unpickling environment. Note that as usual, the callable itself is pickled by name.

- A tuple of arguments for the callable object. Changed in version 2.5: Formerly, this argument could also be None.

- Optionally, the object's state, which will be passed to the object's \_setstate\_ method as described in section 13.1.5. If the object has no \_setstate\_ method, then, as above, the value must be a dictionary and it will be added to the object's __dict__. 

• Optionally, an iterator (and not a sequence) yielding successive list items. These list items will be pickled and appended to the object using either `obj.append(item)` or `obj.extend(list_of_items)`. This is primarily used for list subclasses, but may be used by other classes as long as they have `append()` and `extend()` methods with the appropriate signature. (Whether `append()` or `extend()` is used depends on which pickle protocol version is used as well as the number of items to append, so both must be supported.)

• Optionally, an iterator (not a sequence) yielding successive dictionary items, which should be tuples of the form `(key, value)`. These items will be pickled and stored to the object using `obj[key] = value`. This is primarily used for dictionary subclasses, but may be used by other classes as long as they implement `__setitem__`.

It is sometimes useful to know the protocol version when implementing `__reduce__`. This can be done by implementing a method named `__reduce_ex__` instead of `__reduce__`. `__reduce_ex__`, when it exists, is called in preference over `__reduce__` (you may still provide `__reduce__` for backwards compatibility). The `__reduce_ex__` method will be called with a single integer argument, the protocol version.

The `object` class implements both `__reduce__` and `__reduce_ex__`; however, if a subclass overrides `__reduce__` but not `__reduce_ex__`, the `__reduce_ex__` implementation detects this and calls `__reduce__`.

An alternative to implementing a `__reduce__()` method on the object to be pickled, is to register the callable with the `copy_reg` module. This module provides a way for programs to register "reduction functions" and constructors for user-defined types. Reduction functions have the same semantics and interface as the `__reduce__()` method described above, except that they are called with a single argument, the object to be pickled.

The registered constructor is deemed a "safe constructor" for purposes of unpickling as described above.
13.1.5.3 Pickling and unpickling external objects

For the benefit of object persistence, the pickle module supports the notion of a reference to an object outside the pickled data stream. Such objects are referenced by a ``persistent id'', which is just an arbitrary string of printable ASCII characters. The resolution of such names is not defined by the pickle module; it will delegate this resolution to user defined functions on the pickler and unpickler.\textsuperscript{13.7}

To define external persistent id resolution, you need to set the \texttt{persistent\_id} attribute of the pickler object and the \texttt{persistent\_load} attribute of the unpickler object.

To pickle objects that have an external persistent id, the pickler must have a custom \texttt{persistent\_id()} method that takes an object as an argument and returns either \texttt{None} or the persistent id for that object. When \texttt{None} is returned, the pickler simply pickles the object as normal. When a persistent id string is returned, the pickler will pickle that string, along with a marker so that the unpickler will recognize the string as a persistent id.

To unpickle external objects, the unpickler must have a custom \texttt{persistent\_load()} function that takes a persistent id string and returns the referenced object.

Here's a silly example that \textit{might} shed more light:

```python
import pickle
from cStringIO import StringIO

src = StringIO()
p = pickle.Pickler(src)

def persistent_id(obj):
    if hasattr(obj, 'x'):
        return 'the value %d' % obj.x
    else:
        return None

p.persistent_id = persistent_id
```
class Integer:
    def __init__(self, x):
        self.x = x
    def __str__(self):
        return 'My name is integer %d' % self.x

i = Integer(7)
print i
p.dump(i)

datastream = src.getvalue()
print repr(datastream)
dst = StringIO(datastream)

up = pickle.Unpickler(dst)

class FancyInteger(Integer):
    def __str__(self):
        return 'I am the integer %d' % self.x

    def persistent_load(persid):
        if persid.startswith('the value '):
            value = int(persid.split()[2])
            return FancyInteger(value)
        else:
            raise pickle.UnpicklingError, 'Invalid persistent id'

up.persistent_load = persistent_load

j = up.load()
print j

In the cPickle module, the unpickler's persistent_load attribute can also be set to a Python list, in which case, when the unpickler reaches a persistent id, the persistent id string will simply be appended to this list. This functionality exists so that a pickle data stream can be ``sniffed'' for object references without actually instantiating all the objects in a pickle. Setting persistent_load to a list is usually used in conjunction with the noload() method on the Unpickler.

Footnotes

13.7 The actual mechanism for associating these user defined functions is
slightly different for pickle and cPickle. The description given here works the same for both implementations. Users of the pickle module could also use subclassing to effect the same results, overriding the persistent_id() and persistent_load() methods in the derived classes.

... pickle.\[13.8\]

We’ll leave you with the image of Guido and Jim sitting around sniffing pickles in their living rooms.
13.13.4.1 Introduction

SQLite natively supports the following types: NULL, INTEGER, REAL, TEXT, BLOB.

The following Python types can thus be sent to SQLite without any problem:

<table>
<thead>
<tr>
<th>Python type</th>
<th>SQLite type</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>NULL</td>
</tr>
<tr>
<td>int</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long</td>
<td>INTEGER</td>
</tr>
<tr>
<td>float</td>
<td>REAL</td>
</tr>
<tr>
<td>str (UTF8-encoded)</td>
<td>TEXT</td>
</tr>
<tr>
<td>unicode</td>
<td>TEXT</td>
</tr>
<tr>
<td>buffer</td>
<td>BLOB</td>
</tr>
</tbody>
</table>

This is how SQLite types are converted to Python types by default:

<table>
<thead>
<tr>
<th>SQLite type</th>
<th>Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>None</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int or long, depending on size</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>TEXT</td>
<td>depends on text_factory, unicode by default</td>
</tr>
<tr>
<td>BLOB</td>
<td>buffer</td>
</tr>
</tbody>
</table>

The type system of the sqlite3 module is extensible in two ways: you can store additional Python types in a SQLite database via object adaptation, and you can let the sqlite3 module convert SQLite types to different Python types via converters.
13.13.4.2 Using adapters to

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
As described before, SQLite supports only a limited set of types natively. To use other Python types with SQLite, you must adapt them to one of the sqlite3 module's supported types for SQLite: one of NoneType, int, long, float, str, unicode, buffer.

The sqlite3 module uses Python object adaptation, as described in PEP 246 for this. The protocol to use is PrepareProtocol.

There are two ways to enable the sqlite3 module to adapt a custom Python type to one of the supported ones.

### 13.13.4.2.1 Letting your object adapt itself

This is a good approach if you write the class yourself. Let's suppose you have a class like this:

```python
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y
```

Now you want to store the point in a single SQLite column. First you'll have to choose one of the supported types first to be used for representing the point. Let's just use str and separate the coordinates using a semicolon. Then you need to give your class a method `__conform__` which must return the converted value. The parameter `protocol` will be PrepareProtocol.
**13.13.4.2.2 Registering an adapter callable**

The other possibility is to create a function that converts the type to the string representation and register the function with `register_adapter`.

**Note:** The type/class to adapt must be a new-style class, i.e. it must have `object` as one of its bases.
The `sqlite3` module has two default adapters for Python's built-in `datetime.date` and `datetime.datetime` types. Now let's suppose we want to store `datetime.datetime` objects not in ISO representation, but as a UNIX timestamp.

```python
import sqlite3
import datetime, time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(":memory:")
cur = con.cursor()

now = datetime.datetime.now()
cur.execute("select ?", (now,))
print cur.fetchone()[0]
```

Download as text (original file name: `sqlite3/adapter_datetime.py`).
13.13.4.3 Converting SQLite values to custom Python types

Writing an adapter lets you send custom Python types to SQLite. But to make it really useful we need to make the Python to SQLite to Python roundtrip work.

Enter converters.

Let's go back to the `Point` class. We stored the x and y coordinates separated via semicolons as strings in SQLite.

First, we'll define a converter function that accepts the string as a parameter and constructs a `Point` object from it.

```python
def convert_point(s):
    x, y = map(float, s.split(';'))
    return Point(x, y)
```

Now you need to make the `sqlite3` module know that what you select from the database is actually a point. There are two ways of doing this:

- Implicitly via the declared type
- Explicitly via the column name

Both ways are described in `Module Constants`, section 13.13.1, in the entries for the constants `PARSE_DECLTYPES` and `PARSE_COLNAMES`.

The following example illustrates both approaches.
```python
import sqlite3

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "(%f;%f)" % (self.x, self.y)

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

def convert_point(s):
    x, y = map(float, s.split(";"))
    return Point(x, y)

# Register the adapter
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

# 1) Using declared types
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
print "with declared types:", cur.fetchone()[0]
cur.close()
con.close()

# 1) Using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAME)
cur = con.cursor()
cur.execute("create table test(p)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" as "p [point]" from test')
print "with column names:", cur.fetchone()[0]
cur.close()
con.close()
```

Download as text (original file name: sqlite3/converter_point.py).
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
13.13.4.4 Default adapters and converters

There are default adapters for the date and datetime types in the datetime module. They will be sent as ISO dates/ISO timestamps to SQLite.

The default converters are registered under the name "date" for `datetime.date` and under the name "timestamp" for `datetime.datetime`.

This way, you can use date/timestamps from Python without any additional fiddling in most cases. The format of the adapters is also compatible with the experimental SQLite date/time functions.

The following example demonstrates this.

```python
import sqlite3
import datetime

con = sqlite3.connect(':memory:', detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)"

today = datetime.date.today()
own = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print today, "=>", row[0], type(row[0])
print now, "=>", row[1], type(row[1])

cur.execute('select current_date as "d [date]", current_timestamp as "ts [timestamp]"')
row = cur.fetchone()
print "current_date", row[0], type(row[0])
print "current_timestamp", row[1], type(row[1])
```

Download as text (original file name: sqlite3/pysqlite_datetime.py).
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
13.13.6.1 Using shortcut methods

Using the nonstandard `execute`, `executemany` and `executescript` methods of the `Connection` object, your code can be written more concisely because you don't have to create the (often superfluous) `Cursor` objects explicitly. Instead, the `Cursor` objects are created implicitly and these shortcut methods return the cursor objects. This way, you can execute a SELECT statement and iterate over it directly using only a single call on the `Connection` object.

```python
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")
]

con = sqlite3.connect(":memory:"

# Create the table
con.execute("create table person(firstname, lastname)"

# Fill the table
con.executemany("insert into person(firstname, lastname) values (?,

# Print the table contents
for row in con.execute("select firstname, lastname from person"):
    print row

# Using a dummy WHERE clause to not let SQLite take the shortcut tab
print "I just deleted", con.execute("delete from person where 1=1").
```

Download as text (original file name: sqlite3/shortcut_methods.py).
13.13.6.2 Accessing columns by name instead of by index

One useful feature of the sqlite3 module is the builtin sqlite3.Row class designed to be used as a row factory.

Rows wrapped with this class can be accessed both by index (like tuples) and case-insensitively by name:

```python
import sqlite3
con = sqlite3.connect("mydb")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select name_last, age from people")
for row in cur:
    assert row[0] == row["name_last"]
    assert row["name_last"] == row["nAmE_lAsT"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]
```

Download as text (original file name: sqlite3/rowclass.py).
14.3.1.1 Terminology

**Argument**

A string entered on the command-line, and passed by the shell to `exec1()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (`sys.argv[0]` is the name of the program being executed). UNIX shells also use the term "word".

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read "argument" as "an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]".

**Option**

An argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional UNIX syntax is a hyphen (`-`) followed by a single letter, e.g. "-x" or "-F". Also, traditional UNIX syntax allows multiple options to be merged into a single argument, e.g. "-x -F" is equivalent to "-xF". The GNU project introduced "-" followed by a series of hyphen-separated words, e.g. "-file" or "-dry-run". These are the only two option syntaxes provided by `optparse`.

Some other option syntaxes that the world has seen include:

- A hyphen followed by a few letters, e.g. "-pf" (this is not the same as multiple options merged into a single argument)

- A hyphen followed by a whole word, e.g. "-file" (this is technically equivalent to the previous syntax, but they aren't usually seen in the same program)

- A plus sign followed by a single letter, or a few letters, or a word, e.g. "+f", "+rgb"
• a slash followed by a letter, or a few letters, or a word, e.g. "/f", "/file"

These option syntaxes are not supported by optparse, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you're exclusively targeting VMS, MS-DOS, and/or Windows.

**option argument**

an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With optparse, option arguments may either be in a separate argument from their option:

```
-f foo
--file foo
```

or included in the same argument:

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn't. Lots of people want an "optional option arguments" feature, meaning that some options will take an argument if they see it, and won't if they don't. This is somewhat controversial, because it makes parsing ambiguous: if "-a" takes an optional argument and "-b" is another option entirely, how do we interpret "-ab"? Because of this ambiguity, optparse does not support this feature.

**positional argument**

something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

**required option**

an option that must be supplied on the command-line; note that the phrase "required option" is self-contradictory in English. optparse doesn't prevent you from implementing required options, but doesn't give you much help at it either. See examples/required_1.py and examples/required_2.py in the optparse source distribution for
two ways to implement required options with `optparse`.

For example, consider this hypothetical command-line:

```
prog -v --report /tmp/report.txt foo bar
```

"-v" and "--report" are both options. Assuming that `--report` takes one argument, "/tmp/report.txt" is an option argument. "foo" and "bar" are positional arguments.
14.3.1.2 What are options for?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn't clear, options are usually optional. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main exceptions are find, tar, and dd--all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have "required options". Think about it. If it's required, then it's not optional! If there is a piece of information that your program absolutely requires in order to run successfully, that's what positional arguments are for.

As an example of good command-line interface design, consider the humble cp utility, for copying files. It doesn't make much sense to try to copy files without supplying a destination and at least one source. Hence, cp fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most cp implementations provide a bunch of options to tweak exactly how the files are copied: you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of cp, which is to copy either one file to another, or several files to another directory.
See About this document... for information on suggesting changes.
14.3.1.3 What are positional arguments for?

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn't much matter how you get that information from the user--most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI: if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply--use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That's what options are for. Again, it doesn't matter if they are entries in a config file, widgets in the `Preferences` dialog of a GUI, or command-line options--the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course; too many options can overwhelm users and make your code much harder to maintain.
14.3.2.1 Understanding option actions

Actions tell optparse what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into optparse; adding new actions is an advanced topic covered in section 14.3.5, Extending optparse. Most actions tell optparse to store a value in some variable—for example, take a string from the command line and store it in an attribute of options.

If you don't specify an option action, optparse defaults to store.
14.3.2.2 The store action

The most common option action is store, which tells optparse to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

For example:

```python
parser.add_option("-f", "--file",
                 action="store", type="string", dest="filename")
```

Now let's make up a fake command line and ask optparse to parse it:

```python
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When optparse sees the option string "-f", it consumes the next argument, "foo.txt", and stores it in options.filename. So, after this call to parse_args(), options.filename is "foo.txt".

Some other option types supported by optparse are int and float. Here's an option that expects an integer argument:

```python
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there's no explicit action, since the default is store.

Let's parse another fake command-line. This time, we'll jam the option argument right up against the option: since "-n42" (one argument) is equivalent to "-n 42" (two arguments), the code

```python
(options, args) = parser.parse_args(["-n42"])
print options.num
```

will print "42".

If you don't specify a type, optparse assumes string. Combined with the
fact that the default action is \texttt{store}, that means our first example can be a lot shorter:

\begin{verbatim}
parser.add_option("-f", "--file", dest="filename")
\end{verbatim}

If you don't supply a destination, \texttt{optparse} figures out a sensible default from the option strings: if the first long option string is "-foo-bar", then the default destination is \texttt{foo_bar}. If there are no long option strings, \texttt{optparse} looks at the first short option string: the default destination for "-f" is \texttt{f}.

\texttt{optparse} also includes built-in \texttt{long} and \texttt{complex} types. Adding types is covered in section 14.3.5, Extending \texttt{optparse}.
14.3.2.3 Handling boolean (flag) options

Flag options--set a variable to true or false when a particular option is seen--are quite common. optparse supports them with two separate actions, store_true and store_false. For example, you might have a verbose flag that is turned on with "-v" and off with "-q":

```
pARSER.add_option("-v", action="store_true", dest="verbose")
pARSER.add_option("-q", action="store_false", dest="verbose")
```

Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values--see below.)

When optparse encounters "-v" on the command line, it sets options.verbose to True; when it encounters "-q", options.verbose is set to False.
14.3.2.4 Other actions

Some other actions supported by `optparse` are:

- `store_const`: store a constant value
- `append`: append this option's argument to a list
- `count`: increment a counter by one
- `callback`: call a specified function

These are covered in section 14.3.3, Reference Guide and section 14.3.4, Option Callbacks.
14.3.2.5 Default values

All of the above examples involve setting some variable (the "destination") when certain command-line options are seen. What happens if those options are never seen? Since we didn't supply any defaults, they are all set to None. This is usually fine, but sometimes you want more control. optparse lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want optparse to set verbose to True unless "-q" is seen, then we can do this:

```python
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the destination rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent:

```python
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Consider this:

```python
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose", default=False)
```

Again, the default value for verbose will be True: the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the set_defaults() method of OptionParser, which you can call at any time before calling parse_args():

```python
parser.set_defaults(verbos=True)
parser.add_option(...)  
(options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that
counts. For clarity, try to use one method or the other of setting default values, not both.
14.3.2.6 Generating help

`optparse`'s ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a `help` value for each option, and optionally a short usage message for your whole program. Here's an OptionParser populated with user-friendly (documented) options:

```python
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parsers.add_option("-v", "--verbose",
                 action="store_true", dest="verbose", default=True,
                 help="make lots of noise [default]"
) parsers.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)"
) parsers.add_option("-f", "--filename",
                metavar="FILE", help="write output to FILE"
) parsers.add_option("-m", "--mode",
            default="intermediate",
            help="interaction mode: novice, intermediate, "
            "or expert [default: %default]"
)
```

If `optparse` encounters either "-h" or "-help" on the command-line, or if you just call `parser.print_help()`, it prints the following to standard output:

```
usage: <yourscript> [options] arg1 arg2

options:
  -h, --help                     show this help message and exit
  -v, --verbose                  make lots of noise [default]
  -q, --quiet                    be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE       write output to FILE
  -m MODE, --mode=MODE           interaction mode: novice, intermediate, or
                                 expert [default: intermediate]
```

(If the help output is triggered by a help option, `optparse` exits after printing the help text.)

There's a lot going on here to help `optparse` generate the best possible help
the script defines its own usage message:

   usage = "usage: %prog [options] arg1 arg2"

optparse expands "%prog" in the usage string to the name of the current program, i.e. os.path.basename(sys.argv[0]). The expanded string is then printed before the detailed option help.

If you don't supply a usage string, optparse uses a bland but sensible default: ``usage: %prog [options]'', which is fine if your script doesn't take any positional arguments.

- every option defines a help string, and doesn't worry about line-wrapping--optparse takes care of wrapping lines and making the help output look good.

- options that take a value indicate this fact in their automatically-generated help message, e.g. for the ``mode'' option:

   -m MODE, --mode=MODE

Here, ``MODE'' is called the meta-variable: it stands for the argument that the user is expected to supply to -m/--mode. By default, optparse converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that's not what you want--for example, the --filename option explicitly sets metavar="FILE", resulting in this automatically-generated option description:

   -f FILE, --filename=FILE

This is important for more than just saving space, though: the manually written help text uses the meta-variable ``FILE'' to clue the user in that there's a connection between the semi-formal syntax ``-f FILE'' and the informal semantic description ``write output to FILE''. This is a simple but effective way to make your help text a lot clearer and more useful for end users.

- options that have a default value can include %default in the help string-
  -optparse will replace it with str() of the option's default value. If an
option has no default value (or the default value is None), %default expands to none.
14.3.2.7 Printing a version string

Similar to the brief usage string, optparse can also print a version string for your program. You have to supply the string as the version argument to OptionParser:

```python
def main():
    parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

"%prog" is expanded just like it is in usage. Apart from that, version can contain anything you like. When you supply it, optparse automatically adds a "-version" option to your parser. If it encounters this option on the command line, it expands your version string (by replacing "%prog"), prints it to stdout, and exits.

For example, if your script is called /usr/bin/foo:

```bash
$ /usr/bin/foo --version
foo 1.0
```
14.3.2.8 How optparse handles errors

There are two broad classes of errors that optparse has to worry about: programmer errors and user errors. Programmer errors are usually erroneous calls to parser.add_option(), e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way: raise an exception (either optparse.OptionError or TypeError) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. optparse can automatically detect some user errors, such as bad option arguments (passing "-n 4x" where -n takes an integer argument), missing arguments ("-n" at the end of the command line, where -n takes an argument of any type). Also, you can call parser.error() to signal an application-defined error condition:

```python
(options, args) = parser.parse_args()
[...]
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, optparse handles the error the same way: it prints the program's usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes "4x" to an option that takes an integer:

```
$ /usr/bin/foo -n 4x
usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all:

```
$ /usr/bin/foo -n
usage: foo [options]
```
foo: error: -n option requires an argument

optparse-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling \texttt{parser.error()} from your application code.

If \texttt{optparse}'s default error-handling behaviour does not suite your needs, you'll need to subclass \texttt{OptionParser} and override \texttt{exit()} and/or \texttt{error()}.
14.3.2.9 Putting it all together

Here's what `optparse`-based scripts usually look like:

```python
from optparse import OptionParser

def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option(""'-f", "'--file", dest="filename",
                     help="read data from FILENAME")
    parser.add_option(""'-v", "'--verbose",
                     action="store_true", dest="verbose")
    parser.add_option(""'-q", "'--quiet",
                     action="store_false", dest="verbose")
    [...]
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print "reading %s..." % options.filename
    [...]
    if __name__ == "__main__":
        main()
```

Python Library Reference
14.3.3.1 Creating the parser

The first step in using `optparse` is to create an OptionParser instance:

```python
parser = OptionParser(...)  
```

The OptionParser constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

**usage** (default: "%prog [options]")

The usage summary to print when your program is run incorrectly or with a help option. When `optparse` prints the usage string, it expands `%prog` to `os.path.basename(sys.argv[0])` (or to `prog` if you passed that keyword argument). To suppress a usage message, pass the special value `optparse.SUPPRESS_USAGE`.

**option_list** (default: `[]`)

A list of `Option` objects to populate the parser with. The options in `option_list` are added after any options in `standard_option_list` (a class attribute that may be set by OptionParser subclasses), but before any version or help options. Deprecated; use `add_option()` after creating the parser instead.

**option_class** (default: `optparse.Option`)

Class to use when adding options to the parser in `add_option()`.

**version** (default: `None`)

A version string to print when the user supplies a version option. If you supply a true value for `version`, `optparse` automatically adds a version option with the single option string "-version". The substring "%prog" is expanded the same as for `usage`.

**conflict_handler** (default: "error")

Specifies what to do when options with conflicting option strings are added to the parser; see section 14.3.3, Conflicts between options.

**description** (default: `None`)

A paragraph of text giving a brief overview of your program. `optparse` reformats this paragraph to fit the current terminal width and prints it when
the user requests help (after usage, but before the list of options).

**formatter (default: a new IndentedHelpFormatter)**

An instance of optparse.HelpFormatter that will be used for printing help text. optparse provides two concrete classes for this purpose: IndentedHelpFormatter and TitledHelpFormatter.

**add_help_option (default: True)**

If true, optparse will add a help option (with option strings "-h" and "-help") to the parser.

**prog**

The string to use when expanding "%prog" in usage and version instead of os.path.basename(sys.argv[0]).
14.3.3.2 Populating the parser

There are several ways to populate the parser with options. The preferred way is by using `OptionParser.add_option()`, as shown in section 14.3.2, the tutorial. `add_option()` can be called in one of two ways:

- pass it an Option instance (as returned by `make_option()`)
- pass it any combination of positional and keyword arguments that are acceptable to `make_option()` (i.e., to the Option constructor), and it will create the Option instance for you

The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in:

```python
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
]
parser = OptionParser(option_list=option_list)
```

(`make_option()` is a factory function for creating Option instances; currently it is an alias for the Option constructor. A future version of `optparse` may split Option into several classes, and `make_option()` will pick the right class to instantiate. Do not instantiate Option directly.)
14.3.3.3 Defining options

Each Option instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.

The canonical way to create an Option instance is with the `add_option()` method of `OptionParser`:

```
parser.add_option(opt_str[, ...], attr=value, ...)
```

To define an option with only a short option string:

```
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string:

```
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is `action`, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, `optparse` raises an OptionError exception explaining your mistake.

An option's `action` determines what `optparse` does when it encounters this option on the command-line. The standard option actions hard-coded into `optparse` are:

- **store**
  - store this option's argument (default)
- **store_const**
  - store a constant value
- **store_true**
  - store a true value
- **store_false**
  - store a false value
append
   append this option's argument to a list
append_const
   append a constant value to a list
count
   increment a counter by one
callback
   call a specified function
help
   print a usage message including all options and the documentation for them

(If you don't supply an action, the default is store. For this action, you may also supply type and dest option attributes; see below.)

As you can see, most actions involve storing or updating a value somewhere. optparse always creates a special object for this, conventionally called options (it happens to be an instance of optparse.Values). Option arguments (and various other values) are stored as attributes of this object, according to the dest (destination) option attribute.

For example, when you call

    parser.parse_args()

one of the first things optparse does is create the options object:

    options = Values()

If one of the options in this parser is defined with

    parser.add_option("-f", "--file", action="store", type="string", dest

and the command-line being parsed includes any of the following:

    -ffoo
    -f foo
    --file=foo
    --file foo

then optparse, on seeing this option, will do the equivalent of

    options.filename = "foo"
The type and dest option attributes are almost as important as action, but action is the only one that makes sense for all options.
The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide optparse's behaviour; a few have required attributes, which you must specify for any option using that action.

- **store** [relevant: `type`, `dest`, `nargs`, `choices`]

  The option must be followed by an argument, which is converted to a value according to `type` and stored in `dest`. If `nargs > 1`, multiple arguments will be consumed from the command line; all will be converted according to `type` and stored to `dest` as a tuple. See the ``Option types'' section below.

  If `choices` is supplied (a list or tuple of strings), the type defaults to `choice`.

  If `type` is not supplied, it defaults to `string`.

  If `dest` is not supplied, `optparse` derives a destination from the first long option string (e.g., "-foo-bar" implies `foo_bar`). If there are no long option strings, `optparse` derives a destination from the first short option string (e.g., "-f" implies `f`).

Example:

```
parser.add_option("-f")
parser.add_option("-p", type="float", nargs=3, dest="point")
```

As it parses the command line

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

`optparse` will set

```
options.f = "foo.txt"
```
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"

- **store_const** [required: const; relevant: dest]

  The value `const` is stored in `dest`.

  Example:

  ```python
  parser.add_option("-q", "--quiet",
                   action="store_const", const=0, dest="verbose");
  parser.add_option("-v", "--verbose",
                   action="store_const", const=1, dest="verbose");
  parser.add_option("--noisy",
                   action="store_const", const=2, dest="verbose");
  ```

  If "-noisy" is seen, `optparse` will set

  ```python
  options.verbose = 2
  ```

- **store_true** [relevant: dest]

  A special case of `store_const` that stores a true value to `dest`.

- **store_false** [relevant: dest]

  Like `store_true`, but stores a false value.

  Example:

  ```python
  parser.add_option("--clobber", action="store_true", dest="clobber")
  parser.add_option("--no-clobber", action="store_false", dest="clobber")
  ```

- **append** [relevant: type, dest, nargs, choices]

  The option must be followed by an argument, which is appended to the list in `dest`. If no default value for `dest` is supplied, an empty list is automatically created when `optparse` first encounters this option on the command-line. If `nargs > 1`, multiple arguments are consumed, and a tuple of length `nargs` is appended to `dest`.

  The defaults for `type` and `dest` are the same as for the `store` action.
Example:

```python
caller.add_option("-t", "--tracks", action="append", type="int")
```

If "-t3" is seen on the command-line, `optparse` does the equivalent of:

```python
options.tracks = []
options.tracks.append(int("3"))
```

If, a little later on, "-tracks=4" is seen, it does:

```python
options.tracks.append(int("4"))
```

- **append_const [required: const; relevant: dest]**

  Like `store_const`, but the value `const` is appended to `dest`; as with `append`, `dest` defaults to `None`, and an an empty list is automatically created the first time the option is encountered.

- **count [relevant: dest]**

  Increment the integer stored at `dest`. If no default value is supplied, `dest` is set to zero before being incremented the first time.

  Example:

  ```python
  parser.add_option("-v", action="count", dest="verbosity")
  ```

  The first time "-v" is seen on the command line, `optparse` does the equivalent of:

  ```python
  options.verbosity = 0
  options.verbosity += 1
  ```

  Every subsequent occurrence of "-v" results in

  ```python
  options.verbosity += 1
  ```

- **callback [required: callback; relevant: type, nargs, callback_args, callback_kwargs]**

  Call the function specified by `callback`, which is called as
func(option, opt_str, value, parser, *args, **kwargs)

See section 14.3.4, Option Callbacks for more detail.

- help

Prints a complete help message for all the options in the current option parser. The help message is constructed from the usage string passed to OptionParser's constructor and the help string passed to every option.

If no help string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value optparse.SUPPRESS_HELP.

optparse automatically adds a help option to all OptionParsers, so you do not normally need to create one.

