Python Library Reference

Release 2.3.2

Guido van Rossum
Fred L. Drake, Jr., editor

October 3, 2003

PythonLabs
Email: docs@python.org

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

CHAPTER

ONE

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, grouped 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!

CHAPTER

TWO

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.[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.

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.

__import__(*name*[, *globals*[, *locals*[, *fromlist*]]])

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 `rexec`. 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(), [])`; the statement '`from spam.ham import eggs`' results in '`__import__('spam.ham', globals(), locals(), ['eggs'])`'. 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
```

[1] Most descriptions sorely lack explanations of the exceptions that may be raised — this will be fixed in a future version of this manual.

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.

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.

basestring()
This abstract type is the superclass for `str` and `unicode`. It cannot be called or instantiated, but it can be used to test whether an object is an instance of `str` or `unicode`. `isinstance(obj, basestring)` is equivalent to `isinstance(obj, (str, unicode))`. New in version 2.3.

bool([*x*])
Convert a value to a Boolean, using the standard truth testing procedure. If `x` is false, this returns `False`; otherwise it returns `True`. `bool` is also a class, which is a subclass of `int`. Class `bool` cannot be subclassed further. Its only instances are `False` and `True`.

New in version 2.2.1.

Changed in version 2.3: If no argument is given, this function returns `False`.

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).

callable(*object*)
Return true if the *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 *object* will never succeed. Note that classes are callable (calling a class returns a new instance); class instances are callable if they have a `__call__()` method.

chr(*i*)
Return a string of one character whose ASCII code is the integer *i*. For example, `chr(97)` returns the string `'a'`. This is the inverse of `ord()`. The argument must be in the range [0..255], inclusive; `ValueError` will be raised if *i* is outside that range.

classmethod(*function*)
Return a class method for *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:

```
class C:
    def f(cls, arg1, arg2, ...): ...
    f = classmethod(f)
```

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. New in version 2.2.

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*.

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.

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 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 statemants 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}`:

- `dict({'one': 2, 'two': 3})`
- `dict({'one': 2, 'two': 3}.items())`
- `dict({'one': 2, 'two': 3}.iteritems())`
- `dict(zip(('one', 'two'), (2, 3)))`
- `dict([['two', 3], ['one', 2]])`
- `dict(one=2, two=3)`
- `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([*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:

```
>>> 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 two optional dictionaries. 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:

```
>>> 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.[2]

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 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*]])

Return a new file object (described earlier under Built-in Types). The first two arguments are the same as for `stdio`'s `fopen()`: *filename* is the file name to be opened, *mode* indicates how the file is to be opened: `'r'` for reading, `'w'` for writing (truncating an existing file), and `'a'` opens it for appending (which on *some* UNIX systems means that *all* writes append to the end of the file, regardless of the current seek position).

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 (else it is ignored). If the file cannot be opened, `IOError` is raised.

In addition to the standard `fopen()` values *mode* may be `'U'` or `'rU'`. If Python is built with universal newline support (the default) the file is opened as a text file, but lines may be terminated by any of `'\n'`, the Unix end-of-line convention, `'\r'`, the Macintosh convention or `'\r\n'`, the Windows convention. All of these external representations are seen as `'\n'` by the Python program. If Python is built without universal newline support *mode* `'U'` is the same as normal text mode. Note that file objects so opened also have an attribute called `newlines` which has a value of `None` (if no newlines have yet been seen), `'\n'`, `'\r'`, `'\r\n'`, or a tuple containing all the newline types seen.

If *mode* is omitted, it defaults to `'r'`. When opening a binary file, you should append `'b'` to the *mode* value for improved portability. (It's useful even on systems which don't treat binary and text files differently, where it serves as documentation.) The optional *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 *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.[3]

The `file()` constructor is new in Python 2.2. The previous spelling, `open()`, is retained for compatibility, and is an alias for `file()`.

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 (zero or empty) 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; this behaves identical to `string.atof(`*x*`)`. 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`.

[2] It is used relatively rarely so does not warrant being made into a statement.

[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.

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.

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 1 if the string is the name of one of the object's attributes, 0 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. Note: this always yields an unsigned literal. For example, on a 32-bit machine, `hex(-1)` yields `'0xffffffff'`. When evaluated on a machine with the same word size, this literal is evaluated as -1; at a different word size, it may turn up as a large positive number or raise an `OverflowError` exception.

id(*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 whose lifetimes are disjunct may have the same `id()` value. (Implementation note: this is the address of the object.)

input([*prompt*])
Equivalent to `eval(raw_input(`*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*[, *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`.

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.

`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; this behaves identical to `string.atol(`*x*`)`. 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...*])
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.

min(*s*[, *args...*])
With a single argument *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.

object()
Return a new featureless object. `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.

oct(*x*)
Convert an integer number (of any size) to an octal string. The result is a valid Python expression. Note: this always yields an unsigned literal. For example, on a 32-bit machine, `oct(-1)` yields `'037777777777'`. When evaluated on a machine with the same word size, this literal is evaluated as -1; at a different word size, it may turn up as a large positive number or raise an `OverflowError` exception.

open(*filename*[, *mode*[, *bufsize*]])
An alias for the `file()` function above.

ord(*c*)
Return the ASCII value of a string of one character or a Unicode character. E.g., `ord('a')` returns the integer `97`, `ord(u'\u2020')` returns `8224`. This is the inverse of `chr()` for strings and of `unichr()` for Unicode characters.

pow(*x*, *y*[, *z*])
Return *x* to the power *y*; if *z* is present, return *x* to the power *y*, modulo *z* (computed more efficiently than `pow(x, y) % z`). The arguments must have numeric types. 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.) If the second argument is negative, the third argument must be omitted. If *z* is present, *x* and *y* must be of integer types, and *y* must be non-negative. (This restriction was added in Python 2.2. In Python 2.1 and before, floating 3-argument `pow()` returned platform-dependent results depending on floating-point rounding accidents.)

property([*fget*[, *fset*[, *fdel*[, *doc*]]]])
Return a property attribute for new-style classes (classes that derive from `object`).

fget is a function for getting an attribute value, likewise *fset* is a function for setting, and *fdel* a function for del'ing, an attribute. Typical use is to define a managed attribute x:

```
class C(object):
    def getx(self): return self.__x
    def setx(self, value): self.__x = value
    def delx(self): del self.__x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

New in version 2.2.

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* `+ 1, `*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 largest *start* `+` *i* `*` *step* greater than *stop*. *step* must not be zero (or else `ValueError` is raised). Example:

```
>>> 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)
[]
>>> range(1, 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:

```
>>> 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)`. 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*)

Re-parse and re-initialize an already 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).

There are a number of 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.

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 `from` statement, another is to use `import` and qualified names (*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.

repr(*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 `eval()`.

round(*x*[, *n*])
Return the floating point value *x* rounded to *n* digits after the decimal point. If *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 *n*; if two multiples are equally close, rounding is done away from 0 (so. for example, `round(0.5)` is `1.0` and `round(-0.5)` is `-1.0`).

setattr(*object, name, value*)
This is the counterpart of `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]`'.

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:

```
class C:
    def f(arg1, arg2, ...): ...
    f = staticmethod(f)
```

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. New in version 2.2.

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:

```
class C(B):
    def meth(self, arg):
        super(C, self).meth(arg)
```

New in version 2.2.

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, `''`.

`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 standard module `types` defines names for all built-in types that don't already have built-in names. For instance:

```
>>> import types
>>> x = 'abc'
>>> if type(x) is str: print "It's a string"
...
It's a string
>>> def f(): pass
...
>>> if type(f) is types.FunctionType: print "It's a function"
...
It's a function
```

The `isinstance()` built-in function is recommended for testing the type of an object.

`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 argument must be in the range [0..65535], inclusive. `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.[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

[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.

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`).

`zip`(*seq1*, ...)
This function returns a list of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences. At least one sequence is required, otherwise a `TypeError` is raised. The returned list is truncated in length to the length of the shortest argument sequence. When there are multiple argument sequences 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. New in version 2.0.

2.2 Built-in Types

The following sections describe the standard types that are built into the interpreter. Historically, 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. With the 2.2 release this situation has started to change, although the intended unification of user-defined and built-in types is as yet far from complete.

The principal built-in types are numerics, sequences, mappings, files classes, instances and exceptions.

Some operations are supported by several object types; in particular, all objects can be compared, tested for truth value, and converted to a string (with the `'...'` notation). The latter conversion is implicitly used when an object is written by the `print` statement. (Information on `print` statement and other language statements can be found in the *Python Reference Manual* and the *Python Tutorial*.)

2.2.1 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`.[5]

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.)

2.2.2 Boolean Operations

These are the Boolean operations, ordered by ascending priority:

Operation	Result	Notes
x `or` *y*	if *x* is false, then *y*, else *x*	(1)
x `and` *y*	if *x* is false, then *x*, else *y*	(1)
`not` *x*	if *x* is false, then `True`, else `False`	(2)

[5] Additional information on these special methods may be found in the *Python Reference Manual*.

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.

2.2.3 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* `<=` *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).

This table summarizes the comparison operations:

Operation	Meaning	Notes
`<`	strictly less than	
`<=`	less than or equal	
`>`	strictly greater than	
`>=`	greater than or equal	
`==`	equal	
`!=`	not equal	(1)
`<>`	not equal	(1)
`is`	object identity	
`is not`	negated object identity	

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).

2.2.4 Numeric Types

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. 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.[6] 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):

Operation	Result	Notes
x `+` *y*	sum of *x* and *y*	
x `-` *y*	difference of *x* and *y*	
x `*` *y*	product of *x* and *y*	
x `/` *y*	quotient of *x* and *y*	(1)
x `%` *y*	remainder of *x* `/` *y*	(4)
`-`*x*	*x* negated	
`+`*x*	*x* unchanged	
`abs(`*x*`)`	absolute value or magnitude of *x*	
`int(`*x*`)`	*x* converted to integer	(2)
`long(`*x*`)`	*x* converted to long integer	(2)
`float(`*x*`)`	*x* converted to floating point	
`complex(`*re*`,`*im*`)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.	
c`.conjugate()`	conjugate of the complex number *c*	
`divmod(`*x*`,` *y*`)`	the pair `(`*x* `/` *y*`,` *x* `%` *y*`)`	(3)(4)
`pow(`*x*`,` *y*`)`	*x* to the power *y*	
x `**` *y*	*x* to the power *y*	

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 `floor()` and `ceil()` in the `math` module for well-defined conversions.

(3) See section 2.1, "Built-in Functions," for a full description.

(4) Complex floor division operator, modulo operator, and `divmod()`.

Deprecated since release 2.3. Instead convert to float using `abs()` if appropriate.

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):

[6] As a consequence, the list `[1, 2]` is considered equal to `[1.0, 2.0]`, and similarly for tuples.

Operation	Result	Notes
`x | y`	bitwise *or* of *x* and *y*	
`x ^ y`	bitwise *exclusive or* of *x* and *y*	
`x & y`	bitwise *and* of *x* and *y*	
`x << n`	*x* shifted left by *n* bits	(1), (2)
`x >> n`	*x* shifted right by *n* bits	(1), (3)
`~x`	the bits of *x* inverted	

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 `pow(2, n)` without overflow check.

(3) A right shift by *n* bits is equivalent to division by `pow(2, n)` without overflow check.

2.2.5 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:

`__iter__()`
: 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*:

`__iter__()`
: 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.

`next()`
: Return the next item from the container. If there are no further items, raise the `StopIteration` exception. This method corresponds to the `tp_iternext` 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.

2.2.6 Sequence Types

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 preceeding '`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 builtin 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.[7]

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:

Operation	Result	Notes
x `in` *s*	`1` if an item of *s* is equal to *x*, else `0`	(1)
x `not in` *s*	`0` if an item of *s* is equal to *x*, else `1`	(1)
s `+` *t*	the concatenation of *s* and *t*	
s `*` *n*, *n* `*` *s*	*n* shallow copies of *s* concatenated	(2)
s`[`*i*`]`	*i*'th item of *s*, origin 0	(3)
s`[`*i*`:`*j*`]`	slice of *s* from *i* to *j*	(3), (4)
s`[`*i*`:`*j*`:`*k*`]`	slice of *s* from *i* to *j* with step *k*	(3), (5)
`len(`*s*`)`	length of *s*	
`min(`*s*`)`	smallest item of *s*	
`max(`*s*`)`	largest item of *s*	

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 `lists` is a list containing three copies of the list `[[]]` (a one-element list containing an empty list), but the contained list is shared by each copy. You can create a list of different lists this way:

[7] They must have since the parser can't tell the type of the operands.

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

(3) If *i* or *j* is negative, the index is relative to the end of the string: `len(`*s*`) + `*i* or `len(`*s*`) + `*j* is substituted. But note that `-0` is still `0`.

(4) The slice of *s* from *i* to *j* is defined as the sequence of items with index *k* such that *i* `<=` *k* `<` *j*. If *i* or *j* is greater than `len(`*s*`)`, use `len(`*s*`)`. If *i* is omitted, use `0`. If *j* is omitted, use `len(`*s*`)`. If *i* is greater than or equal to *j*, the slice is empty.

(5) The slice of *s* from *i* to *j* with step *k* is defined as the sequence of items with index *x* `=` *i* `+` *n*`*`*k* such that `0` `<=` *n* `<` `abs(i-j)`. If *i* or *j* is greater than `len(`*s*`)`, use `len(`*s*`)`. If *i* or *j* are omitted then they become "end" values (which end depends on the sign of *k*). Note, *k* cannot be zero.

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.

`center``(`*width*`)`
Return centered in a string of length *width*. Padding is done using spaces.

`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 `ValueError`. Other possible values are `'ignore'` and `replace'`. New in version 2.2.

`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 `ValueError`. Other possible values are `'ignore'` and `'replace'`. New in version 2.0.

`endswith``(`*suffix*`[`, *start*`[`, *end*`]])`
Return `True` if the string ends with the specified *suffix*, otherwise return `False`. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

`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.

isalpha()
Return true if all characters in the string are alphabetic and there is at least one character, false otherwise.

isdigit()
Return true if there are only digit characters, false otherwise.

islower()
Return true if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

isspace()
Return true if there are only whitespace characters in the string and the string is not empty, false otherwise.

istitle()
Return true if the string is a titlecased string: uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise.

isupper()
Return true if all cased characters in the string are uppercase and there is at least one cased character, false otherwise.

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*)
Return the string left justified in a string of length *width*. Padding is done using spaces. The original string is returned if *width* is less than `len(s)`.

lower()
Return a copy of the string converted to lowercase.

lstrip([*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 2.2.2: Support for the *chars* argument.

replace(*old, new*[, *maxsplit*])
Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *maxsplit* is given, only the first *maxsplit* 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*)
Return the string right justified in a string of length *width*. Padding is done using spaces. The original string is returned if *width* is less than `len(s)`.

rstrip([*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 2.2.2: 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. If *sep* is not specified or `None`, any whitespace string is a separator.

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`. With optional *start*, test string beginning at that position. With optional *end*, stop comparing string at that position.

strip([*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 2.2.2: Support for the *chars* argument.

swapcase()
Return a copy of the string with uppercase characters converted to lowercase and vice versa.

title()
Return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase.

translate(*table*[, *deletechars*])
Return a copy of the string where all characters occurring in the optional argument *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 `translate()` method does not accept the optional *deletechars* argument. Instead, it returns a copy of the *s* 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 `None`. Unmapped characters are left untouched. Characters mapped to `None` are deleted. Note, a more flexible approach is to create a custom character mapping codec using the `codecs` module (see `encodings.cp1251` for an example).

upper()
Return a copy of the string converted to uppercase.

zfill(*width*)
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.

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. [8] 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.

[8] To format only a tuple you should therefore provide a singleton tuple whose only element is the tuple to be formatted.

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.