Example:

```python
from optparse import OptionParser, SUPPRESS_HELP

parser = OptionParser()
parser.add_option("-h", "--help", action="help"),
parser.add_option("-v", action="store_true", dest="verbose",
    help="Be moderately verbose")
parser.add_option("--file", dest="filename",
    help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

If optparse sees either "-h" or "-help" on the command line, it will print something like the following help message to stdout (assuming sys.argv[0] is "foo.py"):

```plaintext
usage: foo.py [options]

options:
  -h, --help            Show this help message and exit
  -v                   Be moderately verbose
  --file=FILENAME      Input file to read data from
```

After printing the help message, optparse terminates your process with sys.exit(0).

- version
Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the `print_version()` method of OptionParser. Generally only relevant if the `version` argument is supplied to the OptionParser constructor. As with help options, you will rarely create `version` options, since `optparse` automatically adds them when needed.
14.3.3.5 Option attributes

The following option attributes may be passed as keyword arguments to `parser.add_option()`. If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, `optparse` raises OptionError.

- **action** (default: "store")

  Determines `optparse`'s behaviour when this option is seen on the command line; the available options are documented above.

- **type** (default: "string")

  The argument type expected by this option (e.g., "string" or "int"); the available option types are documented below.

- **dest** (default: derived from option strings)

  If the option's action implies writing or modifying a value somewhere, this tells `optparse` where to write it: `dest` names an attribute of the `options` object that `optparse` builds as it parses the command line.

- **default** (deprecated)

  The value to use for this option's destination if the option is not seen on the command line. Deprecated; use `parser.set_defaults()` instead.

- **nargs** (default: 1)

  How many arguments of type `type` should be consumed when this option is seen. If > 1, `optparse` will store a tuple of values to `dest`.

- **const**

  For actions that store a constant value, the constant value to store.
• **choices**

  For options of type "choice", the list of strings the user may choose from.

• **callback**

  For options with action "callback", the callable to call when this option is seen. See section [14.3.4](#), Option Callbacks for detail on the arguments passed to callable.

• **callback_args**, **callback_kwargs**

  Additional positional and keyword arguments to pass to callback after the four standard callback arguments.

• **help**

  Help text to print for this option when listing all available options after the user supplies a help option (such as "-help"). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value `SUPPRESS_HELP`.

• **metavar** (default: derived from option strings)

  Stand-in for the option argument(s) to use when printing help text. See section [14.3.2](#), the tutorial for an example.
14.3.3.6 Standard option types

`optparse` has six built-in option types: `string`, `int`, `long`, `choice`, `float` and `complex`. If you need to add new option types, see section 14.3.5, Extending `optparse`.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type `int` or `long`) are parsed as follows:

- if the number starts with `0x`, it is parsed as a hexadecimal number
- if the number starts with `0`, it is parsed as an octal number
- if the number starts with `0b`, is is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling either `int()` or `long()` with the appropriate base (2, 8, 10, or 16). If this fails, so will `optparse`, although with a more useful error message.

`float` and `complex` option arguments are converted directly with `float()` and `complex()`, with similar error-handling.

`choice` options are a subtype of `string` options. The `choices` option attribute (a sequence of strings) defines the set of allowed option arguments. `optparse.check_choice()` compares user-supplied option arguments against this master list and raises `OptionValueError` if an invalid string is given.
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
14.3.3.7 Parsing arguments

The whole point of creating and populating an OptionParser is to call its `parse_args()` method:

```python
(options, args) = parser.parse_args(args=None, options=None)
```

where the input parameters are

- **args**
  - the list of arguments to process (default: `sys.argv[1:]`)
- **options**
  - object to store option arguments in (default: a new instance of `optparse.Values`)

and the return values are

- **options**
  - the same object that was passed in as `options`, or the `optparse.Values` instance created by `optparse`
- **args**
  - the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply `options`, it will be modified with repeated `setattr()` calls (roughly one for every option argument stored to an option destination) and returned by `parse_args()`.

If `parse_args()` encounters any errors in the argument list, it calls the OptionParser's `error()` method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional UNIX exit status for command-line errors).
14.3.3.8 Querying and manipulating

Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
14.3.3.8 Querying and manipulating your option parser

Sometimes, it's useful to poke around your option parser and see what's there. OptionParser provides a couple of methods to help you out:

**has_option(opt_str)**
Return true if the OptionParser has an option with option string `opt_str` (e.g., "-q" or "-verbose").

**get_option(opt_str)**
Returns the Option instance with the option string `opt_str`, or None if no options have that option string.

**remove_option(opt_str)**
If the OptionParser has an option corresponding to `opt_str`, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If `opt_str` does not occur in any option belonging to this OptionParser, raises ValueError.
14.3.3.9 Conflicts between options

If you're not careful, it's easy to define options with conflicting option strings:

```
    parser.add_option("-n", "--dry-run", ...)  
    [...]  
    parser.add_option("-n", "--noisy", ...)  
```

(This is particularly true if you've defined your own OptionParser subclass with some standard options.)

Every time you add an option, `optparse` checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor:

```
    parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call:

```
    parser.set_conflict_handler(handler)
```

The available conflict handlers are:

**error** *(default)*

- assume option conflicts are a programming error and raise `OptionConflictError`

**resolve**

- resolve option conflicts intelligently (see below)

As an example, let's define an OptionParser that resolves conflicts intelligently and add conflicting options to it:

```
    parser = OptionParser(conflict_handler="resolve")  
    parser.add_option("-n", "--dry-run", ..., help="do no harm")  
    parser.add_option("-n", "--noisy", ..., help="be noisy")
```

At this point, `optparse` detects that a previously-added option is already using the "-n" option string. Since `conflict_handler` is "resolve", it
resolves the situation by removing "-n" from the earlier option's list of option strings. Now "-dry-run" is the only way for the user to activate that option. If the user asks for help, the help message will reflect that:

```python
options:
    --dry-run     do no harm
    [...]        
    -n, --noisy  be noisy
```

It's possible to whittle away the option strings for a previously-added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, `optparse` removes that option completely, so it doesn't show up in help text or anywhere else. Carrying on with our existing OptionParser:

```python
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original `-n/-dry-run` option is no longer accessible, so `optparse` removes it, leaving this help text:

```python
options:
    [...]        
    -n, --noisy  be noisy
    --dry-run    new dry-run option
```

---

**Python Library Reference**

Previous: [14.3.3.8 Querying and manipulating](#)  
Up: [14.3.3 Reference Guide](#)  
Next: [14.3.3.10 Cleanup](#)  

Release 2.5, documentation updated on 19th September, 2006.  
See [About this document...](#) for information on suggesting changes.
14.3.3.10 Cleanup

OptionParser instances have several cyclic references. This should not be a problem for Python's garbage collector, but you may wish to break the cyclic references explicitly by calling `destroy()` on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.
14.3.3.11 Other methods

OptionParser supports several other public methods:

- **set_usage(usage)**

  Set the usage string according to the rules described above for the usage constructor keyword argument. Passing `None` sets the default usage string; use `SUPPRESS_USAGE` to suppress a usage message.

- **enable_interspersed_args()**, **disable_interspersed_args()**

  Enable/disable positional arguments interspersed with options, similar to GNU getopt (enabled by default). For example, if "-a" and "-b" are both simple options that take no arguments, optparse normally accepts this syntax:

  ```
  prog -a arg1 -b arg2
  ```

  and treats it as equivalent to

  ```
  prog -a -b arg1 arg2
  ```

  To disable this feature, call `disable_interspersed_args()`. This restores traditional UNIX syntax, where option parsing stops with the first non-option argument.

- **set_defaults(dest=value, ...)**

  Set default values for several option destinations at once. Using `set_defaults()` is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several "mode" options all set the same destination, any one of them can set the default, and the last one wins:

  ```python
  parser.add_option("--advanced", action="store_const",
  ```
To avoid this confusion, use `set_defaults()`:

```python
default="advanced")  # overridden below
parser.add_option("--advanced", action="store_const",
dest="mode", const="advanced")  # overrides above setting

parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
dest="mode", const="novice")
```
14.3.4.1 Defining a callback option

As always, the easiest way to define a callback option is by using the `parser.add_option()` method. Apart from `action`, the only option attribute you must specify is `callback`, the function to call:

```python
parser.add_option("-c", action="callback", callback=my_callback)
```

callback is a function (or other callable object), so you must have already defined `my_callback()` when you create this callback option. In this simple case, `optparse` doesn't even know if `-c` takes any arguments, which usually means that the option takes no arguments--the mere presence of `-c` on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky; it's covered later in this section.

`optparse` always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via `callback_args` and `callback_kwargs`. Thus, the minimal callback function signature is:

```python
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option:

**type**

has its usual meaning: as with the `store` or `append` actions, it instructs `optparse` to consume one argument and convert it to `type`. Rather than storing the converted value(s) anywhere, though, `optparse` passes it to your callback function.

**nargs**

also has its usual meaning: if it is supplied and > 1, `optparse` will consume `nargs` arguments, each of which must be convertible to `type`. It then passes a tuple of converted values to your callback.
callback_args
   a tuple of extra positional arguments to pass to the callback

callback_kwargs
   a dictionary of extra keyword arguments to pass to the callback
14.3.4.2 How callbacks are called

All callbacks are called as follows:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

where

**option**

is the Option instance that's calling the callback

**opt_str**

is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, **opt_str** will be the full, canonical option string--e.g. if the user puts "-foo" on the command-line as an abbreviation for "-foobar", then **opt_str** will be "-foobar".)

**value**

is the argument to this option seen on the command-line. **optparse** will only expect an argument if **type** is set; the type of **value** will be the type implied by the option's type. If **type** for this option is **None** (no argument expected), then **value** will be **None**. If **nargs** > 1, **value** will be a tuple of values of the appropriate type.

**parser**

is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes:

**parser.largs**

the current list of leftover arguments, i.e. arguments that have been consumed but are neither options nor option arguments. Feel free to modify **parser.largs**, e.g. by adding more arguments to it. (This list will become **args**, the second return value of **parse_args()**.)

**parser.rargs**

the current list of remaining arguments, i.e. with **opt_str** and **value** (if applicable) removed, and only the arguments following them still there. Feel free to modify **parser.rargs**, e.g. by consuming more arguments.

**parser.values**
the object where option values are by default stored (an instance of optparse.OptionValues). This lets callbacks use the same mechanism as the rest of optparse for storing option values; you don't need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

**args**

is a tuple of arbitrary positional arguments supplied via the callback_args option attribute.

**kwargs**

is a dictionary of arbitrary keyword arguments supplied via callback_kwargs.
14.3.4.3 Raising errors in a callback

The callback function should raise OptionValueError if there are any problems with the option or its argument(s). optparse catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what he did wrong.
14.3.4.4 Callback example 1: trivial callback

Here's an example of a callback option that takes no arguments, and simply records that the option was seen:

```python
def record_foo_seen(option, opt_str, value, parser):
    parser.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Of course, you could do that with the `store_true` action.
14.3.4.5 Callback example 2: check option order

Here's a slightly more interesting example: record the fact that "-a" is seen, but blow up if it comes after "-b" in the command-line.

```python
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1

[...]
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

See About this document... for information on suggesting changes.
14.3.4.6 Callback example 3: check option order (generalized)

If you want to re-use this callback for several similar options (set a flag, but blow up if "-b" has already been seen), it needs a bit of work: the error message and the flag that it sets must be generalized.

```python
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)

parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest=
```

Release 2.5, documentation updated on 19th September, 2006.

See About this document... for information on suggesting changes.
14.3.4.7 Callback example 4: check arbitrary condition

Of course, you could put any condition in there--you're not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this:

```python
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full" % opt_str)
    setattr(parser.values, option.dest, 1)
[[...]
parser.add_option("--foo",
    action="callback", callback=check_moon, dest="foo"
)
```

(The definition of is_moon_full() is left as an exercise for the reader.)
14.3.4.8 Callback example 5: fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a store or append option: if you define type, then the option takes one argument that must be convertible to that type; if you further define nargs, then the option takes nargs arguments.

Here's an example that just emulates the standard store action:

```python
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)

[...]
parsed.add_option("--foo",
    action="callback", callback=store_value,
    type="int", nargs=3, dest="foo")
```

Note that optparse takes care of consuming 3 arguments and converting them to integers for you; all you have to do is store them. (Or whatever; obviously you don't need a callback for this example.)
14.3.4.9 Callback example 6: variable arguments

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as optparse doesn't provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional UNIX command-line parsing that optparse normally handles for you. In particular, callbacks should implement the conventional rules for bare "-" and "-" arguments:

- either "-" or "-" can be option arguments
- bare "-" (if not the argument to some option): halt command-line processing and discard the "-"
- bare "-" (if not the argument to some option): halt command-line processing but keep the "-" (append it to parser.largs)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you're willing to make for your application (which is why optparse doesn't support this sort of thing directly).

Nevertheless, here's a stab at a callback for an option with variable arguments:

```python
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    done = 0
    value = []
    rargs = parser.rargs
    while rargs:
        arg = rargs[0]
        # Stop if we hit an arg like "--foo", ":a", ":fx", ":file=f
        # etc. Note that this also stops on ":3" or ":3.0", so if
        # your option takes numeric values, you will need to handle
        # this.
        if ((arg[2:] == "--" and len(arg) > 2) or
            (arg[1:] == "-" and len(arg) > 1 and arg[1] != ":")): break
```
else:
    value.append(arg)
    del rargs[0]

    setattr(parser.values, option.dest, value)

    [...]  
    parser.add_option("-c", "--callback",
                      action="callback", callback=varargs)

The main weakness with this particular implementation is that negative numbers in the arguments following "-c" will be interpreted as further options (probably causing an error), rather than as arguments to "-c". Fixing this is left as an exercise for the reader.
14.3.5.1 Adding new types

To add new types, you need to define your own subclass of `optparse`'s Option class. This class has a couple of attributes that define `optparse`'s types: TYPES and TYPE_CHECKER.

TYPES is a tuple of type names; in your subclass, simply define a new tuple TYPES that builds on the standard one.

TYPE_CHECKER is a dictionary mapping type names to type-checking functions. A type-checking function has the following signature:

```python
def check_mytype(option, opt, value)
```

where option is an Option instance, opt is an option string (e.g., "-f"), and value is the string from the command line that must be checked and converted to your desired type. check_mytype() should return an object of the hypothetical type mytype. The value returned by a type-checking function will wind up in the OptionValues instance returned by OptionParser.parse_args(), or be passed to a callback as the value parameter.

Your type-checking function should raise OptionValueError if it encounters any problems. OptionValueError takes a single string argument, which is passed as-is to OptionParser's error() method, which in turn prepends the program name and the string "error:" and prints everything to stderr before terminating the process.

Here's a silly example that demonstrates adding a complex option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because optparse 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports:

```python
from copy import copy
```
You need to define your type-checker first, since it's referred to later (in the TYPE_CHECKER class attribute of your Option subclass):

```python
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

Finally, the Option subclass:

```python
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(If we didn't make a copy() of Option.TYPE_CHECKER, we would end up modifying the TYPE_CHECKER attribute of optparse's Option class. This being Python, nothing stops you from doing that except good manners and common sense.)

That's it! Now you can write a script that uses the new option type just like any other optparse-based script, except you have to instruct your OptionParser to use MyOption instead of Option:

```python
parser = OptionParser(option_class=MyOption)
presenter.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to OptionParser; if you don't use add_option() in the above way, you don't need to tell OptionParser which option class to use:

```python
option_list = [MyOption("-c", action="store", type="complex", dest="
```

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.3.5.2 Adding new actions

Adding new actions is a bit trickier, because you have to understand that `optparse` has a couple of classifications for actions:

``store'' actions
  actions that result in `optparse` storing a value to an attribute of the current `OptionValues` instance; these options require a `dest` attribute to be supplied to the `Option` constructor

``typed'' actions
  actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a `type` attribute to the `Option` constructor.

These are overlapping sets: some default ```store''` actions are `store`, `store_const`, `append`, and `count`, while the default ```typed''` actions are `store`, `append`, and `callback`.

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of `Option` (all are lists of strings):

**ACTIONS**
  all actions must be listed in `ACTIONS`

**STORE_ACTIONS**
  ```store''` actions are additionally listed here

**TYPED_ACTIONS**
  ```typed''` actions are additionally listed here

**ALWAYS_TYPED_ACTIONS**
  actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that `optparse` assigns the default type, `string`, to options with no explicit type whose action is listed in `ALWAYS_TYPED_ACTIONS`.

In order to actually implement your new action, you must override `Option`'s `take_action()` method and add a case that recognizes your action.
For example, let’s add an `extend` action. This is similar to the standard `append` action, but instead of taking a single value from the command-line and appending it to an existing list, `extend` will take multiple values in a single comma-delimited string, and extend an existing list with them. That is, if `-names` is an `extend` option of type `string`, the command line

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list

```
["foo", "bar", "blah", "ding", "dong"]
```

Again we define a subclass of `Option`:

```
class MyOption (Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

Features of note:

- `extend` both expects a value on the command-line and stores that value somewhere, so it goes in both `STORE_ACTIONS` and `TYPED_ACTIONS`

- to ensure that `optparse` assigns the default type of `string` to `extend` actions, we put the `extend` action in `ALWAYS_TYPED_ACTIONS` as well

- `MyOption.take_action()` implements just this one new action, and passes control back to `Option.take_action()` for the standard `optparse` actions

- `values` is an instance of the `optparse_parser.Values` class, which provides the very useful `ensure_value()` method. `ensure_value()` is
essentially `getattr()` with a safety valve; it is called as

```
values.ensure_value(attr, value)
```

If the `attr` attribute of `values` doesn't exist or is `None`, then
`ensure_value()` first sets it to `value`, and then returns `value`. This is very
handy for actions like `extend`, `append`, and `count`, all of which
accumulate data in a variable and expect that variable to be of a certain type
(a list for the first two, an integer for the latter). Using `ensure_value()`
means that scripts using your action don't have to worry about setting a
default value for the option destinations in question; they can just leave the
default as `None` and `ensure_value()` will take care of getting it right
when it's needed.
14.5.5.1 StreamHandler

The StreamHandler class, located in the core logging package, sends logging output to streams such as sys.stdout, sys.stderr or any file-like object (or, more precisely, any object which supports write() and flush() methods).

```python
class StreamHandler([strm])
    Returns a new instance of the StreamHandler class. If strm is specified, the instance will use it for logging output; otherwise, sys.stderr will be used.

emit(record)
    If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using traceback.print_exception() and appended to the stream.

flush()
    Flushes the stream by calling its flush() method. Note that the close() method is inherited from Handler and so does nothing, so an explicit flush() call may be needed at times.
```
14.5.5.2 FileHandler

The FileHandler class, located in the core logging package, sends logging output to a disk file. It inherits the output functionality from StreamHandler.

```python
class FileHandler(filename[, mode])
```

Returns a new instance of the FileHandler class. The specified file is opened and used as the stream for logging. If mode is not specified, 'a' is used. By default, the file grows indefinitely.

```python
close()
```

Closes the file.

```python
emit(record)
```

Outputs the record to the file.
14.5.5.3 RotatingFileHandler

The RotatingFileHandler class, located in the logging.handlers module, supports rotation of disk log files.

```python
class RotatingFileHandler( filename[, mode[, maxBytes[, backupCount]]])
```

Returns a new instance of the RotatingFileHandler class. The specified file is opened and used as the stream for logging. If mode is not specified, 'a' is used. By default, the file grows indefinitely.

You can use the maxBytes and backupCount values to allow the file to rollover at a predetermined size. When the size is about to be exceeded, the file is closed and a new file is silently opened for output. Rollover occurs whenever the current log file is nearly maxBytes in length; if maxBytes is zero, rollover never occurs. If backupCount is non-zero, the system will save old log files by appending the extensions ".1", ".2" etc., to the filename. For example, with a backupCount of 5 and a base file name of app.log, you would get app.log, app.log.1, app.log.2, up to app.log.5. The file being written to is always app.log. When this file is filled, it is closed and renamed to app.log.1, and if files app.log.1, app.log.2, etc. exist, then they are renamed to app.log.2, app.log.3 etc. respectively.

doRollover()

Does a rollover, as described above.

emit(record)

Outputs the record to the file, catering for rollover as described previously.
14.5.5.4 TimedRotatingFileHandler

The `TimedRotatingFileHandler` class, located in the `logging.handlers` module, supports rotation of disk log files at certain timed intervals.

```python
class TimedRotatingFileHandler(filename [,when [,interval [,backupCount]]])
```

Returns a new instance of the `TimedRotatingFileHandler` class. The specified file is opened and used as the stream for logging. On rotating it also sets the filename suffix. Rotating happens based on the product of `when` and `interval`.

You can use the `when` to specify the type of `interval`. The list of possible values is, note that they are not case sensitive:

<table>
<thead>
<tr>
<th>Value</th>
<th>Type of interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Seconds</td>
</tr>
<tr>
<td>M</td>
<td>Minutes</td>
</tr>
<tr>
<td>H</td>
<td>Hours</td>
</tr>
<tr>
<td>D</td>
<td>Days</td>
</tr>
<tr>
<td>W</td>
<td>Week day (0=Monday)</td>
</tr>
<tr>
<td>midnight</td>
<td>Roll over at midnight</td>
</tr>
</tbody>
</table>

If `backupCount` is non-zero, the system will save old log files by appending extensions to the filename. The extensions are date-and-time based, using the strftime format `%Y-%m-%d_%H-%M-%S` or a leading portion thereof, depending on the rollover interval. At most `backupCount` files will be kept, and if more would be created when rollover occurs, the oldest one is deleted.

```python
doRollover()
```

Does a rollover, as described above.
**emit** (*record*)

Outputs the record to the file, catering for rollover as described above.
14.5.5.5 SocketHandler

The SocketHandler class, located in the logging.handlers module, sends logging output to a network socket. The base class uses a TCP socket.

```python
class SocketHandler(host, port)
    Returns a new instance of the SocketHandler class intended to communicate with a remote machine whose address is given by host and port.

close()
    Closes the socket.

handleError()

emit()
    Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. If the connection was previously lost, re-establishes the connection. To unpickle the record at the receiving end into a LogRecord, use the makeLogRecord() function.

handleError()
    Handles an error which has occurred during emit(). The most likely cause is a lost connection. Closes the socket so that we can retry on the next event.

makeSocket()
    This is a factory method which allows subclasses to define the precise type of socket they want. The default implementation creates a TCP socket (socket.SOCK_STREAM).

makePickle(record)
```
Pickles the record's attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket.

**send**(*packet*)

Send a pickled string *packet* to the socket. This function allows for partial sends which can happen when the network is busy.
14.5.5.6 DatagramHandler

The DatagramHandler class, located in the logging.handlers module, inherits from SocketHandler to support sending logging messages over UDP sockets.

class DatagramHandler( host, port)

    Returns a new instance of the DatagramHandler class intended to communicate with a remote machine whose address is given by host and port.

emit()

    Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. To unpickle the record at the receiving end into a LogRecord, use the makeLogRecord() function.

makeSocket()

    The factory method of SocketHandler is here overridden to create a UDP socket (socket.SOCK_DGRAM).

send(s)

    Send a pickled string to a socket.
14.5.5.7 SysLogHandler

The SysLogHandler class, located in the logging.handlers module, supports sending logging messages to a remote or local UNIX syslog.

```python
class SysLogHandler([address[, facility]])
    Returns a new instance of the SysLogHandler class intended to communicate with a remote UNIX machine whose address is given by address in the form of a (host, port) tuple. If address is not specified, ('localhost', 514) is used. The address is used to open a UDP socket. If facility is not specified, LOG_USER is used.

close()
    Closes the socket to the remote host.

emit(record)
    The record is formatted, and then sent to the syslog server. If exception information is present, it is not sent to the server.

encodePriority(facility, priority)
    Encodes the facility and priority into an integer. You can pass in strings or integers - if strings are passed, internal mapping dictionaries are used to convert them to integers.
```
14.5.5.8 NTEventLogHandler

The NTEventLogHandler class, located in the logging.handlers module, supports sending logging messages to a local Windows NT, Windows 2000 or Windows XP event log. Before you can use it, you need Mark Hammond's Win32 extensions for Python installed.

class NTEventLogHandler( appname[, dllname[, logtype]])

Returns a new instance of the NTEventLogHandler class. The appname is used to define the application name as it appears in the event log. An appropriate registry entry is created using this name. The dllname should give the fully qualified pathname of a .dll or .exe which contains message definitions to hold in the log (if not specified, 'win32service.pyd' is used - this is installed with the Win32 extensions and contains some basic placeholder message definitions. Note that use of these placeholders will make your event logs big, as the entire message source is held in the log. If you want slimmer logs, you have to pass in the name of your own .dll or .exe which contains the message definitions you want to use in the event log). The logtype is one of 'Application', 'System' or 'Security', and defaults to 'Application'.

close()

At this point, you can remove the application name from the registry as a source of event log entries. However, if you do this, you will not be able to see the events as you intended in the Event Log Viewer - it needs to be able to access the registry to get the .dll name. The current version does not do this (in fact it doesn't do anything).

emit( record)

Determines the message ID, event category and event type, and then logs the message in the NT event log.

getEventCategory( record)

Returns the event category for the record. Override this if you want to
specify your own categories. This version returns 0.

**getEventType**( record )

Returns the event type for the record. Override this if you want to specify your own types. This version does a mapping using the handler's typemap attribute, which is set up in `__init__()` to a dictionary which contains mappings for DEBUG, INFO, WARNING, ERROR and CRITICAL. If you are using your own levels, you will either need to override this method or place a suitable dictionary in the handler's typemap attribute.

**getMessageID**( record )

Returns the message ID for the record. If you are using your own messages, you could do this by having the *msg* passed to the logger being an ID rather than a format string. Then, in here, you could use a dictionary lookup to get the message ID. This version returns 1, which is the base message ID in win32service.pyd.
The **SMTPHandler** class, located in the `logging.handlers` module, supports sending logging messages to an email address via SMTP.

```python
class SMTPHandler( mailhost, fromaddr, toaddrs, subject )
```

Returns a new instance of the `SMTPHandler` class. The instance is initialized with the from and to addresses and subject line of the email. The `toaddrs` should be a list of strings. To specify a non-standard SMTP port, use the (host, port) tuple format for the `mailhost` argument. If you use a string, the standard SMTP port is used.

```python
emit( record )
```

Formats the record and sends it to the specified addressees.

```python
getSubject( record )
```

If you want to specify a subject line which is record-dependent, override this method.
14.5.5.10 MemoryHandler

The MemoryHandler class, located in the logging.handlers module, supports buffering of logging records in memory, periodically flushing them to a target handler. Flushing occurs whenever the buffer is full, or when an event of a certain severity or greater is seen.

MemoryHandler is a subclass of the more general BufferingHandler, which is an abstract class. This buffers logging records in memory. Whenever each record is added to the buffer, a check is made by calling shouldFlush() to see if the buffer should be flushed. If it should, then flush() is expected to do the needful.

class BufferingHandler(capacity)
  Initializes the handler with a buffer of the specified capacity.

class MemoryHandler(capacity[, flushLevel [, target]])
  Returns a new instance of the MemoryHandler class. The instance is initialized with a buffer size of capacity. If flushLevel is not specified, ERROR is used. If no target is specified, the target will need to be set using setTarget() before this handler does anything useful.
close()
    Calls flush(), sets the target to None and clears the buffer.

flush()
    For a MemoryHandler, flushing means just sending the buffered records
    to the target, if there is one. Override if you want different behavior.

setTarget( target)
    Sets the target handler for this handler.

shouldFlush( record)
    Checks for buffer full or a record at the flushLevel or higher.
The HTTPHandler class, located in the logging.handlers module, supports sending logging messages to a Web server, using either "GET" or "POST" semantics.

```
class HTTPHandler( host, url[, method])

    Returns a new instance of the HTTPHandler class. The instance is initialized with a host address, url and HTTP method. The host can be of the form host:port, should you need to use a specific port number. If no method is specified, "GET" is used.
```

```
emit( record)

    Sends the record to the Web server as an URL-encoded dictionary.
```
14.5.10.1 Configuration functions

The following functions configure the logging module. They are located in the `logging.config` module. Their use is optional -- you can configure the logging module using these functions or by making calls to the main API (defined in `logging` itself) and defining handlers which are declared either in `logging` or `logging.handlers`.

`fileConfig(fname[, defaults])`

Reads the logging configuration from a ConfigParser-format file named `fname`. This function can be called several times from an application, allowing an end user the ability to select from various pre-canned configurations (if the developer provides a mechanism to present the choices and load the chosen configuration). Defaults to be passed to ConfigParser can be specified in the `defaults` argument.

`listen([port])`

Starts up a socket server on the specified port, and listens for new configurations. If no port is specified, the module's default `DEFAULT_LOGGING_CONFIG_PORT` is used. Logging configurations will be sent as a file suitable for processing by `fileConfig()`. Returns a `Thread` instance on which you can call `start()` to start the server, and which you can `join()` when appropriate. To stop the server, call `stopListening()`. To send a configuration to the socket, read in the configuration file and send it to the socket as a string of bytes preceded by a four-byte length packed in binary using `struct.pack('>L', n)`.

`stopListening()`

Stops the listening server which was created with a call to `listen()`. This is typically called before calling `join()` on the return value from `listen()`.
Release 2.5, documentation updated on 19th September, 2006. 
See About this document... for information on suggesting changes.
14.5.10.2 Configuration file format

The configuration file format understood by `fileConfig()` is based on ConfigParser functionality. The file must contain sections called `[loggers]`, `[handlers]` and `[formatters]` which identify by name the entities of each type which are defined in the file. For each such entity, there is a separate section which identified how that entity is configured. Thus, for a logger named `log01` in the `[loggers]` section, the relevant configuration details are held in a section `[logger_log01]`. Similarly, a handler called `hand01` in the `[handlers]` section will have its configuration held in a section called `[handler_hand01]`, while a formatter called `form01` in the `[formatters]` section will have its configuration specified in a section called `[formatter_form01]`. The root logger configuration must be specified in a section called `[logger_root]`.

Examples of these sections in the file are given below.

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

The root logger must specify a level and a list of handlers. An example of a root logger section is given below.

```
[logger_root]
level=NOTSET
handlers=hand01
```

The `level` entry can be one of `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL` or `NOTSET`. For the root logger only, `NOTSET` means that all messages will be logged. Level values are evaluated in the context of the `logging` package's namespace.
The **handlers** entry is a comma-separated list of handler names, which must appear in the [handlers] section. These names must appear in the [handlers] section and have corresponding sections in the configuration file.

For loggers other than the root logger, some additional information is required. This is illustrated by the following example.

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

The **level** and **handlers** entries are interpreted as for the root logger, except that if a non-root logger's level is specified as **NOTSET**, the system consults loggers higher up the hierarchy to determine the effective level of the logger. The **propagate** entry is set to 1 to indicate that messages must propagate to handlers higher up the logger hierarchy from this logger, or 0 to indicate that messages are **not** propagated to handlers up the hierarchy. The **qualname** entry is the hierarchical channel name of the logger, that is to say the name used by the application to get the logger.

Sections which specify handler configuration are exemplified by the following.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

The **class** entry indicates the handler's class (as determined by eval() in the logging package's namespace). The **level** is interpreted as for loggers, and **NOTSET** is taken to mean "log everything".

The **formatter** entry indicates the key name of the formatter for this handler. If blank, a default formatter (**logging._defaultFormatter**) is used. If a name is specified, it must appear in the [formatters] section and have a corresponding section in the configuration file.

The **args** entry, when eval()uated in the context of the logging package's namespace, is the list of arguments to the constructor for the handler class. Refer to the constructors for the relevant handlers, or to the examples below, to see
how typical entries are constructed.

[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(("localhost", handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger S

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
Sections which specify formatter configuration are typified by the following.

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

The `format` entry is the overall format string, and the `datefmt` entry is the `strftime()`-compatible date/time format string. If empty, the package substitutes ISO8601 format date/times, which is almost equivalent to specifying the date format string "The ISO8601 format also specifies milliseconds, which are appended to the result of using the above format string, with a comma separator. An example time in ISO8601 format is 2003-01-23 00:29:50,411.

The `class` entry is optional. It indicates the name of the formatter's class (as a dotted module and class name.) This option is useful for instantiating a `Formatter` subclass. Subclasses of `Formatter` can present exception traces in an expanded or condensed format.
14.12.3.1 Win95/98 specific

`popen(cmd, mode='r', bufsize=None)`

Portable `popen()` interface. Find a working `popen` implementation preferring `win32pipe.popen()`. On Windows NT, `win32pipe.popen()` should work; on Windows 9x it hangs due to bugs in the MS C library.
14.14.1.1 Loading dynamic link libraries

c\texttt{types} exports the \texttt{dll}, and on Windows also \texttt{windll} and \texttt{oledll} objects to load dynamic link libraries.

You load libraries by accessing them as attributes of these objects. \texttt{dll} loads libraries which export functions using the standard \texttt{cdecl} calling convention, while \texttt{windll} libraries call functions using the \texttt{stdcall} calling convention. \texttt{oledll} also uses the \texttt{stdcall} calling convention, and assumes the functions return a Windows \texttt{HRESULT} error code. The error code is used to automatically raise \texttt{WindowsError} Python exceptions when the function call fails.

Here are some examples for Windows, note that \texttt{msvcrt} is the MS standard C library containing most standard C functions, and uses the cdecl calling convention:

\begin{verbatim}
>>> from ctypes import *
>>> print windll.kernel32 # doctest: +WINDOWS
<WinDLL 'kernel32', handle ... at ...>
>>> print cdll.msvcrt # doctest: +WINDOWS
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt # doctest: +WINDOWS
>>> Windows\ appendsthe usual '.dll' file suffix automatically.

On Linux, it is required to specify the filename \texttt{including} the extension to load a library, so attribute access does not work. Either the \texttt{LoadLibrary} method of the dll loaders should be used, or you should load the library by creating an instance of CDLL by calling the constructor:

\begin{verbatim}
>>> cdll.LoadLibrary("libc.so.6") # doctest: +LINUX
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")  # doctest: +LINUX
>>> libc # doctest: +LINUX
<CDLL 'libc.so.6', handle ... at ...>
>>> \end{verbatim}
Accessing functions from

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.14.1.2 Accessing functions from loaded dlls

Functions are accessed as attributes of dll objects:

```python
>>> from ctypes import *
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print windll.kernel32.GetModuleHandleA # doctest: +WINDOWS
<_FuncPtr object at 0x...>
>>> print windll.kernel32.MyOwnFunction # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
```n
Note that win32 system dlls like kernel32 and user32 often export ANSI as well as UNICODE versions of a function. The UNICODE version is exported with an W appended to the name, while the ANSI version is exported with an A appended to the name. The win32 GetModuleHandle function, which returns a module handle for a given module name, has the following C prototype, and a macro is used to expose one of them as GetModuleHandle depending on whether UNICODE is defined or not:

```c
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

windll does not try to select one of them by magic, you must access the version you need by specifying GetModuleHandleA or GetModuleHandleW explicitly, and then call it with normal strings or unicode strings respectively.

Sometimes, dlls export functions with names which aren't valid Python identifiers, like "??2@YAPAXI@Z". In this case you have to use getattr to retrieve the function:

```python
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z") # doctest: +WINDOWS
<_FuncPtr object at 0x...>
```
On Windows, some dlls export functions not by name but by ordinal. These functions can be accessed by indexing the dll object with the ordinal number:

```python
>>> cdll.kernel32[1] # doctest: +WINDOWS
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0] # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in 
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>```

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.14.1.3 Calling functions

You can call these functions like any other Python callable. This example uses the `time()` function, which returns system time in seconds since the UNIX epoch, and the `GetModuleHandleA()` function, which returns a win32 module handle.

This example calls both functions with a NULL pointer (None should be used as the NULL pointer):

```python
>>> print libc.time(None) # doctest: +SKIP
1150640792
>>> print hex(windll.kernel32.GetModuleHandleA(None)) # doctest: +WIN
0x1d000000
>>>
```

curses tries to protect you from calling functions with the wrong number of arguments or the wrong calling convention. Unfortunately this only works on Windows. It does this by examining the stack after the function returns, so although an error is raised the function has been called:

```python
>>> windll.kernel32.GetModuleHandleA() # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Procedure probably called with not enough arguments (4 b
>>> windll.kernel32.GetModuleHandleA(0, 0) # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Procedure probably called with too many arguments (4 byt
>>> 
```

The same exception is raised when you call an stdcall function with the cdecl calling convention, or vice versa:

```python
>>> cdll.kernel32.GetModuleHandleA(0) # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Procedure probably called with not enough arguments (4 b
>>> 
>>> windll.msvcrt.printf("spam") # doctest: +WINDOWS
```
To find out the correct calling convention you have to look into the C header file or the documentation for the function you want to call.

On Windows, ctypes uses win32 structured exception handling to prevent crashes from general protection faults when functions are called with invalid argument values:

```python
>>> windll.kernel32.GetModuleHandleA(32)  # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
WindowsError: exception: access violation reading 0x00000020
```

There are, however, enough ways to crash Python with ctypes, so you should be careful anyway.

None, integers, longs, byte strings and unicode strings are the only native Python objects that can directly be used as parameters in these function calls. None is passed as a C NULL pointer, byte strings and unicode strings are passed as pointer to the memory block that contains their data (char * or wchar_t *). Python integers and Python longs are passed as the platforms default C int type, their value is masked to fit into the C type.

Before we move on calling functions with other parameter types, we have to learn more about ctypes data types.
### 14.14.1.4 Fundamental data types

`ctypes` defines a number of primitive C compatible data types:

<table>
<thead>
<tr>
<th>ctypes type</th>
<th>C type</th>
<th>Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_char</td>
<td>char</td>
<td>1-character string</td>
</tr>
<tr>
<td>c_wchar</td>
<td>wchar_t</td>
<td>1-character unicode string</td>
</tr>
<tr>
<td>c_byte</td>
<td>char</td>
<td>int/long</td>
</tr>
<tr>
<td>c_ubyte</td>
<td>unsigned char</td>
<td>int/long</td>
</tr>
<tr>
<td>c_short</td>
<td>short</td>
<td>int/long</td>
</tr>
<tr>
<td>c_ushort</td>
<td>unsigned short</td>
<td>int/long</td>
</tr>
<tr>
<td>c_int</td>
<td>int</td>
<td>int/long</td>
</tr>
<tr>
<td>c_uint</td>
<td>unsigned int</td>
<td>int/long</td>
</tr>
<tr>
<td>c_long</td>
<td>long</td>
<td>int/long</td>
</tr>
<tr>
<td>c_ulong</td>
<td>unsigned long</td>
<td>int/long</td>
</tr>
<tr>
<td>c_longlong</td>
<td>__int64 or long long</td>
<td>int/long</td>
</tr>
<tr>
<td>c_ulonglong</td>
<td>unsigned __int64 or unsigned long long</td>
<td>int/long</td>
</tr>
<tr>
<td>c_float</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>c_double</td>
<td>double</td>
<td>float</td>
</tr>
<tr>
<td>c_char_p</td>
<td>char * (NUL terminated)</td>
<td>string or None</td>
</tr>
<tr>
<td>c_wchar_p</td>
<td>wchar_t * (NUL terminated)</td>
<td>unicode or None</td>
</tr>
<tr>
<td>c_void_p</td>
<td>void *</td>
<td>int/long or None</td>
</tr>
</tbody>
</table>

All these types can be created by calling them with an optional initializer of the correct type and value:

```python
globals() c_int() c_long(0)
globals() c_char_p("Hello, World") c_char_p('Hello, World')
globals() c_ushort(-3) c_ushort(65533)
globals()```
Since these types are mutable, their value can also be changed afterwards:

```python
>>> i = c_int(42)
>>> print i
c_long(42)
>>> print i.value
42
>>> i.value = -99
>>> print i.value
-99
```

Assigning a new value to instances of the pointer types `c_char_p`, `c_wchar_p`, and `c_void_p` changes the memory location they point to, *not the contents* of the memory block (of course not, because Python strings are immutable):

```python
>>> s = "Hello, World"
>>> c_s = c_char_p(s)
>>> print c_s
'Hello, World'
>>> c_s.value = "Hi, there"
>>> print c_s
'Hi, there'
>>> print s  # first string is unchanged
Hello, World
```

You should be careful, however, not to pass them to functions expecting pointers to mutable memory. If you need mutable memory blocks, ctypes has a `create_string_buffer` function which creates these in various ways. The current memory block contents can be accessed (or changed) with the `raw` property, if you want to access it as NUL terminated string, use the `string` property:

```python
>>> from ctypes import *
>>> p = create_string_buffer(3)  # create a 3 byte buffer, initialized to NUL bytes
>>> print sizeof(p), repr(p.raw)
3 '\x00\x00\x00'
>>> p = create_string_buffer("Hello")  # create a buffer containing a NUL terminated string
>>> print sizeof(p), repr(p.raw)
6 'Hello\x00'
>>> print repr(p.value)
'Hello'
>>> p = create_string_buffer("Hello", 10)  # create a 10 byte buffer
```
>>> print sizeof(p), repr(p.raw)
10 'Hello\x00\x00\x00\x00\x00'
>>> p.value = "Hi"
>>> print sizeof(p), repr(p.raw)
10 'Hi\x00lo\x00\x00\x00\x00\x00'

The create_string_buffer function replaces the c_buffer function (which is still available as an alias), as well as the c_string function from earlier ctypes releases. To create a mutable memory block containing unicode characters of the C type wchar_t use the create_unicode_buffer function.
14.14.1.5 Calling functions, continued

Note that `printf` prints to the real standard output channel, not to `sys.stdout`, so these examples will only work at the console prompt, not from within `IDLE` or `PythonWin`:

```python
>>> printf = libc.printf
>>> printf("Hello, %s\n", "World!")
Hello, World!
14
>>> printf("Hello, %S", u"World!")
Hello, World!
13
>>> printf("%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf("%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: Don't know how to c
>>> 
```

As has been mentioned before, all Python types except integers, strings, and unicode strings have to be wrapped in their corresponding `ctypes` type, so that they can be converted to the required C data type:

```python
>>> printf("An int %d, a double %f\n", 1234, c_double(3.14))
Integer 1234, double 3.1400001049
31
>>> 
```
14.14.1.6 Calling functions with your own custom data types

You can also customize ctypes argument conversion to allow instances of your own classes be used as function arguments. ctypes looks for an _as_parameter_ attribute and uses this as the function argument. Of course, it must be one of integer, string, or unicode:

```python
>>> class Bottles(object):
...     def __init__(self, number):
...         self._as_parameter_ = number
... >>> bottles = Bottles(42)
>>> printf("%d bottles of beer\n", bottles)
42 bottles of beer
19
```  

If you don't want to store the instance's data in the _as_parameter_ instance variable, you could define a property which makes the data available.
14.14.1.7 Specifying the required argument types (function prototypes)

It is possible to specify the required argument types of functions exported from DLLs by setting the `argtypes` attribute.

`argtypes` must be a sequence of C data types (the `printf` function is probably not a good example here, because it takes a variable number and different types of parameters depending on the format string, on the other hand this is quite handy to experiment with this feature):

```python
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf("String '%s', Int %d, Double %f\n", "Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
```

Specifying a format protects against incompatible argument types (just as a prototype for a C function), and tries to convert the arguments to valid types:

```python
>>> printf("%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf("%s %d %f", "X", 2, 3)
X 2 3.00000012
12
```

If you have defined your own classes which you pass to function calls, you have to implement a `from_param` class method for them to be able to use them in the `argtypes` sequence. The `from_param` class method receives the Python object passed to the function call, it should do a typecheck or whatever is needed to make sure this object is acceptable, and then return the object itself, it's `_as_parameter_` attribute, or whatever you want to pass as the C function argument in this case. Again, the result should be an integer, string, unicode, a `ctypes` instance, or something having the `_as_parameter_` attribute.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.14.1.8 Return types

By default functions are assumed to return the C int type. Other return types can be specified by setting the restype attribute of the function object.

Here is a more advanced example, it uses the strchr function, which expects a string pointer and a char, and returns a pointer to a string:

```python
>>> strchr = libc.strchr
>>> strchr("abcdef", ord("d")) # doctest: +SKIP
8059983
>>> strchr.restype = c_char_p # c_char_p is a pointer to a string
>>> strchr("abcdef", ord("d"))
'def'
>>> print strchr("abcdef", ord("x"))
None
```

If you want to avoid the `ord("x")` calls above, you can set the argtypes attribute, and the second argument will be converted from a single character Python string into a C char:

```python
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr("abcdef", "d")
'def'
>>> strchr("abcdef", "def")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print strchr("abcdef", "x")
None
>>> strchr("abcdef", "d")
'def'
```

You can also use a callable Python object (a function or a class for example) as the restype attribute, if the foreign function returns an integer. The callable will be called with the integer the C function returns, and the result of this call will be used as the result of your function call. This is useful to check for error return values and automatically raise an exception:
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA # doctest: +W
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
... >>>
>>> GetModuleHandle().restype = ValidHandle # doctest: +W
>>> GetModuleHandle(None) # doctest: +W
486539264
>>> GetModuleHandle("something silly") # doctest: +W
WindowsError: [Errno 126] The specified module could not be found.

WinError is a function which will call Windows FormatMessage() api to get the string representation of an error code, and returns an exception. WinError takes an optional error code parameter, if no one is used, it calls GetLastError() to retrieve it.

Please note that a much more powerful error checking mechanism is available through the errcheck attribute; see the reference manual for details.
14.14.1.9 Passing pointers (or: passing parameters by reference)

Sometimes a C api function expects a pointer to a data type as parameter, probably to write into the corresponding location, or if the data is too large to be passed by value. This is also known as passing parameters by reference.

ctypes exports the byref function which is used to pass parameters by reference. The same effect can be achieved with the pointer function, although pointer does a lot more work since it constructs a real pointer object, so it is faster to use byref if you don't need the pointer object in Python itself:

```python
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer('\000' * 32)
>>> print i.value, f.value, repr(s.value)
0 0.0 ''
>>> libc.sscanf("1 3.14 Hello", "%d %f %s", ...          byref(i), byref(f), s)
3
>>> print i.value, f.value, repr(s.value)
1 3.14000001049 'Hello'
```
Structures and unions must derive from the `Structure` and `Union` base classes which are defined in the `ctypes` module. Each subclass must define a `_fields_` attribute. `_fields_` must be a list of 2-tuples, containing a field name and a field type.

The field type must be a `ctypes` type like `c_int`, or any other derived `ctypes` type: structure, union, array, pointer.

Here is a simple example of a `POINT` structure, which contains two integers named `x` and `y`, and also shows how to initialize a structure in the constructor:

```python
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ["x", c_int],
...     ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print point.x, point.y
10 20
>>> point = POINT(y=5)
>>> print point.x, point.y
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: too many initializers
>>>
```

You can, however, build much more complicated structures. Structures can itself contain other structures by using a structure as a field type.

Here is a `RECT` structure which contains two `POINTs` named `upperleft` and `lowerright`:

```python
>>> class RECT(Structure):
...     _fields_ = ["upperleft", POINT],
...     ("lowerright", POINT)]
...
>>> rc = RECT(point)
```
Nested structures can also be initialized in the constructor in several ways:

```python
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

Fields descriptors can be retrieved from the class, they are useful for debugging because they can provide useful information:

```python
>>> print POINT.x
<Field type=c_long, ofs=0, size=4>
```
14.14.1.11 Structure/union alignment and byte order

By default, Structure and Union fields are aligned in the same way the C compiler does it. It is possible to override this behaviour by specifying a _pack_ class attribute in the subclass definition. This must be set to a positive integer and specifies the maximum alignment for the fields. This is what #pragma pack(n) also does in MSVC.

ctypes uses the native byte order for Structures and Unions. To build structures with non-native byte order, you can use one of the BigEndianStructure, LittleEndianStructure, BigEndianUnion, and LittleEndianUnion base classes. These classes cannot contain pointer fields.
14.14.1.12 Bit fields in structures and unions

It is possible to create structures and unions containing bit fields. Bit fields are only possible for integer fields, the bit width is specified as the third item in the _fields_ tuples:

```python
>>> class Int(Structure):
...     _fields_ = ["first_16", c_int, 16],
...     ("second_16", c_int, 16]
... >>> print Int.first_16
<Field type=c_long, ofs=0:0, bits=16>
>>> print Int.second_16
<Field type=c_long, ofs=0:16, bits=16>
```
14.14.1.13 Arrays

Arrays are sequences, containing a fixed number of instances of the same type.

The recommended way to create array types is by multiplying a data type with a positive integer:

```
TenPointsArrayType = POINT * 10
```

Here is an example of an somewhat artificial data type, a structure containing 4 POINTs among other stuff:

```
>>> from ctypes import *
>>> class POINT(Structure):
...    _fields_ = ("x", c_int), ("y", c_int)
...    ...
>>> class MyStruct(Structure):
...    _fields_ = ["a", c_int],
...                ("b", c_float),
...                ("point_array", POINT * 4)
>>> print len(MyStruct().point_array)
4
```

Instances are created in the usual way, by calling the class:

```
arr = TenPointsArrayType()
for pt in arr:
    print pt.x, pt.y
```

The above code print a series of 0 0 lines, because the array contents is initialized to zeros.

Initializers of the correct type can also be specified:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> for i in ii: print i,
<__main__.c_long_Array_10 object at 0x...>
```

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.14.14 Pointers

Pointer instances are created by calling the `pointer` function on a `ctypes` type:

```python
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)

>>> pi.contents
<5000000000
```

Pointer instances have a `contents` attribute which returns the object to which the pointer points, the `i` object above:

```python
>>> pi.contents
<5000000000
```

Note that `ctypes` does not have OOR (original object return), it constructs a new, equivalent object each time you retrieve an attribute:

```python
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
```

Assigning another `c_int` instance to the pointer's `contents` attribute would cause the pointer to point to the memory location where this is stored:

```python
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
<5000000000
```

Pointer instances can also be indexed with integers:

```python
>>> pi[0]
99
```
Assigning to an integer index changes the pointed to value:

```python
>>> print i
c_long(99)
>>> pi[0] = 22
>>> print i
c_long(22)
```

It is also possible to use indexes different from 0, but you must know what you're doing, just as in C: You can access or change arbitrary memory locations. Generally you only use this feature if you receive a pointer from a C function, and you know that the pointer actually points to an array instead of a single item.

Behind the scenes, the `pointer` function does more than simply create pointer instances, it has to create pointer types first. This is done with the `POINTER` function, which accepts any `ctypes` type, and returns a new type:

```python
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
```

Calling the pointer type without an argument creates a NULL pointer. NULL pointers have a `False` boolean value:

```python
>>> null_ptr = POINTER(c_int)()
>>> print bool(null_ptr)
False
```

`ctypes` checks for NULL when dereferencing pointers (but dereferencing non-NULL pointers would crash Python):

```python
>>> null_ptr[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: NULL pointer access
```
>>> null_ptr[0] = 1234
Traceback (most recent call last):
...
ValueError: NULL pointer access
>>>

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.14.1.15 Type conversions

Usually, ctypes does strict type checking. This means, if you have `POINTER(c_int)` in the `argtypes` list of a function or as the type of a member field in a structure definition, only instances of exactly the same type are accepted. There are some exceptions to this rule, where ctypes accepts other objects. For example, you can pass compatible array instances instead of pointer types. So, for `POINTER(c_int)`, ctypes accepts an array of `c_int`:

```python
>>> class Bar(Structure):
...     _fields_ = ["count", c_int], ["values", POINTER(c_int)]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print bar.values[i]
... 1
2
3
```

To set a `POINTER` type field to NULL, you can assign `None`:

```python
>>> bar.values = None
```

XXX list other conversions...

Sometimes you have instances of incompatible types. In C, you can cast one type into another type. ctypes provides a `cast` function which can be used in the same way. The `Bar` structure defined above accepts `POINTER(c_int)` pointers or `c_int` arrays for its `values` field, but not instances of other types:

```python
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: incompatible types, c_byte_Array_4 instance instead of LP
>>>```
For these cases, the `cast` function is handy.

The `cast` function can be used to cast a ctypes instance into a pointer to a different ctypes data type. `cast` takes two parameters, a ctypes object that is or can be converted to a pointer of some kind, and a ctypes pointer type. It returns an instance of the second argument, which references the same memory block as the first argument:

```python
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
```

So, `cast` can be used to assign to the `values` field of `Bar` the structure:

```python
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print bar.values[0]
0
```
14.14.1.16 Incomplete Types

Incomplete Types are structures, unions or arrays whose members are not yet specified. In C, they are specified by forward declarations, which are defined later:

```c
struct cell; /* forward declaration */

struct {
    char *name;
    struct cell *next;
} cell;
```

The straightforward translation into ctypes code would be this, but it does not work:

```python
>>> class cell(Structure):
...     _fields_ = ["name", c_char_p],
...             ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
```

because the new class **cell** is not available in the class statement itself. In ctypes, we can define the **cell** class and set the **_fields_** attribute later, after the class statement:

```python
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...     _fields_ = ["name", c_char_p],
...             ("next", POINTER(cell))]
```

Lets try it. We create two instances of **cell**, and let them point to each other, and finally follow the pointer chain a few times:

```python
>>> c1 = cell()
```
>>> c1.name = "foo"
>>> c2 = cell()
>>> c2.name = "bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...    print p.name,
...    p = p.next[0]
...
foo bar foo bar foo bar foo bar

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.14.17 Callback functions

callable allows to create C callable function pointers from Python callables. These are sometimes called callback functions.

First, you must create a class for the callback function, the class knows the calling convention, the return type, and the number and types of arguments this function will receive.

The CFUNCTYPE factory function creates types for callback functions using the normal cdecl calling convention, and, on Windows, the WINFUNCTYPE factory function creates types for callback functions using the stdcall calling convention.

Both of these factory functions are called with the result type as first argument, and the callback functions expected argument types as the remaining arguments.

I will present an example here which uses the standard C library's qsort function, this is used to sort items with the help of a callback function. qsort will be used to sort an array of integers:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>> qsort
```

qsort must be called with a pointer to the data to sort, the number of items in the data array, the size of one item, and a pointer to the comparison function, the callback. The callback will then be called with two pointers to items, and it must return a negative integer if the first item is smaller than the second, a zero if they are equal, and a positive integer else.

So our callback function receives pointers to integers, and must return an integer. First we create the type for the callback function:

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>> CMPFUNC
```
For the first implementation of the callback function, we simply print the arguments we get, and return 0 (incremental development ;-):

```python
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a, b
...     return 0
... >>>
```

Create the C callable callback:

```python
>>> cmp_func = CMPFUNC(py_cmp_func)
>>> And we're ready to go:

```python
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func) # doctest: +WINDOWS
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long o
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long o
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long o
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long o
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long o
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long o
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long o
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long o
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long o
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long o
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long o
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long o
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long o
```  

We know how to access the contents of a pointer, so let's redefine our callback:

```python
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a[0], b[0]
...     return 0
... >>>
```

Here is what we get on Windows:

```python
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func) # doctest: +WINDOWS
py_cmp_func 7 1
py_cmp_func 33 1
py_cmp_func 99 1
py_cmp_func 5 1
py_cmp_func 7 5
py_cmp_func 33 5
py_cmp_func 99 5
py_cmp_func 7 99
```
It is funny to see that on linux the sort function seems to work much more efficient, it is doing less comparisons:

```python
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func) # doctest: +LINUX
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
```

Ah, we're nearly done! The last step is to actually compare the two items and return a useful result:

```python
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a[0], b[0]
...     return a[0] - b[0]
... print
```

Final run on Windows:

```python
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func)) # doctest
py_cmp_func 33 7
py_cmp_func 99 33
py_cmp_func 5 99
py_cmp_func 1 99
py_cmp_func 33 7
py_cmp_func 1 33
py_cmp_func 5 33
py_cmp_func 5 7
py_cmp_func 1 7
py_cmp_func 5 1
```

and on Linux:

```python
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func)) # doctest
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
```
It is quite interesting to see that the Windows qsort function needs more comparisons than the linux version!

As we can easily check, our array sorted now:

```python
>>> for i in ia: print i,
... 1 5 7 33 99
>>> 
```

**Important note for callback functions:**

Make sure you keep references to CFUNCTYPE objects as long as they are used from C code. ctypes doesn't, and if you don't, they may be garbage collected, crashing your program when a callback is made.
14.14.1.18 Accessing values exported from dlls

Sometimes, a dll not only exports functions, it also exports variables. An example in the Python library itself is the `Py_OptimizeFlag`, an integer set to 0, 1, or 2, depending on the `-O` or `-OO` flag given on startup.

`ctypes` can access values like this with the `in_dll` class methods of the type. `pythonapi` is a predefined symbol giving access to the Python C api:

```python
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print opt_flag
c_long(0)
```

If the interpreter would have been started with `-O`, the sample would have printed `c_long(1)`, or `c_long(2)` if `-OO` would have been specified.

An extended example which also demonstrates the use of pointers accesses the `PyImport_FrozenModules` pointer exported by Python.

Quoting the Python docs: *This pointer is initialized to point to an array of \`struct _frozen\` records, terminated by one whose members are all NULL or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.*

So manipulating this pointer could even prove useful. To restrict the example size, we show only how this table can be read with `ctypes`:

```python
>>> from ctypes import *

>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...          ("code", POINTER(c_ubyte)),
...          ("size", c_int)]
...
```

We have defined the `struct _frozen` data type, so we can get the pointer to
the table:

```python
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")
```

Since `table` is a pointer to the array of `struct_frozen` records, we can iterate over it, but we just have to make sure that our loop terminates, because pointers have no size. Sooner or later it would probably crash with an access violation or whatever, so it's better to break out of the loop when we hit the NULL entry:

```python
>>> for item in table:
...    print item.name, item.size
...    if item.name is None:
...        break
...  __hello__ 104
  __phello__ -104
  __phello__.spam 104
None 0
```

The fact that standard Python has a frozen module and a frozen package (indicated by the negative size member) is not wellknown, it is only used for testing. Try it out with `import __hello__` for example.
14.14.1.19 Surprises

There are some edges in ctypes where you may be expect something else than what actually happens.

Consider the following example:

```python
>>> from ctypes import *
>>> class POINT(Structure):
...    _fields_ = ("x", c_int), ("y", c_int)
...>>> class RECT(Structure):
...    _fields_ = ("a", POINT), ("b", POINT)
...>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print rc.a.x, rc.a.y, rc.b.x, rc.b.y
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print rc.a.x, rc.a.y, rc.b.x, rc.b.y
3 4 3 4
>>>```

Hm. We certainly expected the last statement to print 3 4 1 2. What happened? Here are the steps of the `rc.a, rc.b = rc.b, rc.a` line above:

```python
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
```  

Note that `temp0` and `temp1` are objects still using the internal buffer of the `rc` object above. So executing `rc.a = temp0` copies the buffer contents of `temp0` into `rc`'s buffer. This, in turn, changes the contents of `temp1`. So, the last assignment `rc.b = temp1`, doesn't have the expected effect.

Keep in mind that retrieving subobjects from Structure, Unions, and Arrays doesn't copy the subobject, instead it retrieves a wrapper object accessing the
root-object's underlying buffer.

Another example that may behave different from what one would expect is this:

```python
>>> s = c_char_p()
>>> s.value = "abc def ghi"
>>> s.value
'abc def ghi'
>>> s.value is s.value
False
``` 

Why is it printing `False`? ctypes instances are objects containing a memory block plus some descriptors accessing the contents of the memory. Storing a Python object in the memory block does not store the object itself, instead the contents of the object is stored. Accessing the contents again constructs a new Python each time!
14.14.1.20 Variable-sized data types

cypes provides some support for variable-sized arrays and structures (this was added in version 0.9.9.7).

The resize function can be used to resize the memory buffer of an existing ctypes object. The function takes the object as first argument, and the requested size in bytes as the second argument. The memory block cannot be made smaller than the natural memory block specified by the objects type, a ValueError is raised if this is tried:

```python
>>> short_array = (c_short * 4)()
>>> print sizeof(short_array)
8
>>> resize(short_array, 4)
Traceback (most recent call last):
 ... 
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>> 
```

This is nice and fine, but how would one access the additional elements contained in this array? Since the type still only knows about 4 elements, we get errors accessing other elements:

```python
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
 ... 
IndexError: invalid index
>>> 
```

Another way to use variable-sized data types with ctypes is to use the dynamic nature of Python, and (re-)define the data type after the required size is already known, on a case by case basis.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.14.1.21 Bugs, ToDo and non-implemented things

Enumeration types are not implemented. You can do it easily yourself, using c_int as the base class.

long double is not implemented.
14.14.2.1 Finding shared libraries

When programming in a compiled language, shared libraries are accessed when compiling/linking a program, and when the program is run.

The purpose of the `find_library` function is to locate a library in a way similar to what the compiler does (on platforms with several versions of a shared library the most recent should be loaded), while the ctypes library loaders act like when a program is run, and call the runtime loader directly.

The `ctypes.util` module provides a function which can help to determine the library to load.

`find_library(name)`

Try to find a library and return a pathname. `name` is the library name without any prefix like `lib`, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependend.

On Linux, `find_library` tries to run external programs (/sbin/ldconfig, gcc, and objdump) to find the library file. It returns the filename of the library file. Here are some examples:

```python
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
```

On OS X, `find_library` tries several predefined naming schemes and paths to locate the library, and returns a full pathname if successful:

```python
>>> from ctypes.util import find_library
```
On Windows, `find_library` searches along the system search path, and returns the full pathname, but since there is no predefined naming scheme a call like `find_library("c")` will fail and return `None`.

If wrapping a shared library with `ctypes`, it *may* be better to determine the shared library name at development type, and hardcode that into the wrapper module instead of using `find_library` to locate the library at runtime.
14.14.2.2 Loading shared libraries

There are several ways to loaded shared libraries into the Python process. One way is to instantiate one of the following classes:

**class CDLL** *(name, mode=DEFAULT_MODE, handle=None)*

Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return int.

**class OleDLL** *(name, mode=DEFAULT_MODE, handle=None)*

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the stdcall calling convention, and are assumed to return the windows specific HRESULT code. HRESULT values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an WindowsError is automatically raised.