6. Length modifier (optional).

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:

```
>>> 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:

Flag	Meaning
'#'	The value conversion will use the "alternate form" (where defined below).
'0'	The conversion will be zero padded for numeric values.
'-'	The converted value is left adjusted (overrides the '0' conversion if both are given).
' '	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
'+'	A sign character ('+' or '-') will precede the conversion (overrides a "space" flag).

The length modifier may be h, l, and L may be present, but are ignored as they are not necessary for Python.

The conversion types are:

Conversion	Meaning	Notes
'd'	Signed integer decimal.	
'i'	Signed integer decimal.	
'o'	Unsigned octal.	(1)
'u'	Unsigned decimal.	
'x'	Unsigned hexidecimal (lowercase).	(2)
'X'	Unsigned hexidecimal (uppercase).	(2)
'e'	Floating point exponential format (lowercase).	
'E'	Floating point exponential format (uppercase).	
'f'	Floating point decimal format.	
'F'	Floating point decimal format.	
'g'	Same as 'e' if exponent is greater than -4 or less than precision, 'f' otherwise.	
'G'	Same as 'E' if exponent is greater than -4 or less than precision, 'F' otherwise.	
'c'	Single character (accepts integer or single character string).	
'r'	String (converts any python object using repr()).	(3)
's'	String (converts any python object using str()).	(4)
'%'	No argument is converted, results in a '%' character in the result.	

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 `%r` conversion was added in Python 2.0.

(4) If the object or format provided is a `unicode` string, the resulting string will also be `unicode`.

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.[9] All other errors raise exceptions.

Additional string operations are defined in standard modules `string` and `re`.

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.

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):

Operation	Result	Notes
s`[`*i*`] = `*x*	item *i* of *s* is replaced by *x*	
s`[`*i*`:`*j*`] = `*t*	slice of *s* from *i* to *j* is replaced by *t*	
`del` *s*`[`*i*`:`*j*`]`	same as *s*`[`*i*`:`*j*`] = []`	
s`[`*i*`:`*j*`:`*k*`] = `*t*	the elements of *s*`[`*i*`:`*j*`:`*k*`]` are replaced by those of *t*	(1)
`del` *s*`[`*i*`:`*j*`:`*k*`]`	removes the elements of *s*`[`*i*`:`*j*`:`*k*`]` from the list	
s`.append(`*x*`)`	same as *s*`[len(`*s*`):len(`*s*`)] = [`*x*`]`	(2)
s`.extend(`*x*`)`	same as *s*`[len(`*s*`):len(`*s*`)] = `*x*	(3)
s`.count(`*x*`)`	return number of *i*'s for which *s*`[`*i*`] == `*x*	
s`.index(`*x*`[, `*i*`[, `*j*`]])`	return smallest *k* such that *s*`[`*k*`] == `*x* and *i* `<= k <` *j*	(4)
s`.insert(`*i*`, `*x*`)`	same as *s*`[`*i*`:`*i*`] = [`*x*`]`	(5)
s`.pop([`*i*`])`	same as *x* `=` *s*`[`*i*`];` `del` *s*`[`*i*`];` `return` *x*	(6)
s`.remove(`*x*`)`	same as `del` *s*`[`*s*`.index(`*x*`)]`	(4)
s`.reverse()`	reverses the items of *s* in place	(7)
s`.sort([`*cmpfunc=None*`])`	sort the items of *s* in place	(7), (8), (9), (10)

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) Raises an exception when *x* is not a list object. The `extend()` method is experimental and not supported by mutable sequence types other than lists.

(4) Raises `ValueError` when *x* is not found in *s*. When a negative index is passed as the second or third parameter to the `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, `index()` didn't have arguments for specifying start and stop positions.

[9] 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.

(5) When a negative index is passed as the first parameter to the `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 `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 `sort()` and `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 `sort()` method takes an optional argument specifying a 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. Note that this slows the sorting process down considerably; for example to sort a list in reverse order it is much faster to call `sort()` followed by `reverse()` than to use `sort()` with a comparison function that reverses the ordering of the elements. Passing `None` as the comparison function is semantically equivalent to calling `sort()` with no comparison function. Changed in version 2.3: Support for `None` as an equivalent to omitting *cmpfunc* was added.

As an example of using the *cmpfunc* argument to the `sort()` method, consider sorting a list of sequences by the second element of that list:

```
def mycmp(a, b):
    return cmp(a[1], b[1])

mylist.sort(mycmp)
```

A more time-efficient approach for reasonably-sized data structures can often be used:

```
tmplist = [(x[1], x) for x in mylist]
tmplist.sort()
mylist = [x for (key, x) in tmplist]
```

(9) Whether the `sort()` method is stable is not defined by the language (a sort is stable if it guarantees not to change the relative order of elements that compare equal). In the C implementation of Python, sorts were stable only by accident through Python 2.2. The C implementation of Python 2.3 introduced a stable `sort()` method, but code that intends to be portable across implementations and versions must not rely on stability.

(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 makes the list appear empty for the duration, and raises `ValueError` if it can detect that the list has been mutated during a sort.

2.2.7 Mapping Types

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):

Operation	Result	Notes
`len(`*a*`)`	the number of items in *a*	
a`[`*k*`]`	the item of *a* with key *k*	(1)
a`[`*k*`] = `*v*	set *a*`[`*k*`]` to *v*	
`del` *a*`[`*k*`]`	remove *a*`[`*k*`]` from *a*	(1)
a`.clear()`	remove all items from `a`	
a`.copy()`	a (shallow) copy of `a`	
a`.has_key(`*k*`)`	`True` if *a* has a key *k*, else `False`	
k `in` *a*	Equivalent to *a*.has_key(*k*)	(2)
k `not in` *a*	Equivalent to `not` *a*.has_key(*k*)	(2)
a`.items()`	a copy of *a*'s list of (*key*, *value*) pairs	(3)
a`.keys()`	a copy of *a*'s list of keys	(3)
a`.update(`*b*`)`	`for` *k* `in` *b*`.keys(): a[`*k*`] = b[`*k*`]`	
a`.fromkeys(`*seq*`[, `*value*`])`	Creates a new dictionary with keys from *seq* and values set to *value*	(7)
a`.values()`	a copy of *a*'s list of values	(3)
a`.get(`*k*`[, `*x*`])`	*a*`[`*k*`]` if *k* `in` *a*, else *x*	(4)
a`.setdefault(`*k*`[, `*x*`])`	*a*`[`*k*`]` if *k* `in` *a*, else *x* (also setting it)	(5)
a`.pop(`*k*`[, `*x*`])`	*a*`[`*k*`]` if *k* `in` *a*, else *x* (and remove k)	(8)
a`.popitem()`	remove and return an arbitrary (*key*, *value*) pair	(6)
a`.iteritems()`	return an iterator over (*key*, *value*) pairs	(2), (3)
a`.iterkeys()`	return an iterator over the mapping's keys	(2), (3)
a`.itervalues()`	return an iterator over the mapping's values	(2), (3)

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 random order. 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())`' provides the same value for `pairs`. Another way to create the same list is '`pairs = [(v, k) for (k, v) in `*a*`.iteritems()]`'.

(4) 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, `None` is returned.

(5) `setdefault()` is like `get()`, except that if *k* is missing, *x* is both returned and inserted into the dictionary as the value of *k*.

(6) `popitem()` is useful to destructively iterate over a dictionary, as often used in set algorithms.

(7) `fromkeys()` is a class method that returns a new dictionary. *value* defaults to `None`. New in version 2.3.

(8) `pop()` raises a `KeyError` when no default value is given and the key is not found. New in version 2.3.

2.2.8 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."[10] 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.

[10] `file()` is new in Python 2.2. The older built-in `open()` is an alias for `file()`.

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.

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[11] (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+'`),

[11] 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!).

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+'`).

`tell``()`
Return the file's current position, like `stdio`'s `ftell()`.

`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 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 (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.

2.2.9 Other Built-in Types

The interpreter supports several other kinds of objects. Most of these support only one or two operations.

Modules

The only special operation on a module is attribute access: *m*`.`*name*, where *m* is a module and *name* accesses a name defined in *m*'s symbol table. Module attributes can be assigned to. (Note that the `import` statement is not, strictly speaking, an operation on a module object; `import` *foo* does not require a module object named *foo* to exist, rather it requires an (external) *definition* for a module named *foo* somewhere.)

A special member of every module is `__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 `__dict__` attribute is not possible (you can write *m*`.__dict__['a'] = 1`, which defines *m*`.a` to be `1`, but you can't write *m*`.__dict__ = {}`).

Modules built into the interpreter are written like this: `<module 'sys' (built-in)>`. If loaded from a file, they are written as `<module 'os' from '/usr/local/lib/python2.3/os.pyc'>`.

Classes and Class Instances

See chapters 3 and 7 of the *Python Reference Manual* for these.

Functions

Function objects are created by function definitions. The only operation on a function object is to call it: *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.

The implementation adds two special read-only attributes: *f*`.func_code` is a function's *code object* (see below) and *f*`.func_globals` is the dictionary used as the function's global namespace (this is the same as *m*`.__dict__` where *m* is the module in which the function *f* was defined).

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.*

Functions have another special attribute *f*`.__dict__` (a.k.a. *f*`.func_dict`) which contains the namespace used to support function attributes. `__dict__` and `func_dict` can be accessed directly or set to a dictionary object. A function's dictionary cannot be deleted.

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:

```
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.

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.

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'>`.

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`.

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`.

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.

Internal Objects

See the *Python Reference Manual* for this information. It describes stack frame objects, traceback objects, and slice objects.

2.2.10 Special Attributes

The implementation adds a few special read-only attributes to several object types, where they are relevant:

`__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.

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 `PendingDeprecationWarning`. 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 `Exception`, the associated value is present as the exception instance's `args` attribute, and possibly on other attributes as well.

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` base class. 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 `Exception`
The root class for exceptions. All built-in exceptions are derived from this class. All user-defined exceptions should also be derived from this class, but this is not (yet) enforced. The `str()` function, when applied to an instance of this class (or most derived classes) returns the string value of the argument or arguments, or an empty string if no arguments were given to the constructor. When used as a sequence, this accesses the arguments given to the constructor (handy for backward compatibility with old code). The arguments are also available on the instance's `args` attribute, as a tuple.

exception `StandardError`
The base class for all built-in exceptions except `StopIteration` and `SystemExit`. `StandardError` itself is derived from the root class `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, `TypeError` is raised.)

exception `EOFError`
Raised when one of the built-in functions (`input()` or `raw_input()`) hits an end-of-file condition (EOF) without reading any data. (N.B.: the `read()` and `readline()` methods of file objects return an empty string when they hit EOF.)

exception `FloatingPointError`
Raised when a floating point operation fails. This exception is always defined, but can only be raised when Python is configured with the **--with-fpectl** option, or the `WANT_SIGFPE_HANDLER` symbol is defined in the 'pyconfig.h' file.

exception `IOError`
Raised when an I/O operation (such as a `print` statement, the built-in `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 `EnvironmentError`. See the discussion above for more information on exception instance attributes.

exception `ImportError`
Raised when an `import` statement fails to find the module definition or when a `from ... import` fails to find a name that is to be imported.

exception `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, `TypeError` is raised.)

exception `KeyError`
Raised when a mapping (dictionary) key is not found in the set of existing keys.

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.

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 atttributes `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 `Exception` 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()`).

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 `errno` and `strerror` values are created from the return values of the `GetLastError()` and `FormatMessage()` functions from the Windows Platform API. This is a subclass of `OSError`. New in version 2.0.

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.

The class hierarchy for built-in exceptions is:

```
Exception
 +-- SystemExit
 +-- StopIteration
 +-- StandardError
 |    +-- KeyboardInterrupt
 |    +-- ImportError
 |    +-- EnvironmentError
 |    |    +-- IOError
 |    |    +-- OSError
 |    |         +-- WindowsError
 |    +-- EOFError
 |    +-- RuntimeError
 |    |    +-- NotImplementedError
 |    +-- NameError
 |    |    +-- UnboundLocalError
 |    +-- AttributeError
 |    +-- SyntaxError
 |    |    +-- IndentationError
 |    |         +-- TabError
 |    +-- TypeError
 |    +-- AssertionError
 |    +-- LookupError
 |    |    +-- IndexError
 |    |    +-- KeyError
 |    +-- ArithmeticError
 |    |    +-- OverflowError
 |    |    +-- ZeroDivisionError
 |    |    +-- FloatingPointError
 |    +-- ValueError
 |    |    +-- UnicodeError
 |    |         +-- UnicodeEncodeError
 |    |         +-- UnicodeDecodeError
 |    |         +-- UnicodeTranslateError
 |    +-- ReferenceError
 |    +-- SystemError
 |    +-- MemoryError
 +---Warning
+-- UserWarning
+-- DeprecationWarning
+-- PendingDeprecationWarning
+-- SyntaxWarning
+-- OverflowWarning
+-- RuntimeWarning
+-- FutureWarning
```

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.

CHAPTER

THREE

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.
gc	Interface to the cycle-detecting garbage collector.
weakref	Support for weak references and weak dictionaries.
fpectl	Provide control for floating point exception handling.
atexit	Register and execute cleanup functions.
types	Names for built-in types.
UserDict	Class wrapper for dictionary objects.
UserList	Class wrapper for list objects.
UserString	Class wrapper for string objects.
operator	All Python's standard operators as built-in functions.
inspect	Extract information and source code from live objects.
traceback	Print or retrieve a stack traceback.
linecache	This module provides random access to individual lines from text files.
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.
copy	Shallow and deep copy operations.
marshal	Convert Python objects to streams of bytes and back (with different constraints).
warnings	Issue warning messages and control their disposition.
imp	Access the implementation of the `import` statement.
pkgutil	Utilities to support extension of packages.
code	Base classes for interactive Python interpreters.
codeop	Compile (possibly incomplete) Python code.
pprint	Data pretty printer.
repr	Alternate `repr()` implementation with size limits.
new	Interface to the creation of runtime implementation objects.
site	A standard way to reference site-specific modules.
user	A standard way to reference user-specific modules.
__builtin__	The set of built-in functions.
__main__	The environment where the top-level script is run.
__future__	Future statement definitions

3.1 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-signigicant byte first) platforms, and `'little'` on little-endian (least-significant byte first) platforms. New in version 2.0.

`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.

`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__._`.

`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 `exctype, 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 occured 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/python`*version*`/config'`, and shared library modules are installed in `exec_prefix + '/lib/python`*version*`/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.

getcheckinterval()
Return the interpreter's "check interval"; see `setcheckinterval()`. New in version 2.3.

getdefaultencoding()
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.

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, `ValueError` is raised. The default for *depth* is zero, returning the frame at the top of the call stack.

This function should be used for internal and specialized purposes only.

getwindowsversion()
Return a tuple containing five components, describing the Windows version currently running. The elements are *major*, *minor*, *build*, *platform*, and *text*. *text* contains a string while all other values are integers.

platform may be one of the following values:

0 (`VER_PLATFORM_WIN32s`)Win32s on Windows 3.1.

1 (`VER_PLATFORM_WIN32_WINDOWS`)Windows 95/98/ME

2 (`VER_PLATFORM_WIN32_NT`)Windows NT/2000/XP

3 (`VER_PLATFORM_WIN32_CE`)Windows CE.

This function wraps the Win32 `GetVersionEx()` function; see the Microsoft Documentation for more information about these fields.

Availability: Windows. New in version 2.3.

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 ‘`hexversion`’ since it only really looks meaningful when viewed as the result of passing it to the built-in `hex()` function. The `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 9, "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, thread-safety 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/python`*version*`'` while the platform independent header files (all except 'pyconfig.h') are stored in `prefix + '/include/python`*version*`'`, 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` module implementation and, where needed, by `sitecustomize`. Once used by the `site` module, it is removed from the `sys` 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(0)`. To share symbols across extension modules, call as `sys.setdlopenflags(dl.RTLD_NOW | dl.RTLD_GLOBAL)`. Symbolic names for the flag modules can be either found in the `dl` module, or in the `DLFCN` module. If `DLFCN` is not available, it can be generated from '/usr/include/dlfcn.h' using the **h2py** 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 10 for more information on the Python profiler. The system's profile function is called similarly to the system's trace function (see `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 9.2, "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.