**class WinDLL** *(name, mode=DEFAULT_MODE, handle=None)*

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the stdcall calling convention, and are assumed to return int by default.

On Windows CE only the standard calling convention is used, for convenience the WinDLL and OleDLL use the standard calling convention on this platform.

The Python GIL is released before calling any function exported by these libraries, and reaquired afterwards.

**class PyDLL** *(name, mode=DEFAULT_MODE, handle=None)*

Instances of this class behave like CDLL instances, except that the Python GIL is not released during the function call, and after the function execution the Python error flag is checked. If the error flag is set, a Python exception
is raised.

Thus, this is only useful to call Python C api functions directly.

All these classes can be instantiated by calling them with at least one argument, the pathname of the shared library. If you have an existing handle to an already loaded shard library, it can be passed as the handle named parameter, otherwise the underlying platforms dlopen or LoadLibrary function is used to load the library into the process, and to get a handle to it.

The mode parameter can be used to specify how the library is loaded. For details, consult the dlopen(3) manpage, on Windows, mode is ignored.

RTLD_GLOBAL
Flag to use as mode parameter. On platforms where this flag is not available, it is defined as the integer zero.

RTLD_LOCAL
Flag to use as mode parameter. On platforms where this is not available, it is the same as RTLD_GLOBAL.

DEFAULT_MODE
The default mode which is used to load shared libraries. On OSX 10.3, this is RTLD_GLOBAL, otherwise it is the same as RTLD_LOCAL.

Instances of these classes have no public methods, however __getattr__ and __getitem__ have special behaviour: functions exported by the shared library can be accessed as attributes of by index. Please note that both __getattr__ and __getitem__ cache their result, so calling them repeatedly returns the same object each time.

The following public attributes are available, their name starts with an underscore to not clash with exported function names:

_handle
The system handle used to access the library.

_name
The name of the library passed in the contractor.
Shared libraries can also be loaded by using one of the prefabricated objects, which are instances of the `LibraryLoader` class, either by calling the `LoadLibrary` method, or by retrieving the library as attribute of the loader instance.

**class LibraryLoader** *(dlltype)*

Class which loads shared libraries. `dlltype` should be one of the `CDLL`, `PyDLL`, `WinDLL`, or `OleDLL` types.

`__getattr__` has special behaviour: It allows to load a shared library by accessing it as attribute of a library loader instance. The result is cached, so repeated attribute accesses return the same library each time.

**LoadLibrary** *(name)*

Load a shared library into the process and return it. This method always returns a new instance of the library.

These prefabricated library loaders are available:

**cdll**

Creates `CDLL` instances.

**windll**

Windows only: Creates `WinDLL` instances.

**oledll**

Windows only: Creates `OleDLL` instances.

**pydll**

Creates `PyDLL` instances.

For accessing the C Python api directly, a ready-to-use Python shared library object is available:

**pythonapi**

An instance of `PyDLL` that exposes Python C api functions as attributes. Note that all these functions are assumed to return C `int`, which is of course not always the truth, so you have to assign the correct `restype` attribute to use these functions.
14.14.2.3 Foreign functions

As explained in the previous section, foreign functions can be accessed as attributes of loaded shared libraries. The function objects created in this way by default accept any number of arguments, accept any ctypes data instances as arguments, and return the default result type specified by the library loader. They are instances of a private class:

```python
class _FuncPtr
    Base class for C callable foreign functions.
```

Instances of foreign functions are also C compatible data types; they represent C function pointers.

This behaviour can be customized by assigning to special attributes of the foreign function object.

**restype**

Assign a ctypes type to specify the result type of the foreign function. Use `None` for `void` a function not returning anything.

It is possible to assign a callable Python object that is not a ctypes type, in this case the function is assumed to return a `C int`, and the callable will be called with this integer, allowing to do further processing or error checking. Using this is deprecated, for more flexible postprocessing or error checking use a ctypes data type as `restype` and assign a callable to the `errcheck` attribute.

**argtypes**

Assign a tuple of ctypes types to specify the argument types that the function accepts. Functions using the `stdcall` calling convention can only be called with the same number of arguments as the length of this tuple; functions using the C calling convention accept additional, unspecified arguments as well.

When a foreign function is called, each actual argument is passed to the
from_param class method of the items in the argtypes tuple, this method allows to adapt the actual argument to an object that the foreign function accepts. For example, a c_char_p item in the argtypes tuple will convert a unicode string passed as argument into an byte string using ctypes conversion rules.

New: It is now possible to put items in argtypes which are not ctypes types, but each item must have a from_param method which returns a value usable as argument (integer, string, ctypes instance). This allows to define adapters that can adapt custom objects as function parameters.

errcheck
Assign a Python function or another callable to this attribute. The callable will be called with three or more arguments:

callable( result, func, arguments)

result is what the foreign function returns, as specified by the restype attribute.

func is the foreign function object itself, this allows to reuse the same callable object to check or postprocess the results of several functions.

arguments is a tuple containing the parameters originally passed to the function call, this allows to specialize the behaviour on the arguments used.

The object that this function returns will be returned from the foreign function call, but it can also check the result value and raise an exception if the foreign function call failed.

exception ArgumentError()
This exception is raised when a foreign function call cannot convert one of the passed arguments.

Next: 14.14.2.4 Function prototypes

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.14.2.4 Function prototypes

Foreign functions can also be created by instantiating function prototypes. Function prototypes are similar to function prototypes in C; they describe a function (return type, argument types, calling convention) without defining an implementation. The factory functions must be called with the desired result type and the argument types of the function.

**CFUNCTYPE( restype, *argtypes)**

The returned function prototype creates functions that use the standard C calling convention. The function will release the GIL during the call.

**WINFUNCTYPE( restype, *argtypes)**

Windows only: The returned function prototype creates functions that use the stdcall calling convention, except on Windows CE where WINFUNCTYPE is the same as CFUNCTYPE. The function will release the GIL during the call.

**PYFUNCTYPE( restype, *argtypes)**

The returned function prototype creates functions that use the Python calling convention. The function will *not* release the GIL during the call.

Function prototypes created by the factory functions can be instantiated in different ways, depending on the type and number of the parameters in the call.

**prototype( address)**

Returns a foreign function at the specified address.

**prototype( callable)**

Create a C callable function (a callback function) from a Python callable.

**prototype( func_spec[, paramflags])**

Returns a foreign function exported by a shared library. **func_spec** must
be a 2-tuple \((name\_or\_ordinal, \ library)\). The first item is the name of the exported function as string, or the ordinal of the exported function as small integer. The second item is the shared library instance.

**prototype**\((vtbl\_index, name[, \ paramflags[, \ iid]])\)

Returns a foreign function that will call a COM method. \(vtbl\_index\) is the index into the virtual function table, a small nonnegative integer. \(name\) is name of the COM method. \(iid\) is an optional pointer to the interface identifier which is used in extended error reporting.

COM methods use a special calling convention: They require a pointer to the COM interface as first argument, in addition to those parameters that are specified in the \texttt{argtypes} tuple.

The optional \texttt{paramflags} parameter creates foreign function wrappers with much more functionality than the features described above.

\texttt{paramflags} must be a tuple of the same length as \texttt{argtypes}.

Each item in this tuple contains further information about a parameter, it must be a tuple containing 1, 2, or 3 items.

The first item is an integer containing flags for the parameter:

1  
Specifies an input parameter to the function.

2  
Output parameter. The foreign function fills in a value.

4  
Input parameter which defaults to the integer zero.

The optional second item is the parameter name as string. If this is specified, the foreign function can be called with named parameters.

The optional third item is the default value for this parameter.

This example demonstrates how to wrap the Windows \texttt{MessageBoxA} function so that it supports default parameters and named arguments. The C declaration
from the windows header file is this:

```c
WINUSERAPI int WINAPI MessageBoxA(
    HWND hWnd,
    LPCSTR lpText,
    LPCSTR lpCaption,
    UINT uType);
```

Here is the wrapping with ctypes:

```python
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCSTR, LPCSTR, c_uint)
>>> paramflags = (1, "hwnd", 0),
                (1, "text", "Hi"),
                (1, "caption", None),
                (1, "flags", 0)
>>> MessageBox = prototype(('MessageBoxA', windll.user32), paramflags)

The MessageBox foreign function can now be called in these ways:

```python
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

A second example demonstrates output parameters. The win32 GetWindowRect function retrieves the dimensions of a specified window by copying them into RECT structure that the caller has to supply. Here is the C declaration:

```c
WINUSERAPI BOOL WINAPI GetWindowRect(
    HWND hWnd,
    LPRECT lpRect);
```

Here is the wrapping with ctypes:

```python
>>> from ctypes import POINTER, WINFUNCTYPE, windll
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(('GetWindowRect', windll.user32), paramflags)

Functions with output parameters will automatically return the output parameter value if there is a single one, or a tuple containing the output parameter values when there are more than one, so the GetWindowRect function now returns a
RECT instance, when called.

Output parameters can be combined with the errcheck protocol to do further output processing and error checking. The win32 GetWindowRect api function returns a BOOL to signal success or failure, so this function could do the error checking, and raises an exception when the api call failed:

```python
>>> def errcheck(result, func, args):
    ...    if not result:
    ...        raise WinError()
    ...    return args
>>> GetWindowRect.errcheck = errcheck
>>>```

If the errcheck function returns the argument tuple it receives unchanged, ctypes continues the normal processing it does on the output parameters. If you want to return a tuple of window coordinates instead of a RECT instance, you can retrieve the fields in the function and return them instead, the normal processing will no longer take place:

```python
>>> def errcheck(result, func, args):
    ...    if not result:
    ...        raise WinError()
    ...    rc = args[1]
    ...    return rc.left, rc.top, rc.bottom, rc.right
>>> >>>
>>> GetWindowRect.errcheck = errcheck
>>>```
14.14.2.5 Utility functions

**addressof**(obj)

Returns the address of the memory buffer as integer. obj must be an instance of a ctypes type.

**alignment**(obj_or_type)

Returns the alignment requirements of a ctypes type. obj_or_type must be a ctypes type or instance.

**byref**(obj)

Returns a light-weight pointer to obj, which must be an instance of a ctypes type. The returned object can only be used as a foreign function call parameter. It behaves similar to **pointer**(obj), but the construction is a lot faster.

**cast**(obj, type)

This function is similar to the cast operator in C. It returns a new instance of type which points to the same memory block as obj. type must be a pointer type, and obj must be an object that can be interpreted as a pointer.

**create_string_buffer**(init_or_size[, size])

This function creates a mutable character buffer. The returned object is a ctypes array of c_char.

init_or_size must be an integer which specifies the size of the array, or a string which will be used to initialize the array items.

If a string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows to specify the size of the array if the length of the string should not be used.
If the first parameter is a unicode string, it is converted into an 8-bit string according to ctypes conversion rules.

**create_unicode_buffer**(init_or_size[, size])
This function creates a mutable unicode character buffer. The returned object is a ctypes array of c_wchar.

init_or_size must be an integer which specifies the size of the array, or a unicode string which will be used to initialize the array items.

If a unicode string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows to specify the size of the array if the length of the string should not be used.

If the first parameter is a 8-bit string, it is converted into an unicode string according to ctypes conversion rules.

**DllCanUnloadNow()**
Windows only: This function is a hook which allows to implement inprocess COM servers with ctypes. It is called from the DllCanUnloadNow function that the _ctypes extension dll exports.

**DllGetClassObject()**
Windows only: This function is a hook which allows to implement inprocess COM servers with ctypes. It is called from the DllGetClassObject function that the _ctypes extension dll exports.

**FormatError**(code)
Windows only: Returns a textual description of the error code. If no error code is specified, the last error code is used by calling the Windows api function GetLastError.

**GetLastError()**
Windows only: Returns the last error code set by Windows in the calling thread.
**memmove** *(dst, src, count)*
Same as the standard C memmove library function: copies *count* bytes from *src* to *dst*. *dst* and *src* must be integers or ctypes instances that can be converted to pointers.

**memset** *(dst, c, count)*
Same as the standard C memset library function: fills the memory block at address *dst* with *count* bytes of value *c*. *dst* must be an integer specifying an address, or a ctypes instance.

**POINTER** *(type)*
This factory function creates and returns a new ctypes pointer type. Pointer types are cached and reused internally, so calling this function repeatedly is cheap. *type* must be a ctypes type.

**pointer** *(obj)*
This function creates a new pointer instance, pointing to *obj*. The returned object is of the type `POINTER(type(obj))`.

Note: If you just want to pass a pointer to an object to a foreign function call, you should use `byref(obj)` which is much faster.

**resize** *(obj, size)*
This function resizes the internal memory buffer of *obj*, which must be an instance of a ctypes type. It is not possible to make the buffer smaller than the native size of the objects type, as given by `sizeof(type(obj))`, but it is possible to enlarge the buffer.

**set_conversion_mode** *(encoding, errors)*
This function sets the rules that ctypes objects use when converting between 8-bit strings and unicode strings. *encoding* must be a string specifying an encoding, like 'utf-8' or 'mbcs', *errors* must be a string specifying the error handling on encoding/decoding errors. Examples of possible values are "strict", "replace", or "ignore".

`set_conversion_mode` returns a 2-tuple containing the previous conversion rules. On windows, the initial conversion rules are ('mbcs', 'ignore'), on other systems ('ascii', 'strict').
**sizeof**(obj_or_type)

Returns the size in bytes of a ctypes type or instance memory buffer. Does the same as the C `sizeof()` function.

**string_at**(address[, size])

This function returns the string starting at memory address address. If size is specified, it is used as size, otherwise the string is assumed to be zero-terminated.

**WinError**(code=None, descr=None)

Windows only: this function is probably the worst-named thing in ctypes. It creates an instance of WindowsError. If code is not specified, `GetLastError` is called to determine the error code. If descr is not specified, `FormatError` is called to get a textual description of the error.

**wstring_at**(address)

This function returns the wide character string starting at memory address address as unicode string. If size is specified, it is used as the number of characters of the string, otherwise the string is assumed to be zero-terminated.
14.14.2.6 Data types

class _CData

This non-public class is the common base class of all ctypes data types. Among other things, all ctypes type instances contain a memory block that hold C compatible data; the address of the memory block is returned by the `addressof()` helper function. Another instance variable is exposed as `_objects`; this contains other Python objects that need to be kept alive in case the memory block contains pointers.

Common methods of ctypes data types, these are all class methods (to be exact, they are methods of the metaclass):

`from_address(address)`

This method returns a ctypes type instance using the memory specified by address which must be an integer.

`from_param(obj)`

This method adapts obj to a ctypes type. It is called with the actual object used in a foreign function call, when the type is present in the foreign functions `argtypes` tuple; it must return an object that can be used as function call parameter.

All ctypes data types have a default implementation of this classmethod, normally it returns `obj` if that is an instance of the type. Some types accept other objects as well.

`in_dll(name, library)`

This method returns a ctypes type instance exported by a shared library. `name` is the name of the symbol that exports the data, `library` is the loaded shared library.

Common instance variables of ctypes data types:

`_b_base_`
Sometimes ctypes data instances do not own the memory block they contain, instead they share part of the memory block of a base object. The \_b\_base\_ readonly member is the root ctypes object that owns the memory block.

\_b\_needsfree\_

This readonly variable is true when the ctypes data instance has allocated the memory block itself, false otherwise.

\_objects

This member is either None or a dictionary containing Python objects that need to be kept alive so that the memory block contents is kept valid. This object is only exposed for debugging; never modify the contents of this dictionary.
14.14.2.7 Fundamental data types

class _SimpleCData

This non-public class is the base class of all fundamental ctypes data types. It is mentioned here because it contains the common attributes of the fundamental ctypes data types. _SimpleCData is a subclass of _CData, so it inherits their methods and attributes.

Instances have a single attribute:

value

This attribute contains the actual value of the instance. For integer and pointer types, it is an integer, for character types, it is a single character string, for character pointer types it is a Python string or unicode string.

When the value attribute is retrieved from a ctypes instance, usually a new object is returned each time. ctypes does not implement original object return, always a new object is constructed. The same is true for all other ctypes object instances.

Fundamental data types, when returned as foreign function call results, or, for example, by retrieving structure field members or array items, are transparently converted to native Python types. In other words, if a foreign function has a restype of c_char_p, you will always receive a Python string, not a c_char_p instance.

Subclasses of fundamental data types do not inherit this behaviour. So, if a foreign functions restype is a subclass of c_void_p, you will receive an instance of this subclass from the function call. Of course, you can get the value of the pointer by accessing the value attribute.

These are the fundamental ctypes data types:

class c_byte

Represents the C signed char datatype, and interprets the value as small
integer. The constructor accepts an optional integer initializer; no overflow checking is done.

**class c_char**
Represents the C char datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

**class c_char_p**
Represents the C char * datatype, which must be a pointer to a zero-terminated string. The constructor accepts an integer address, or a string.

**class c_double**
Represents the C double datatype. The constructor accepts an optional float initializer.

**class c_float**
Represents the C double datatype. The constructor accepts an optional float initializer.

**class c_int**
Represents the C signed int datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias to `c_long`.

**class c_int8**
Represents the C 8-bit signed int datatype. Usually an alias for `c_byte`.

**class c_int16**
Represents the C 16-bit signed int datatype. Usually an alias for `c_short`.

**class c_int32**
Represents the C 32-bit signed int datatype. Usually an alias for `c_int`.

**class c_int64**
Represents the C 64-bit signed int datatype. Usually an alias for `c_longlong`. 
**class c_long**
Represents the C signed long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

**class c_longlong**
Represents the C signed long long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

**class c_short**
Represents the C signed short datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

**class c_size_t**
Represents the C size_t datatype.

**class c_ubyte**
Represents the C unsigned char datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

**class c_uint**
Represents the C unsigned int datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where sizeof(int) == sizeof(long) it is an alias for c_ulong.

**class c_uint8**
Represents the C 8-bit unsigned int datatype. Usually an alias for c_ubyte.

**class c_uint16**
Represents the C 16-bit unsigned int datatype. Usually an alias for c_ushort.

**class c_uint32**
Represents the C 32-bit unsigned int datatype. Usually an alias for c_uint.

**class c_uint64**
Represents the C 64-bit unsigned int datatype. Usually an alias for
class c_ulong
Represents the C unsigned long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class c_ulonglong
Represents the C unsigned long long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class c_ushort
Represents the C unsigned short datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class c_void_p
Represents the C void * type. The value is represented as integer. The constructor accepts an optional integer initializer.

class c_wchar
Represents the C wchar_t datatype, and interprets the value as a single character unicode string. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class c_wchar_p
Represents the C wchar_t * datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.

class HRESULT
Windows only: Represents a HRESULT value, which contains success or error information for a function or method call.

py_object : classdesc*

Represents the C PyObject * datatype. Calling this with an without an argument creates a NULL PyObject * pointer.

The ctypes.wintypes module provides quite some other Windows specific data types, for example HWND, WPARAM, or DWORD. Some useful structures like
MSG or RECT are also defined.
14.14.2.8 Structured data types

class Union(*args, **kw)
    Abstract base class for unions in native byte order.

class BigEndianStructure(*args, **kw)
    Abstract base class for structures in big endian byte order.

class LittleEndianStructure(*args, **kw)
    Abstract base class for structures in little endian byte order.

Structures with non-native byte order cannot contain pointer type fields, or any other data types containing pointer type fields.

class Structure(*args, **kw)
    Abstract base class for structures in native byte order.

Concrete structure and union types must be created by subclassing one of these types, and at least define a _fields_ class variable. ctypes will create descriptors which allow reading and writing the fields by direct attribute accesses. These are the

_fields_
    A sequence defining the structure fields. The items must be 2-tuples or 3-tuples. The first item is the name of the field, the second item specifies the type of the field; it can be any ctypes data type.

    For integer type fields like c_int, a third optional item can be given. It must be a small positive integer defining the bit width of the field.

    Field names must be unique within one structure or union. This is not checked, only one field can be accessed when names are repeated.

    It is possible to define the _fields_ class variable after the class statement that defines the Structure subclass, this allows to create data types
that directly or indirectly reference themselves:

```python
class List(Structure):
    pass
List._fields_ = [('pnext', POINTER(List)),
                 ...
```

The `_fields_` class variable must, however, be defined before the type is first used (an instance is created, `sizeof()` is called on it, and so on). Later assignments to the `_fields_` class variable will raise an AttributeError.

Structure and union subclass constructors accept both positional and named arguments. Positional arguments are used to initialize the fields in the same order as they appear in the `_fields_` definition, named arguments are used to initialize the fields with the corresponding name.

It is possible to defined sub-subclasses of structure types, they inherit the fields of the base class plus the `_fields_` defined in the sub-subclass, if any.

`_pack_`

An optional small integer that allows to override the alignment of structure fields in the instance. `_pack_` must already be defined when `_fields_` is assigned, otherwise it will have no effect.

`_anonymous_`

An optional sequence that lists the names of unnamed (anonymous) fields. `_anonymous_` must be already defined when `_fields_` is assigned, otherwise it will have no effect.

The fields listed in this variable must be structure or union type fields. `ctypes` will create descriptors in the structure type that allows to access the nested fields directly, without the need to create the structure or union field.

Here is an example type (Windows):

```python
class _U(Union):
    _fields_ = [('lptdesc', POINTER(TYPEDESC)),
                ('lpadesc', POINTER(ARRAYDESC)),
                ]
```
class TYPEDESC(Structure):
    _fields_ = ["u", _U],
               ("vt", VARTYPE)]

    _anonymous_ = ("u",)

The TYPEDESC structure describes a COM data type, the vt field specifies which one of the union fields is valid. Since the u field is defined as anonymous field, it is now possible to access the members directly off the TYPEDESC instance. td.lptdesc and td.u.lptdesc are equivalent, but the former is faster since it does not need to create a temporary union instance:

    td = TYPEDESC()
    td.vt = VT_PTR
    td.lptdesc = POINTER(some_type)
    td.u.lptdesc = POINTER(some_type)

It is possible to defined sub-subclasses of structures, they inherit the fields of the base class. If the subclass definition has a separate _fields_ variable, the fields specified in this are appended to the fields of the base class.

Structure and union constructors accept both positional and keyword arguments. Positional arguments are used to initialize member fields in the same order as they are appear in _fields_. Keyword arguments in the constructor are interpreted as attribute assignments, so they will initialize _fields_ with the same name, or create new attributes for names not present in _fields_.

Next: 14.14.2.9 Arrays and pointers

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
14.14.2.9 Arrays and pointers

XXX

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
15.3.4.1 Semaphore Example

Semaphores are often used to guard resources with limited capacity, for example, a database server. In any situation where the size of the resource size is fixed, you should use a bounded semaphore. Before spawning any worker threads, your main thread would initialize the semaphore:

```python
maxconnections = 5
...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Once spawned, worker threads call the semaphore's acquire and release methods when they need to connect to the server:

```python
pool_sema.acquire()
conn = connectdb()
... use connection ...
conn.close()
pool_sema.release()
```

The use of a bounded semaphore reduces the chance that a programming error which causes the semaphore to be released more than it's acquired will go undetected.
17.1.1.1 Convenience Functions

This module also defines two shortcut functions:

**call**(*popenargs, **kwargs*)

Run command with arguments. Wait for command to complete, then return the returncode attribute.

The arguments are the same as for the Popen constructor. Example:

```python
retcode = call(['ls', '-l'])
```

**check_call**(*popenargs, **kwargs*)

Run command with arguments. Wait for command to complete. If the exit code was zero then return, otherwise raise CalledProcessError. The CalledProcessError object will have the return code in the returncode attribute.

The arguments are the same as for the Popen constructor. Example:

```python
check_call(['ls', '-l'])
```
17.1.1.2 Exceptions

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent. Additionally, the exception object will have one extra attribute called child_traceback, which is a string containing traceback information from the childs point of view.

The most common exception raised is OSError. This occurs, for example, when trying to execute a non-existent file. Applications should prepare for OSError exceptions.

A ValueError will be raised if Popen is called with invalid arguments.

check_call() will raise CalledProcessError, if the called process returns a non-zero return code.
17.1.1.3 Security

Unlike some other popen functions, this implementation will never call /bin/sh implicitly. This means that all characters, including shell metacharacters, can safely be passed to child processes.
17.1.3.1 Replacing /bin/sh shell backquote

```python
output=`mycmd myarg`
==> output = Popen(['mycmd', 'myarg'], stdout=PIPE).communicate()[0]
```
17.1.3.2 Replacing shell pipe line

```
output=`dmesg | grep hda`
==> 
p1 = Popen(['dmesg'], stdout=PIPE)
p2 = Popen(['grep', 'hda'], stdin=p1.stdout, stdout=PIPE)
output = p2.communicate()[0]
```
17.1.3.3 Replacing os.system()

```python
sts = os.system("mycmd" + " myarg")
==>  
p = Popen("mycmd" + " myarg", shell=True)  
sts = os.waitpid(p.pid, 0)
```

Notes:

- Calling the program through the shell is usually not required.
- It's easier to look at the `returncode` attribute than the exit status.

A more realistic example would look like this:

```python
try:  
    retcode = call("mycmd" + " myarg", shell=True)  
    if retcode < 0:  
        print >>sys.stderr, "Child was terminated by signal", -retco  
    else:  
        print >>sys.stderr, "Child returned", retcode  
except OSError, e:  
    print >>sys.stderr, "Execution failed:", e
```

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
17.1.3.4 Replacing os.spawn*

P_NOWAIT example:

```
    pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
    ==> pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT example:

```
    retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
    ==> retcode = call(["/bin/mycmd", "myarg"])
```

Vector example:

```
    os.spawnvp(os.P_NOWAIT, path, args)
    ==> Popen([path] + args[1:])
```

Environment example:

```
    os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
    ==> Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```
17.1.3.5 Replacing os.popen*

```
pipe = os.popen(cmd, mode='r', bufsize)
==> pipe = Popen(cmd, shell=True, bufsize=bufsize, stdout=PIPE).stdout

pipe = os.popen(cmd, mode='w', bufsize)
==> pipe = Popen(cmd, shell=True, bufsize=bufsize, stdin=PIPE).stdin

(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
==> p = Popen(cmd, shell=True, bufsize=bufsize, stdin=PIPE, stdout=PIPE, close_fds=True)
   (child_stdin, child_stdout) = (p.stdin, p.stdout)

(child_stdin, child_stdout, child_stderr) = os.popen3(cmd, mode, bufsize)
==> p = Popen(cmd, shell=True, bufsize=bufsize, stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
   (child_stdin, child_stdout, child_stderr) = (p.stdin, p.stdout, p.stderr)

(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
==> p = Popen(cmd, shell=True, bufsize=bufsize, stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
   (child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```
17.1.3.6 Replacing popen2.*

Note: If the cmd argument to popen2 functions is a string, the command is executed through /bin/sh. If it is a list, the command is directly executed.

```python
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, m
==>
p = Popen(["somestring"], shell=True, bufsize=bufsize,
stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)

(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufs
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

The popen2.Popen3 and popen2.Popen4 basically works as subprocess.Popen, except that:

- subprocess.Popen raises an exception if the execution fails
- the `capturestderr` argument is replaced with the `stderr` argument.
- `stdin=PIPE` and `stdout=PIPE` must be specified.
- `popen2` closes all file descriptors by default, but you have to specify `close_fds=True` with subprocess.Popen.
20.1.2.1 How To Use This Section

This section is designed in two parts: the first half (roughly) covers background material, while the second half can be taken to the keyboard as a handy reference.

When trying to answer questions of the form ``how do I do blah'', it is often best to find out how to do `blah' in straight Tk, and then convert this back into the corresponding Tkinter call. Python programmers can often guess at the correct Python command by looking at the Tk documentation. This means that in order to use Tkinter, you will have to know a little bit about Tk. This document can't fulfill that role, so the best we can do is point you to the best documentation that exists. Here are some hints:

- The authors strongly suggest getting a copy of the Tk man pages. Specifically, the man pages in the man directory are most useful. The man3 man pages describe the C interface to the Tk library and thus are not especially helpful for script writers.

- Addison-Wesley publishes a book called Tcl and the Tk Toolkit by John Ousterhout (ISBN 0-201-63337-X) which is a good introduction to Tcl and Tk for the novice. The book is not exhaustive, and for many details it defers to the man pages.

- Tkinter.py is a last resort for most, but can be a good place to go when nothing else makes sense.

See Also:

ActiveState Tcl Home Page

The Tk/Tcl development is largely taking place at ActiveState.

Tcl and the Tk Toolkit

The book by John Ousterhout, the inventor of Tcl.
20.1.2.2 A Simple Hello World Program

from Tkinter import *

class Application(Frame):
    def say_hi(self):
        print "hi there, everyone!"

    def createWidgets(self):
        self.QUIT = Button(self)
        self.QUIT["text"] = "QUIT"
        self.QUIT["fg"] = "red"
        self.QUIT["command"] = self.quit

        self.QUIT.pack({"side": "left"})

        self.hi_there = Button(self)
        self.hi_there["text"] = "Hello",
        self.hi_there["command"] = self.say_hi

        self.hi_there.pack({"side": "left"})

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

root = Tk()
app = Application(master=root)
app.mainloop()
root.destroy()
20.1.6.1 Setting Options

Options control things like the color and border width of a widget. Options can be set in three ways:

At object creation time, using keyword arguments

```
fred = Button(self, fg = "red", bg = "blue")
```

After object creation, treating the option name like a dictionary index

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

Use the config() method to update multiple attrs subsequent to object creation

```
fred.config(fg = "red", bg = "blue")
```

For a complete explanation of a given option and its behavior, see the Tk man pages for the widget in question.

Note that the man pages list "STANDARD OPTIONS" and "WIDGET SPECIFIC OPTIONS" for each widget. The former is a list of options that are common to many widgets, the latter are the options that are idiosyncratic to that particular widget. The Standard Options are documented on the options(3) man page.

No distinction between standard and widget-specific options is made in this document. Some options don't apply to some kinds of widgets. Whether a given widget responds to a particular option depends on the class of the widget; buttons have a command option, labels do not.

The options supported by a given widget are listed in that widget's man page, or can be queried at runtime by calling the config() method without arguments,
or by calling the `keys()` method on that widget. The return value of these calls is a dictionary whose key is the name of the option as a string (for example, 'relief') and whose values are 5-tuples.

Some options, like bg are synonyms for common options with long names (bg is shorthand for "background"). Passing the `config()` method the name of a shorthand option will return a 2-tuple, not 5-tuple. The 2-tuple passed back will contain the name of the synonym and the "real" option (such as ('bg', 'background')).