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__
__stdout__
__stderr__
These objects contain the original values of `stdin`, `stderr` and `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 `1000`. When set to `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 *'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:

```
>>> 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: *major*, *minor*, *micro*, *releaselevel*, and *serial*. All values except *releaselevel* are integers; the release level is `'alpha'`, `'beta'`, `'candidate'`, or `'final'`. The `version_info` value corresponding to the Python version 2.0 is `(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 `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` (section 3.28):
This describes how to use .pth files to extend `sys.path`.

3.2 gc — Garbage Collector interface

The `gc` module is only available if the interpreter was built with the optional cyclic garbage detector (enabled by default). If this was not enabled, an `ImportError` is raised by attempts to import this module.

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)`.

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()
Run a full collection. All generations are examined and the number of unreachable objects found is returned.

set_debug(*flags*)
Set the garbage collection debugging flags. Debugging information will be written to `sys.stderr`. See below for a list of debugging flags which can be combined using bit operations to control debugging.

get_debug()
Return the debugging flags currently set.

get_objects()
Returns a list of all objects tracked by the collector, excluding the list returned. New in version 2.2.

set_threshold(*threshold0*[, *threshold1*[, *threshold2*]])
Set the garbage collection thresholds (the collection frequency). Setting *threshold0* to zero disables collection.

The GC classifies objects into three generations depending on how many collection sweeps they have sur-

vived. 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 *threshold0*, collection starts. Initially only generation `0` is examined. If generation `0` has been examined more than *threshold1* times since generation `1` has been examined, then generation `1` is examined as well. Similarly, *threshold2* controls the number of collections of generation `1` before collecting generation `2`.

`get_threshold``()`
Return the current collection thresholds as a tuple of `(`*threshold0*`, `*threshold1*`, `*threshold2*`)`.

`get_referrers``(`**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()`.

New in version 2.2.

`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):

`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.[1] 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.

[1] Prior to Python 2.2, the list contained all instance objects in unreachable cycles, not only those with `__del__()` methods.

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).

3.3 weakref — Weak references

New in version 2.1.

The weakref module allows the Python programmer to create *weak references* to objects.

In the discussion which follows, the term *referent* means the object which is referred to by a weak reference.

XXX — need to say more here!

Not all objects can be weakly referenced; those objects which can include class instances, functions written in Python (but not in C), and methods (both bound and unbound). Extension types can easily be made to support weak references; see section 3.3.3, "Weak References in Extension Types," for more information.

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, it 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.

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.

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.

`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.

3.3.1 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:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
1
```

If the referent no longer exists, calling the reference object returns `None`:

```
>>> 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:

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print "Object has been allocated; 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.

3.3.2 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.

```
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]
```

3.3.3 Weak References in Extension Types

One of the goals of the 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. Also, it needs to add `Py_TPFLAGS_HAVE_WEAKREFS` to the tp_flags slot. 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 | Py_TPFLAGS_HAVE_WEAKREFS   /* 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 */

    self->in_weakreflist = NULL;

    return (PyObject *) self;
}
```

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`:

```
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. */
}
```

3.4 `fpectl` — Floating point exception control

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

```
>>> 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:

`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.

3.4.1 Example

The following example demonstrates how to start up and test operation of the `fpectl` module.

```
>>> 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
```

3.4.2 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'.

3.5 `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`.

`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.

See Also:

Module `readline` (section 7.20):
Useful example of `atexit` to read and write `readline` history files.

3.5.1 `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.

```
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:

```
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')
```

3.6 `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 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:

```
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:

```
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.

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.

3.7 `UserDict` — Class wrapper for dictionary 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 `dict` type.

This module defines a class that acts as a wrapper around dictionary objects. It is a useful base class for your own dictionary-like classes, which can inherit from them and override existing methods or add new ones. In this way one can add new behaviors to dictionaries.

The module also defines a mixin 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).

The `UserDict` module defines the `UserDict` class and `DictMixin`:

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.

In addition to supporting the methods and operations of mappings (see section 2.2.7), `UserDict` instances provide the following attribute:

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, progessively 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()`.

3.8 `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:

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 2.2.6), `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.

3.9 `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 for `MutableString`.

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 2.2.6, "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.

3.10 `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*)
ne(*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 informations 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:

not_(*o*)
__not__(*o*)

Return the outcome of `not` *o*. (Note that there is no `__not__()` method for object instances; only the interpreter core defines this operation. The result is affected by the `__nonzero__()` and `__len__()` 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.

is_not(*a*, *b*)

Return *a* `is not` *b*. Tests object identity.

The mathematical and bitwise operations are the most numerous:

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*)
__div__(*a*, *b*)

Return *a* `/` *b* when `__future__.division` is not in effect. This is also known as "classic" division.

floordiv(*a*, *b*)
__floordiv__(*a*, *b*)
Return *a* `//` *b*. New in version 2.2.

inv(*o*)
invert(*o*)
__inv__(*o*)
__invert__(*o*)
Return the bitwise inverse of the number *o*. This is equivalent to `~`*o*. The names `invert()` and `__invert__()` were added in Python 2.0.

lshift(*a*, *b*)
__lshift__(*a*, *b*)
Return *a* shifted left by *b*.

mod(*a*, *b*)
__mod__(*a*, *b*)
Return *a* `%` *b*.

mul(*a*, *b*)
__mul__(*a*, *b*)
Return *a* `*` *b*, for *a* and *b* numbers.

neg(*o*)
__neg__(*o*)
Return *o* negated.

or_(*a*, *b*)
__or__(*a*, *b*)
Return the bitwise or of *a* and *b*.

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 division. New in version 2.2.

xor(*a*, *b*)
__xor__(*a*, *b*)
Return the bitwise exclusive or of *a* and *b*.

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*.

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:

```
>>> class C:
...     pass
...
>>> import operator
>>> o = C()
>>> operator.isMappingType(o)
1
```

`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. **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, and for all instance objects. **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. **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 `256` to their character equivalents.

```
>>> import operator
>>> d = {}
>>> keys = range(256)
>>> vals = map(chr, keys)
>>> map(operator.setitem, [d]*len(keys), keys, vals)
```

3.10.1 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.

Operation	Syntax	Function
Addition	*a* + *b*	add(*a*, *b*)
Concatenation	*seq1* + *seq2*	concat(*seq1*, *seq2*)
Containment Test	*o* in *seq*	contains(*seq*, *o*)
Division	*a* / *b*	div(*a*, *b*) # without __future__.division
Division	*a* / *b*	truediv(*a*, *b*) # with __future__.division
Division	*a* // *b*	floordiv(*a*, *b*)
Bitwise And	*a* & *b*	and_(*a*, *b*)
Bitwise Exclusive Or	*a* ^ *b*	xor(*a*, *b*)
Bitwise Inversion	~ *a*	invert(*a*)
Bitwise Or	*a* | *b*	or_(*a*, *b*)
Exponentiation	*a* ** *b*	pow(*a*, *b*)
Identity	*a* is *b*	is_(*a*, *b*)
Identity	*a* is not *b*	is_not(*a*, *b*)
Indexed Assignment	*o*[*k*] = *v*	setitem(*o*, *k*, *v*)
Indexed Deletion	del *o*[*k*]	delitem(*o*, *k*)
Indexing	*o*[*k*]	getitem(*o*, *k*)
Left Shift	*a* << *b*	lshift(*a*, *b*)
Modulo	*a* % *b*	mod(*a*, *b*)
Multiplication	*a* * *b*	mul(*a*, *b*)
Negation (Arithmetic)	- *a*	neg(*a*)
Negation (Logical)	not *a*	not_(*a*)
Right Shift	*a* >> *b*	rshift(*a*, *b*)
Sequence Repitition	*seq* * *i*	repeat(*seq*, *i*)
Slice Assignment	*seq*[*i*:*j*] = *values*	setslice(*seq*, *i*, *j*, *values*)
Slice Deletion	del *seq*[*i*:*j*]	delslice(*seq*, *i*, *j*)
Slicing	*seq*[*i*:*j*]	getslice(*seq*, *i*, *j*)
String Formatting	*s* % *o*	mod(*s*, *o*)
Subtraction	*a* - *b*	sub(*a*, *b*)
Truth Test	*o*	truth(*o*)
Ordering	*a* < *b*	lt(*a*, *b*)
Ordering	*a* <= *b*	le(*a*, *b*)
Equality	*a* == *b*	eq(*a*, *b*)
Difference	*a* != *b*	ne(*a*, *b*)
Ordering	*a* >= *b*	ge(*a*, *b*)
Ordering	*a* > *b*	gt(*a*, *b*)

3.11 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.

3.11.1 Types and members

The getmembers() function retrieves the members of an object such as a class or module. The nine 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:

Type	Attribute	Description	Notes
module	__doc__	documentation string	
	__file__	filename (missing for built-in modules)	
class	__doc__	documentation string	
	__module__	name of module in which this class was defined	
method	__doc__	documentation string	
	__name__	name with which this method was defined	
	im_class	class object that asked for this method	(1)
	im_func	function object containing implementation of method	
	im_self	instance to which this method is bound, or `None`	
function	__doc__	documentation string	
	__name__	name with which this function was defined	
	func_code	code object containing compiled function bytecode	
	func_defaults	tuple of any default values for arguments	
	func_doc	(same as __doc__)	
	func_globals	global namespace in which this function was defined	
	func_name	(same as __name__)	
traceback	tb_frame	frame object at this level	
	tb_lasti	index of last attempted instruction in bytecode	
	tb_lineno	current line number in Python source code	
	tb_next	next inner traceback object (called by this level)	
frame	f_back	next outer frame object (this frame's caller)	
	f_builtins	built-in namespace seen by this frame	
	f_code	code object being executed in this frame	
	f_exc_traceback	traceback if raised in this frame, or `None`	
	f_exc_type	exception type if raised in this frame, or `None`	
	f_exc_value	exception value if raised in this frame, or `None`	
	f_globals	global namespace seen by this frame	
	f_lasti	index of last attempted instruction in bytecode	
	f_lineno	current line number in Python source code	
	f_locals	local namespace seen by this frame	
	f_restricted	0 or 1 if frame is in restricted execution mode	
	f_trace	tracing function for this frame, or `None`	
code	co_argcount	number of arguments (not including * or ** args)	
	co_code	string of raw compiled bytecode	
	co_consts	tuple of constants used in the bytecode	
	co_filename	name of file in which this code object was created	
	co_firstlineno	number of first line in Python source code	
	co_flags	bitmap: 1=optimized | 2=newlocals | 4=*arg | 8=**arg	
	co_lnotab	encoded mapping of line numbers to bytecode indices	
	co_name	name with which this code object was defined	
	co_names	tuple of names of local variables	
	co_nlocals	number of local variables	
	co_stacksize	virtual machine stack space required	
	co_varnames	tuple of names of arguments and local variables	
builtin	__doc__	documentation string	
	__name__	original name of this function or method	
	__self__	instance to which a method is bound, or `None`	

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 algortihm 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.

isclass(*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.

iscode(*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) and getsets and members (defined in C). 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.

3.11.2 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.

3.11.3 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; 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, argformat, varargsformat, varkwformat, defaultformat*])
Format a pretty argument spec from the four values returned by `getargspec()`. The other four arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

formatargvalues(*args*[, *varargs, varkw, locals, argformat, varargsformat, varkwformat, valueformat*])
Format a pretty argument spec from the four values returned by `getargvalues()`. The other four 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.

3.11.4 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. The optional *context* argument specifies the number of lines of context to return, which are centered around the current line.

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.

`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. The optional second argument specifies the number of lines of context to return, which are centered around the current line.

`getouterframes`(*frame*[, *context*])
Get a list of frame records for a frame and all higher (calling) frames.

`getinnerframes`(*traceback*[, *context*])
Get a list of frame records for a traceback's frame and all lower frames.

`currentframe`()
Return the frame object for the caller's stack frame.

`stack`([*context*])
Return a list of frame records for the stack above the caller's frame.

`trace`([*context*])
Return a list of frame records for the stack below the current exception.

Stackframes stored directly or indirectly in local variables can easily cause reference cycles. 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:

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

3.12 `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.)

`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.

3.12.1 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.

```
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)
```

3.13 `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*)
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`.

`clearcache`()
Clear the cache. Use this function if you no longer need lines from files previously read using `getline()`.

`checkcache`()
Check the cache for validity. Use this function if files in the cache may have changed on disk, and you require the updated version.

Example:

```
>>> import linecache
>>> linecache.getline('/etc/passwd', 4)
'sys:x:3:3:sys:/dev:/bin/sh\n'
```

3.14 `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,"[2] or "flattening", however,

[2] Don't confuse this with the `marshal` module

to avoid confusion, the terms used here are “pickling” and “unpickling”.

This documentation describes both the `pickle` module and the `cPickle` module.

3.14.1 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.

3.14.2 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: The *bin* parameter is deprecated and only provided for backwards compatibility. You should use the *protocol* parameter instead.

A binary format, which is slightly more efficient, can be chosen by specifying a true value for the *bin* argument to the `Pickler` constructor or the `dump()` and `dumps()` functions. A *protocol* version ¿= 1 implies use of a binary format.

3.14.3 Usage

To serialize an object hierarchy, you first create a pickler, then you call the pickler's `dump()` method. To deserialize 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.

The `pickle` module provides the following functions to make this process more convenient:

`dump`(*object*, *file*[, *protocol*[, *bin*]])
Write a pickled representation of *object* to the open file object *file*. This is equivalent to `Pickler(`*file*`,` *protocol*`,` *bin*`).dump(`*object*`)`.

If the *protocol* parameter is ommitted, 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. The *bin* parameter is deprecated and only provided for backwards compatibility. You should use the *protocol* parameter instead.

If the optional *bin* argument is true, the binary pickle format is used; otherwise the (less efficient) text pickle format is used (for backwards compatibility, this is the default).

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.

`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`(*object*[, *protocol*[, *bin*]])
Return the pickled representation of the object as a string, instead of writing it to a file.

If the *protocol* parameter is ommitted, 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. The *bin* parameter is deprecated and only provided for backwards compatibility. You should use the *protocol* parameter instead.

If the optional *bin* argument is true, the binary pickle format is used; otherwise the (less efficient) text pickle format is used (this is the default).

`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,[3] `Pickler` and `Unpickler`:

class `Pickler`(*file*[, *protocol*[, *bin*]])
This takes a file-like object to which it will write a pickle data stream.

If the *protocol* parameter is ommitted, protocol 0 is used. If *protocol* is specified as a negative value, the highest protocol version will be used.

Changed in version 2.3: The *bin* parameter is deprecated and only provided for backwards compatibility. You should use the *protocol* parameter instead.

Optional *bin* if true, tells the pickler to use the more efficient binary pickle format, otherwise the ASCII format is used (this is the default).

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:

`dump`(*object*)
Write a pickled representation of *object* to the open file object given in the constructor. Either the binary or ASCII format will be used, depending on the value of the *bin* flag passed to the constructor.

`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.

Note: Prior to Python 2.3, `clear_memo()` was only available on the picklers created by `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:

```
mypickler.memo.clear()
```

Code that does not need to support older versions of Python should simply use `clear_memo()`.

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[4].

`Unpickler` objects are defined as:

[3] 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 3.14.6 for more details.

[4] *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.

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 3.14.5 below for more details.

Note: the `noload()` method is currently only available on `Unpickler` objects created with the `cPickle` module. `pickle` module `Unpickler`s do not have the `noload()` method.

3.14.4 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, 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 3.14.5 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.

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[5].

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:

```
class Foo:
    attr = 'a class attr'

picklestring = pickle.dumps(Foo)
```

[5]The exception raised will likely be an `ImportError` or an `AttributeError` but it could be something else.

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.

3.14.5 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 3.14.6 for more details.