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>option name</td>
<td>'relief'</td>
</tr>
<tr>
<td>1</td>
<td>option name for database lookup</td>
<td>'relief'</td>
</tr>
<tr>
<td>2</td>
<td>option class for database lookup</td>
<td>'Relief'</td>
</tr>
<tr>
<td>3</td>
<td>default value</td>
<td>'raised'</td>
</tr>
<tr>
<td>4</td>
<td>current value</td>
<td>'groove'</td>
</tr>
</tbody>
</table>

Example:

```python
>>> print fred.config()
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

Of course, the dictionary printed will include all the options available and their values. This is meant only as an example.
20.1.6.2 The Packer

The packer is one of Tk's geometry-management mechanisms.

Geometry managers are used to specify the relative positioning of the positioning of widgets within their container - their mutual master. In contrast to the more cumbersome placer (which is used less commonly, and we do not cover here), the packer takes qualitative relationship specification - above, to the left of, filling, etc - and works everything out to determine the exact placement coordinates for you.

The size of any master widget is determined by the size of the "slave widgets" inside. The packer is used to control where slave widgets appear inside the master into which they are packed. You can pack widgets into frames, and frames into other frames, in order to achieve the kind of layout you desire. Additionally, the arrangement is dynamically adjusted to accommodate incremental changes to the configuration, once it is packed.

Note that widgets do not appear until they have had their geometry specified with a geometry manager. It's a common early mistake to leave out the geometry specification, and then be surprised when the widget is created but nothing appears. A widget will appear only after it has had, for example, the packer's pack() method applied to it.

The pack() method can be called with keyword-option/value pairs that control where the widget is to appear within its container, and how it is to behave when the main application window is resized. Here are some examples:

```python
fred.pack() # defaults to side = "top"
fred.pack(side = "left")
fred.pack(expand = 1)
```
20.1.6.3 Packer Options

For more extensive information on the packer and the options that it can take, see the man pages and page 183 of John Ousterhout's book.

anchor
Anchor type. Denotes where the packer is to place each slave in its parcel.

expand
Boolean, 0 or 1.

fill
Legal values: 'x', 'y', 'both', 'none'.

ipadx and ipady
A distance - designating internal padding on each side of the slave widget.

padx and pady
A distance - designating external padding on each side of the slave widget.

side
Legal values are: 'left', 'right', 'top', 'bottom'.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
The current-value setting of some widgets (like text entry widgets) can be connected directly to application variables by using special options. These options are variable, textvariable, onvalue, offvalue, and value. This connection works both ways: if the variable changes for any reason, the widget it's connected to will be updated to reflect the new value.

Unfortunately, in the current implementation of Tkinter it is not possible to hand over an arbitrary Python variable to a widget through a variable or textvariable option. The only kinds of variables for which this works are variables that are subclassed from a class called Variable, defined in the Tkinter module.

There are many useful subclasses of Variable already defined: StringVar, IntVar, DoubleVar, and BooleanVar. To read the current value of such a variable, call the get() method on it, and to change its value you call the set() method. If you follow this protocol, the widget will always track the value of the variable, with no further intervention on your part.

For example:

```python
class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        # here is the application variable
        self.contents = StringVar()
        # set it to some value
        self.contents.set("this is a variable")
        # tell the entry widget to watch this variable
        self.entrythingy["textvariable"] = self.contents

        # and here we get a callback when the user hits return.
        # we will have the program print out the value of the
        # application variable when the user hits return
```
self.entrythingy.bind('<Key-Return>',
    self.print_contents)

def print_contents(self, event):
    print "hi. contents of entry is now ---->", \
    self.contents.get()
20.1.6.5 The Window Manager

In Tk, there is a utility command, `wm`, for interacting with the window manager. Options to the `wm` command allow you to control things like titles, placement, icon bitmaps, and the like. In Tkinter, these commands have been implemented as methods on the `Wm` class. Toplevel widgets are subclassed from the `Wm` class, and so can call the `Wm` methods directly.

To get at the toplevel window that contains a given widget, you can often just refer to the widget's master. Of course if the widget has been packed inside of a frame, the master won't represent a toplevel window. To get at the toplevel window that contains an arbitrary widget, you can call the `_root()` method. This method begins with an underscore to denote the fact that this function is part of the implementation, and not an interface to Tk functionality.

Here are some examples of typical usage:

```python
from Tkinter import *
class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

    # create the application
    myapp = App()

    # # here are method calls to the window manager class
    # #
    # myapp.master.title("My Do-Nothing Application")
    # myapp.master.maxsize(1000, 400)

    # start the program
    myapp.mainloop()
```
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20.1.6.6 Tk Option Data Types

anchor
Legal values are points of the compass: "n", "ne", "e", "se", "s", "sw", "w", "nw", and also "center".

bitmap
There are eight built-in, named bitmaps: 'error', 'gray25', 'gray50', 'hourglass', 'info', 'questhead', 'question', 'warning'. To specify an X bitmap filename, give the full path to the file, preceded with an @, as in "@/usr/contrib(bitmap/gumby.bit".

boolean
You can pass integers 0 or 1 or the strings "yes" or "no".

callback
This is any Python function that takes no arguments. For example:

```python
def print_it():
    print "hi there"
fred["command"] = print_it
```

color
Colors can be given as the names of X colors in the rgb.txt file, or as strings representing RGB values in 4 bit: "#RGB", 8 bit: "#RRGGBB", 12 bit "#RRGGBBB", or 16 bit "#RRRGGGGBBBB" ranges, where R,G,B here represent any legal hex digit. See page 160 of Ousterhout's book for details.

cursor
The standard X cursor names from cursorfont.h can be used, without the XC_ prefix. For example to get a hand cursor (XC_hand2), use the string "hand2". You can also specify a bitmap and mask file of your own. See page 179 of Ousterhout's book.
distance
Screen distances can be specified in either pixels or absolute distances. Pixels are given as numbers and absolute distances as strings, with the trailing character denoting units: c for centimetres, i for inches, m for millimetres, p for printer's points. For example, 3.5 inches is expressed as "3.5i".

font
Tk uses a list font name format, such as {courier 10 bold}. Font sizes with positive numbers are measured in points; sizes with negative numbers are measured in pixels.

gemoetry
This is a string of the form "widthxheight", where width and height are measured in pixels for most widgets (in characters for widgets displaying text). For example: fred["geometry"] = "200x100".

justify
Legal values are the strings: "left", "center", "right", and "fill".

region
This is a string with four space-delimited elements, each of which is a legal distance (see above). For example: "2 3 4 5" and "3i 2i 4.5i 2i" and "3c 2c 4c 10.43c" are all legal regions.

relief
Determines what the border style of a widget will be. Legal values are: "raised", "sunken", "flat", "groove", and "ridge".

scrollcommand
This is almost always the set() method of some scrollbar widget, but can be any widget method that takes a single argument. Refer to the file Demo/tkinter/matt/canvas-with-scrollbars.py in the Python source distribution for an example.

wrap:
Must be one of: "none", "char", or "word".
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20.1.6.7 Bindings and Events

The bind method from the widget command allows you to watch for certain events and to have a callback function trigger when that event type occurs. The form of the bind method is:

\[
def bind(self, sequence, func, add=''):\]

where:

- **sequence**
  is a string that denotes the target kind of event. (See the bind man page and page 201 of John Ousterhout's book for details).

- **func**
  is a Python function, taking one argument, to be invoked when the event occurs. An Event instance will be passed as the argument. (Functions deployed this way are commonly known as *callbacks*.)

- **add**
  is optional, either "" or "+". Passing an empty string denotes that this binding is to replace any other bindings that this event is associated with. Preceeding with a "+" means that this function is to be added to the list of functions bound to this event type.

For example:

\[
def turnRed(self, event):
    event.widget["activeforeground"] = "red"
    self.button.bind("<Enter>", self.turnRed)
\]

Notice how the widget field of the event is being accessed in the \texttt{turnRed()} callback. This field contains the widget that caught the X event. The following table lists the other event fields you can access, and how they are denoted in Tk, which can be useful when referring to the Tk man pages.
The index Parameter

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20.1.6.8 The index Parameter

A number of widgets require ```index``` parameters to be passed. These are used to point at a specific place in a Text widget, or to particular characters in an Entry widget, or to particular menu items in a Menu widget.

**Entry widget indexes (index, view index, etc.)**
Entry widgets have options that refer to character positions in the text being displayed. You can use these `Tkinter` functions to access these special points in text widgets:

- **AtEnd()**
  refers to the last position in the text

- **AtInsert()**
  refers to the point where the text cursor is

- **AtSelFirst()**
  indicates the beginning point of the selected text

- **AtSelLast()**
  denotes the last point of the selected text and finally

- **At(x, y)**
  refers to the character at pixel location x, y (with y not used in the case of a text entry widget, which contains a single line of text).

**Text widget indexes**
The index notation for Text widgets is very rich and is best described in the Tk man pages.

**Menu indexes (menu.invoke(), menu.entryconfig(), etc.)**
Some options and methods for menus manipulate specific menu entries. Anytime a menu index is needed for an option or a parameter, you may pass in:
• an integer which refers to the numeric position of the entry in the widget, counted from the top, starting with 0;
• the string 'active', which refers to the menu position that is currently under the cursor;
• the string "last" which refers to the last menu item;
• An integer preceded by @, as in @6, where the integer is interpreted as a y pixel coordinate in the menu's coordinate system;
• the string "none", which indicates no menu entry at all, most often used with menu.activate() to deactivate all entries, and finally,
• a text string that is pattern matched against the label of the menu entry, as scanned from the top of the menu to the bottom. Note that this index type is considered after all the others, which means that matches for menu items labelled last, active, or none may be interpreted as the above literals, instead.
20.1.6.9 Images

Bitmap/Pixelmap images can be created through the subclasses of Tkinter.Image:

- **BitmapImage** can be used for X11 bitmap data.
- **PhotoImage** can be used for GIF and PPM/PGM color bitmaps.

Either type of image is created through either the **file** or the **data** option (other options are available as well).

The image object can then be used wherever an image option is supported by some widget (e.g. labels, buttons, menus). In these cases, Tk will not keep a reference to the image. When the last Python reference to the image object is deleted, the image data is deleted as well, and Tk will display an empty box wherever the image was used.
20.2.2.1 Basic Widgets

class Balloon()

A Balloon that pops up over a widget to provide help. When the user moves the cursor inside a widget to which a Balloon widget has been bound, a small pop-up window with a descriptive message will be shown on the screen.

class ButtonBox()

The ButtonBox widget creates a box of buttons, such as is commonly used for Ok Cancel.

class ComboBox()

The ComboBox widget is similar to the combo box control in MS Windows. The user can select a choice by either typing in the entry subwidget or selecting from the listbox subwidget.

class Control()

The Control widget is also known as the SpinBox widget. The user can adjust the value by pressing the two arrow buttons or by entering the value directly into the entry. The new value will be checked against the user-defined upper and lower limits.

class LabelEntry()

The LabelEntry widget packages an entry widget and a label into one mega widget. It can be used to simplify the creation of `entry-form` type of interface.

class LabelFrame()

The LabelFrame widget packages a frame widget and a label into one mega widget. To create widgets inside a LabelFrame widget, one creates the new widgets relative to the frame subwidget and manage them inside the frame subwidget.
class Meter()
    The Meter widget can be used to show the progress of a background job which may take a long time to execute.

class OptionMenu()
    The OptionMenu creates a menu button of options.

class PopupMenu()
    The PopupMenu widget can be used as a replacement of the tk_popup command. The advantage of the Tix PopupMenu widget is it requires less application code to manipulate.

class Select()
    The Select widget is a container of button subwidgets. It can be used to provide radio-box or check-box style of selection options for the user.

class StdButtonBox()
    The StdButtonBox widget is a group of standard buttons for Motif-like dialog boxes.
20.2.2.2 File Selectors

class DirList()

The DirList widget displays a list view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class DirTree()

The DirTree widget displays a tree view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class DirSelectDialog()

The DirSelectDialog widget presents the directories in the file system in a dialog window. The user can use this dialog window to navigate through the file system to select the desired directory.

class DirSelectBox()

The DirSelectBox is similar to the standard Motif(TM) directory-selection box. It is generally used for the user to choose a directory. DirSelectBox stores the directories mostly recently selected into a ComboBox widget so that they can be quickly selected again.

class ExFileSelectBox()

The ExFileSelectBox widget is usually embedded in a tixExFileSelectDialog widget. It provides a convenient method for the user to select files. The style of the ExFileSelectBox widget is very similar to the standard file dialog on MS Windows 3.1.

class FileSelectBox()

The FileSelectBox is similar to the standard Motif(TM) file-selection box. It is generally used for the user to choose a file. FileSelectBox stores the files mostly recently selected into a ComboBox widget so that they can be quickly selected again.
class FileEntry()

The FileEntry widget can be used to input a filename. The user can type in the filename manually. Alternatively, the user can press the button widget that sits next to the entry, which will bring up a file selection dialog.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20.2.2.3 Hierarchical ListBox

**class HList()**

The `HList` widget can be used to display any data that have a hierarchical structure, for example, file system directory trees. The list entries are indented and connected by branch lines according to their places in the hierarchy.

**class CheckList()**

The `CheckList` widget displays a list of items to be selected by the user. CheckList acts similarly to the Tk checkbutton or radiobutton widgets, except it is capable of handling many more items than checkbuttons or radiobuttons.

**class Tree()**

The `Tree` widget can be used to display hierarchical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.
20.2.2.4 Tabular ListBox

class TList()

The TList widget can be used to display data in a tabular format. The list entries of a TList widget are similar to the entries in the Tk listbox widget. The main differences are (1) the TList widget can display the list entries in a two dimensional format and (2) you can use graphical images as well as multiple colors and fonts for the list entries.
20.2.2.5 Manager Widgets

class PanedWindow()

The PanedWindow widget allows the user to interactively manipulate the sizes of several panes. The panes can be arranged either vertically or horizontally. The user changes the sizes of the panes by dragging the resize handle between two panes.

class ListNoteBook()

The ListNoteBook widget is very similar to the TixNoteBook widget: it can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages (windows). At one time only one of these pages can be shown. The user can navigate through these pages by choosing the name of the desired page in the hlist subwidget.

class NoteBook()

The NoteBook widget can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages. At one time only one of these pages can be shown. The user can navigate through these pages by choosing the visual `tabs'' at the top of the NoteBook widget.
20.2.2.6 Image Types

The Tix module adds:

- **Pixmap** capabilities to all Tix and Tkinter widgets to create color images from XPM files.

- **Compound** image types can be used to create images that consists of multiple horizontal lines; each line is composed of a series of items (texts, bitmaps, images or spaces) arranged from left to right. For example, a compound image can be used to display a bitmap and a text string simultaneously in a Tk Button widget.
20.2.2.7 Miscellaneous Widgets

class InputOnly()

    The InputOnly widgets are to accept inputs from the user, which can be done with the bind command (UNIX only).
20.2.2.8 Form Geometry Manager

In addition, `Tix` augments `Tkinter` by providing:

```
class Form():
    The `Form` geometry manager based on attachment rules for all Tk widgets.
```

Python Library Reference

Previous: 20.2.2.7 Miscellaneous Widgets Up: 20.2.2 Tix Widgets Next: 20.2.3 Tix Commands

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
20.5.1.1 File menu

New window
create a new editing window

Open...
open an existing file

Open module...
open an existing module (searches sys.path)

Class browser
show classes and methods in current file

Path browser
show sys.path directories, modules, classes and methods

Save
save current window to the associated file (unsaved windows have a * before and after the window title)

Save As...
save current window to new file, which becomes the associated file

Save Copy As...
save current window to different file without changing the associated file

Close
close current window (asks to save if unsaved)

Exit
close all windows and quit IDLE (asks to save if unsaved)
20.5.1.2 Edit menu

**Undo**
Undo last change to current window (max 1000 changes)

**Redo**
Redo last undone change to current window

**Cut**
Copy selection into system-wide clipboard; then delete selection

**Copy**
Copy selection into system-wide clipboard

**Paste**
Insert system-wide clipboard into window

**Select All**
Select the entire contents of the edit buffer

**Find...**
Open a search dialog box with many options

**Find again**
Repeat last search

**Find selection**
Search for the string in the selection

**Find in Files...**
Open a search dialog box for searching files

**Replace...**
Open a search-and-replace dialog box

**Go to line**
Ask for a line number and show that line

**Indent region**
Shift selected lines right 4 spaces

**Dedent region**
Shift selected lines left 4 spaces

**Comment out region**
Insert ## in front of selected lines

**Uncomment region**
Remove leading # or ## from selected lines

**Tabify region**
Turn *leading* stretches of spaces into tabs

**Untabify region**
Turn *all* tabs into the right number of spaces

**Expand word**
Expand the word you have typed to match another word in the same buffer; repeat to get a different expansion

**Format Paragraph**
Reformat the current blank-line-separated paragraph

**Import module**
Import or reload the current module

**Run script**
Execute the current file in the __main__ namespace
20.5.1.3 Windows menu

**Zoom Height**

toggles the window between normal size (24x80) and maximum height.

The rest of this menu lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).
20.5.1.4 Debug menu (in the Python Shell window only)

Go to file/line
look around the insert point for a filename and linenumber, open the file, and show the line.

Open stack viewer
show the stack traceback of the last exception

Debugger toggle
Run commands in the shell under the debugger

JIT Stack viewer toggle
Open stack viewer on traceback
20.5.2.1 Automatic indentation

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, Backspace deletes up to 4 spaces if they are there. Tab inserts 1-4 spaces (in the Python Shell window one tab). See also the indent/dedent region commands in the edit menu.
20.5.2.2 Python Shell window

- `c-c` interrupts executing command
- `c-d` sends end-of-file; closes window if typed at a ">>>>" prompt

- `Alt-p` retrieves previous command matching what you have typed
- `Alt-n` retrieves next
- `Return` while on any previous command retrieves that command
- `Alt-/` (Expand word) is also useful here
20.5.3.1 Command line usage

idle.py [-c command] [-d] [-e] [-s] [-t title] [arg] ...

- `c command` run this command
- `-d` enable debugger
- `-e` edit mode; arguments are files to be edited
- `-s` run $IDLESTARTUP or $PYTHONSTARTUP first
- `-t title` set title of shell window

If there are arguments:

1. If `-e` is used, arguments are files opened for editing and `sys.argv` reflects the arguments passed to IDLE itself.

2. Otherwise, if `-c` is used, all arguments are placed in `sys.argv[1:]`, with `sys.argv[0]` set to '-c'.

3. Otherwise, if neither `-e` nor `-c` is used, the first argument is a script which is executed with the remaining arguments in `sys.argv[1:]` and `sys.argv[0]` set to the script name. If the script name is '-', no script is executed but an interactive Python session is started; the arguments are still available in `sys.argv`. 
21.1.2.1 The NullTranslations class

Translation classes are what actually implement the translation of original source file message strings to translated message strings. The base class used by all translation classes is NullTranslations; this provides the basic interface you can use to write your own specialized translation classes. Here are the methods of NullTranslations:

__init__(fp)

Takes an optional file object fp, which is ignored by the base class. Initializes "protected" instance variables __info and __charset which are set by derived classes, as well as __fallback, which is set through add_fallback. It then calls self.__parse(fp) if fp is not None.

__parse(fp)

No-op'd in the base class, this method takes file object fp, and reads the data from the file, initializing its message catalog. If you have an unsupported message catalog file format, you should override this method to parse your format.

add_fallback(fallback)

Add fallback as the fallback object for the current translation object. A translation object should consult the fallback if it cannot provide a translation for a given message.

gettext(message)

If a fallback has been set, forward gettext() to the fallback. Otherwise, return the translated message. Overridden in derived classes.

lgettext(message)

If a fallback has been set, forward lgettext() to the fallback. Otherwise, return the translated message. Overridden in derived classes.

New in version 2.4.
**uggettext**(message)

If a fallback has been set, forward `uggettext()` to the fallback. Otherwise, return the translated message as a Unicode string. Overridden in derived classes.

**ngettext**(singular, plural, n)

If a fallback has been set, forward `ngettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

New in version 2.3.

**lngettext**(singular, plural, n)

If a fallback has been set, forward `ngettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

New in version 2.4.

**ungettext**(singular, plural, n)

If a fallback has been set, forward `ungettext()` to the fallback. Otherwise, return the translated message as a Unicode string. Overridden in derived classes.

New in version 2.3.

**info()**

Return the ``protected`` `_info` variable.

**charset()**

Return the ``protected`` `_charset` variable.

**output_charset()**

Return the ``protected`` `_output_charset` variable, which defines the encoding used to return translated messages.

New in version 2.4.

**set_output_charset**(charset)
Change the ``protected'' _output_charset variable, which defines the encoding used to return translated messages.

New in version 2.4.

install([unicode [, names]])

If the unicode flag is false, this method installs self.gettext() into the built-in namespace, binding it to "_". If unicode is true, it binds self.ugettext() instead. By default, unicode is false.

If the names parameter is given, it must be a sequence containing the names of functions you want to install in the builtin namespace in addition to _(). Supported names are 'gettext' (bound to self.gettext() or self.ugettext() according to the unicode flag), 'ngettext' (bound to self.ngettext() or self.ungettext() according to the unicode flag), 'lgettext' and 'lngettext'.

Note that this is only one way, albeit the most convenient way, to make the _ function available to your application. Because it affects the entire application globally, and specifically the built-in namespace, localized modules should never install _. Instead, they should use this code to make _ available to their module:

```python
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

This puts _ only in the module's global namespace and so only affects calls within this module.

Changed in version 2.5: Added the names parameter.
21.1.2.2 The GNUTranslations class

The gettext module provides one additional class derived from NullTranslations: GNUTranslations. This class overrides _parse() to enable reading GNU gettext format .mo files in both big-endian and little-endian format. It also coerces both message ids and message strings to Unicode.

GNUTranslations parses optional meta-data out of the translation catalog. It is convention with GNU gettext to include meta-data as the translation for the empty string. This meta-data is in RFC 822-style key: value pairs, and should contain the Project-Id-Version key. If the key Content-Type is found, then the charset property is used to initialize the "protected" _charset instance variable, defaulting to None if not found. If the charset encoding is specified, then all message ids and message strings read from the catalog are converted to Unicode using this encoding. The ugettext() method always returns a Unicode, while the gettext() returns an encoded 8-bit string. For the message id arguments of both methods, either Unicode strings or 8-bit strings containing only US-ASCII characters are acceptable. Note that the Unicode version of the methods (i.e. ugettext() and ungettext()) are the recommended interface to use for internationalized Python programs.

The entire set of key/value pairs are placed into a dictionary and set as the "protected" _info instance variable.

If the .mo file's magic number is invalid, or if other problems occur while reading the file, instantiating a GNUTranslations class can raise IOError.

The following methods are overridden from the base class implementation:

gettext( message)
Look up the message id in the catalog and return the corresponding message string, as an 8-bit string encoded with the catalog's charset encoding, if known. If there is no entry in the catalog for the message id, and a fallback has been set, the look up is forwarded to the fallback's
gettext() method. Otherwise, the message id is returned.

lgettext(message)
Equivalent to gettext(), but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with set_output_charset().

New in version 2.4.

ugettext(message)
Look up the message id in the catalog and return the corresponding message string, as a Unicode string. If there is no entry in the catalog for the message id, and a fallback has been set, the look up is forwarded to the fallback's ugettext() method. Otherwise, the message id is returned.

ngettext(singular, plural, n)
Do a plural-forms lookup of a message id. singular is used as the message id for purposes of lookup in the catalog, while n is used to determine which plural form to use. The returned message string is an 8-bit string encoded with the catalog's charset encoding, if known.

If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's ngettext() method. Otherwise, when n is 1 singular is returned, and plural is returned in all other cases.

New in version 2.3.

lngettext(singular, plural, n)
Equivalent to gettext(), but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with set_output_charset().

New in version 2.4.

ungettext(singular, plural, n)
Do a plural-forms lookup of a message id. singular is used as the message id for purposes of lookup in the catalog, while n is used to determine which plural form to use. The returned message string is a Unicode string.
If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's `ungettext()` method. Otherwise, when \(n\) is 1 *singular* is returned, and *plural* is returned in all other cases.

Here is an example:

```python
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ungettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

New in version 2.3.
21.1.2.3 Solaris message catalog support

The Solaris operating system defines its own binary .mo file format, but since no documentation can be found on this format, it is not supported at this time.
21.1.2.4 The Catalog constructor

GNOME uses a version of the gettext module by James Henstridge, but this version has a slightly different API. Its documented usage was:

```python
import gettext
cat = gettextCatalog(domain, localedir)
_ = cat.gettext
print _('hello world')
```

For compatibility with this older module, the function Catalog() is an alias for the translation() function described above.

One difference between this module and Henstridge's: his catalog objects supported access through a mapping API, but this appears to be unused and so is not currently supported.
21.1.3.1 Localizing your module

If you are localizing your module, you must take care not to make global changes, e.g. to the built-in namespace. You should not use the GNU gettext API but instead the class-based API.

Let's say your module is called ``spam'' and the module's various natural language translation .mo files reside in /usr/share/locale in GNU gettext format. Here's what you would put at the top of your module:

```python
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.lgettext
```

If your translators were providing you with Unicode strings in their .po files, you'd instead do:

```python
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.ugettext
```

---

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
21.1.3.2 Localizing your application

If you are localizing your application, you can install the \_( )\ function globally into the built-in namespace, usually in the main driver file of your application. This will let all your application-specific files just use \_( ' . . ' )\ without having to explicitly install it in each file.

In the simple case then, you need only add the following bit of code to the main driver file of your application:

```python
import gettext
gettext.install('myapplication')
```

If you need to set the locale directory or the `unicode` flag, you can pass these into the `install()` function:

```python
import gettext
gettext.install('myapplication', '/usr/share/locale', unicode=1)
```
21.1.3.3 Changing languages on the fly

If your program needs to support many languages at the same time, you may want to create multiple translation instances and then switch between them explicitly, like so:

```python
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```
In most coding situations, strings are translated where they are coded. Occasionally however, you need to mark strings for translation, but defer actual translation until later. A classic example is:

```python
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python',
]
#
for a in animals:
    print a
```

Here, you want to mark the strings in the `animals` list as being translatable, but you don’t actually want to translate them until they are printed.

Here is one way you can handle this situation:

```python
def _(message): return message
animals = [_(m) for m in
           'mollusk',
           'albatross',
           'rat',
           'penguin',
           'python',
          ]
del _
#
for a in animals:
    print _(a)
```

This works because the dummy definition of `_` simply returns the string unchanged. And this dummy definition will temporarily override any definition of `_` in the built-in namespace (until the `del` command). Take care, though if you have a previous definition of `_` in the local namespace.
Note that the second use of \_( )\ will not identify ``a'' as being translatable to the \texttt{pygettext} program, since it is not a string.

Another way to handle this is with the following example:

\begin{verbatim}
    def N_(message): return message

    animals = [N_('mollusk'),
               N_('albatross'),
               N_('rat'),
               N_('penguin'),
               N_('python'),
               ]

    # ...
    for a in animals:
        print \_(a)
\end{verbatim}

In this case, you are marking translatable strings with the function \texttt{N\_()},\footnote{21.5} which won't conflict with any definition of \_( ). However, you will need to teach your message extraction program to look for translatable strings marked with \texttt{N\_()}. \texttt{pygettext} and \texttt{xpot} both support this through the use of command line switches.

---

**Footnotes**

\footnote{21.5} The choice of \texttt{N\_()} here is totally arbitrary; it could have just as easily been \texttt{MarkThisStringForTranslation()}.
21.1.3.5 gettext() vs. lgettext()

In Python 2.4 the lgettext() family of functions were introduced. The intention of these functions is to provide an alternative which is more compliant with the current implementation of GNU gettext. Unlike gettext(), which returns strings encoded with the same codeset used in the translation file, lgettext() will return strings encoded with the preferred system encoding, as returned by locale.getpreferredencoding(). Also notice that Python 2.4 introduces new functions to explicitly choose the codeset used in translated strings. If a codeset is explicitly set, even lgettext() will return translated strings in the requested codeset, as would be expected in the GNU gettext implementation.
23.2.3.1 Which Docstrings Are Examined?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched.

In addition, if M.__test__ exists and "is true", it must be a dict, and each entry maps a (string) name to a function object, class object, or string. Function and class object docstrings found from M.__test__ are searched, and strings are treated as if they were docstrings. In output, a key K in M.__test__ appears with name

\[<\text{name of M}>.__\text{test__}.K\]

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes.

Changed in version 2.4: A "private name" concept is deprecated and no longer documented.
23.2.3.2 How are Docstring Examples Recognized?

In most cases a copy-and-paste of an interactive console session works fine, but doctest isn't trying to do an exact emulation of any specific Python shell. All hard tab characters are expanded to spaces, using 8-column tab stops. If you don't believe tabs should mean that, too bad: don't use hard tabs, or write your own DocTestParser class.

Changed in version 2.4: Expanding tabs to spaces is new; previous versions tried to preserve hard tabs, with confusing results.

```python
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...    print "yes"
... else:
...    print "no"
...    print "NO"
...    print "NO!!"
...
no
NO
NO!!
>>>```

Any expected output must immediately follow the final '>>> ' or '...
line containing the code, and the expected output (if any) extends to the next '>>> ' or all-whitespace line.

The fine print:

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put `<BLANKLINE>` in your doctest example each place a blank line is expected. Changed in version 2.4: `<BLANKLINE>` was added; there was no way to use expected output containing empty lines in previous versions.
• Output to stdout is captured, but not output to stderr (exception tracebacks are captured via a different means).

• If you continue a line via backslashing in an interactive session, or for any other reason use a backslash, you should use a raw docstring, which will preserve your backslashes exactly as you type them:

```python
>>> def f(x):
    ...   r'''Backslashes in a raw docstring: m\n'''
>>> print f.__doc__
Backslashes in a raw docstring: m
```

Otherwise, the backslash will be interpreted as part of the string. For example, the "\" above would be interpreted as a newline character. Alternatively, you can double each backslash in the doctest version (and not use a raw string):

```python
>>> def f(x):
    ...   ''Backslashes in a raw docstring: m\n''
>>> print f.__doc__
Backslashes in a raw docstring: m
```

• The starting column doesn't matter:

```python
>>> assert "Easy!"
>>> import math
    >>> math.floor(1.9)
1.0
```

and as many leading whitespace characters are stripped from the expected output as appeared in the initial '>>> ' line that started the example.
23.2.3.3 What's the Execution Context?

By default, each time `doctest` finds a docstring to test, it uses a shallow copy of M's globals, so that running tests doesn't change the module's real globals, and so that one test in M can't leave behind crumbs that accidentally allow another test to work. This means examples can freely use any names defined at top-level in M, and names defined earlier in the docstring being run. Examples cannot see names defined in other docstrings.

You can force use of your own own dict as the execution context by passing `globs=your_dict` to `testmod()` or `testfile()` instead.
23.2.3.4 What About Exceptions?

No problem, provided that the traceback is the only output produced by the example: just paste in the traceback. Since tracebacks contain details that are likely to change rapidly (for example, exact file paths and line numbers), this is one case where doctest works hard to be flexible in what it accepts.

Simple example:

```python
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
```

That doctest succeeds if `ValueError` is raised, with the "list.remove(x): x not in list" detail as shown.

The expected output for an exception must start with a traceback header, which may be either of the following two lines, indented the same as the first line of the example:

```plaintext
Traceback (most recent call last):
Traceback (innermost last):
```

The traceback header is followed by an optional traceback stack, whose contents are ignored by doctest. The traceback stack is typically omitted, or copied verbatim from an interactive session.

The traceback stack is followed by the most interesting part: the line(s) containing the exception type and detail. This is usually the last line of a traceback, but can extend across multiple lines if the exception has a multi-line detail:

```python
>>> raise ValueError('multi
				line
detail')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: multi
  line
detail
```
The last three lines (starting with `ValueError`) are compared against the exception's type and detail, and the rest are ignored.

Best practice is to omit the traceback stack, unless it adds significant documentation value to the example. So the last example is probably better as:

```python
>>> raise ValueError('multi
	line
detail')
Traceback (most recent call last):
...
ValueError: multi
line
detail
```

Note that tracebacks are treated very specially. In particular, in the rewritten example, the use of "..." is independent of doctest's ELLIPSIS option. The ellipsis in that example could be left out, or could just as well be three (or three hundred) commas or digits, or an indented transcript of a Monty Python skit.

Some details you should read once, but won't need to remember:

- Doctest can't guess whether your expected output came from an exception traceback or from ordinary printing. So, e.g., an example that expects "ValueError: 42 is prime" will pass whether `ValueError` is actually raised or if the example merely prints that traceback text. In practice, ordinary output rarely begins with a traceback header line, so this doesn't create real problems.

- Each line of the traceback stack (if present) must be indented further than the first line of the example, or start with a non-alphanumeric character. The first line following the traceback header indented the same and starting with an alphanumeric is taken to be the start of the exception detail. Of course this does the right thing for genuine tracebacks.

- When the `IGNORE_EXCEPTION_DETAIL` doctest option is is specified, everything following the leftmost colon is ignored.

- The interactive shell omits the traceback header line for some `SyntaxErrors`. But doctest uses the traceback header line to distinguish exceptions from non-exceptions. So in the rare case where you need to test a `SyntaxError` that omits the traceback header, you will need to manually add the traceback header line to your test example.
For some SyntaxErrors, Python displays the character position of the syntax error, using a ^ marker:

```python
>>> 1 1
  File "<stdin>", line 1
  1 1
  ^
  SyntaxError: invalid syntax
```

Since the lines showing the position of the error come before the exception type and detail, they are not checked by doctest. For example, the following test would pass, even though it puts the ^ marker in the wrong location:

```python
>>> 1 1
  Traceback (most recent call last):
  File "<stdin>", line 1
  1 1
  ^
  SyntaxError: invalid syntax
```

Changed in version 2.4: The ability to handle a multi-line exception detail, and the IGNORE_EXCEPTION_DETAIL doctest option, were added.

---

Footnotes

... traceback.\(^{23.1}\)

Examples containing both expected output and an exception are not supported. Trying to guess where one ends and the other begins is too error-prone, and that also makes for a confusing test.
23.2.3.5 Option Flags and Directives

A number of option flags control various aspects of doctest's behavior. Symbolic names for the flags are supplied as module constants, which can be or'ed together and passed to various functions. The names can also be used in doctest directives (see below).

The first group of options define test semantics, controlling aspects of how doctest decides whether actual output matches an example's expected output:

**DONT_ACCEPT_TRUE_FOR_1**
By default, if an expected output block contains just 1, an actual output block containing just 1 or just True is considered to be a match, and similarly for 0 versus False. When DONT_ACCEPT_TRUE_FOR_1 is specified, neither substitution is allowed. The default behavior caters to that Python changed the return type of many functions from integer to boolean; doctests expecting "little integer" output still work in these cases. This option will probably go away, but not for several years.

**DONT_ACCEPT_BLANKLINE**
By default, if an expected output block contains a line containing only the string `<BLANKLINE>`, then that line will match a blank line in the actual output. Because a genuinely blank line delimits the expected output, this is the only way to communicate that a blank line is expected. When DONT_ACCEPT_BLANKLINE is specified, this substitution is not allowed.

**NORMALIZE_WHITESPACE**
When specified, all sequences of whitespace (blanks and newlines) are treated as equal. Any sequence of whitespace within the expected output will match any sequence of whitespace within the actual output. By default, whitespace must match exactly. NORMALIZE_WHITESPACE is especially useful when a line of expected output is very long, and you want to wrap it across multiple lines in your source.

**ELLIPSIS**
When specified, an ellipsis marker (\ldots) in the expected output can match any substring in the actual output. This includes substrings that span line boundaries, and empty substrings, so it's best to keep usage of this simple. Complicated uses can lead to the same kinds of "oops, it matched too much!" surprises that .* is prone to in regular expressions.

**IGNORE_EXCEPTION_DETAIL**

When specified, an example that expects an exception passes if an exception of the expected type is raised, even if the exception detail does not match. For example, an example expecting "ValueError: 42" will pass if the actual exception raised is "ValueError: 3*14", but will fail, e.g., if TypeError is raised.

Note that a similar effect can be obtained using ELLIPSIS, and IGNORE_EXCEPTION_DETAIL may go away when Python releases prior to 2.4 become uninteresting. Until then, IGNORE_EXCEPTION_DETAIL is the only clear way to write a doctest that doesn't care about the exception detail yet continues to pass under Python releases prior to 2.4 (doctest directives appear to be comments to them). For example,

```
>>> (1, 2)[3] = 'moo' #doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

passes under Python 2.4 and Python 2.3. The detail changed in 2.4, to say "does not" instead of "doesn't".

**SKIP**

When specified, do not run the example at all. This can be useful in contexts where doctest examples serve as both documentation and test cases, and an example should be included for documentation purposes, but should not be checked. E.g., the example's output might be random; or the example might depend on resources which would be unavailable to the test driver.

The SKIP flag can also be used for temporarily "commenting out" examples.
COMPARISON_FLAGS
   A bitmask or’ing together all the comparison flags above.

The second group of options controls how test failures are reported:

REPORT_UDIFF
   When specified, failures that involve multi-line expected and actual outputs are displayed using a unified diff.

REPORT_CDIFF
   When specified, failures that involve multi-line expected and actual outputs will be displayed using a context diff.

REPORT_NDIFF
   When specified, differences are computed by difflib.Differ, using the same algorithm as the popular ndiff.py utility. This is the only method that marks differences within lines as well as across lines. For example, if a line of expected output contains digit 1 where actual output contains letter l, a line is inserted with a caret marking the mismatching column positions.

REPORT_ONLY_FIRST_FAILURE
   When specified, display the first failing example in each doctest, but suppress output for all remaining examples. This will prevent doctest from reporting correct examples that break because of earlier failures; but it might also hide incorrect examples that fail independently of the first failure. When REPORT_ONLY_FIRST_FAILURE is specified, the remaining examples are still run, and still count towards the total number of failures reported; only the output is suppressed.

REPORTING_FLAGS
   A bitmask or’ing together all the reporting flags above.

"Doctest directives" may be used to modify the option flags for individual examples. Doctest directives are expressed as a special Python comment following an example's source code:

```python
directive ::= "#" "doctest:"
            directive_options

directive_options ::= directive_option (",")
```

Whitespace is not allowed between the + or - and the directive option name. The directive option name can be any of the option flag names explained above.

An example's doctest directives modify doctest's behavior for that single example. Use + to enable the named behavior, or - to disable it.

For example, this test passes:

```python
>>> print range(20) # doctest: +NORMALIZE_WHITESPACE
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Without the directive it would fail, both because the actual output doesn't have two blanks before the single-digit list elements, and because the actual output is on a single line. This test also passes, and also requires a directive to do so:

```python
>>> print range(20) # doctest:+ELLIPSIS
[0, 1, ..., 18, 19]
```

Multiple directives can be used on a single physical line, separated by commas:

```python
>>> print range(20) # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
[0, 1, ..., 18, 19]
```

If multiple directive comments are used for a single example, then they are combined:

```python
>>> print range(20) # doctest: +ELLIPSIS...
... # doctest: +NORMALIZE_WHITESPACE
[0, 1, ..., 18, 19]
```

As the previous example shows, you can add "..." lines to your example
containing only directives. This can be useful when an example is too long for a directive to comfortably fit on the same line:

```python
>>> print range(5) + range(10,20) + range(30,40) + range(50,60)
... # doctest: +ELLIPSIS
[0, ..., 4, 10, ..., 19, 30, ..., 39, 50, ..., 59]
```

Note that since all options are disabled by default, and directives apply only to the example they appear in, enabling options (via + in a directive) is usually the only meaningful choice. However, option flags can also be passed to functions that run doctests, establishing different defaults. In such cases, disabling an option via - in a directive can be useful.

Changed in version 2.4: Constants `DONT_ACCEPT_BLANKLINE`, `NORMALIZE_WHITESPACE`, `ELLIPSIS`, `IGNORE_EXCEPTION_DETAIL`, `REPORT_UDIFF`, `REPORT_CDIFF`, `REPORT_NDIFF`, `REPORT_ONLY_FIRST_FAILURE`, `COMPARISON_FLAGS` and `REPORTING_FLAGS` were added; by default `<BLANKLINE>` in expected output matches an empty line in actual output; and doctest directives were added. Changed in version 2.5: Constant `SKIP` was added.

There's also a way to register new option flag names, although this isn't useful unless you intend to extend `doctest` internals via subclassing:

```python
register_optionflag(name)
```

Create a new option flag with a given name, and return the new flag's integer value. `register_optionflag()` can be used when subclassing `OutputChecker` or `DocTestRunner` to create new options that are supported by your subclasses. `register_optionflag` should always be called using the following idiom:

```python
MY_FLAG = register_optionflag('MY_FLAG')
```

New in version 2.4.
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
23.2.3.6 Warnings

doctest is serious about requiring exact matches in expected output. If even a single character doesn't match, the test fails. This will probably surprise you a few times, as you learn exactly what Python does and doesn't guarantee about output. For example, when printing a dict, Python doesn't guarantee that the key-value pairs will be printed in any particular order, so a test like

```python
>>> foo()
{'Hermione': 'hippogryph', 'Harry': 'broomstick'}
```

is vulnerable! One workaround is to do

```python
>>> foo() == {'Hermione': 'hippogryph', 'Harry': 'broomstick'}
True
```

instead. Another is to do

```python
>>> d = foo().items()
>>> d.sort()
>>> d
[('Harry', 'broomstick'), ('Hermione', 'hippogryph')]
```

There are others, but you get the idea.

Another bad idea is to print things that embed an object address, like

```python
>>> id(1.0) # certain to fail some of the time
7948648
>>> class C: pass
>>> C() # the default repr() for instances embeds an address
<__main__.C instance at 0x00AC18F0>
```

The ELLIPSIS directive gives a nice approach for the last example:

```python
>>> C() # doctest: +ELLIPSIS
<__main__.C instance at 0x...>
```

Floating-point numbers are also subject to small output variations across platforms, because Python defers to the platform C library for float formatting,
and C libraries vary widely in quality here.

```python
>>> 1./7  # risky
0.14285714285714285
>>> print 1./7 # safer
0.142857142857
>>> print round(1./7, 6) # much safer
0.142857
```

Numbers of the form $I/2^J$ are safe across all platforms, and I often contrive doctest examples to produce numbers of that form:

```python
>>> 3./4  # utterly safe
0.75
```

Simple fractions are also easier for people to understand, and that makes for better documentation.
23.2.6.1 DocTest Objects

class DocTest( examples, globs, name, filename, lineno, docstring)

A collection of doctest examples that should be run in a single namespace. The constructor arguments are used to initialize the member variables of the same names. New in version 2.4.

DocTest defines the following member variables. They are initialized by the constructor, and should not be modified directly.

examples
A list of Example objects encoding the individual interactive Python examples that should be run by this test.

globs
The namespace (aka globals) that the examples should be run in. This is a dictionary mapping names to values. Any changes to the namespace made by the examples (such as binding new variables) will be reflected in globs after the test is run.

name
A string name identifying the DocTest. Typically, this is the name of the object or file that the test was extracted from.

filename
The name of the file that this DocTest was extracted from; or None if the filename is unknown, or if the DocTest was not extracted from a file.

lineno
The line number within filename where this DocTest begins, or None if the line number is unavailable. This line number is zero-based with respect to the beginning of the file.

docstring
The string that the test was extracted from, or `None` if the string is
unavailable, or if the test was not extracted from a string.
23.2.6.2 Example Objects

class Example( source, want[, exc_msg][, lineno][, indent][, options])

A single interactive example, consisting of a Python statement and its expected output. The constructor arguments are used to initialize the member variables of the same names. New in version 2.4.

Example defines the following member variables. They are initialized by the constructor, and should not be modified directly.

**source**
A string containing the example's source code. This source code consists of a single Python statement, and always ends with a newline; the constructor adds a newline when necessary.

**want**
The expected output from running the example's source code (either from stdout, or a traceback in case of exception). **want** ends with a newline unless no output is expected, in which case it's an empty string. The constructor adds a newline when necessary.

**exc_msg**
The exception message generated by the example, if the example is expected to generate an exception; or None if it is not expected to generate an exception. This exception message is compared against the return value of traceback.format_exception_only(). **exc_msg** ends with a newline unless it's None. The constructor adds a newline if needed.

**lineno**
The line number within the string containing this example where the example begins. This line number is zero-based with respect to the beginning of the containing string.

**indent**
The example's indentation in the containing string, i.e., the number of space
characters that precede the example's first prompt.

**options**

A dictionary mapping from option flags to `True` or `False`, which is used to override default options for this example. Any option flags not contained in this dictionary are left at their default value (as specified by the `DocTestRunner`'s `optionflags`). By default, no options are set.
23.2.6.3 DocTestFinder objects

class DocTestFinder([verbose][, parser][, recurse][, exclude_empty])

A processing class used to extract the DocTests that are relevant to a
given object, from its docstring and the docstrings of its contained objects.
DocTests can currently be extracted from the following object types:
modules, functions, classes, methods, staticmethods, classmethods, and
properties.

The optional argument verbose can be used to display the objects searched
by the finder. It defaults to False (no output).

The optional argument parser specifies the DocTestParser object (or a
drop-in replacement) that is used to extract doctests from docstrings.

If the optional argument recurse is false, then DocTestFinder.find() will only examine the given object, and not any contained objects.

If the optional argument exclude_empty is false, then DocTestFinder.find() will include tests for objects with empty
docstrings.

New in version 2.4.

DocTestFinder defines the following method:

find( obj[, name][, module][, globs][, extraglobs])

Return a list of the DocTests that are defined by obj's docstring, or by any
of its contained objects' docstrings.

The optional argument name specifies the object's name; this name will be
used to construct names for the returned DocTests. If name is not
specified, then obj.__name__ is used.

The optional parameter module is the module that contains the given object.
If the module is not specified or is None, then the test finder will attempt to automatically determine the correct module. The object's module is used:

- As a default namespace, if `globs` is not specified.
- To prevent the DocTestFinder from extracting DocTests from objects that are imported from other modules. (Contained objects with modules other than `module` are ignored.)
- To find the name of the file containing the object.
- To help find the line number of the object within its file.

If `module` is `False`, no attempt to find the module will be made. This is obscure, of use mostly in testing doctest itself: if `module` is `False`, or is `None` but cannot be found automatically, then all objects are considered to belong to the (non-existent) module, so all contained objects will (recursively) be searched for doctests.

The globals for each `DocTest` is formed by combining `globs` and `extraglobs` (bindings in `extraglobs` override bindings in `globs`). A new shallow copy of the globals dictionary is created for each `DocTest`. If `globs` is not specified, then it defaults to the module's `__dict__`, if specified, or `{}` otherwise. If `extraglobs` is not specified, then it defaults to `{}`.
23.2.6.4 DocTestParser objects

class DocTestParser()

A processing class used to extract interactive examples from a string, and use them to create a DocTest object. New in version 2.4.

DocTestParser defines the following methods:

get_doctest( string, globs, name, filename, lineno)

Extract all doctest examples from the given string, and collect them into a DocTest object.

globs, name, filename, and lineno are attributes for the new DocTest object. See the documentation for DocTest for more information.

get_examples( string[, name])

Extract all doctest examples from the given string, and return them as a list of Example objects. Line numbers are 0-based. The optional argument name is a name identifying this string, and is only used for error messages.

parse( string[, name])

Divide the given string into examples and intervening text, and return them as a list of alternating Examples and strings. Line numbers for the Examples are 0-based. The optional argument name is a name identifying this string, and is only used for error messages.
23.2.6.5 DocTestRunner objects

class DocTestRunner( [checker][, verbose][, optionflags])

A processing class used to execute and verify the interactive examples in a DocTest.

The comparison between expected outputs and actual outputs is done by an OutputChecker. This comparison may be customized with a number of option flags; see section 23.2.3 for more information. If the option flags are insufficient, then the comparison may also be customized by passing a subclass of OutputChecker to the constructor.

The test runner's display output can be controlled in two ways. First, an output function can be passed to TestRunner.run(); this function will be called with strings that should be displayed. It defaults to sys.stdout.write. If capturing the output is not sufficient, then the display output can be also customized by subclassing DocTestRunner, and overriding the methods report_start, report_success, report_unexpected_exception, and report_failure.

The optional keyword argument checker specifies the OutputChecker object (or drop-in replacement) that should be used to compare the expected outputs to the actual outputs of doctest examples.

The optional keyword argument verbose controls the DocTestRunner's verbosity. If verbose is True, then information is printed about each example, as it is run. If verbose is False, then only failures are printed. If verbose is unspecified, or None, then verbose output is used iff the command-line switch -v is used.

The optional keyword argument optionflags can be used to control how the test runner compares expected output to actual output, and how it displays failures. For more information, see section 23.2.3.

New in version 2.4.
DocTestParser defines the following methods:

\textbf{report\_start}(\texttt{out, test, example})

Report that the test runner is about to process the given example. This method is provided to allow subclasses of \texttt{DocTestRunner} to customize their output; it should not be called directly.

\textit{example} is the example about to be processed. \textit{test} is the test containing \textit{example}. \textit{out} is the output function that was passed to \texttt{DocTestRunner.run()}. 

\textbf{report\_success}(\texttt{out, test, example, got})

Report that the given example ran successfully. This method is provided to allow subclasses of \texttt{DocTestRunner} to customize their output; it should not be called directly.

\textit{example} is the example about to be processed. \textit{got} is the actual output from the example. \textit{test} is the test containing \textit{example}. \textit{out} is the output function that was passed to \texttt{DocTestRunner.run()}. 

\textbf{report\_failure}(\texttt{out, test, example, got})

Report that the given example failed. This method is provided to allow subclasses of \texttt{DocTestRunner} to customize their output; it should not be called directly.

\textit{example} is the example about to be processed. \textit{got} is the actual output from the example. \textit{test} is the test containing \textit{example}. \textit{out} is the output function that was passed to \texttt{DocTestRunner.run()}. 

\textbf{report\_unexpected\_exception}(\texttt{out, test, example, exc\_info})

Report that the given example raised an unexpected exception. This method is provided to allow subclasses of \texttt{DocTestRunner} to customize their output; it should not be called directly.

\textit{example} is the example about to be processed. \textit{exc\_info} is a tuple containing information about the unexpected exception (as returned by \texttt{sys.exc\_info()}). \textit{test} is the test containing \textit{example}. \textit{out} is the output function that was passed to \texttt{DocTestRunner.run()}. 
**run**(*test[, compileflags][, out][, clear_globs]*)

Run the examples in *test* (a DocTest object), and display the results using the writer function *out*.

The examples are run in the namespace *test.globs*. If *clear_globs* is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use *clear_globs=False*.

*compileflags* gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to *globs*.

The output of each example is checked using the DocTestRunner's output checker, and the results are formatted by the DocTestRunner.report_* methods.

**summarize**(*[verbose]*)

Print a summary of all the test cases that have been run by this DocTestRunner, and return a tuple "(*failure_count*,  *test_count*)".

The optional *verbose* argument controls how detailed the summary is. If the verbosity is not specified, then the DocTestRunner's verbosity is used.
23.2.6.6 OutputChecker objects

class OutputChecker()

A class used to check the whether the actual output from a doctest example matches the expected output. OutputChecker defines two methods: check_output, which compares a given pair of outputs, and returns true if they match; and output_difference, which returns a string describing the differences between two outputs. New in version 2.4.

OutputChecker defines the following methods:

check_output( want, got, optionflags)

Return True iff the actual output from an example (got) matches the expected output (want). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See section 23.2.3 for more information about option flags.

output_difference( example, got, optionflags)

Return a string describing the differences between the expected output for a given example (example) and the actual output (got). optionflags is the set of option flags used to compare want and got.
30.1.6.1 Emulation of compile()

While many useful operations may take place between parsing and bytecode generation, the simplest operation is to do nothing. For this purpose, using the parser module to produce an intermediate data structure is equivalent to the code

```python
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

The equivalent operation using the parser module is somewhat longer, and allows the intermediate internal parse tree to be retained as an AST object:

```python
>>> import parser
>>> ast = parser.expr('a + 5')
>>> code = ast.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

An application which needs both AST and code objects can package this code into readily available functions:

```python
import parser

def load_suite(source_string):
    ast = parser.suite(source_string)
    return ast, ast.compile()

def load_expression(source_string):
    ast = parser.expr(source_string)
    return ast, ast.compile()
```
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
30.1.6.2 Information Discovery

Some applications benefit from direct access to the parse tree. The remainder of this section demonstrates how the parse tree provides access to module documentation defined in docstrings without requiring that the code being examined be loaded into a running interpreter via `import`. This can be very useful for performing analyses of untrusted code.

Generally, the example will demonstrate how the parse tree may be traversed to distill interesting information. Two functions and a set of classes are developed which provide programmatic access to high level function and class definitions provided by a module. The classes extract information from the parse tree and provide access to the information at a useful semantic level, one function provides a simple low-level pattern matching capability, and the other function defines a high-level interface to the classes by handling file operations on behalf of the caller. All source files mentioned here which are not part of the Python installation are located in the `Demo/parser/` directory of the distribution.

The dynamic nature of Python allows the programmer a great deal of flexibility, but most modules need only a limited measure of this when defining classes, functions, and methods. In this example, the only definitions that will be considered are those which are defined in the top level of their context, e.g., a function defined by a `def` statement at column zero of a module, but not a function defined within a branch of an `if ... else` construct, though there are some good reasons for doing so in some situations. Nesting of definitions will be handled by the code developed in the example.

To construct the upper-level extraction methods, we need to know what the parse tree structure looks like and how much of it we actually need to be concerned about. Python uses a moderately deep parse tree so there are a large number of intermediate nodes. It is important to read and understand the formal grammar used by Python. This is specified in the file `Grammar/Grammar` in the distribution. Consider the simplest case of interest when searching for docstrings: a module consisting of a docstring and nothing else. (See file `docstring.py`.)
Using the interpreter to take a look at the parse tree, we find a bewildering mass of numbers and parentheses, with the documentation buried deep in nested tuples.

```python
>>> import parser
>>> import pprint
>>> ast = parser.suite(open('docstring.py').read())
>>> tup = ast.totuple()
>>> pprint.pprint(tup)
(257,
 (264,
  (265,
   (266,
    (267,
     (307,
      (287,
       (288,
        (289,
         (290,
          (292,
           (293,
            (294,
             (295,
              (296,
               (297,
                (298,
                 (299,
                  (300, (3, """Some documentation.""
                 )))))))))))))))),
  (4, '.replaceAll
  ))),
 (4, 'println
 (0, 'println

The numbers at the first element of each node in the tree are the node types; they map directly to terminal and non-terminal symbols in the grammar. Unfortunately, they are represented as integers in the internal representation, and the Python structures generated do not change that. However, the `symbol` and `token` modules provide symbolic names for the node types and dictionaries which map from the integers to the symbolic names for the node types.

In the output presented above, the outermost tuple contains four elements: the integer 257 and three additional tuples. Node type 257 has the symbolic name `file_input`. Each of these inner tuples contains an integer as the first element; these integers, 264, 4, and 0, represent the node types `stmt`,
NEWLINE, and ENDMARKER, respectively. Note that these values may change depending on the version of Python you are using; consult symbol.py and token.py for details of the mapping. It should be fairly clear that the outermost node is related primarily to the input source rather than the contents of the file, and may be disregarded for the moment. The stmt node is much more interesting. In particular, all docstrings are found in subtrees which are formed exactly as this node is formed, with the only difference being the string itself. The association between the docstring in a similar tree and the defined entity (class, function, or module) which it describes is given by the position of the docstring subtree within the tree defining the described structure.

By replacing the actual docstring with something to signify a variable component of the tree, we allow a simple pattern matching approach to check any given subtree for equivalence to the general pattern for docstrings. Since the example demonstrates information extraction, we can safely require that the tree be in tuple form rather than list form, allowing a simple variable representation to be ['variable_name']. A simple recursive function can implement the pattern matching, returning a Boolean and a dictionary of variable name to value mappings. (See file example.py.)

```python
from types import ListType, TupleType

def match(pattern, data, vars=None):
    if vars is None:
        vars = {}
    if type(pattern) is ListType:
        vars[pattern[0]] = data
        return 1, vars
    if type(pattern) is not TupleType:
        return (pattern == data), vars
    if len(data) != len(pattern):
        return 0, vars
    for pattern, data in map(None, pattern, data):
        same, vars = match(pattern, data, vars)
        if not same:
            break
    return same, vars
```

Using this simple representation for syntactic variables and the symbolic node types, the pattern for the candidate docstring subtrees becomes fairly readable. (See file example.py.)

```python
import symbol
import token
```
DOCSTRING_STMT_PATTERN = (
    symbol.stmt,
    (symbol.simple_stmt,
    (symbol.small_stmt,
      (symbol.expr_stmt,
        (symbol.testlist,
          (symbol.test,
            (symbol.and_test,
              (symbol.not_test,
                (symbol.comparison,
                  (symbol.expr,
                    (symbol.xor_expr,
                      (symbol.and_expr,
                        (symbol.shift_expr,
                          (symbol.arith_expr,
                            (symbol.term,
                              (symbol.factor,
                                (symbol.power,
                                  (symbol.atom,
                                    (token.STRING, ['docstring'])))
                  ))))))))))
        ))))))
    (token.NEWLINE, '')
))

Using the \texttt{match()} function with this pattern, extracting the module docstring from the parse tree created previously is easy:

```python
>>> found, vars = match(DOCSTRING_STMT_PATTERN, tup[1])
>>> found
1
>>> vars
{'docstring': '\"\"\"Some documentation.\n\"\"\"'}
```

Once specific data can be extracted from a location where it is expected, the question of where information can be expected needs to be answered. When dealing with docstrings, the answer is fairly simple: the docstring is the first \texttt{stmt} node in a code block (\texttt{file_input} or \texttt{suite} node types). A module consists of a single \texttt{file_input} node, and class and function definitions each contain exactly one \texttt{suite} node. Classes and functions are readily identified as subtrees of code block nodes which start with \texttt{(stmt, (compound_stmt, (classdef, ... or (stmt, (compound_stmt, (funcdef, ....}

Note that these subtrees cannot be matched by \texttt{match()} since it does not support multiple sibling nodes to match without regard to number. A more elaborate matching function could be used to overcome this limitation, but this is sufficient for the example.
Given the ability to determine whether a statement might be a docstring and extract the actual string from the statement, some work needs to be performed to walk the parse tree for an entire module and extract information about the names defined in each context of the module and associate any docstrings with the names. The code to perform this work is not complicated, but bears some explanation.

The public interface to the classes is straightforward and should probably be somewhat more flexible. Each "major" block of the module is described by an object providing several methods for inquiry and a constructor which accepts at least the subtree of the complete parse tree which it represents. The ModuleInfo constructor accepts an optional name parameter since it cannot otherwise determine the name of the module.

The public classes include ClassInfo, FunctionInfo, and ModuleInfo. All objects provide the methods get_name(), get_docstring(), get_class_names(), and get_class_info(). The ClassInfo objects support get_method_names() and get_method_info() while the other classes provide get_function_names() and get_function_info().

Within each of the forms of code block that the public classes represent, most of the required information is in the same form and is accessed in the same way, with classes having the distinction that functions defined at the top level are referred to as "methods." Since the difference in nomenclature reflects a real semantic distinction from functions defined outside of a class, the implementation needs to maintain the distinction. Hence, most of the functionality of the public classes can be implemented in a common base class, SuiteInfoBase, with the accessors for function and method information provided elsewhere. Note that there is only one class which represents function and method information; this parallels the use of the def statement to define both types of elements.

Most of the accessor functions are declared in SuiteInfoBase and do not need to be overridden by subclasses. More importantly, the extraction of most information from a parse tree is handled through a method called by the SuiteInfoBase constructor. The example code for most of the classes is clear when read alongside the formal grammar, but the method which recursively creates new information objects requires further examination. Here is the
relevant part of the `SuiteInfoBase` definition from example.py:

```python
class SuiteInfoBase:
    _docstring = ''
    _name = ''

def __init__(self, tree = None):
    self._class_info = {}
    self._function_info = {}  
    if tree:
        self._extract_info(tree)

def _extract_info(self, tree):
    # extract docstring
    if len(tree) == 2:
        found, vars = match(DOCSTRING_STMT_PATTERN[1], tree[1])
    else:
        found, vars = match(DOCSTRING_STMT_PATTERN, tree[3])
    if found:
        self._docstring = eval(vars['docstring'])
    # discover inner definitions
    for node in tree[1:]:  
        found, vars = match(COMPOUND_STMT_PATTERN, node)
        if found:
            cstmt = vars['compound']
            if cstmt[0] == symbol.funcdef:
                name = cstmt[2][1]
                self._function_info[name] = FunctionInfo(cstmt)
            elif cstmt[0] == symbol.classdef:
                name = cstmt[2][1]
                self._class_info[name] = ClassInfo(cstmt)
```

After initializing some internal state, the constructor calls the `_extract_info()` method. This method performs the bulk of the information extraction which takes place in the entire example. The extraction has two distinct phases: the location of the docstring for the parse tree passed in, and the discovery of additional definitions within the code block represented by the parse tree.