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, a class can define a method `__getinitargs__()`, which should return a *tuple* containing the arguments to be passed to the class constructor (i.e. `__init__()`). The `__getinitargs__()` method is called at pickle time; the tuple it returns is incorporated in the pickle for the instance.

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[6]. 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.[7]

Warning: For new-style classes, if `__getstate__()` returns a false value, the `__setstate__()` method will not be called.

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. When a tuple is returned, it must be of length two or three, with the following semantics:

- A callable object, which in the unpickling environment 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, or `None`. **Deprecated since release 2.3.** Use the tuple of arguments instead
- Optionally, the object's state, which will be passed to the object's `__setstate__()` method as described in section 3.14.5. 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__`.

[6] These methods can also be used to implement copying class instances.

[7] This protocol is also used by the shallow and deep copying operations defined in the `copy` module.

Upon unpickling, the callable will be called (provided that it meets the above criteria), passing in the tuple of arguments; it should return the unpickled object.

If the second item was `None`, then instead of calling the callable directly, its `__basicnew__()` method is called without arguments. It should also return the unpickled object.

Deprecated since release 2.3. Use the tuple of arguments instead

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.

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[8].

To define external persistent id resolution, you need to set the `persistent_id` attribute of the pickler object and the `persistent_load` attribute of the unpickler object.

To pickle objects that have an external persistent id, the pickler must have a custom `persistent_id()` method that takes an object as an argument and returns either `None` or the persistent id for that object. When `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 `persistent_load()` function that takes a persistent id string and returns the referenced object.

Here's a silly example that *might* shed more light:

[8] 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.

```
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[9]. Setting `persistent_load` to a list is usually used in conjunction with the `noload()` method on the Unpickler.

3.14.6 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

[9] We'll leave you with the image of Guido and Jim sitting around sniffing pickles in their living rooms.

depending on whether you're using `pickle` or `cPickle`.[10].

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 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.

3.14.7 Example

Here's a simple example of 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 it
        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:                 # until line count is restored
            fh.readline()
            count = count - 1
        self.__dict__.update(dict)   # update attributes
        self.fh = fh                 # save the file object
```

[10] 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`.

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` (section 3.16):
Pickle interface constructor registration for extension types.

Module `shelve` (section 3.17):
Indexed databases of objects; uses `pickle`.

Module `copy` (section 3.18):
Shallow and deep object copying.

Module `marshal` (section 3.19):
High-performance serialization of built-in types.

3.15 `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[11].

There are additional minor differences in API between `cPickle` and `pickle`, however for most applications, they are interchangable. More documentation is provided in the `pickle` module documentation, which includes a list of the documented differences.

3.16 `copy_reg` — Register `pickle` support functions

[11] 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.

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*.

3.17 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*[,*binary=None*]]]])
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* pararameter 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. The *binary* parameter is deprecated and provided for backwards compatibility only.

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.

3.17.1 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.

class `Shelf`(*dict*[, *protocol=None*[, *writeback=False*[, *binary=None*]]])
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. The *binary* parameter is deprecated and provided for backwards compatibility only.

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.

class `BsdDbShelf`(*dict*[, *protocol=None*[, *writeback=False*[, *binary=None*]]])
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*, *writeback*, and *binary* parameters have the same interpretation as for the `Shelf` class.

class `DbfilenameShelf`(*filename*[, *flag='c'*[, *protocol=None*[, *writeback=False*[, *binary=None*]]]])
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*, *writeback*, and *binary* parameters have the same interpretation as for the `Shelf` class.

3.17.2 Example

To summarize the interface (`key` is a string, `data` is an arbitrary object):

```
import shelve

d = shelve.open(filename) # open -- file may get suffix added by low-level
                          # 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 no
                # 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
list = 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` (section 7.10):
Generic interface to `dbm`-style databases.

Module `bsddb` (section 7.13):
BSD `db` database interface.

Module `dbhash` (section 7.11):
Thin layer around the `bsddb` which provides an `open` function like the other database modules.

Module `dbm` (section 8.6):
Standard UNIX database interface.

Module `dumbdbm` (section 7.14):
Portable implementation of the `dbm` interface.

Module `gdbm` (section 8.7):
GNU database interface, based on the `dbm` interface.

Module `pickle` (section 3.14):
Object serialization used by `shelve`.

Module `cPickle` (section 3.15):
High-performance version of `pickle`.

3.18 copy — Shallow and deep copy operations

This module provides generic (shallow and deep) copying operations.

Interface summary:

```
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 version does not copy types like module, class, function, method, stack trace, stack frame, file, socket, window, array, or any similar types.

Classes can use the same interfaces to control copying that they use to control pickling: they can define methods called `__getinitargs__()`, `__getstate__()` and `__setstate__()`. 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` (section 3.14):
Discussion of the special methods used to support object state retrieval and restoration.

3.19 `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).[12]

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 `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:

`dump`(*value, file*)
Write the value on the open file. The value must be a supported type. The file must be an open file object such as `sys.stdout` or returned by `open()` or `posix.popen()`. It must be opened in binary mode (`'wb'` or `'w+b'`).

If the value has (or contains an object that has) an unsupported type, a `ValueError` exception is raised — but garbage data will also be written to the file. The object will not be properly read back by `load()`.

`load`(*file*)
Read one value from the open file and return it. If no valid value is read, raise `EOFError`, `ValueError` or `TypeError`. The file must be an open file object opened in binary mode (`'rb'` or `'r+b'`).

Warning: If an object containing an unsupported type was marshalled with `dump()`, `load()` will substitute `None` for the unmarshallable type.

[12] 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.

dumps(*value*)
Return the string that would be written to a file by dump(*value*, *file*). The value must be a supported type. Raise a ValueError exception if value has (or contains an object that has) an unsupported type.

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.

3.20 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 overidden; the default implementation of this function formats the message by calling formatwarning(), which is also available for use by custom implementations.

3.20.1 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:

Class	Description
Warning	This is the base class of all warning category classes. It is a subclass of Exception.
UserWarning	The default category for warn().
DeprecationWarning	Base category for warnings about deprecated features.
SyntaxWarning	Base category for warnings about dubious syntactic features.
RuntimeWarning	Base category for warnings about dubious runtime features.
FutureWarning	Base category for warnings about constructs that will change semantically in the future.

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.

3.20.2 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:

- *action* is one of the following strings:

Value	Disposition
`"error"`	turn matching warnings into exceptions
`"ignore"`	never print matching warnings
`"always"`	always print matching warnings
`"default"`	print the first occurrence of matching warnings for each location where the warning is issued
`"module"`	print the first occurrence of matching warnings for each module where the warning is issued
`"once"`	print only the first occurrence of matching warnings, regardless of location

- *message* is a string containing a regular expression that the warning message must match (the match is compiled to always be case-insensitive)
- *category* is a class (a subclass of `Warning`) of which the warning category must be a subclass in order to match
- *module* is a string containing a regular expression that the module name must match (the match is compiled to be case-sensitive)
- *lineno* is an integer that the line number where the warning occurred must match, or `0` to match all line numbers

Since the `Warning` class is derived from the built-in `Exception` class, to turn a warning into an error we simply raise `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`).

3.20.3 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:

```
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*]])

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.

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 front of the warnings filter. Entries inserted later override entries inserted earlier, if both match a particular warning. Omitted arguments default to a value that matches everything.

resetwarnings()
Reset the warnings filter. This discards the effect of all previous calls to `filterwarnings()`, including that of the **-W** command line options.

3.21 `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 indicate 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.

PY_RESOURCE
The module was found as a Macintosh resource. This value can only be returned on a 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/freeze/' 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 '`init`*name*`()`' 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.

3.21.1 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.)

```
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.

3.22 `pkgutil` — Package extension utility

New in version 2.3.

Warning: This is an experimental module. It may be withdrawn or completely changed up to an including the release of Python 2.3 beta 1.

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':

```
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 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).

3.23 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 cotains an invalid literal.

3.23.1 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 '<input>', 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.

3.23.2 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.

3.24 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:

compile_command(*source*[, *filename*[, *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 indentical 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

```
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:

```
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.

3.25 `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 builtin 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.

The `pprint` module defines one class:

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.

```
>>> 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']
>>>
>>> import parser
>>> tup = parser.ast2tuple(
...     parser.suite(open('pprint.py').read()))[1][1][1]
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
(266, (267, (307, (287, (288, (...))))))
```

The `PrettyPrinter` class supports several derivative functions:

`pformat`(*object*)
Return the formatted representation of *object* as a string. The default parameters for formatting are used.

`pprint`(*object*[, *stream*])
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. The default parameters for formatting are used.

```
>>> 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',
 '/usr/local/lib/python1.5/sunos5',
 '/usr/local/lib/python1.5/sharedmodules',
 '/usr/local/lib/python1.5/tkinter']
```

`isreadable`(*object*)
Determine if the formatted representation of *object* is "readable," or can be used to reconstruct the value using `eval()`. This always returns false for recursive objects.

```
>>> pprint.isreadable(stuff)
0
```

isrecursive(*object*)
Determine if *object* requires a recursive representation.

One more support function is also defined:

saferepr(*object*)
Return a string representation of *object*, protected against recursive data structures. If the representation of *object* exposes a recursive entry, the recursive reference will be represented as '<Recursion on *typename* with id=*number*>'. The representation is not otherwise formatted.

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=682968>, '', '/usr/local/lib/python1.5', '/usr/loca
l/lib/python1.5/test', '/usr/local/lib/python1.5/sunos5', '/usr/local/lib/python
1.5/sharedmodules', '/usr/local/lib/python1.5/tkinter']"
```

3.25.1 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 additionaly entries for containers to this dictionary. The fourth 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.

3.26 `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:

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.

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.

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.

3.26.1 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
Limits on the number of entries represented for the named object type. The default for maxdict is 4, for the others, 6.

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)*'.

3.26.2 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:

```
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>'
```

3.27 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.

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*]])
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.

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*)
This function returns a new module object with name *name*. *name* must be a string.

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*.

3.28 site — Site-specific configuration hook

This module is automatically imported during initialization.

In earlier versions of Python (up to and including 1.5a3), scripts or modules that needed to use site-specific modules would place '`import site`' somewhere near the top of their code. This is no longer necessary.

This 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 (on Macintosh or Windows) or it uses first 'lib/python2.3/site-packages' and then 'lib/site-python' (on UNIX). 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'; 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 2.3.2 library is then installed in '/usr/local/lib/python2.3' (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/python2.3/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:

```
# 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.

3.29 `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

```
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:

```
import user
try:
    verbose = user.spam_verbose  # user's verbosity preference
except AttributeError:
    verbose = 0                  # default verbosity
```

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` (section 3.28):
Site-wide customization mechanism.

3.30 `__builtin__` — Built-in functions

This module provides direct access to all 'built-in' identifiers of Python; e.g. `__builtin__.open` is the full name for the built-in function `open()`. See section 2.1, "Built-in Functions."

3.31 `__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:

```
if __name__ == "__main__":
    main()
```

3.32 `__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:

```
FeatureName = "_Feature(" OptionalRelease "," MandatoryRelease ","
                       CompilerFlag ")"
```

where, normally, OptionalRelease is less then MandatoryRelease, and both are 5-tuples of the same form as `sys.version_info`:

```
(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"; string
 PY_RELEASE_SERIAL # the 3; an int
)
```

OptionalRelease records the first release in which the feature was accepted.

In the case of MandatoryReleases that have 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 `_Future` instances.

No feature description will ever be deleted from `__future__`.

CHAPTER

FOUR

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.
fpformat	General floating point formatting functions.
StringIO	Read and write strings as if they were files.
cStringIO	Faster version of `StringIO`, but not subclassable.
textwrap	Text wrapping and filling
encodings.idna	Internationalized Domain Names implementation
unicodedata	Access the Unicode Database.
stringprep	String preparation, as per RFC 3453

Information on the methods of string objects can be found in section 2.2.6, "String Methods."

4.1 `string` — Common string operations

This module defines some constants useful for checking character classes and some useful string functions. See the module `re` for string functions based on regular expressions.

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.

Many of the functions provided by this module are also defined as methods of string and Unicode objects; see "String Methods" (section 2.2.6) for more information on those. 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 *base*. The string must consist of one or more digits, optionally preceded by a sign ('+' or '-'). The *base* argument has the same meaning as for `atoi()`. A trailing 'l' or 'L' is not allowed, except if the base is 0. Note that when invoked without *base* or with *base* set to 10, this behaves identical to the built-in function `long()` when passed a string.

`capitalize`(*word*)
Return a copy of *word* with only its first character capitalized.

`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.

`expandtabs`(*s*[, *tabsize*])
Expand tabs in a string, i.e. replace 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, sub*[, *start*[,*end*]])
Return the lowest index in *s* where the substring *sub* is found such that *sub* is wholly contained in *s*[*start*:*end*]. Return -1 on failure. Defaults for *start* and *end* and interpretation of negative values is the same as for slices.

rfind(*s, sub*[, *start*[, *end*]])
Like find() but find the highest index.

index(*s, sub*[, *start*[, *end*]])
Like find() but raise ValueError when the substring is not found.

rindex(*s, sub*[, *start*[, *end*]])
Like rfind() but raise ValueError when the substring is not found.

count(*s, sub*[, *start*[, *end*]])
Return the number of (non-overlapping) occurrences of substring *sub* in string *s*[*start*:*end*]. Defaults for *start* and *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.

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().

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).

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 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*[, *maxsplit*])
Return a copy of string *str* with all occurrences of substring *old* replaced by *new*. If the optional argument *maxsplit* is given, the first *maxsplit* occurrences are replaced.

4.2 `re` — Regular expression operations

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.

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\nfoo2\n' 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.

`(?:...)` 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*`>...)` 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*.

`(?#...)` A comment; the contents of the parentheses are simply ignored.

`(?=...)` 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'`.

`(?!...)` 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:

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

This example looks for a word following a hyphen:

```
>>> 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.

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 '`$`'.

`\`*number* Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, `(.+) \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 *number* is 0, or *number* is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value *number*. Inside the '`[`' and '`]`' of a character class, all numeric escapes are treated as characters.

`\A` Matches only at the start of the string.

`\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 `\b` is defined as the boundary between `\w` and `\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, `\b` represents the backspace character, for compatibility with Python's string literals.

`\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 `\b`, so is also subject to the settings of `LOCALE` and `UNICODE`.

`\d` Matches any decimal digit; this is equivalent to the set `[0-9]`.

`\D` Matches any non-digit character; this is equivalent to the set `[^0-9]`.

`\s` Matches any whitespace character; this is equivalent to the set `[ \t\n\r\f\v]`.

`\S` Matches any non-whitespace character; this is equivalent to the set `[^ \t\n\r\f\v]`.

`\w` 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 ⌜`[0-9_]`⌟ plus whatever characters are defined as alphanumeric for the current locale. If `UNICODE` is set, this will match the characters ⌜`[0-9_]`⌟ plus whatever is classified as alphanumeric in the Unicode character properties database.

`\W` 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 ⌜`[0-9_]`⌟, and not defined as alphanumeric for the current locale. If `UNICODE` is set, this will match anything other than ⌜`[0-9_]`⌟ and characters marked as alphanumeric in the Unicode character properties database.

`\Z` 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
\r      \t      \v      \x
\\
```

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.

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 \n
```

4.2.3 Module Contents

The module defines the following functions and constants, and an exception:

`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
LOCALE
Make `\w`, `\W`, `\b`, and `\B` 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`, and `\B` 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.)