The initial if test determines whether the nested suite is of the ``short form'' or the ``long form." The short form is used when the code block is on the same line as the definition of the code block, as in

```python
    def square(x): "Square an argument."; return x ** 2
```  

while the long form uses an indented block and allows nested definitions:
def make_power(exp):
    "Make a function that raises an argument to the exponent `exp'."
    def raiser(x, y=exp):
        return x ** y
    return raiser

When the short form is used, the code block may contain a docstring as the first, and possibly only, small_stmt element. The extraction of such a docstring is slightly different and requires only a portion of the complete pattern used in the more common case. As implemented, the docstring will only be found if there is only one small_stmt node in the simple_stmt node. Since most functions and methods which use the short form do not provide a docstring, this may be considered sufficient. The extraction of the docstring proceeds using the match() function as described above, and the value of the docstring is stored as an attribute of the SuiteInfoBase object.

After docstring extraction, a simple definition discovery algorithm operates on the stmt nodes of the suite node. The special case of the short form is not tested; since there are no stmt nodes in the short form, the algorithm will silently skip the single simple_stmt node and correctly not discover any nested definitions.

Each statement in the code block is categorized as a class definition, function or method definition, or something else. For the definition statements, the name of the element defined is extracted and a representation object appropriate to the definition is created with the defining subtree passed as an argument to the constructor. The representation objects are stored in instance variables and may be retrieved by name using the appropriate accessor methods.

The public classes provide any accessors required which are more specific than those provided by the SuiteInfoBase class, but the real extraction algorithm remains common to all forms of code blocks. A high-level function can be used to extract the complete set of information from a source file. (See file example.py.)

def get_docs(fileName):
    import os
    import parser

    source = open(fileName).read()
    basename = os.path.basename(os.path.splitext(fileName)[0])
    ast = parser.suite(source)
This provides an easy-to-use interface to the documentation of a module. If information is required which is not extracted by the code of this example, the code may be extended at clearly defined points to provide additional capabilities.
Abstract:

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for rapid application development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

This reference manual describes the syntax and ``core semantics'' of the language. It is terse, but attempts to be exact and complete. The semantics of non-essential built-in object types and of the built-in functions and modules are described in the Python Library Reference. For an informal introduction to the language, see the Python Tutorial. For C or C++ programmers, two additional manuals exist: Extending and Embedding the Python Interpreter describes the high-level picture of how to write a Python extension module, and the Python/C API Reference Manual describes the interfaces available to C/C++ programmers in detail.
Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
Contents

1. Introduction
   - 1.1 Alternate Implementations
   - 1.2 Notation

2. Lexical analysis
   - 2.1 Line structure
     - 2.1.1 Logical lines
     - 2.1.2 Physical lines
     - 2.1.3 Comments
     - 2.1.4 Encoding declarations
     - 2.1.5 Explicit line joining
     - 2.1.6 Implicit line joining
     - 2.1.7 Blank lines
     - 2.1.8 Indentation
     - 2.1.9 Whitespace between tokens
   - 2.2 Other tokens
   - 2.3 Identifiers and keywords
     - 2.3.1 Keywords
     - 2.3.2 Reserved classes of identifiers
   - 2.4 Literals
     - 2.4.1 String literals
     - 2.4.2 String literal concatenation
     - 2.4.3 Numeric literals
     - 2.4.4 Integer and long integer literals
     - 2.4.5 Floating point literals
     - 2.4.6 Imaginary literals
   - 2.5 Operators
   - 2.6 Delimiters

3. Data model
   - 3.1 Objects, values and types
   - 3.2 The standard type hierarchy
   - 3.3 New-style and classic classes
   - 3.4 Special method names
     - 3.4.1 Basic customization
     - 3.4.2 Customizing attribute access
     - 3.4.3 Customizing class creation
3.4.4 Emulating callable objects
3.4.5 Emulating container types
3.4.6 Additional methods for emulation of sequence types
3.4.7 Emulating numeric types
3.4.8 Coercion rules
3.4.9 With Statement Context Managers

4. Execution model
   - 4.1 Naming and binding
     - 4.1.1 Interaction with dynamic features
   - 4.2 Exceptions

5. Expressions
   - 5.1 Arithmetic conversions
   - 5.2 Atoms
     - 5.2.1 Identifiers (Names)
     - 5.2.2 Literals
     - 5.2.3 Parenthesized forms
     - 5.2.4 List displays
     - 5.2.5 Generator expressions
     - 5.2.6 Dictionary displays
     - 5.2.7 String conversions
   - 5.3 Primaries
     - 5.3.1 Attribute references
     - 5.3.2 Subscriptions
     - 5.3.3 Slicings
     - 5.3.4 Calls
   - 5.4 The power operator
   - 5.5 Unary arithmetic operations
   - 5.6 Binary arithmetic operations
   - 5.7 Shifting operations
   - 5.8 Binary bit-wise operations
   - 5.9 Comparisons
   - 5.10 Boolean operations
   - 5.11 Lambdas
   - 5.12 Expression lists
   - 5.13 Evaluation order
   - 5.14 Summary

6. Simple statements
   - 6.1 Expression statements
   - 6.2 Assert statements
6.3 Assignment statements
   - 6.3.1 Augmented assignment statements
6.4 The pass statement
6.5 The del statement
6.6 The print statement
6.7 The return statement
6.8 The yield statement
6.9 The raise statement
6.10 The break statement
6.11 The continue statement
6.12 The import statement
   - 6.12.1 Future statements
6.13 The global statement
6.14 The exec statement

7. Compound statements
   - 7.1 The if statement
   - 7.2 The while statement
   - 7.3 The for statement
   - 7.4 The try statement
   - 7.5 The with statement
   - 7.6 Function definitions
   - 7.7 Class definitions

8. Top-level components
   - 8.1 Complete Python programs
   - 8.2 File input
   - 8.3 Interactive input
   - 8.4 Expression input

A. History and License
   - A.1 History of the software
   - A.2 Terms and conditions for accessing or otherwise using Python
   - A.3 Licenses and Acknowledgements for Incorporated Software
      - A.3.1 Mersenne Twister
      - A.3.2 Sockets
      - A.3.3 Floating point exception control
      - A.3.4 MD5 message digest algorithm
      - A.3.5 Asynchronous socket services
      - A.3.6 Cookie management
      - A.3.7 Profiling
      - A.3.8 Execution tracing
- A.3.9 UUencode and UUdecode functions
- A.3.10 XML Remote Procedure Calls

Index
About this document ...


This document was generated using the \LaTeX2HTML translator.


The application of \LaTeX2HTML to the Python documentation has been heavily tailored by Fred L. Drake, Jr. Original navigation icons were contributed by Christopher Petrilli.
Comments and Questions

General comments and questions regarding this document should be sent by email to docs@python.org. If you find specific errors in this document, either in the content or the presentation, please report the bug at the Python Bug Tracker at SourceForge. If you are able to provide suggested text, either to replace existing incorrect or unclear material, or additional text to supplement what's already available, we'd appreciate the contribution. There's no need to worry about text markup; our documentation team will gladly take care of that.

Questions regarding how to use the information in this document should be sent to the Python news group, comp.lang.python, or the Python mailing list (which is gated to the newsgroup and carries the same content).

For any of these channels, please be sure not to send HTML email. Thanks.
3.4.2.1 More attribute access for new-style classes

The following methods only apply to new-style classes.

__getattribute__(self, name)

Called unconditionally to implement attribute accesses for instances of the class. If the class also defines __getattr__(), the latter will not be called unless __getattribute__() either calls it explicitly or raises an AttributeError. This method should return the (computed) attribute value or raise an AttributeError exception. In order to avoid infinite recursion in this method, its implementation should always call the base class method with the same name to access any attributes it needs, for example, "object.__getattribute__(self, name)".
3.4.2.2 Implementing Descriptors

The following methods only apply when an instance of the class containing the method (a so-called descriptor class) appears in the class dictionary of another new-style class, known as the owner class. In the examples below, ``the attribute'' refers to the attribute whose name is the key of the property in the owner class' `__dict__'. Descriptors can only be implemented as new-style classes themselves.

```python
__get__( self, instance, owner)
```

Called to get the attribute of the owner class (class attribute access) or of an instance of that class (instance attribute access). `owner` is always the owner class, while `instance` is the instance that the attribute was accessed through, or `None` when the attribute is accessed through the `owner`. This method should return the (computed) attribute value or raise an `AttributeError` exception.

```python
__set__( self, instance, value)
```

Called to set the attribute on an instance `instance` of the owner class to a new value, `value`.

```python
__delete__( self, instance)
```

Called to delete the attribute on an instance `instance` of the owner class.
3.4.2.3 Invoking Descriptors

In general, a descriptor is an object attribute with "binding behavior", one whose attribute access has been overridden by methods in the descriptor protocol: __get__(), __set__(), and __delete__(). If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, a.x has a lookup chain starting with a.__dict__['x'], then type(a).__dict__['x'], and continuing through the base classes of type(a) excluding metaclasses.

However, if the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined and how they were called. Note that descriptors are only invoked for new style objects or classes (ones that subclass object() or type()).

The starting point for descriptor invocation is a binding, a.x. How the arguments are assembled depends on a:

Direct Call
The simplest and least common call is when user code directly invokes a descriptor method: x.__get__(a).

Instance Binding
If binding to a new-style object instance, a.x is transformed into the call: type(a).__dict__['x'].__get__(a, type(a)).

Class Binding
If binding to a new-style class, A.x is transformed into the call: A.__dict__['x'].__get__(None, A).

Super Binding
If `a` is an instance of `super`, then the binding `super(B, obj).m()` searches `obj.__class__.__mro__` for the base class `A` immediately preceding `B` and then invokes the descriptor with the call: `A.__dict__['m'].__get__(obj, A)`.

For instance bindings, the precedence of descriptor invocation depends on the which descriptor methods are defined. Data descriptors define both `__get__()` and `__set__()`. Non-data descriptors have just the `__get__()` method. Data descriptors always override a redefinition in an instance dictionary. In contrast, non-data descriptors can be overridden by instances.

Python methods (including `staticmethod()` and `classmethod()`)) are implemented as non-data descriptors. Accordingly, instances can redefine and override methods. This allows individual instances to acquire behaviors that differ from other instances of the same class.

The `property()` function is implemented as a data descriptor. Accordingly, instances cannot override the behavior of a property.
3.4.2.4 __slots__

By default, instances of both old and new-style classes have a dictionary for attribute storage. This wastes space for objects having very few instance variables. The space consumption can become acute when creating large numbers of instances.

The default can be overridden by defining __slots__ in a new-style class definition. The __slots__ declaration takes a sequence of instance variables and reserves just enough space in each instance to hold a value for each variable. Space is saved because __dict__ is not created for each instance.

__slots__

This class variable can be assigned a string, iterable, or sequence of strings with variable names used by instances. If defined in a new-style class, __slots__ reserves space for the declared variables and prevents the automatic creation of __dict__ and __weakref__ for each instance. New in version 2.2.

Notes on using __slots__

- Without a __dict__ variable, instances cannot be assigned new variables not listed in the __slots__ definition. Attempts to assign to an unlisted variable name raises AttributeError. If dynamic assignment of new variables is desired, then add '__dict__' to the sequence of strings in the __slots__ declaration. Changed in version 2.3: Previously, adding '__dict__' to the __slots__ declaration would not enable the assignment of new attributes not specifically listed in the sequence of instance variable names.

- Without a __weakref__ variable for each instance, classes defining __slots__ do not support weak references to its instances. If weak reference support is needed, then add '__weakref__' to the sequence of strings in the __slots__ declaration. Changed in version 2.3: Previously, adding '__weakref__' to the __slots__ declaration would not enable support
for weak references.

- **__slots__** are implemented at the class level by creating descriptors (3.4.2) for each variable name. As a result, class attributes cannot be used to set default values for instance variables defined by **__slots__**: otherwise, the class attribute would overwrite the descriptor assignment.

- If a class defines a slot also defined in a base class, the instance variable defined by the base class slot is inaccessible (except by retrieving its descriptor directly from the base class). This renders the meaning of the program undefined. In the future, a check may be added to prevent this.

- The action of a **__slots__** declaration is limited to the class where it is defined. As a result, subclasses will have a **__dict__** unless they also define **__slots__**.

- **__slots__** do not work for classes derived from ``variable-length'' built-in types such as long, str and tuple.

- Any non-string iterable may be assigned to **__slots__**. Mappings may also be used; however, in the future, special meaning may be assigned to the values corresponding to each key.
Front Matter

Copyright © 2001-2006 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

See the end of this document for complete license and permissions information.

Abstract:

This library reference manual documents Python’s extensions for the Macintosh. It should be used in conjunction with the Python Library Reference, which documents the standard library and built-in types.

This manual assumes basic knowledge about the Python language. For an informal introduction to Python, see the Python Tutorial; the Python Reference Manual remains the highest authority on syntactic and semantic questions. Finally, the manual entitled Extending and Embedding the Python Interpreter describes how to add new extensions to Python and how to embed it in other applications.
Contents

- 1. Using Python on a Macintosh
  - 1.1 Getting and Installing MacPython
    - 1.1.1 How to run a Python script
    - 1.1.2 Running scripts with a GUI
    - 1.1.3 configuration
  - 1.2 The IDE
    - 1.2.1 Using the "Python Interactive" window
    - 1.2.2 Writing a Python Script
    - 1.2.3 Executing a script from within the IDE
    - 1.2.4 "Save as" versus "Save as Applet"
  - 1.3 The Package Manager
- 2. MacPython Modules
  - 2.1 macpath -- MacOS path manipulation functions
  - 2.2 macfs -- Various file system services
    - 2.2.1 FSSpec Objects
    - 2.2.2 Alias Objects
    - 2.2.3 FInfo Objects
  - 2.3 ic -- Access to Internet Config
    - 2.3.1 IC Objects
  - 2.4 MacOS -- Access to Mac OS interpreter features
  - 2.5 macostools -- Convenience routines for file manipulation
  - 2.6 findertools -- The finder's Apple Events interface
  - 2.7 EasyDialogs -- Basic Macintosh dialogs
    - 2.7.1 ProgressBar Objects
  - 2.8 FrameWork -- Interactive application framework
    - 2.8.1 Application Objects
    - 2.8.2 Window Objects
    - 2.8.3 ControlsWindow Object
    - 2.8.4 ScrolledWindow Object
    - 2.8.5 DialogWindow Objects
  - 2.9 autoGIL -- Global Interpreter Lock handling in event loops
- 3. MacPython OSA Modules
  - 3.1 gensuitemodule -- Generate OSA stub packages
  - 3.2 aetools -- OSA client support
  - 3.3 aepack -- Conversion between Python variables and AppleEvent
data containers

- 3.4 aetypes -- AppleEvent objects
- 3.5 MiniAEFrame -- Open Scripting Architecture server support
  - 3.5.1 AEServer Objects

4. MacOS Toolbox Modules
- 4.1 Carbon.AE -- Apple Events
- 4.2 Carbon.AH -- Apple Help
- 4.3 Carbon.App -- Appearance Manager
- 4.4 Carbon.CF -- Core Foundation
- 4.5 Carbon.CG -- Core Graphics
- 4.6 Carbon.CarbonEvt -- Carbon Event Manager
- 4.7 Carbon.Cm -- Component Manager
- 4.8 Carbon.Ctl -- Control Manager
- 4.9 Carbon.Dlg -- Dialog Manager
- 4.10 Carbon.Evt -- Event Manager
- 4.11 Carbon.Fm -- Font Manager
- 4.12 Carbon.Folder -- Folder Manager
- 4.13 Carbon.Help -- Help Manager
- 4.14 Carbon.List -- List Manager
- 4.15 Carbon.Menu -- Menu Manager
- 4.16 Carbon.Mlte -- MultiLingual Text Editor
- 4.17 Carbon.Qd -- QuickDraw
- 4.18 Carbon.Qdoffs -- QuickDraw Offscreen
- 4.19 Carbon.Qt -- QuickTime
- 4.20 Carbon.Res -- Resource Manager and Handles
- 4.21 Carbon.Scrap -- Scrap Manager
- 4.22 Carbon.Snd -- Sound Manager
- 4.23 Carbon.TE -- TextEdit
- 4.24 Carbon.Win -- Window Manager
- 4.25 ColorPicker -- Color selection dialog

5. Undocumented Modules
- 5.1 applesingle -- AppleSingle decoder
- 5.2 buildtools -- Helper module for BuildApplet and Friends
- 5.3 cfmfile -- Code Fragment Resource module
- 5.4 icopen -- Internet Config replacement for open()
- 5.5 macerrors -- Mac OS Errors
- 5.6 macresource -- Locate script resources
- 5.7 Nav -- NavServices calls
- 5.8 PixMapWrapper -- Wrapper for PixMap objects
5.9 videoreader -- Read QuickTime movies
5.10 W -- Widgets built on FrameWork

- A. History and License
  - A.1 History of the software
  - A.2 Terms and conditions for accessing or otherwise using Python
  - A.3 Licenses and Acknowledgements for Incorporated Software
    - A.3.1 Mersenne Twister
    - A.3.2 Sockets
    - A.3.3 Floating point exception control
    - A.3.4 MD5 message digest algorithm
    - A.3.5 Asynchronous socket services
    - A.3.6 Cookie management
    - A.3.7 Profiling
    - A.3.8 Execution tracing
    - A.3.9 UUencode and UUdecode functions
    - A.3.10 XML Remote Procedure Calls

- Module Index
- Index
Module Index

This index only lists modules documented in this manual. The *Global Module Index* lists all modules that are documented in this set of manuals.

Some module names are followed by an annotation indicating what platform they are available on.

```
aepack (Mac)
aetools (Mac)
aetypes (Mac)
aplesingle
autoGIL (Mac)
buildtools (Mac)
Carbon.AE (Mac)
Carbon.AH (Mac)
Carbon.App (Mac)
Carbon.CaronEvt
Carbon.CF (Mac)
Carbon.CG (Mac)
Carbon.Cm (Mac)
Carbon.Ctl
(Mac)
Carbon.Dlg
(Mac)
Carbon.Evt
(Mac)
Carbon.Fm
(Mac)
Carbon.Folder
(Mac)
Carbon.Help
(Mac)
Carbon.List
(Mac)
Carbon.Menu
(Mac)
Carbon.Mlte
(Mac)
Carbon.Qd
(Mac)
Carbon.Qdoffs
(Mac)
Carbon.Qt
(Mac)
Carbon.Res
(Mac)
Carbon.Scrap
(Mac)
Carbon.Snd
(Mac)
Carbon.TE
(Mac)
Carbon.Win
(Mac)
cfmfile
(Mac)
ColorPicker
(Mac)
EasyDialogs
(Mac)
findertools
(Mac)
Framework
(Mac)
gensuitemodule
(Mac)
ico
(Mac)
icopen
(Mac)
macerrrors
(Mac)
macfs
(Mac)
MacOS
(Mac)
macostools
macpath
macresource
MiniAEFrame
Nav (Mac)
PixMapWrap
 Videoreader
W (Mac)
```
About this document ...

Macintosh Library Modules, 19th September, 2006, Release 2.5

This document was generated using the LaTeX2HTML translator.


The application of LaTeX2HTML to the Python documentation has been heavily tailored by Fred L. Drake, Jr. Original navigation icons were contributed by Christopher Petrilli.
Comments and Questions

General comments and questions regarding this document should be sent by email to docs@python.org. If you find specific errors in this document, either in the content or the presentation, please report the bug at the Python Bug Tracker at SourceForge. If you are able to provide suggested text, either to replace existing incorrect or unclear material, or additional text to supplement what's already available, we'd appreciate the contribution. There's no need to worry about text markup; our documentation team will gladly take care of that.

Questions regarding how to use the information in this document should be sent to the Python news group, comp.lang.python, or the Python mailing list (which is gated to the newsgroup and carries the same content).

For any of these channels, please be sure not to send HTML email. Thanks.
Front Matter

Abstract:

Python is an interpreted, object-oriented programming language. This document describes how to write modules in C or C++ to extend the Python interpreter with new modules. Those modules can define new functions but also new object types and their methods. The document also describes how to embed the Python interpreter in another application, for use as an extension language. Finally, it shows how to compile and link extension modules so that they can be loaded dynamically (at run time) into the interpreter, if the underlying operating system supports this feature.

This document assumes basic knowledge about Python. For an informal introduction to the language, see the Python Tutorial. The Python Reference Manual gives a more formal definition of the language. The Python Library Reference documents the existing object types, functions and modules (both built-in and written in Python) that give the language its wide application range.

For a detailed description of the whole Python/C API, see the separate Python/C API Reference Manual.

Extending and Embedding the Python Interpreter

Previous: Extending and Embedding the Up: Extending and Embedding the
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
1. Extending and Embedding the Python with
Contents

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

- 2. Defining New Types
  - 2.1 The Basics
    - 2.1.1 Adding data and methods to the Basic example
    - 2.1.2 Providing finer control over data attributes
    - 2.1.3 Supporting cyclic garbage collection
  - 2.2 Type Methods
    - 2.2.1 Finalization and De-allocation
    - 2.2.2 Object Presentation
    - 2.2.3 Attribute Management
    - 2.2.4 Object Comparison
    - 2.2.5 Abstract Protocol Support
    - 2.2.6 Weak Reference Support
    - 2.2.7 More Suggestions

- 3. Building C and C++ Extensions with distutils
  - 3.1 Distributing your extension modules

- 4. Building C and C++ Extensions on Windows
  - 4.1 A Cookbook Approach
  - 4.2 Differences Between Unix and Windows
4.3 Using DLLs in Practice

5. Embedding Python in Another Application
   5.1 Very High Level Embedding
   5.2 Beyond Very High Level Embedding: An overview
   5.3 Pure Embedding
   5.4 Extending Embedded Python
   5.5 Embedding Python in C++
   5.6 Linking Requirements

A. Reporting Bugs

B. History and License
   B.1 History of the software
   B.2 Terms and conditions for accessing or otherwise using Python
   B.3 Licenses and Acknowledgements for Incorporated Software
      B.3.1 Mersenne Twister
      B.3.2 Sockets
      B.3.3 Floating point exception control
      B.3.4 MD5 message digest algorithm
      B.3.5 Asynchronous socket services
      B.3.6 Cookie management
      B.3.7 Profiling
      B.3.8 Execution tracing
      B.3.9 UUencode and UUdecode functions
      B.3.10 XML Remote Procedure Calls
About this document ...

Extending and Embedding the Python Interpreter, 19th September, 2006, Release 2.5

This document was generated using the LaTeX2HTML translator.


The application of LaTeX2HTML to the Python documentation has been heavily tailored by Fred L. Drake, Jr. Original navigation icons were contributed by Christopher Petrilli.
Comments and Questions

General comments and questions regarding this document should be sent by email to docs@python.org. If you find specific errors in this document, either in the content or the presentation, please report the bug at the Python Bug Tracker at SourceForge. If you are able to provide suggested text, either to replace existing incorrect or unclear material, or additional text to supplement what's already available, we'd appreciate the contribution. There's no need to worry about text markup; our documentation team will gladly take care of that.

Questions regarding how to use the information in this document should be sent to the Python news group, comp.lang.python, or the Python mailing list (which is gated to the newsgroup and carries the same content).

For any of these channels, please be sure not to send HTML email. Thanks.
2.2.3.1 Generic Attribute Management

New in version 2.2.

Most extension types only use simple attributes. So, what makes the attributes simple? There are only a couple of conditions that must be met:

1. The name of the attributes must be known when PyType_Ready() is called.

2. No special processing is needed to record that an attribute was looked up or set, nor do actions need to be taken based on the value.

Note that this list does not place any restrictions on the values of the attributes, when the values are computed, or how relevant data is stored.

When PyType_Ready() is called, it uses three tables referenced by the type object to create descriptors which are placed in the dictionary of the type object. Each descriptor controls access to one attribute of the instance object. Each of the tables is optional; if all three are NULL, instances of the type will only have attributes that are inherited from their base type, and should leave the tp_getattro and tp_setattro fields NULL as well, allowing the base type to handle attributes.

The tables are declared as three fields of the type object:

```c
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
```

If tp_methods is not NULL, it must refer to an array of PyMethodDef structures. Each entry in the table is an instance of this structure:

```c
typedef struct PyMethodDef {
    char    *ml_name;    /* method name */
    PyCFunction ml_meth;    /* implementation function */
    int      ml_flags;    /* flags */
    char    *ml_doc;    /* docstring */
} PyMethodDef;
```
One entry should be defined for each method provided by the type; no entries are needed for methods inherited from a base type. One additional entry is needed at the end; it is a sentinel that marks the end of the array. The \texttt{ml\_name} field of the sentinel must be \texttt{NULL}.

XXX Need to refer to some unified discussion of the structure fields, shared with the next section.

The second table is used to define attributes which map directly to data stored in the instance. A variety of primitive C types are supported, and access may be read-only or read-write. The structures in the table are defined as:

```c
typedef struct PyMemberDef {
    char *name;
    int type;
    int offset;
    int flags;
    char *doc;
} PyMemberDef;
```

For each entry in the table, a descriptor will be constructed and added to the type which will be able to extract a value from the instance structure. The \texttt{type} field should contain one of the type codes defined in the \texttt{structmember.h} header; the value will be used to determine how to convert Python values to and from C values. The \texttt{flags} field is used to store flags which control how the attribute can be accessed.

XXX Need to move some of this to a shared section!

The following flag constants are defined in \texttt{structmember.h}; they may be combined using bitwise-OR.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>READONLY</td>
<td>Never writable.</td>
</tr>
<tr>
<td>R0</td>
<td>Shorthand for \texttt{READONLY}.</td>
</tr>
<tr>
<td>READ_RESTRICTED</td>
<td>Not readable in restricted mode.</td>
</tr>
<tr>
<td>WRITE_RESTRICTED</td>
<td>Not writable in restricted mode.</td>
</tr>
<tr>
<td>RESTRICTED</td>
<td>Not readable or writable in restricted mode.</td>
</tr>
</tbody>
</table>
An interesting advantage of using the `tp_members` table to build descriptors that are used at runtime is that any attribute defined this way can have an associated doc string simply by providing the text in the table. An application can use the introspection API to retrieve the descriptor from the class object, and get the doc string using its `__doc__` attribute.

As with the `tp_methods` table, a sentinel entry with a name value of NULL is required.
2.2.3.2 Type-specific Attribute Management

For simplicity, only the char* version will be demonstrated here; the type of the name parameter is the only difference between the char* and PyObject* flavors of the interface. This example effectively does the same thing as the generic example above, but does not use the generic support added in Python 2.2. The value in showing this is two-fold: it demonstrates how basic attribute management can be done in a way that is portable to older versions of Python, and explains how the handler functions are called, so that if you do need to extend their functionality, you'll understand what needs to be done.

The tp_getattr handler is called when the object requires an attribute lookup. It is called in the same situations where the __getattr__() method of a class would be called.

A likely way to handle this is (1) to implement a set of functions (such as newdatatype_getSize() and newdatatype_setSize() in the example below), (2) provide a method table listing these functions, and (3) provide a getattr function that returns the result of a lookup in that table. The method table uses the same structure as the tp_methods field of the type object.

Here is an example:

```c
static PyMethodDef newdatatype_methods[] = {
    {"getSize", (PyCFunction)newdatatype_getSize, METH_VARARGS,
       "Return the current size."},
    {"setSize", (PyCFunction)newdatatype_setSize, METH_VARARGS,
       "Set the size."},
    {NULL, NULL, 0, NULL} /* sentinel */
};

static PyObject *
newdatatype_getattr(newdatatypeobject *obj, char *name)
{
    return Py_FindMethod(newdatatype_methods, (PyObject *)obj, name)
}
```

The tp_setattr handler is called when the __setattr__() or
__delattr__() method of a class instance would be called. When an attribute should be deleted, the third parameter will be NULL. Here is an example that simply raises an exception; if this were really all you wanted, the tp_setattr handler should be set to NULL.

```c
static int
newdatatype_setattr(newdatatypeobject *obj, char *name, PyObject *v)
{
    (void)PyErr_Format(PyExc_RuntimeError, "Read-only attribute: \%s",
                       name);
    return -1;
}
```

Extending and Embedding the Python Interpreter

Previous: 2.2.3.1 Generic Attribute Management Up: 2.2.3 Attribute Management Next: 2.2.4 Object Comparison

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.
Front Matter

Copyright © 2001-2006 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

See the end of this document for complete license and permissions information.

Abstract:

This manual documents the API used by C and C++ programmers who want to write extension modules or embed Python. It is a companion to Extending and Embedding the Python Interpreter, which describes the general principles of extension writing but does not document the API functions in detail.