```
>>> re.split('\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split('\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
```

This function combines and extends the functionality of the old `regsub.split()` and `regsub.splitx()`.

`findall`(*pattern, string*)
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.

`finditer`(*pattern, string*)
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.

`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:

```
>>> 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:

```
>>> 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 BBBB")`' 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<0>`' substitutes in the entire substring matched by the RE.

`subn`(*pattern, repl, string*[, *count*])
Perform the same operation as `sub()`, but return a tuple (*new_string*, *number_of_subs_made*).

`escape`(*string*)
Return *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:

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)`.

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.

split(*string*[, *maxsplit* `= 0`])
Identical to the `split()` function, using the compiled pattern.

findall(*string*)
Identical to the `findall()` function, using the compiled pattern.

finditer(*string*)
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:

```
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`(\[*default*\])
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

```
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 applyied to the string `'ab'`, while the expression `(a)(b)` will have `lastindex == 2`, if applyied 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` in-

stance.

`string`
The string passed to `match()` or `search()`.

4.2.6 Examples

Simulating `scanf()`

Python does not currently have an equivalent to `scanf()`. Regular expressions are generally more powerful, though also more verbose, than `scanf()` format strings. The table below offers some more-or-less equivalent mappings between `scanf()` format tokens and regular expressions.

`scanf()` Token	Regular Expression
%c	`.`
%5c	`.{5}`
%d	`[-+]?\d+`
%e, %E, %f, %g	`[-+]?(\d+(\.\d*)?|\d*\.\d+)([eE][-+]?\d+)?`
%i	`[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)`
%o	`0[0-7]*`
%s	`\S+`
%u	`\d+`
%x, %X	`0[xX][\dA-Fa-f]+`

To extract the filename and numbers from a string like

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

you would use a `scanf()` format like

```
%s - %d errors, %d warnings
```

The equivalent regular expression would be

```
(\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 RuntimeError exception with the message `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/python2.3/sre.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:

Format	C Type	Python	Notes
'x'	pad byte	no value	
'c'	`char`	string of length 1	
'b'	`signed char`	integer	
'B'	`unsigned char`	integer	
'h'	`short`	integer	
'H'	`unsigned short`	integer	
'i'	`int`	integer	
'I'	`unsigned int`	long	
'l'	`long`	integer	
'L'	`unsigned long`	long	
'q'	`long long`	long	(1)
'Q'	`unsigned long long`	long	(1)
'f'	`float`	float	
'd'	`double`	float	
's'	`char[]`	string	
'p'	`char[]`	string	
'P'	`void *`	integer	

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 characters).

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:

Character	Byte order	Size and alignment
‘@’	native	native
‘=’	native	standard
‘<’	little-endian	standard
‘>’	big-endian	standard
‘!’	network (= big-endian)	standard

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):

```
>>> 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 `'llh0l'` 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` (section 5.12):
Packed binary storage of homogeneous data.

Module `xdrlib` (section 12.17):
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 Obershelp 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 Obershelp 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-Obershelp 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:

Code	Meaning
`'- '`	line unique to sequence 1
`'+ '`	line unique to sequence 2
`'  '`	line common to both sequences
`'?  '`	line not present in either input sequence

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.

`context_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.

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.

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['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.

```
>>> 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:

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(1),
...              'ore\ntree\nemu\n'.splitlines(1))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print ''.join(restore(diff, 1)),
one
two
three
>>> print ''.join(restore(diff, 2)),
ore
tree
emu
```

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 a 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.

4.4.1 SequenceMatcher Objects

The `SequenceMatcher` class has this constructor:

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 *b* is equivalent to passing `lambda x: 0`; in other words, no elements are ignored. For example, pass:

```
lambda x: x in " \t"
```

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:

`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.

`set_seq1`(*a*)
Set the first sequence to be compared. The second sequence to be compared is not changed.

`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*.

```
>>> 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:

```
>>> 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`.

```
>>> 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 preceeding tuple, and, likewise, *j1* equal to the previous *j2*.

The *tag* values are strings, with these meanings:

Value	Meaning
`'replace'`	*a*`[`*i1*`:`*i2*`]` should be replaced by *b*`[`*j1*`:`*j2*`]`.
`'delete'`	*a*`[`*i1*`:`*i2*`]` should be deleted. Note that *j1* `==` *j2* in this case.
`'insert'`	*b*`[`*j1*`:`*j2*`]` should be inserted at *a*`[`*i1*`:`*i1*`]`. Note that *i1* `==` *i2* in this case.
`'equal'`	*a*`[`*i1*`:`*i2*`]` `==` *b*`[`*j1*`:`*j2*`]` (the sub-sequences are equal).

For example:

```
>>> 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]))
 delete a[0:1] (q) b[0:0] ()
  equal a[1:3] (ab) b[0:2] (ab)
replace a[3:4] (x) b[2:3] (y)
  equal a[4:6] (cd) b[3:5] (cd)
 insert a[6:6] () b[5:6] (f)
```

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*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()`:

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

4.4.2 SequenceMatcher Examples

This example compares two strings, considering blanks to be "junk:"

```
>>> 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:

```
>>> print round(s.ratio(), 3)
0.866
```

If you're only interested in where the sequences match, `get_matching_blocks()` is handy:

```
>>> 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()`:

```
>>> 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.

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:

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.

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.

`Differ` objects are used (deltas generated) via a single method:

`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.

4.4.4 Differ Example

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 `readlines()` method of file-like objects):

```
>>> 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'
>>> text2 = '''  1. Beautiful is better than ugly.
...   3.   Simple is better than complex.
...   4. Complicated is better than complex.
...   5. Flat is better than nested.
... '''.splitlines(1)
```

Next we instantiate a Differ object:

```
>>> d = Differ()
```

Note that when instantiating a `Differ` object we may pass functions to filter out line and character "junk." See the `Differ()` constructor for details.

Finally, we compare the two:

```
>>> result = list(d.compare(text1, text2))
```

`result` is a list of strings, so let's pretty-print it:

```
>>> 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',
 '?     ++                                \n',
 '-   4. Complex is better than complicated.\n',
 '?            ^                     ---- ^  \n',
 '+   4. Complicated is better than complex.\n',
 '?           ++++ ^                      ^  \n',
 '+   5. Flat is better than nested.\n']
```

As a single multi-line string it looks like this:

```
>>> 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 `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:

```
>>> import fpformat
>>> fpformat.fix(1.23, 1)
'1.2'
```

4.6 `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 2.2.8).

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.

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.

4.7 `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.

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.

4.8 `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.

`wrap`(*text*[, *width*[, ...]])
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`.

`fill`(*text*[, *width*[, ...]])
Wraps the single paragraph in *text*, and returns a single string containing the wrapped paragraph. `fill()` is shorthand for

```
"\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.

`dedent`(*text*)
Remove any whitespace than can be uniformly removed from the left of every line in *text*.

This is typically used to make triple-quoted strings line up with the left edge of screen/whatever, while still presenting it in the source code in indented form.

For example:

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
      world
    '''
    print repr(s)          # prints '    hello\n      world\n    '
    print repr(dedent(s))  # prints 'hello\n  world\n'
```

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 `width`, `TextWrapper` guarantees that no output line will be longer than `width` characters.

`expand_tabs`
(default: `True`) If true, then all tab characters in *text* will be expanded to spaces using the `expand_tabs()` method of *text*.

`replace_whitespace`
(default: `True`) If true, each whitespace character (as defined by `string.whitespace`) remaining after tab expansion will be replaced by a single space. **Note:** If `expand_tabs` is false and `replace_whitespace` is true, each tab character will be replaced by a single space, which is *not* the same as tab expansion.

`initial_indent`
(default: `''`) String that will be prepended to the first line of wrapped output. Counts towards the length of the first line.

`subsequent_indent`
(default: `''`) String that will be prepended to all lines of wrapped output except the first. Counts towards the length of each line except the first.

`fix_sentence_endings`
(default: `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 '.', '!', or '?', possibly followed by one of '"' or ''', followed by a space. One problem with this is algorithm is that it is unable to detect the difference between "Dr." in

```
[...] Dr. Frankenstein's monster [...]
```

and "Spot." in

```
[...] See Spot. See Spot run [...]
```

`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.

4.9 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 tuple of functions `(`*encoder*`,` *decoder*`,` *stream_reader*`,` *stream_writer*`)` taking 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.

stream_reader and *stream_writer*: These have to be factory functions providing the following interface:

`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 a codec tuple in the Python codec registry and returns the function tuple 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 codecs tuple is found, a `LookupError` is raised. Otherwise, the codecs tuple 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:

`getencoder`(*encoding*)
Lookup up the codec for the given encoding and return its encoder function.

Raises a `LookupError` in case the encoding cannot be found.

`getdecoder`(*encoding*)
Lookup up the codec for the given encoding and return its decoder function.

Raises a `LookupError` in case the encoding cannot be found.

`getreader`(*encoding*)
Lookup 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.

`getwriter`(*encoding*)
Lookup 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.

`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 register 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.

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.

See Also:

http://sourceforge.net/projects/python-codecs/

A SourceForge project working on additional support for Asian codecs for use with Python. They are in the early stages of development at the time of this writing — look in their FTP area for downloadable files.

4.9.1 Codec Base Classes

The `codecs` defines a set of base classes which define the interface and can also be used to easily write you 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:

Value	Meaning
`'strict'`	Raise `UnicodeError` (or a subclass); this is the default.
`'ignore'`	Ignore the character and continue with the next.
`'replace'`	Replace with a suitable replacement character; Python will use the official U+FFFD REPLACEMI
`'xmlcharrefreplace'`	Replace with the appropriate XML character reference (only for encoding).
`'backslashreplace'`	Replace with backslashed escape sequences (only for encoding).

The set of allowed values can be extended via `register_error`.

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 `StreamWriter` and `StreamReader` classes provide generic working interfaces which can be used to implement new encodings submodules very easily. See `encodings.utf_8` for an example on how this is done.

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 to the Python codec registry.

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()`.

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 attribute from the underlying stream.

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 to the Python codec registry.

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()`.

read([*size*])
Decodes data from the stream and returns the resulting object.

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.

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.

readline(*[size]*)
Read one line from the input stream and return the decoded data.

Unlike the `readlines()` method, this method inherits the line breaking knowledge from the underlying stream's `readline()` method – there is currently no support for line breaking using the codec decoder due to lack of line buffering. Sublcasses should however, if possible, try to implement this method using their own knowledge of line breaking.

size, if given, is passed as size argument to the stream's `readline()` method.

readlines(*[sizehint]*)
Read all lines available on the input stream and return them as list of lines.

Line breaks are implemented using the codec's decoder method and are included in the list entries.

sizehint, if given, is passed as *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 attribute 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.

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.

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 attribute from the underlying stream.

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 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 attribute from the underlying stream.

4.9.2 Standard Encodings

Python comes with a number of codecs builtin, 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

Codec	Aliases	Languages
ascii	646, us-ascii	English
cp037	IBM037, IBM039	English
cp424	EBCDIC-CP-HE, IBM424	Hebrew
cp437	437, IBM437	English
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	Western Europe
cp737		Greek
cp775	IBM775	Baltic languages
cp850	850, IBM850	Western Europe
cp852	852, IBM852	Central and Eastern Europe
cp855	855, IBM855	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
cp856		Hebrew
cp857	857, IBM857	Turkish
cp860	860, IBM860	Portuguese
cp861	861, CP-IS, IBM861	Icelandic
cp862	862, IBM862	Hebrew
cp863	863, IBM863	Canadian
cp864	IBM864	Arabic
cp865	865, IBM865	Danish, Norwegian
cp869	869, CP-GR, IBM869	Greek
cp874		Thai
cp875		Greek
cp1006		Urdu
cp1026	ibm1026	Turkish
cp1140	ibm1140	Western Europe
cp1250	windows-1250	Central and Eastern Europe
cp1251	windows-1251	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
cp1252	windows-1252	Western Europe
cp1253	windows-1253	Greek
cp1254	windows-1254	Turkish
cp1255	windows-1255	Hebrew
cp1256	windows1256	Arabic
cp1257	windows-1257	Baltic languages
cp1258	windows-1258	Vietnamese
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	West Europe
iso8859_2	iso-8859-2, latin2, L2	Central and Eastern Europe
iso8859_3	iso-8859-3, latin3, L3	Esperanto, Maltese
iso8859_4	iso-8859-4, latin4, L4	Baltic languagues
iso8859_5	iso-8859-5, cyrillic	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
iso8859_6	iso-8859-6, arabic	Arabic
iso8859_7	iso-8859-7, greek, greek8	Greek
iso8859_8	iso-8859-8, hebrew	Hebrew
iso8859_9	iso-8859-9, latin5, L5	Turkish
iso8859_10	iso-8859-10, latin6, L6	Nordic languages
iso8859_13	iso-8859-13	Baltic languages

Codec	Aliases	Languages
iso8859_14	iso-8859-14, latin8, L8	Celtic languages
iso8859_15	iso-8859-15	Western Europe
koi8_r		Russian
koi8_u		Ukrainian
mac_cyrillic	maccyrillic	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
mac_greek	macgreek	Greek
mac_iceland	maciceland	Icelandic
mac_latin2	maclatin2, maccentraleurope	Central and Eastern Europe
mac_roman	macroman	Western Europe
mac_turkish	macturkish	Turkish
utf_16	U16, utf16	all languages
utf_16_be	UTF-16BE	all languages (BMP only)
utf_16_le	UTF-16LE	all languages (BMP only)
utf_7	U7	all languages
utf_8	U8, UTF, utf8	all languages

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.

Codec	Aliases	Operand type	Purpose
base64_codec	base64, base-64	byte string	Convert operand to MIME base64
hex_codec	hex	byte string	Convert operand to hexadecimal representa
idna		Unicode string	Implements RFC 3490. New in version 2.3
mbcs	dbcs	Unicode string	Windows only: Encode operand according t
palmos		Unicode string	Encoding of PalmOS 3.5
punycode		Unicode string	Implements RFC 3492. New in version 2.3
quopri_codec	quopri, quoted-printable, quotedprintable	byte string	Convert operand to MIME quoted printable
raw_unicode_escape		Unicode string	Produce a string that is suitable as raw Unic
rot_13	rot13	byte string	Returns the Caesar-cypher encryption of the
string_escape		byte string	Produce a string that is suitable as string lite
undefined		any	Raise an exception for all conversion. Can b
unicode_escape		Unicode string	Produce a string that is suitable as Unicode
unicode_internal		Unicode string	Return the internal represenation of the ope
uu_codec	uu	byte string	Convert the operand using uuencode
zlib_codec	zip, zlib	byte string	Compress the operand using gzip

4.9.3 `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.Alliancefrançaise.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.

4.10 `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 3.2.0 which is publically available from ftp://ftp.unicode.org/.

The module uses the same names and symbols as defined by the UnicodeData File Format 3.2.0 (see http://www.unicode.org/Public/UNIDATA/UnicodeData.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.

mirrored(*unichr*)
Returns the mirrored property of 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 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.

4.11 `stringprep` — Internet String Preparation

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 tableC.5 (Surrogate codes).

in_table_c6(*code*)
Determine whether *code* is in tableC.6 (Inappropriate for plain text).

in_table_c7(*code*)
Determine whether *code* is in tableC.7 (Inappropriate for canonical representation).

in_table_c8(*code*)
Determine whether *code* is in tableC.8 (Change display properties or are deprecated).

in_table_c9(*code*)
Determine whether *code* is in tableC.9 (Tagging characters).

in_table_d1(*code*)
Determine whether *code* is in tableD.1 (Characters with bidirectional property “R” or “AL”).

in_table_d2(*code*)
Determine whether *code* is in tableD.2 (Characters with bidirectional property “L”).

CHAPTER

FIVE

Miscellaneous Services

The modules described in this chapter provide miscellaneous services that are available in all Python versions. Here's an overview:

pydoc	Documentation generator and online help system.
doctest	A framework for verifying examples in docstrings.
unittest	Unit testing framework for Python.
test	Regression tests package containing the testing suite for Python.
test.test_support	Support for Python regression tests.
math	Mathematical functions (`sin()` etc.).
cmath	Mathematical functions for complex numbers.
random	Generate pseudo-random numbers with various common distributions.
whrandom	Floating point pseudo-random number generator.
bisect	Array bisection algorithms for binary searching.
heapq	Heap queue algorithm (a.k.a. priority queue).
array	Efficient arrays of uniformly typed numeric values.
sets	Implementation of sets of unique elements.
itertools	Functions creating iterators for efficient looping.
ConfigParser	Configuration file parser.
fileinput	Perl-like iteration over lines from multiple input streams, with "save in place" capability.
xreadlines	Efficient iteration over the lines of a file.
calendar	Functions for working with calendars, including some emulation of the UNIX **cal** program.
cmd	Build line-oriented command interpreters.
shlex	Simple lexical analysis for UNIX shell-like languages.

5.1 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`'.

5.2 `doctest` — Test docstrings represent reality

The `doctest` module searches a module's docstrings for text that looks like an interactive Python session, then executes all such sessions to verify they still work exactly as shown. Here's a complete but small example:

```
"""
This is module example.

Example 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)
    265252859812191058636308480000000L
    >>> factorial(30L)
    265252859812191058636308480000000L
    >>> 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)
    265252859812191058636308480000000L

    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:
        try:
            result *= factor
        except OverflowError:
            result *= long(factor)
        factor += 1
    return result

def _test():
    import doctest, example
    return doctest.testmod(example)

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
Running example.__doc__
Trying: factorial(5)
Expecting: 120
ok
0 of 1 examples failed in example.__doc__
Running example.factorial.__doc__
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
Trying: factorial(30)
Expecting: 265252859812191058636308480000000L
ok
```

And so on, eventually ending with:

```
Trying: factorial(1e100)
Expecting:
Traceback (most recent call last):
    ...
OverflowError: n too large
ok
0 of 8 examples failed in example.factorial.__doc__
2 items passed all tests:
   1 tests in example
   8 tests in example.factorial
9 tests in 2 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 docstrings in 'doctest.py' contain detailed information about all aspects of `doctest`, and we'll just cover the more important points here.

5.2.1 Normal Usage

In normal use, end each module `M` with:

```
def _test():
    import doctest, M           # replace M with your module's name
    return doctest.testmod(M)   # ditto

if __name__ == "__main__":
    _test()
```

If you want to test the module as the main module, you don't need to pass M to `testmod()`; in this case, it will test the current module.

Then running the module as a script causes the examples in the docstrings to get executed and verified:

```
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.'`.

Run it with the **-v** switch instead:

```
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=1` to `testmod()`, or prohibit it by passing `verbose=0`. In either of those cases, `sys.argv` is not examined by `testmod()`.

In any case, `testmod()` returns a 2-tuple of ints `(`*f*`, `*t*`)`, where *f* is the number of docstring examples that failed and *t* is the total number of docstring examples attempted.

5.2.2 Which Docstrings Are Examined?

See the docstrings in 'doctest.py' for all the details. They're unsurprising: the module docstring, and all function, class and method docstrings are searched. Optionally, the tester can be directed to exclude docstrings attached to

objects with private names. 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 even if the tester has been directed to skip over private names in the rest of the module. In output, a key `K` in `M.__test__` appears with name

```
<name of M>.__test__.K
```

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes. While private names reached from `M`'s globals can be optionally skipped, all names reached from `M.__test__` are searched.

5.2.3 What's the Execution Context?

By default, each time `testmod()` finds a docstring to test, it uses a *copy* of `M`'s globals, so that running tests on a module 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.

You can force use of your own dict as the execution context by passing `globs=your_dict` to `testmod()` instead. Presumably this would be a copy of `M.__dict__` merged with the globals from other imported modules.

5.2.4 What About Exceptions?

No problem, as long as the only output generated by the example is the traceback itself. For example:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
>>>
```

Note that only the exception type and value are compared (specifically, only the last line in the traceback). The various "File" lines in between can be left out (unless they add significantly to the documentation value of the example).

5.2.5 Advanced Usage

Several module level functions are available for controlling how doctests are run.

debug(*module*, *name*)

Debug a single docstring containing doctests.

Provide the *module* (or dotted name of the module) containing the docstring to be debugged and the *name* (within the module) of the object with the docstring to be debugged.

The doctest examples are extracted (see function `testsource()`), and written to a temporary file. The Python debugger, pdb, is then invoked on that file. New in version 2.3.

testmod()

This function provides the most basic interface to the doctests. It creates a local instance of class `Tester`, runs appropriate methods of that class, and merges the results into the global `Tester` instance, `master`.

To get finer control than `testmod()` offers, create an instance of `Tester` with custom policies, or run methods of `master` directly. See `Tester.__doc__` for details.

testsource(*module*, *name*)

Extract the doctest examples from a docstring.

Provide the *module* (or dotted name of the module) containing the tests to be extracted and the *name* (within the module) of the object with the docstring containing the tests to be extracted.

The doctest examples are returned as a string containing Python code. The expected output blocks in the examples are converted to Python comments. New in version 2.3.

DocTestSuite([*module*])

Convert doctest tests for a module to a `unittest.TestSuite`.

The returned `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 `DocTestTestFailure` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

The optional *module* argument 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.

Example using one of the many ways that the `unittest` module can use a `TestSuite`:

```
import unittest
import doctest
import my_module_with_doctests

suite = doctest.DocTestSuite(my_module_with_doctests)
runner = unittest.TextTestRunner()
runner.run(suite)
```

New in version 2.3. **Warning:** This function does not currently search `M.__test__` and its search technique does not exactly match `testmod()` in every detail. Future versions will bring the two into convergence.

5.2.6 How are Docstring Examples Recognized?

In most cases a copy-and-paste of an interactive console session works fine—just make sure the leading whitespace is rigidly consistent (you can mix tabs and spaces if you're too lazy to do it right, but `doctest` is not in the business of guessing what you think a tab means).

```
>>> # 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.

- 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 need to double the backslash in the docstring version. This is simply because you're in a string, and so the backslash must be escaped for it to survive intact. Like:

```
>>> if "yes" == \\
...     "y" +   \\
...     "es":
...     print 'yes'
yes
```

- The starting column doesn't matter:

```
  >>> 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 triggered it.

5.2.7 Warnings

1. `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

```
>>> foo()
{"Hermione": "hippogryph", "Harry": "broomstick"}
>>>
```

is vulnerable! One workaround is to do

```
>>> foo() == {"Hermione": "hippogryph", "Harry": "broomstick"}
1
>>>
```

instead. Another is to do

```
>>> 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

```
>>> id(1.0) # certain to fail some of the time
7948648
>>>
```

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.

```
>>> 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:

```
>>> 3./4  # utterly safe
0.75
```

Simple fractions are also easier for people to understand, and that makes for better documentation.

2. Be careful if you have code that must only execute once.

   If you have module-level code that must only execute once, a more foolproof definition of `_test()` is

   ```
   def _test():
       import doctest, sys
       doctest.testmod()
   ```

3. WYSIWYG isn't always the case, starting in Python 2.3. The string form of boolean results changed from `'0'` and `'1'` to `'False'` and `'True'` in Python 2.3. This makes it clumsy to write a doctest showing boolean results that passes under multiple versions of Python. In Python 2.3, by default, and as a special case, if an expected output block consists solely of `'0'` and the actual output block consists solely of `'False'`, that's accepted as an exact match, and similarly for `'1'` versus `'True'`. This behavior can be turned off by passing the new (in 2.3) module constant `DONT_ACCEPT_TRUE_FOR_1` as the value of `testmod()`'s new (in 2.3) optional *optionflags* argument. Some years after the integer spellings of booleans are history, this hack will probably be removed again.

5.2.8 Soapbox

The first word in "doctest" is "doc," and that's why the author wrote `doctest`: to keep documentation up to date. It so happens that `doctest` makes a pleasant unit testing environment, but that's not its primary purpose.

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 possible, show just a few normal cases, show endcases, show interesting subtle cases, and show an example of each kind of exception that can be raised. You're probably testing for endcases and subtle cases anyway in an interactive shell: `doctest` wants to make it as easy as possible to capture those sessions, and will verify they continue to work as designed forever after.

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.

For exhaustive testing, or testing boring cases that add no value to the docs, define a `__test__` dict instead. That's what it's for.

5.3 `unittest` — Unit testing framework

New in version 2.1.

The Python unit testing framework, often 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.

PyUnit 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, PyUnit 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. PyUnit 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 PyUnit-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. PyUnit provide 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:

PyUnit Web Site
(http://pyunit.sourceforge.net/)
: The source for further information on PyUnit.

Simple Smalltalk Testing: With Patterns
(http://www.XProgramming.com/testfram.htm)
: Kent Beck's original paper on testing frameworks using the pattern shared by `unittest`.

5.3.1 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:

```
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 3 tests in 0.000s

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:

```
suite = unittest.TestSuite()
suite.addTest(unittest.makeSuite(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.

5.3.2 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 PyUnit, test cases are represented by instances of the `TestCase` class in the `unittest` module. 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 test case subclass will simply override the `runTest()` method in order to perform specific testing code:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def runTest(self):
        widget = Widget("The widget")
        self.failUnless(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 when the test case runs, an exception will be raised, and the testing framework will identify the test case as a *failure*. Other exceptions that do not arise from checks made through the `assert*()` and `fail*()` methods are identified by the testing framework as dfnerrors.

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:

```
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:

```
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:

```
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 regardless of whether or not `runTest()` succeeded.

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, PyUnit provides a simpler mechanism:

```
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:

```
defaultSizeTestCase = WidgetTestCase("testDefaultSize")
resizeTestCase = WidgetTestCase("testResize")
```

Test case instances are grouped together according to the features they test. PyUnit provides a mechanism for this: the `test suite`, represented by the class `TestSuite` in the `unittest` module:

```
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:

```
class WidgetTestSuite(unittest.TestSuite):
    def __init__(self):
        unittest.TestSuite.__init__(self,map(WidgetTestCase,
                                              ("testDefaultSize",
                                               "testResize")))
```

(The latter is admittedly not for the faint-hearted!)

Since it is a common pattern to create a `TestCase` subclass with many similarly named test functions, there is a convenience function called `makeSuite()` provided in the `unittest` module that constructs a test suite that comprises all of the test cases in a test case class:

```
suite = unittest.makeSuite(WidgetTestCase,'test')
```

Note that when using the `makeSuite()` function, the order in which the various test cases will be run by the test suite is the order determined by sorting the test function names using the `cmp()` built-in 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 'widgettests.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.

5.3.3 Re-using old test code

Some users will find that they have existing test code that they would like to run from PyUnit, without converting every old test function to a `TestCase` subclass.

For this reason, PyUnit 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 optionally be wrapped.

Given the following test function:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

one can create an equivalent test case instance as follows:

```
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:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

Note: PyUnit supports the use of `AssertionError` as an indicator of test failure, but does not recommend it. Future versions may treat `AssertionError` differently.

5.3.4 Classes and functions

class `TestCase`()
Instances of the `TestCase` class represent the smallest testable units in a set of tests. 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 failures.

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, but all the contained tests and test suites are executed. Additional methods are provided to add test cases and suites to the aggregation. If *tests* is given, it must be a sequence of individual tests that will be added to the suite.

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` class. When loading from a module, it considers all `TestCase`-derived classes. For each such class, it creates an instance for each method with a name beginning with the string '`test`'.

`defaultTestLoader`
Instance of the `TestLoader` class which can be shared. If no customization of the `TestLoader` is needed, this instance can always be used instead of creating new instances.

class `TextTestRunner`([*stream*[, *descriptions*[, *verbosity*]]])
A basic test runner implementation which prints results on standard output. 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:

```
if __name__ == '__main__':
    unittest.main()
```

In some cases, the existing tests may have be written using the `doctest` module. If so, that module provides a `DocTestSuite` class that can automatically build `unittest.TestSuite` instances from the existing test code. New in version 2.3.

5.3.5 TestCase 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 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 and used, but is not made available to the caller. This is equivalent to simply calling the `TestCase` instance.

`debug`()
Run the test without collecting the result. This allows exceptions raised by the test to be propogated 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.

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 the this test object. For `TestCase` instances, this will always be `1`, but this method is also implemented by the `TestSuite` class, which can return larger values.

defaultTestResult()
Return the default type of test result object to be used to run this test.

id()
Return a string identifying the specific test case. This is usually the full name of the test method, including the module and class names.

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.

5.3.6 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 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 set of tests that make up the suite.

addTests(*tests*)
Add all the tests from a sequence of TestCase and TestSuite instances to this test suite.

The run() method is also slightly different:

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.

In the typical usage of a TestSuite object, the run() method is invoked by a TestRunner rather than by the end-user test harness.

5.3.7 TestResult Objects

A TestResult object stores the results of a set of tests. The TestCase and TestSuite classes ensure that results are properly stored; 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.

Each instance holds the total number of tests run, and collections of failures and errors that occurred among those test runs. The collections contain tuples of (*testcase*, *traceback*), where *traceback* is a string containing a formatted version of the traceback for the exception.

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 pairs of TestCase instances and the formatted tracebacks for tests which raised an exception but did not signal a test failure. Changed in version 2.2: Contains formatted tracebacks instead of sys.exc_info() results.

failures
A list containing pairs of TestCase instances and the formatted tracebacks for tests which signalled a failure in the code under test. Changed in version 2.2: Contains formatted tracebacks instead of sys.exc_info() results.

testsRun
The number of tests which have been started.

wasSuccessful()
Returns true if all tests run so far have passed, otherwise returns false.

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.

stopTest(*test*)
Called when the test case *test* has been executed, regardless of the outcome.

addError(*test, err*)
Called when the test case *test* raises an exception without signalling a test failure. *err* is a tuple of the form returned by `sys.exc_info()`: (*type, value, traceback*).

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*).

addSuccess(*test*)
This method is called for a test that does not fail; *test* is the test case object.

One additional method is available for `TestResult` objects:

stop()
This method can be called to signal that the set of tests being run should be aborted. Once this has been called, the `TestRunner` object return to its caller without running any additional tests. This is used by the `TextTestRunner` class to stop the test framework when the user signals an interrupt from the keyboard. Interactive tools which provide runners can use this in a similar manner.

5.3.8 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 the `defaultTestLoader` module attribute. Using a subclass or instance would allow 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 class `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, or a callable object which returns a `TestCase` or `TestSuite` instance. 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 a 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*.

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'`.

`sortTestMethodsUsing`
Function to be used to compare method names when sorting them in `getTestCaseNames()`. The default value is the built-in `cmp()` function; it can 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.

5.3.9 Getting Extended Error Information

Some applications can make use of more error information (for example, an integrated development environment, or IDE). Such an application can retrieve supplemental information about errors and failures by using an alternate `TestResult` implementation, and extending the `defaultTestResult()` method of the `TestCase` class to provide it.

Here is a brief example of a `TestResult` subclass which stores the actual exception and traceback objects. (Be aware that storing traceback objects can cause a great deal of memory not to be reclaimed when it otherwise would be, which can have effects that affect the behavior of the tests.)

```
import unittest

class MyTestCase(unittest.TestCase):
    def defaultTestResult(self):
        return MyTestResult()

class MyTestResult(unittest.TestResult):
    def __init__(self):
        self.errors_tb = []
        self.failures_tb = []

    def addError(self, test, err):
        self.errors_tb.append((test, err))
        unittest.TestResult.addError(self, test, err)

    def addFailure(self, test, err):
        self.failures_tb.append((test, err))
        unittest.TestResult.addFailure(self, test, err)
```

Tests written using `MyTestCase` as the base class, instead of `TestCase`, will allow tools to extract additional information from the results object.

5.4 `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` (section 5.3):
Writing PyUnit regression tests.

Module `doctest` (section 5.2):
Tests embedded in documentation strings.