Warning: The current version of this document is incomplete. I hope that it is nevertheless useful. I will continue to work on it, and release new versions from time to time, independent from Python source code releases.
Contents

- 1. Introduction
  - 1.1 Include Files
  - 1.2 Objects, Types and Reference Counts
    - 1.2.1 Reference Counts
    - 1.2.2 Types
  - 1.3 Exceptions
  - 1.4 Embedding Python
  - 1.5 Debugging Builds
- 2. The Very High Level Layer
- 3. Reference Counting
- 4. Exception Handling
  - 4.1 Standard Exceptions
  - 4.2 Deprecation of String Exceptions
- 5. Utilities
  - 5.1 Operating System Utilities
  - 5.2 Process Control
  - 5.3 Importing Modules
  - 5.4 Data marshalling support
  - 5.5 Parsing arguments and building values
- 6. Abstract Objects Layer
  - 6.1 Object Protocol
  - 6.2 Number Protocol
  - 6.3 Sequence Protocol
  - 6.4 Mapping Protocol
  - 6.5 Iterator Protocol
  - 6.6 Buffer Protocol
- 7. Concrete Objects Layer
  - 7.1 Fundamental Objects
    - 7.1.1 Type Objects
    - 7.1.2 The None Object
  - 7.2 Numeric Objects
    - 7.2.1 Plain Integer Objects
    - 7.2.2 Boolean Objects
    - 7.2.3 Long Integer Objects
    - 7.2.4 Floating Point Objects
- 7.2.5 Complex Number Objects
- 7.3 Sequence Objects
  - 7.3.1 String Objects
  - 7.3.2 Unicode Objects
  - 7.3.3 Buffer Objects
  - 7.3.4 Tuple Objects
  - 7.3.5 List Objects
- 7.4 Mapping Objects
  - 7.4.1 Dictionary Objects
- 7.5 Other Objects
  - 7.5.1 File Objects
  - 7.5.2 Instance Objects
  - 7.5.3 Function Objects
  - 7.5.4 Method Objects
  - 7.5.5 Module Objects
  - 7.5.6 Iterator Objects
  - 7.5.7 Descriptor Objects
  - 7.5.8 Slice Objects
  - 7.5.9 Weak Reference Objects
  - 7.5.10 CObjects
  - 7.5.11 Cell Objects
  - 7.5.12 Generator Objects
  - 7.5.13 DateTime Objects
  - 7.5.14 Set Objects
- 8. Initialization, Finalization, and Threads
  - 8.1 Thread State and the Global Interpreter Lock
  - 8.2 Profiling and Tracing
  - 8.3 Advanced Debugger Support
- 9. Memory Management
  - 9.1 Overview
  - 9.2 Memory Interface
  - 9.3 Examples
- 10. Object Implementation Support
  - 10.1 Allocating Objects on the Heap
  - 10.2 Common Object Structures
  - 10.3 Type Objects
  - 10.4 Mapping Object Structures
  - 10.5 Number Object Structures
  - 10.6 Sequence Object Structures
10.7 Buffer Object Structures
10.8 Supporting the Iterator Protocol
10.9 Supporting Cyclic Garbage Collection

- **A. Reporting Bugs**
- **B. History and License**
  - **B.1 History of the software**
  - **B.2 Terms and conditions for accessing or otherwise using Python**
  - **B.3 Licenses and Acknowledgements for Incorporated Software**
    - B.3.1 Mersenne Twister
    - B.3.2 Sockets
    - B.3.3 Floating point exception control
    - B.3.4 MD5 message digest algorithm
    - B.3.5 Asynchronous socket services
    - B.3.6 Cookie management
    - B.3.7 Profiling
    - B.3.8 Execution tracing
    - B.3.9 UUencode and UUdecode functions
    - B.3.10 XML Remote Procedure Calls

- **Index**
About this document ...


This document was generated using the \LaTeX2HTML translator.


The application of \LaTeX2HTML to the Python documentation has been heavily tailored by Fred L. Drake, Jr. Original navigation icons were contributed by Christopher Petrilli.
Comments and Questions

General comments and questions regarding this document should be sent by email to docs@python.org. If you find specific errors in this document, either in the content or the presentation, please report the bug at the Python Bug Tracker at SourceForge. If you are able to provide suggested text, either to replace existing incorrect or unclear material, or additional text to supplement what's already available, we'd appreciate the contribution. There's no need to worry about text markup; our documentation team will gladly take care of that.

Questions regarding how to use the information in this document should be sent to the Python news group, comp.lang.python, or the Python mailing list (which is gated to the newsgroup and carries the same content).

For any of these channels, please be sure not to send HTML email. Thanks.
1.2.1.1 Reference Count Details

The reference count behavior of functions in the Python/C API is best explained in terms of *ownership of references*. Ownership pertains to references, never to objects (objects are not owned: they are always shared). "Owning a reference" means being responsible for calling Py_Decref on it when the reference is no longer needed. Ownership can also be transferred, meaning that the code that receives ownership of the reference then becomes responsible for eventually decref'ing it by calling Py_Decref() or Py_XDecref() when it's no longer needed--or passing on this responsibility (usually to its caller). When a function passes ownership of a reference on to its caller, the caller is said to receive a new reference. When no ownership is transferred, the caller is said to borrow the reference. Nothing needs to be done for a borrowed reference.

Conversely, when a calling function passes it a reference to an object, there are two possibilities: the function steals a reference to the object, or it does not. *Stealing a reference* means that when you pass a reference to a function, that function assumes that it now owns that reference, and you are not responsible for it any longer.

Few functions steal references; the two notable exceptions are PyTuple_SetItem() and PyTuple_SetItem(), which steal a reference to the item (but not to the tuple or list into which the item is put!). These functions were designed to steal a reference because of a common idiom for populating a tuple or list with newly created objects; for example, the code to create the tuple (1, 2, "three") could look like this (forgetting about error handling for the moment; a better way to code this is shown below):

```c
PyObject *t;
    t = PyTuple_New(3);
    PyTuple_SetItem(t, 0, PyInt_FromLong(1L));
    PyTuple_SetItem(t, 1, PyInt_FromLong(2L));
    PyTuple_SetItem(t, 2, PyString_FromString("three"));
```

Here, PyInt_FromLong() returns a new reference which is immediately stolen by PyTuple_SetItem(). When you want to keep using an object
although the reference to it will be stolen, use Py_INCREF() to grab another reference before calling the reference-stealing function.

Incidentally, PyTuple_SetItem() is the only way to set tuple items; PySequence_SetItem() and PyObject_SetItem() refuse to do this since tuples are an immutable data type. You should only use PyTuple_SetItem() for tuples that you are creating yourself.

Equivalent code for populating a list can be written using PyList_New() and PyList_SetItem().

However, in practice, you will rarely use these ways of creating and populating a tuple or list. There's a generic function, Py_BuildValue(), that can create most common objects from C values, directed by a format string. For example, the above two blocks of code could be replaced by the following (which also takes care of the error checking):

```c
PyObject *tuple, *list;

tuple = Py_BuildValue("(iis)", 1, 2, "three");
list = Py_BuildValue("[iis]", 1, 2, "three");
```

It is much more common to use PyObject_SetItem() and friends with items whose references you are only borrowing, like arguments that were passed in to the function you are writing. In that case, their behaviour regarding reference counts is much saner, since you don't have to increment a reference count so you can give a reference away (``have it be stolen''). For example, this function sets all items of a list (actually, any mutable sequence) to a given item:

```c
int
set_all(PyObject *target, PyObject *item)
{
    int i, n;

    n = PyObject_Length(target);
    if (n < 0)
        return -1;
    for (i = 0; i < n; i++) {
        PyObject *index = PyInt_FromLong(i);
        if (!index)
            return -1;
        if (PyObject_SetItem(target, index, item) < 0)
            return -1;
        Py_DECREF(index);
    }
```
The situation is slightly different for function return values. While passing a reference to most functions does not change your ownership responsibilities for that reference, many functions that return a reference to an object give you ownership of the reference. The reason is simple: in many cases, the returned object is created on the fly, and the reference you get is the only reference to the object. Therefore, the generic functions that return object references, like PyObject_GetItem() and PySequence_GetItem(), always return a new reference (the caller becomes the owner of the reference).

It is important to realize that whether you own a reference returned by a function depends on which function you call only -- the plumage (the type of the object passed as an argument to the function) doesn't enter into it! Thus, if you extract an item from a list using PyList_GetItem(), you don't own the reference -- but if you obtain the same item from the same list using PySequence_GetItem() (which happens to take exactly the same arguments), you do own a reference to the returned object.

Here is an example of how you could write a function that computes the sum of the items in a list of integers; once using PyList_GetItem(), and once using PySequence_GetItem().

```c
long
sum_list(PyObject *list)
{
    int i, n;
    long total = 0;
    PyObject *item;

    n = PyList_Size(list);
    if (n < 0)
        return -1; /* Not a list */
    for (i = 0; i < n; i++) {
        item = PyList_GetItem(list, i); /* Can't fail */
        if (!PyInt_Check(item)) continue; /* Skip non-integers */
        total += PyInt_AsLong(item);
    }
    return total;
}

long
sum_sequence(PyObject *sequence)
```
{  
    int i, n;
    long total = 0;
    PyObject *item;
    n = PySequence_Length(sequence);
    if (n < 0)  
        return -1; /* Has no length */
    for (i = 0; i < n; i++) {  
        item = PySequence_GetItem(sequence, i);
        if (item == NULL)  
            return -1; /* Not a sequence, or other failure */
        if (PyInt_Check(item))
            total += PyInt_AsLong(item);
        Py_DECREF(item); /* Discard reference ownership */
    }
    return total;
}
7.2.5.1 Complex Numbers as C Structures

Note that the functions which accept these structures as parameters and return them as results do so by value rather than dereferencing them through pointers. This is consistent throughout the API.

**Py_complex**

The C structure which corresponds to the value portion of a Python complex number object. Most of the functions for dealing with complex number objects use structures of this type as input or output values, as appropriate. It is defined as:

```c
typedef struct {
    double real;
    double imag;
} Py_complex;
```

**Py_complex **_Py_c_sum**( Py_complex left, Py_complex right)

Return the sum of two complex numbers, using the C Py_complex representation.

**Py_complex **_Py_c_diff**( Py_complex left, Py_complex right)

Return the difference between two complex numbers, using the C Py_complex representation.

**Py_complex **_Py_c_neg**( Py_complex complex)

Return the negation of the complex number complex, using the C Py_complex representation.

**Py_complex **_Py_c_prod**( Py_complex left, Py_complex right)

Return the product of two complex numbers, using the C Py_complex representation.

**Py_complex **_Py_c_quot**( Py_complex dividend, Py_complex divisor)

Return the quotient of two complex numbers, using the C Py_complex representation.
representation.

Py_complex \_Py\_c\_pow(Py_complex num, Py_complex exp)

Return the exponentiation of num by exp, using the C Py_complex representation.
7.2.5.2 Complex Numbers as Python Objects

**PyComplexObject**
This subtype of PyObject represents a Python complex number object.

PyTypeObject **PyComplex_Type**
This instance of PyTypeObject represents the Python complex number type. It is the same object as complex and types.ComplexType.

int **PyComplex_Check**( PyObject *p)
Return true if its argument is a PyComplexObject or a subtype of PyComplexObject. Changed in version 2.2: Allowed subtypes to be accepted.

int **PyComplex_CheckExact**( PyObject *p)
Return true if its argument is a PyComplexObject, but not a subtype of PyComplexObject. New in version 2.2.

PyObject* **PyComplex_FromCComplex**( Py_complex v)
*Return value: New reference.*
Create a new Python complex number object from a C Py_complex value.

PyObject* **PyComplex_FromDoubles**( double real, double imag)
*Return value: New reference.*
Return a new PyComplexObject object from real and imag.

double **PyComplex_RealAsDouble**( PyObject *op)
Return the real part of op as a C double.

double **PyComplex_ImagAsDouble**( PyObject *op)
Return the imaginary part of op as a C double.
Py_complex  \texttt{PyComplex\_AsCComplex}( \texttt{PyObject \, *op} )

Return the Py\_complex value of the complex number \textit{op}.
7.3.2.1 Built-in Codecs

Python provides a set of builtin codecs which are written in C for speed. All of these codecs are directly usable via the following functions.

Many of the following APIs take two arguments encoding and errors. These parameters encoding and errors have the same semantics as the ones of the builtin unicode() Unicode object constructor.

Setting encoding to NULL causes the default encoding to be used which is ASCII. The file system calls should use `Py_FileSystemDefaultEncoding` as the encoding for file names. This variable should be treated as read-only: On some systems, it will be a pointer to a static string, on others, it will change at run-time (such as when the application invokes setlocale).

Error handling is set by errors which may also be set to NULL meaning to use the default handling defined for the codec. Default error handling for all builtin codecs is ``strict'' (`ValueError` is raised).

The codecs all use a similar interface. Only deviation from the following generic ones are documented for simplicity.

These are the generic codec APIs:

```c
PyObject* PyUnicode_Decode( const char *s, Py_ssize_t size, const char *encoding, const char *errors)
```

Create a Unicode object by decoding size bytes of the encoded string s. 
*encoding* and *errors* have the same meaning as the parameters of the same name in the `unicode()` builtin function. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

```c
const Py_UNICODE *s, Py_ssize_t size,
```
PyObject* **PyUnicode_Encode**( const char *encoding, const char *errors)

*Return value: New reference.*
Encode the Py_UNICODE buffer of the given size and return a Python string object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode *encode()* method. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

PyObject* **PyUnicode_AsEncodedString**( PyObject *unicode, char *encoding, const char *errors)

*Return value: New reference.*
Encode a Unicode object and return the result as Python string object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode *encode()* method. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

These are the UTF-8 codec APIs:

PyObject* **PyUnicode_DecodeUTF8**( const char *s, Py_ssize_t size, const char *errors)

*Return value: New reference.*
Create a Unicode object by decoding *size* bytes of the UTF-8 encoded string *s*. Return NULL if an exception was raised by the codec.

PyObject* **PyUnicode_DecodeUTF8Stateful**( size, const char *errors, Py_ssize_t *consumed)

*Return value: New reference.*
If consumed is NULL, behave like PyUnicode_DecodeUTF8(). If consumed is not NULL, trailing incomplete UTF-8 byte sequences will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in consumed. New in version 2.4.
PyObject* **PyUnicode_EncodeUTF8**( size, const char *errors)

*Return value:* New reference.
Encode the Py_UNICODE buffer of the given size using UTF-8 and return a Python string object. Return NULL if an exception was raised by the codec.

PyObject* **PyUnicode_AsUTF8String**( PyObject *unicode)

*Return value:* New reference.
Encode a Unicode objects using UTF-8 and return the result as Python string object. Error handling is ``strict``. Return NULL if an exception was raised by the codec.

These are the UTF-16 codec APIs:

PyObject* **PyUnicode_DecodeUTF16**( const char *s, Py_ssize_t size, const char *errors, int *byteorder)

*Return value:* New reference.
Decode length bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. errors (if non-NULL) defines the error handling. It defaults to ``strict``.

If byteorder is non-NULL, the decoder starts decoding using the given byte order:

*byteorder == -1: little endian
*byteorder == 0: native order
*byteorder == 1: big endian

and then switches according to all byte order marks (BOM) it finds in the input data. BOMs are not copied into the resulting Unicode string. After completion, *byteorder is set to the current byte order at the end of input data.

If byteorder is NULL, the codec starts in native order mode.

Return NULL if an exception was raised by the codec.

const char *s,
Py_ssize_t size, const
PyObject* **PyUnicode_DecodeUTF16Stateful** (char *errors, int *byteorder, Py_ssize_t *consumed)

*Return value: New reference.*

If `consumed` is NULL, behave like `PyUnicode_DecodeUTF16()`. If `consumed` is not NULL, `PyUnicode_DecodeUTF16Stateful()` will not treat trailing incomplete UTF-16 byte sequences (such as an odd number of bytes or a split surrogate pair) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in `consumed`. New in version 2.4.

```c
PyObject* PyUnicode_EncodeUTF16 (const Py_UNICODE* s, Py_ssize_t size, const char* errors, int byteorder)
```

*Return value: New reference.*

Return a Python string object holding the UTF-16 encoded value of the Unicode data in `s`. If `byteorder` is not 0, output is written according to the following byte order:

- `byteorder == -1`: little endian
- `byteorder == 0`: native byte order (writes a BOM mark)
- `byteorder == 1`: big endian

If byteorder is 0, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.

If `Py_UNICODE_WIDE` is defined, a single `Py_UNICODE` value may get represented as a surrogate pair. If it is not defined, each `Py_UNICODE` values is interpreted as an UCS-2 character.

Return NULL if an exception was raised by the codec.

PyObject* **PyUnicode_AsUTF16String** (PyObject *`unicode`)  

*Return value: New reference.*

Return a Python string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is ``strict''. Return NULL if an exception was raised by the codec.

These are the ``Unicode Escape'' codec APIs:
PyObject* **PyUnicode_DecodeUnicodeEscape**( Py_ssize_t size, const char *errors)

*Return value: New reference.*
Create a Unicode object by decoding *size* bytes of the Unicode-Escape encoded string *s*. Return NULL if an exception was raised by the codec.

PyObject* **PyUnicode_EncodeUnicodeEscape**( const Py_UNICODE *s, Py_ssize_t size)

*Return value: New reference.*
Encode the Py_UNICODE buffer of the given size using Unicode-Escape and return a Python string object. Return NULL if an exception was raised by the codec.

PyObject* **PyUnicode_AsUnicodeEscapeString**( PyObject *unicode)

*Return value: New reference.*
Encode a Unicode objects using Unicode-Escape and return the result as Python string object. Error handling is ``strict``. Return NULL if an exception was raised by the codec.

These are the ``Raw Unicode Escape`` codec APIs:

PyObject* **PyUnicode_DecodeRawUnicodeEscape**( Py_ssize_t size, const char *errors)

*Return value: New reference.*
Create a Unicode object by decoding *size* bytes of the Raw-Unicode-Escape encoded string *s*. Return NULL if an exception was raised by the codec.

PyObject* **PyUnicode_EncodeRawUnicodeEscape**( const Py_UNICODE *s, Py_ssize_t size, const char *errors)

*Return value: New reference.*
Encode the Py_UNICODE buffer of the given size using Raw-Unicode-Escape and return a Python string object. Return NULL if an exception was raised by the codec.
PyObject* `PyUnicode_AsRawUnicodeEscapeString`(PyObject* `unicode`)

*Return value: New reference.*
Encode a Unicode objects using Raw-Unicode-Escape and return the result as Python string object. Error handling is ``strict``. Return NULL if an exception was raised by the codec.

These are the Latin-1 codec APIs: Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

PyObject* `PyUnicode_DecodeLatin1`(const char* `s`, Py_ssize_t `size`, const char* `errors`)

*Return value: New reference.*
Create a Unicode object by decoding size bytes of the Latin-1 encoded string s. Return NULL if an exception was raised by the codec.

PyObject* `PyUnicode_EncodeLatin1`(Py_ssize_t `size`, const char* `errors`)

*Return value: New reference.*
Encode the Py_UNICODE buffer of the given size using Latin-1 and return a Python string object. Return NULL if an exception was raised by the codec.

PyObject* `PyUnicode_AsLatin1String`(PyObject* `unicode`)

*Return value: New reference.*
Encode a Unicode objects using Latin-1 and return the result as Python string object. Error handling is ``strict``. Return NULL if an exception was raised by the codec.

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

PyObject* `PyUnicode_DecodeASCII`(const char* `s`, Py_ssize_t `size`, const char* `errors`)

*Return value: New reference.*
Create a Unicode object by decoding size bytes of the ASCII encoded string s. Return NULL if an exception was raised by the codec.
PyObject* PyUnicode_EncodeASCII(const Py_UNICODE *s, Py_ssize_t size, const char *errors)

*Return value: New reference.*
Encode the Py_UNICODE buffer of the given size using ASCII and return a Python string object. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_AsASCIIString( PyObject *unicode)

*Return value: New reference.*
Encode a Unicode objects using ASCII and return the result as Python string object. Error handling is ``strict''. Return NULL if an exception was raised by the codec.

These are the mapping codec APIs:

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the encodings package). The codec uses mapping to encode and decode characters.

Decoding mappings must map single string characters to single Unicode characters, integers (which are then interpreted as Unicode ordinals) or None (meaning "undefined mapping" and causing an error).

Encoding mappings must map single Unicode characters to single string characters, integers (which are then interpreted as Latin-1 ordinals) or None (meaning "undefined mapping" and causing an error).

The mapping objects provided must only support the __getitem__ mapping interface.

If a character lookup fails with a LookupError, the character is copied as-is meaning that its ordinal value will be interpreted as Unicode or Latin-1 ordinal resp. Because of this, mappings only need to contain those mappings which map characters to different code points.

PyObject* PyUnicode_DecodeCharmap( PyObject *mapping, const char *errors)
Create a Unicode object by decoding size bytes of the encoded string s using the given mapping object. Return NULL if an exception was raised by the codec. If mapping is NULL latin-1 decoding will be done. Else it can be a dictionary mapping byte or a unicode string, which is treated as a lookup table. Byte values greater that the length of the string and U+FFFE "characters" are treated as "undefined mapping". Changed in version 2.4: Allowed unicode string as mapping argument.

PyObject* PyUnicode_EncodeCharmap( const Py_UNICODE* s, Py_ssize_t size, PyObject* mapping, const char* errors)

Encode the Py_UNICODE buffer of the given size using the given mapping object and return a Python string object. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_AsCharmapString( PyObject* unicode, PyObject* mapping)

Encode a Unicode objects using the given mapping object and return the result as Python string object. Error handling is ``strict''. Return NULL if an exception was raised by the codec.

The following codec API is special in that maps Unicode to Unicode.

PyObject* PyUnicode_TranslateCharmap( const Py_UNICODE* s, Py_ssize_t size, PyObject* table, const char* errors)

Translate a Py_UNICODE buffer of the given length by applying a character mapping table to it and return the resulting Unicode object. Return NULL when an exception was raised by the codec.

The mapping table must map Unicode ordinal integers to Unicode ordinal integers or None (causing deletion of the character).

Mapping tables need only provide the __getitem__() interface;
dictionaries and sequences work well. Unmapped character ordinals (ones which cause a LookupError) are left untouched and are copied as-is.

These are the MBCS codec APIs. They are currently only available on Windows and use the Win32 MBCS converters to implement the conversions. Note that MBCS (or DBCS) is a class of encodings, not just one. The target encoding is defined by the user settings on the machine running the codec.

*PyObject* **PyUnicode_DecodeMBCS**( const char *s, Py_ssize_t size, const char *errors)

*Return value:* New reference.
Create a Unicode object by decoding size bytes of the MBCS encoded string s. Return NULL if an exception was raised by the codec.

*PyObject* **PyUnicode_DecodeMBCSStateful**( const char *errors, int *consumed)

If consumed is NULL, behave like PyUnicode_DecodeMBCS(). If consumed is not NULL, PyUnicode_DecodeMBCSStateful() will not decode trailing lead byte and the number of bytes that have been decoded will be stored in consumed. New in version 2.5.

*PyObject* **PyUnicode_EncodeMBCS**( const Py_UNICODE *s, Py_ssize_t size, const char *errors)

*Return value:* New reference.
Encode the Py_UNICODE buffer of the given size using MBCS and return a Python string object. Return NULL if an exception was raised by the codec.

*PyObject* **PyUnicode_AsMBCSString**( PyObject *unicode)

*Return value:* New reference.
Encode a Unicode objects using MBCS and return the result as Python string object. Error handling is `strict`. Return NULL if an exception was raised by the codec.
Methods and Slot

Release 2.5, documentation updated on 19th September, 2006. 
See About this document... for information on suggesting changes.
7.3.2.2 Methods and Slot Functions

The following APIs are capable of handling Unicode objects and strings on input (we refer to them as strings in the descriptions) and return Unicode objects or integers as appropriate.

They all return `NULL` or `-1` if an exception occurs.

**PyObject* **`PyUnicode_Concat`**( PyObject *left, PyObject *right)**

*Return value: New reference.*
Concat two strings giving a new Unicode string.

**PyObject* **`PyUnicode_Split`**( PyObject *s, PyObject *sep, Py_ssize_t maxsplit)**

*Return value: New reference.*
Split a string giving a list of Unicode strings. If `sep` is `NULL`, splitting will be done at all whitespace substrings. Otherwise, splits occur at the given separator. At most `maxsplit` splits will be done. If negative, no limit is set. Separators are not included in the resulting list.

**PyObject* **`PyUnicode_Splitlines`**( PyObject *s, int keepend)**

*Return value: New reference.*
Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If `keepend` is 0, the Line break characters are not included in the resulting strings.

**PyObject* **`PyUnicode_Translate`**( PyObject *str, PyObject *table, const char *errors)**

*Return value: New reference.*
Translate a string by applying a character mapping table to it and return the resulting Unicode object.

The mapping table must map Unicode ordinal integers to Unicode ordinal integers or None (causing deletion of the character).
Mapping tables need only provide the `__getitem__()` interface; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a `LookupError`) are left untouched and are copied as-is.

`errors` has the usual meaning for codecs. It may be `NULL` which indicates to use the default error handling.

```python
PyObject* PyUnicode_Join( PyObject *separator, PyObject *seq)
```

*Return value: New reference.*
Join a sequence of strings using the given separator and return the resulting Unicode string.

```python
int PyUnicode_Tailmatch( PyObject *str, PyObject *substr, Py_ssize_t start, Py_ssize_t end, int direction)
```

*Return value: New reference.*
Return 1 if `substr` matches `str[start:end]` at the given tail end (`direction` == -1 means to do a prefix match, `direction` == 1 a suffix match), 0 otherwise. Return -1 if an error occurred.

```python
Py_ssize_t PyUnicode_Find( PyObject *str, PyObject *substr, Py_ssize_t start, Py_ssize_t end, int direction)
```

*Return value:* Index of the first ```match```. A value of -1 indicates that no match was found, and -2 indicates that an error occurred and an exception has been set.

```python
Py_ssize_t PyUnicode_Count( PyObject *str, PyObject *substr, Py_ssize_t start, Py_ssize_t end)
```

*Return value:* Number of non-overlapping occurrences of `substr` in ```str[start:end]```. Return -1 if an error occurred.

```python
PyObject* PyUnicode_Replace( PyObject *str, PyObject *substr, PyObject *replstr, Py_ssize_t maxcount)
```

*Return value: New reference.*
Replace at most `maxcount` occurrences of `substr` in `str` with `replstr` and return the resulting Unicode object. `maxcount` == -1 means replace all occurrences.
int PyUnicode_Compare( PyObject *left, PyObject *right)

Compare two strings and return -1, 0, 1 for less than, equal, and greater than, respectively.

int PyUnicode_RichCompare( PyObject *left, PyObject *right, int op)

Rich compare two unicode strings and return one of the following:

- NULL in case an exception was raised
- Py_True or Py_False for successful comparisons
- Py_NotImplemented in case the type combination is unknown

Note that Py_EQ and Py_NE comparisons can cause a UnicodeWarning in case the conversion of the arguments to Unicode fails with a UnicodeDecodeError.

Possible values for op are Py_GT, Py_GE, Py_EQ, Py_NE, Py_LT, and Py_LE.

PyObject* PyUnicode_Format( PyObject *format, PyObject *args)

Return a new string object from format and args; this is analogous to format % args. The args argument must be a tuple.

int PyUnicode_Contains( PyObject *container, PyObject *element)

Check whether element is contained in container and return true or false accordingly.

element has to coerce to a one element Unicode string. -1 is returned if there was an error.
Contents

- 1 Introduction
- 2 Directory Structure
- 3 Style Guide
- 4 LaTeX Primer
  - 4.1 Syntax
  - 4.2 Hierarchical Structure
  - 4.3 Common Environments
- 5 Document Classes
- 6 Special Markup Constructs
  - 6.1 Markup for the Preamble
  - 6.2 Meta-information Markup
  - 6.3 Information Units
  - 6.4 Showing Code Examples
  - 6.5 Inline Markup
  - 6.6 Miscellaneous Text Markup
  - 6.7 Module-specific Markup
  - 6.8 Library-level Markup
  - 6.9 Table Markup
  - 6.10 Reference List Markup
  - 6.11 Index-generating Markup
  - 6.12 Grammar Production Displays
  - 6.13 Graphical Interface Components
- 7 Processing Tools
  - 7.1 External Tools
  - 7.2 Internal Tools
  - 7.3 Working on Cygwin
- 8 Including Graphics
- 9 Future Directions
  - 9.1 Structured Documentation
  - 9.2 Discussion Forums
Release 2.5, documentation updated on 19th September, 2006. See About this document... for information on suggesting changes.
About this document ...

Documenting Python, 19th September, 2006, Release 2.5

This document was generated using the LaTeX2HTML translator.


The application of LaTeX2HTML to the Python documentation has been heavily tailored by Fred L. Drake, Jr. Original navigation icons were contributed by Christopher Petrilli.
Comments and Questions

General comments and questions regarding this document should be sent by email to docs@python.org. If you find specific errors in this document, either in the content or the presentation, please report the bug at the Python Bug Tracker at SourceForge. If you are able to provide suggested text, either to replace existing incorrect or unclear material, or additional text to supplement what's already available, we'd appreciate the contribution. There's no need to worry about text markup; our documentation team will gladly take care of that.

Questions regarding how to use the information in this document should be sent to the Python news group, comp.lang.python, or the Python mailing list (which is gated to the newsgroup and carries the same content).

For any of these channels, please be sure not to send HTML email. Thanks.
About this document ...

**Installing Python Modules**, 19th September, 2006, Release 2.5

This document was generated using the [LaTeX2HTML](http://www.latex2html.com) translator.


The application of [LaTeX2HTML](http://www.latex2html.com) to the Python documentation has been heavily tailored by Fred L. Drake, Jr. Original navigation icons were contributed by Christopher Petrilli.
Comments and Questions

General comments and questions regarding this document should be sent by email to docs@python.org. If you find specific errors in this document, either in the content or the presentation, please report the bug at the Python Bug Tracker at SourceForge. If you are able to provide suggested text, either to replace existing incorrect or unclear material, or additional text to supplement what's already available, we'd appreciate the contribution. There's no need to worry about text markup; our documentation team will gladly take care of that.

Questions regarding how to use the information in this document should be sent to the Python news group, comp.lang.python, or the Python mailing list (which is gated to the newsgroup and carries the same content).

For any of these channels, please be sure not to send HTML email. Thanks.
Module Index

This index only lists modules documented in this manual. The Global Module Index lists all modules that are documented in this set of manuals.

distutils.archive_util
distutils.bcppcompiler
distutils.ccompiler
distutils.cmd
distutils.command.bdist
distutils.command.bdist_dumb
distutils.command.bdist_packager
distutils.command.bdist_rpm
distutils.command.bdist_wininst
distutils.command.build
distutils.command.build_clib
distutils.command.build_ext
distutils.command.build_py
distutils.command.build_scripts
distutils.command.clean
distutils.command.config
distutils.command.install
distutils.command.install_data
distutils.command.install_headers
distutils.command.install_lib
distutils.command.install_scripts
distutils.command.register
distutils.command.sdist
About this document ...

**Distributing Python Modules**, 19th September, 2006, Release 2.5

This document was generated using the [LaTeX2HTML](http://www.latex2html.org) translator.


The application of [LaTeX2HTML](http://www.latex2html.org) to the Python documentation has been heavily tailored by Fred L. Drake, Jr. Original navigation icons were contributed by Christopher Petrilli.
Comments and Questions

General comments and questions regarding this document should be sent by email to docs@python.org. If you find specific errors in this document, either in the content or the presentation, please report the bug at the Python Bug Tracker at SourceForge. If you are able to provide suggested text, either to replace existing incorrect or unclear material, or additional text to supplement what's already available, we'd appreciate the contribution. There's no need to worry about text markup; our documentation team will gladly take care of that.

Questions regarding how to use the information in this document should be sent to the Python news group, comp.lang.python, or the Python mailing list (which is gated to the newsgroup and carries the same content).

For any of these channels, please be sure not to send HTML email. Thanks.