5.4.1 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 have the name of all the test methods start with ‘`''test_''`’ as well as the module’s name. 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:

```
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,
                              MyTestCase2,
                              ... list other tests ...
                             )

if __name__ == '__main__':
    test_main()
```

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).
- 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:

```
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.

5.4.2 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.

5.5 `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.

5.6 `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:

acos(*x*)
Return the arc cosine of *x*.

asin(*x*)
Return the arc sine of *x*.

atan(*x*)
Return the arc tangent of *x*.

atan2(*y*, *x*)
Return `atan(`*y* `/` *x*`)`.

ceil(*x*)
Return the ceiling of *x* as a float.

cos(*x*)
Return the cosine of *x*.

cosh(*x*)
Return the hyperbolic cosine of *x*.

degrees(*x*)
Converts angle *x* from radians to degrees.

exp(*x*)
Return `e**`*x*.

fabs(*x*)
Return the absolute value of *x*.

floor(*x*)
Return the floor of *x* as a float.

fmod(*x*, *y*)
Return `fmod(`*x*`, `*y*`)`, as defined by the platform C library. Note that the Python expression *x* `%` *y* may not return the same result.

frexp(*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* `* 2**`*e*. If *x* is zero, returns `(0.0, 0)`, otherwise `0.5 <= abs(`*m*`) < 1`.

hypot(*x*, *y*)
Return the Euclidean distance, `sqrt(`*x*`*`*x* `+` *y*`*`*y*`)`.

ldexp(*x*, *i*)
Return *x* `* (2**`*i*`)`.

log(*x*[, *base*])
Returns the logarithm of *x* to the given *base*. If the *base* is not specified, returns the natural logarithm of *x*. Changed in version 2.3: *base* argument added.

log10(*x*)
Return the base-10 logarithm of *x*.

modf(*x*)
Return the fractional and integer parts of *x*. Both results carry the sign of *x*. The integer part is returned as a float.

pow(*x*, *y*)

Return *x*`**`*y*.

`radians`(*x*)
Converts angle *x* from degrees to radians.

`sin`(*x*)
Return the sine of *x*.

`sinh`(*x*)
Return the hyperbolic sine of *x*.

`sqrt`(*x*)
Return the square root of *x*.

`tan`(*x*)
Return the tangent of *x*.

`tanh`(*x*)
Return the hyperbolic tangent of *x*.

Note that `frexp()` and `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).

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` (section 5.7):
Complex number versions of many of these functions.

5.7 `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 ∞, continuous from below. The other extends left from -1 along the real axis to -∞, 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 -∞, 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 $\pm$-∞`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 ∞`j` and continuous

from the right, and one from -1j down to -∞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 ∞j, continuous from the left. The other extends from -1j along the imaginary axis to -∞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 ∞, continuous from above. The other extends from -1 along the real axis to -∞, continuous from above. (This should probably be changed so the right cut becomes continuous from the other side.)

cos(*x*)
Return the cosine of *x*.

cosh(*x*)
Return the hyperbolic cosine of *x*.

exp(*x*)
Return the exponential value e**x.

log(*x*)
Return the natural logarithm of *x*. There is one branch cut, from 0 along the negative real axis to -∞, continuous from above.

log10(*x*)
Return the base-10 logarithm of *x*. This has the same branch cut as log().

sin(*x*)
Return the sine of *x*.

sinh(*x*)
Return the hyperbolic sine of *x*.

sqrt(*x*)
Return the square root of *x*. This has the same branch cut as log().

tan(*x*)
Return the tangent of *x*.

tanh(*x*)
Return the hyperbolic tangent of *x*.

The module also defines two mathematical constants:

pi
The mathematical constant *pi*, as a real.

e
The mathematical constant *e*, as a real.

Note that the selection of functions is similar, but not identical, to that in module 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 math.sqrt(-1) raise an exception than return a complex number. Also note that the functions defined in 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:

See Also:

Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothings's sign bit. In Iserles, A., and Powell, M. (eds.), *The state of the art in numerical analysis*. Clarendon Press (1987) pp165-211.

5.8 `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 ensure 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.

As an example of subclassing, the `random` module provides the `WichmannHill` class which 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. 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 *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 conjuction 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..

Functions for integers:

`randrange``(`[*start*,]*stop*[, *step*]`)`
: Return a randomly selected element from `range(`*start*`, `*stop*`, `*step*`)`. This is equivalent to `choice(range(`*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* `<=` *N* `<=` *b*.

Functions for sequences:

`choice``(`*seq*`)`

Return a random element from the non-empty sequence *seq*.

shuffle(*x*[, *random*])
Shuffle the sequence *x* in place. The optional argument *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 `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 `xrange` 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* `<=` *N* `<` *b*.

betavariate(*alpha, beta*)
Beta distribution. Conditions on the parameters are *alpha* `> -1` and *beta* `> -1`. Returned values range between 0 and 1.

cunifvariate(*mean, arc*)
Circular uniform distribution. *mean* is the mean angle, and *arc* is the range of the distribution, centered around the mean angle. Both values must be expressed in radians, and can range between 0 and *pi*. Returned values range between *mean* `-` *arc*`/2` and *mean* `+` *arc*`/2` and are normalized to between 0 and *pi*.

Deprecated since release 2.3. Instead, use (*mean* `+` *arc* `*` `(random.random() - 0.5)) %math.pi`.

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**pi*, 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**pi*.

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 Generator

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.

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, January pp.3-30 1998.

Wichmann, B. A. & Hill, I. D., "Algorithm AS 183: An efficient and portable pseudo-random number generator", *Applied Statistics* 31 (1982) 188-190.

5.9 whrandom — Pseudo-random number generator

Deprecated since release 2.1. Use `random` instead.

Note: This module was an implementation detail of the `random` module in releases of Python prior to 2.1. It is no longer used. Please do not use this module directly; use `random` instead.

This module implements a Wichmann-Hill pseudo-random number generator class that is also named `whrandom`. Instances of the `whrandom` class conform to the Random Number Generator interface described in section **??**. They also offer the following method, specific to the Wichmann-Hill algorithm:

seed([*x, y, z*])
Initializes the random number generator from the integers *x*, *y* and *z*. When the module is first imported, the random number is initialized using values derived from the current time. If *x*, *y*, and *z* are either omitted or `0`, the seed will be computed from the current system time. If one or two of the parameters are `0`, but not all three, the zero values are replaced by ones. This causes some apparently different seeds to be equal, with the corresponding result on the pseudo-random series produced by the generator.

choice(*seq*)
Chooses a random element from the non-empty sequence *seq* and returns it.

randint(*a, b*)
Returns a random integer *N* such that *a*<=*N*<=*b*.

random()
Returns the next random floating point number in the range [0.0 ... 1.0).

seed(*x, y, z*)
Initializes the random number generator from the integers *x*, *y* and *z*. When the module is first imported, the random number is initialized using values derived from the current time.

uniform(*a, b*)
Returns a random real number *N* such that *a*<=*N*<*b*.

When imported, the `whrandom` module also creates an instance of the `whrandom` class, and makes the methods of that instance available at the module level. Therefore one can write either `N = whrandom.random()` or:

```
generator = whrandom.whrandom()
N = generator.random()
```

Note that using separate instances of the generator leads to independent sequences of pseudo-random numbers.

See Also:

Module `random` (section 5.8):
Generators for various random distributions and documentation for the Random Number Generator interface.

Wichmann, B. A. & Hill, I. D., "Algorithm AS 183: An efficient and portable pseudo-random number generator", *Applied Statistics* 31 (1982) 188-190.

5.10 `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.

`insort`(...)
Alias for `insort_right()`.

5.10.1 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.

```
>>> 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']
```

The bisect module can be used with the Queue module to implement a priority queue (example courtesy of Fredrik Lundh):

```
import Queue, bisect

class PriorityQueue(Queue.Queue):
    def _put(self, item):
        bisect.insort(self.queue, item)

# usage
queue = PriorityQueue(0)
queue.put((2, "second"))
queue.put((1, "first"))
queue.put((3, "third"))
priority, value = queue.get()
```

5.11 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*`] <= `*heap*`[2*`*k*`+1]` and *heap*`[`*k*`] <= `*heap*`[2*`*k*`+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, `IndexError` is raised. This is more efficient than `heappop()` followed by `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.

Example of use:

```
>>> 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
>>>
```

5.11.1 Theory

(This explanation is due to Franois Pinard. The Python code for this module was contributed by Kevin O'Connor.)

Heaps are arrays for which `a[`*k*`] <= a[2*`*k*`+1]` and `a[`*k*`] <= 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               6

  7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

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[1]. 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. :-)

5.12 `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 *type code*, which is a single character. The following type codes are defined:

Type code	C Type	Python Type	Minimum size in bytes
`'c'`	char	character	1
`'b'`	signed char	int	1
`'B'`	unsigned char	int	1
`'u'`	Py_UNICODE	Unicode character	2
`'h'`	signed short	int	2
`'H'`	unsigned short	int	2
`'i'`	signed int	int	2
`'I'`	unsigned int	long	2
`'l'`	signed long	int	4
`'L'`	unsigned long	long	4
`'f'`	float	float	4
`'d'`	double	float	8

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 `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:

[1]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! :-)

array(*typecode*[, *initializer*])
Return a new array whose items are restricted by *typecode*, and initialized from the optional *initializer* value, which must be a list or a string. The list or string is passed to the new array's `fromlist()`, `fromstring()`, or `fromunicode()` method (see below) to add initial items to the array.

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 occurences of *x* in the array.

extend(*a*)
Append array items from *a* to the end of the array. The two arrays must have *exactly* the same type code; if not, `TypeError` will be raised.

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 occurence 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 occurence 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.)

tounicode()
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` (section 4.3):
Packing and unpacking of heterogeneous binary data.

Module `xdrlib` (section 12.17):
Packing and unpacking of External Data Representation (XDR) data as used in some remote procedure call systems.

The Numerical Python Manual
(http://numpy.sourceforge.net/numdoc/HTML/numdoc.htm)
The Numeric Python extension (NumPy) defines another array type; see http://numpy.sourceforge.net/

for further information about Numerical Python. (A PDF version of the NumPy manual is available at http://numpy.sourceforge.net/numdoc/numdoc.pdf).

5.13 `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. 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 5.13.3.

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 5.13.3.

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.

5.13.1 Set Objects

Instances of `Set` and `ImmutableSet` both provide the following operations:

Operation	Equivalent	Result
`len(`*s*`)`		cardinality of set *s*
x `in` *s*		test *x* for membership in *s*
x `not in` *s*		test *x* for non-membership in *s*
s`.issubset(`*t*`)`	*s* `<=` *t*	test whether every element in *s* is in *t*
s`.issuperset(`*t*`)`	*s* `>=` *t*	test whether every element in *t* is in *s*
s`.union(`*t*`)`	*s* — *t*	new set with elements from both *s* and *t*
s`.intersection(`*t*`)`	*s* & *t*	new set with elements common to *s* and *t*
s`.difference(`*t*`)`	*s* - *t*	new set with elements in *s* but not in *t*
s`.symmetric_difference(`*t*`)`	*s* ^ *t*	new set with elements in either *s* or *t* but not both
s`.copy()`		new set with a shallow copy of *s*

Note, this 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`:

Operation	Result
`hash(`*s*`)`	returns a hash value for *s*

The following table lists operations available in `Set` but not found in `ImmutableSet`:

Operation	Equivalent	Result
s`.union_update(`*t*`)`	*s* —= *t*	return set *s* with elements added from *t*
s`.intersection_update(`*t*`)`	*s* &= *t*	return set *s* keeping only elements also found in *t*
s`.difference_update(`*t*`)`	*s* -= *t*	return set *s* after removing elements found in *t*
s`.symmetric_difference_update(`*t*`)`	*s* ^= *t*	return set *s* with elements from *s* or *t* but not both
s`.add(`*x*`)`		add element *x* to set *s*
s`.remove(`*x*`)`		remove *x* from set *s*; raises KeyError if not present
s`.discard(`*x*`)`		removes *x* from set *s* if present
s`.pop()`		remove and return an arbitrary element from *s*; raises KeyError if
s`.clear()`		remove all elements from set *s*

Changed in version 2.3.1: Earlier versions had an `update()` method; use `union_update()` instead.

Note, this non-operator versions of `union_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.

5.13.2 Example

```
>>> 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           # intersection
>>> fulltime_management = managers - engineers - programmers # difference
>>> engineers.add('Marvin')                                 # add element
>>> print engineers
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
>>> employees.issuperset(engineers)           # superset test
False
>>> employees.union_update(engineers)         # update from another set
>>> employees.issuperset(engineers)
True
>>> for group in [engineers, programmers, management, employees]:
...     group.discard('Susan')                # unconditionally remove element
...     print group
...
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
Set(['Janice', 'Jack', 'Sam'])
Set(['Jane', 'Zack', 'Jack'])
Set(['Jack', 'Sam', 'Jane', 'Marvin', 'Janice', 'John', 'Zack'])
```

5.13.3 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 `__as_immutable__()` method which returns an immutable equivalent.

Since `Set` objects have a `__as_immutable__()` 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 `__as_temporarily_immutable__()` 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 `__as_temporarily_immutable__()` 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.

5.14 `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: `tabulate(f)` which produces a sequence `f(0), f(1), ...`. This toolbox provides `imap()` and `count()` which can be combined to form `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 `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 C 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:

The Standard ML Basis Library, *The Standard ML Basis Library*.

Haskell, A Purely Functional Language, *Definition of Haskell and the Standard Libraries*.

5.14.1 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:

```
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*. 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:

```
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:

```
def cycle(iterable):
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
              yield element
```

Note, this is the only member of the toolkit that 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
```

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:

```
def ifilter(predicate, iterable):
    if predicate is None:
        def predicate(x):
            return x
    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:

```
def ifilterfalse(predicate, iterable):
    if predicate is None:
        def predicate(x):
            return x
    for x in iterable:
        if not predicate(x):
            yield x
```

imap(*function, *iterables*)
Make an iterator that computes the function using arguments from each of the iterables. If *function* is set to `None`, then `imap()` returns the arguments as a tuple. Like `map()` but stops when the shortest iterable is exhausted instead of filling in `None` for shorter iterables. The reason for the difference is that infinite iterator arguments are typically an error for `map()` (because the output is fully evaluated) but represent a common and useful way of supplying arguments to `imap()`. Equivalent to:

```
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)
```

islice(*iterable*, [*start*,] *stop* [, *step*])

Make an iterator that returns selected elements from the iterable. If *start* is non-zero, then elements from the iterable are skipped until start is reached. Afterward, elements are returned consecutively unless *step* is set higher than one which results in items being skipped. If *stop* is `None`, then iteration continues until the iterator is exhausted, if at all; otherwise, it stops at the specified position. Unlike regular slicing, `islice()` does not support negative values for *start*, *stop*, or *step*. Can be used to extract related fields from data where the internal structure has been flattened (for example, a multi-line report may list a name field on every third line). Equivalent to:

```
def islice(iterable, *args):
    s = slice(*args)
    next, stop, step = s.start or 0, s.stop, s.step or 1
    for cnt, element in enumerate(iterable):
        if cnt < next:
            continue
        if stop is not None and cnt >= stop:
            break
        yield element
        next += 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:

```
def izip(*iterables):
    iterables = map(iter, iterables)
    while iterables:
        result = [i.next() for i in iterables]
        yield tuple(result)
```

Changed in version 2.3.1: When no iterables are specified, returns a zero length iterator instead of raising a TypeError exception.

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:

```
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:

```
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:

```
def takewhile(predicate, iterable):
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

5.14.2 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(1,4), repeat(3)):
...    print cube
...
1
8
27

>>> reportlines = ['EuroPython', 'Roster', '', 'alex', '', 'laura',
                   '', 'martin', '', 'walter', '', 'samuele']
>>> for name in islice(reportlines, 3, None, 2):
...    print name.title()
...
Alex
Laura
Martin
Walter
Samuele
```

This section shows how itertools can be combined to create other more powerful itertools. Note that `enumerate()` and `iteritems()` already have efficient implementations in Python. They are only included here to illustrate how higher level tools can be created from building blocks.

```
    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(pred, seq):
        "Returns True if pred(x) is True for every element in the iterable"
        return False not in imap(pred, seq)

    def some(pred, seq):
        "Returns True if pred(x) is True at least one element in the iterable"
        return True in imap(pred, seq)

    def no(pred, seq):
        "Returns True if pred(x) is False for every element in the iterable"
        return True not in imap(pred, seq)

    def quantify(pred, seq):
        "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"
        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 window(seq, n=2):
        "Returns a sliding window (of width n) over data from the iterable"
        "   s -> (s0,s1,...s[n-1]), (s1,s2,...,sn), ...                   "
        it = iter(seq)
        result = tuple(islice(it, n))
        if len(result) == n:
            yield result
        for elem in it:
            result = result[1:] + (elem,)
            yield result

    def tee(iterable):
        "Return two independent iterators from a single iterable"
        def gen(next, data={}, cnt=[0]):
            dpop = data.pop
            for i in count():
                if i == cnt[0]:
                    item = data[i] = next()
                    cnt[0] += 1
                else:
                    item = dpop(i)
                yield item
        next = iter(iterable).next
        return (gen(next), gen(next))
```

5.15 `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 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*.

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 when multiple sections with the same name are found, or 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 `MAX_INTERPOLATION_DEPTH`. Subclass of `InterpolationError`.

exception `InterpolationMissingOptionError`

Exception raised when an option referenced from a value does not exist. Subclass of `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 `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.

`MAX_INTERPOLATION_DEPTH`
The maximum depth for recursive interpolation for `get()` when the *raw* parameter is false. This is relevant only for the `ConfigParser` class.

See Also:

Module `shlex` (section 5.20):
Support for a creating UNIX shell-like mini-languages which can be used as an alternate format for application configuration files.

5.15.1 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 1; otherwise return 0. New in version 1.6.

`read``(`*filenames*`)`
Read and parse a list of filenames. 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:

```
import ConfigParser, os

config = ConfigParser.ConfigParser()
config.readfp(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')])
```

`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`. 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 1; otherwise return 0. 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.

5.15.2 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*]])
Create a generator which will return a tuple `(name, value)` for each option in the given *section*. Optional arguments have the same meaning as for the `get()` method. New in version 2.3.

5.16 `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:

```
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. 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.

The following function is the primary interface of this module:

`input``(`[*files*[, *inplace*[, *backup*]]]`)`
: 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.

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`.

`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 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:

class `FileInput``(`[*files*[, *inplace*[, *backup*]]]`)`
: Class `FileInput` is the implementation; its methods `filename()`, `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.

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.

5.17 `xreadlines` — Efficient iteration over a file

New in version 2.1.

Deprecated since release 2.3. Use `for line in file` instead.

This module defines a new object type which can efficiently iterate over the lines of a file. An xreadlines object is a sequence type which implements simple in-order indexing beginning at `0`, as required by `for` statement or the `filter()` function.

Thus, the code

```
import xreadlines, sys

for line in xreadlines.xreadlines(sys.stdin):
    pass
```

has approximately the same speed and memory consumption as

```
while 1:
    lines = sys.stdin.readlines(8*1024)
    if not lines: break
    for line in lines:
        pass
```

except the clarity of the `for` statement is retained in the former case.

`xreadlines`(*fileobj*)
Return a new xreadlines object which will iterate over the contents of *fileobj*. *fileobj* must have a `readlines()` method that supports the *sizehint* parameter. **Note:** Because the `readlines()` method buffers data, this effectively ignores the effects of setting the file object as unbuffered.

An xreadlines object *s* supports the following sequence operation:

Operation	Result
s[*i*]	*i*'th line of *s*

If successive values of *i* are not sequential starting from `0`, this code will raise `RuntimeError`.

After the last line of the file is read, this code will raise an `IndexError`.

5.18 `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 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.

`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:

```
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 `1` if *year* is a leap year, otherwise `0`.

`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`).

`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. 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()`. 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. 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()`. The earliest year for which a calendar can be generated is platform-dependent. 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.

See Also:

Module `time` (section 6.10):
Low-level time related functions.

5.19 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.

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..

5.19.1 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).

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.

`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.

`completedefault`(*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.)

5.20 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, (e.g. in run control files for Python applications) or for parsing quoted strings.

See Also:

Module `ConfigParser` (section 5.15):
Parser for configuration files similar to the Windows '.ini' files.

5.20.1 Module Contents

The `shlex` module defines the following functions:

`split`(*s*[, *comments*=`False`])
Split the string *s* using shell-like syntax. If *comments* is `False`, 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 classes:

class `shlex`([*instream*=`sys.stdin`[, *infile*=`None`[, *posix*=`False`]]])
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 5.20.2.

5.20.2 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 (e.g. `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.

5.20.3 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;

CHAPTER

SIX

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.
os.path	Common pathname manipulations.
dircache	Return directory listing, with cache mechanism.
stat	Utilities for interpreting the results of `os.stat()`, `os.lstat()` and `os.fstat()`.
statcache	Stat files, and remember results.
statvfs	Constants for interpreting the result of `os.statvfs()`.
filecmp	Compare files efficiently.
popen2	Subprocesses with accessible standard I/O streams.
datetime	Basic date and time types.
time	Time access and conversions.
sched	General purpose event scheduler.
mutex	Lock and queue for mutual exclusion.
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.
getopt	Portable parser for command line options; support both short and long option names.
optparse	Powerful, flexible, extensible, easy-to-use command-line parsing library.
tempfile	Generate temporary files and directories.
errno	Standard errno system symbols.
glob	UNIX shell style pathname pattern expansion.
fnmatch	UNIX shell style filename pattern matching.
shutil	High-level file operations, including copying.
locale	Internationalization services.
gettext	Multilingual internationalization services.
logging	Logging module for Python based on PEP 282.

6.1 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`!

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`.

6.1.1 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.

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:** 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, this mapping may be passed to the appropriate process-creation functions to cause child processes to use a modified environment.

`chdir``(`*path*`)`
`fchdir``(`*fd*`)`
`getcwd``()`
These functions are described in "Files and Directories" (section 6.1.4).

`ctermid``()`
Return the filename corresponding to the controlling terminal of the process. Availability: UNIX.

`getegid``()`
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.

`geteuid``()`
Return the current process' effective user id. Availability: UNIX.

`getgid``()`
Return the real group id of the current process. Availability: UNIX.

`getgroups``()`
Return list of supplemental group ids associated with the current process. Availability: UNIX.

`getlogin``()`

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.

`getpgid`(*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.

`getpgrp`()
Return the id of the current process group. Availability: UNIX.

`getpid`()
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.

`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.

6.1.2 File Object Creation

These functions create new file objects.

fdopen(*fd*[, *mode*[, *bufsize*]])
Return an open file object connected to the file descriptor *fd*. The *mode* and *bufsize* arguments have the same meaning as the corresponding arguments to the built-in `open()` function. Availability: Macintosh, UNIX, Windows.

Changed in version 2.3: When specified, the *mode* argument must now start with one of the letters '`r`', '`w`', or '`a`', otherwise a `ValueError` is raised.

popen(*command*[, *mode*[, *bufsize*]])
Open a pipe to or from *command*. The return value is an open file object connected to the pipe, which can be read or written depending on whether *mode* is `'r'` (default) or `'w'`. The *bufsize* argument has the same meaning as the corresponding argument to the built-in `open()` function. The exit status of the command (encoded in the format specified for `wait()`) is available as the return value of the `close()` method of the file object, except that when the exit status is zero (termination without errors), `None` is returned. Availability: 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 `_popen()` function from the libraries provided with Windows. Newer versions of Python do not use the broken implementation from the Windows libraries.

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: UNIX, Windows.

For each of these `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'`.

These methods do not make it possible to retrieve the return code 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 6.8.2).

popen2(*cmd*[, *mode*[, *bufsize*]])
Executes *cmd* as a sub-process. Returns the file objects (*child_stdin*, *child_stdout*). Availability: UNIX, Windows. New in version 2.0.

popen3(*cmd*[, *mode*[, *bufsize*]])
Executes *cmd* as a sub-process. Returns the file objects (*child_stdin*, *child_stdout*, *child_stderr*). Availability: UNIX, Windows. New in version 2.0.

popen4(*cmd*[, *mode*[, *bufsize*]])

Executes *cmd* as a sub-process. Returns the file objects (*child_stdin*, *child_stdout_and_stderr*). Availability: UNIX, Windows. New in version 2.0.

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.

6.1.3 File Descriptor Operations

These functions operate on I/O streams referred to using file descriptors.

close(*fd*)
Close file descriptor *fd*. 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 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: UNIX, Windows.

fdatasync(*fd*)
Force write of file with filedescriptor *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: 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: 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 filedescriptor *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: 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: UNIX.

isatty(*fd*)
Return `True` if the file descriptor *fd* is open and connected to a tty(-like) device, else `False`. Availability: UNIX.

lseek(*fd, pos, how*)
Set the current position of file descriptor *fd* to position *pos*, modified by *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.

open(*file*, *flags*[, *mode*])
Open the file *file* and set various flags according to *flags* and possibly its mode according to *mode*. The default *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).

Note: this function is intended for low-level I/O. For normal usage, use the built-in function `open()`, which returns a "file object" with `read()` and `write()` methods (and many more).

openpty()
Open a new pseudo-terminal pair. Return a pair of file descriptors (*master*, *slave*) for the pty and the tty, respectively. For a (slightly) more portable approach, use the `pty` module. Availability: Some flavors of UNIX.

pipe()
Create a pipe. Return a pair of file descriptors (*r*, *w*) usable for reading and writing, respectively. Availability: UNIX, Windows.

read(*fd*, *n*)
Read at most *n* bytes from file descriptor *fd*. Return a string containing the bytes read. If the end of the file referred to by *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: 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: 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: 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.

O_RDONLY
O_WRONLY
O_RDWR
O_NDELAY
O_NONBLOCK
O_APPEND
O_DSYNC
O_RSYNC
O_SYNC
O_NOCTTY
O_CREAT
O_EXCL
O_TRUNC
Options for the *flag* argument to the `open()` function. These can be bit-wise OR'd together. Availability: Macintosh, UNIX, Windows.

O_BINARY
Option for the *flag* argument to the `open()` function. This can be bit-wise OR'd together with those listed above. Availability: Macintosh, Windows.

O_NOINHERIT
O_SHORT_LIVED
O_TEMPORARY
O_RANDOM
O_SEQUENTIAL
O_TEXT
Options for the *flag* argument to the `open()` function. These can be bit-wise OR'd together. Availability: Windows.

6.1.4 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 `1` if access is allowed, `0` if not. See the UNIX man page *access*(2) for more information. Availability: UNIX, Windows.

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()
Return a string representing the current working directory. Availability: Macintosh, UNIX, Windows.

getcwdu()
Return a Unicode object representing the current working directory. Availability: UNIX, Windows. New in version 2.3.

chroot(*path*)
Change the root directory of the current process to *path*. Availability: UNIX. New in version 2.2.

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):

- `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: UNIX, Windows.

chown(*path, uid, gid*)
Change the owner and group id of *path* to the numeric *uid* and *gid*. Availability: 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: UNIX. New in version 2.3.

link(*src, dst*)
Create a hard link pointing to *src* named *dst*. Availability: 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: 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: UNIX.

FIFOs are pipes that can be accessed like regular files. FIFOs exist until they are deleted (for example with `os.unlink()`). Generally, FIFOs are used as rendezvous between "client" and "server" type processes: the server opens the FIFO for reading, and the client opens it for writing. Note that `mkfifo()` doesn't open the FIFO — it just creates the rendezvous point.

mknod(*path*[, *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 a device major number from a raw device number. New in version 2.3.

minor(*device*)
Extracts a device minor number from a raw device number. 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, Windows.

makedirs(*path*[, *mode*])
Recursive directory creation function. Like `mkdir()`, but makes all intermediate-level directories needed to contain the leaf directory. Throws an `error` exception if the leaf directory already exists or cannot be created. The default *mode* is `0777` (octal). This function does not properly handle UNC paths (only relevant on Windows systems; Universal Naming Convention paths are those that use the '`\\host\path`' syntax). New in version 1.5.2.

pathconf(*path*, *name*)
Return system configuration information relevant to a named 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: 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.

pathconf_names
Dictionary mapping names accepted by `pathconf()` and `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: UNIX.

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 `os.path.join(os.path.dirname(`*path*`),` *result*`)`. Availability: 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, directories corresponding to rightmost path segments will be pruned way until either the whole path is consumed or an error is raised (which is ignored, because it generally means that a parent directory is not empty). Throws an `error` exception 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()`.

Note: this function can fail with the new directory structure made if you lack permissions needed to remove the leaf directory or file. New in version 1.5.2.

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` (time of most recent content modification or metadata change).

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).

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_attrs` (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.) Availability: Macintosh, UNIX, Windows.

Changed in version 2.2: Added access to values as attributes of the returned object.

stat_float_times([*newvalue*])
Determine whether `stat_result` represents time stamps as float objects. If newval is True, future calls to stat() return floats, if it is False, future calls return ints. If newval is omitted, return the current setting.

For compatibility with older Python versions, accessing `stat_result` as a tuple always returns integers. For compatibility with Python 2.2, accessing the time stamps by field name also returns integers. Applications that want to determine the fractions of a second in a time stamp can use this function to have time stamps represented as floats. Whether they will actually observe non-zero fractions depends on the system.

Future Python releases will change the default of this setting; applications that cannot deal with floating point time stamps can then use this function to turn the feature off.

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_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() instead. Availability: 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() 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. Changed in version 2.0: Added support for None for *times*. Availability: Macintosh, UNIX, Windows.

walk(*top*[, *topdown=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 *dirnames* 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 os.error 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:

```
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:

```
import os
from os.path import join
# Delete everything reachable from the directory named in 'top'.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(join(root, name))
    for name in dirs:
        os.rmdir(join(root, name))
```

New in version 2.3.

6.1.5 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: UNIX, Windows.

`execl``(`*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 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 `execl*()` 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 'p' near the end (`execlp()`, `execlpe()`, `execvp()`, and `execvpe()`) 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, `execl()`, `execle()`, `execv()`, and `execve()`, will not use the PATH variable to locate the executable; *path* must contain an appropriate absolute or relative path.

For `execle()`, `execlpe()`, `execve()`, and `execvpe()` (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 `execl()`, `execlp()`, `execv()`, and `execvp()` all cause the new process to inherit the environment of the current process. Availability: UNIX, Windows.

_exit(*n*)
Exit to the system with status *n*, without calling cleanup handlers, flushing stdio buffers, etc. Availability: 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.

EX_OK
Exit code that means no error occurred. Availability: 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: UNIX. New in version 2.3.

EX_DATAERR
Exit code that means the input data was incorrect. Availability: UNIX. New in version 2.3.

EX_NOINPUT
Exit code that means an input file did not exist or was not readable. Availability: UNIX. New in version 2.3.

EX_NOUSER
Exit code that means a specified user did not exist. Availability: UNIX. New in version 2.3.

EX_NOHOST
Exit code that means a specified host did not exist. Availability: UNIX. New in version 2.3.

EX_UNAVAILABLE
Exit code that means that a required service is unavailable. Availability: UNIX. New in version 2.3.

EX_SOFTWARE
Exit code that means an internal software error was detected. Availability: 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: 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: UNIX. New in version 2.3.

EX_CANTCREAT
Exit code that means a user specified output file could not be created. Availability: UNIX. New in version 2.3.

EX_IOERR
Exit code that means that an error occurred while doing I/O on some file. Availability: 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: UNIX. New in version 2.3.

EX_PROTOCOL
Exit code that means that a protocol exchange was illegal, invalid, or not understood. Availability: 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: UNIX. New in version 2.3.

EX_CONFIG
Exit code that means that some kind of configuration error occurred. Availability: UNIX. New in version 2.3.

EX_NOTFOUND
Exit code that means something like "an entry was not found". Availability: 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: UNIX.

forkpty()
Fork a child process, using a new pseudo-terminal as the child's controlling terminal. Return a pair of (*pid*, *fd*), where *pid* is `0` in the child, the new child's process id in the parent, and *fd* is the file descriptor of the master end of the pseudo-terminal. For a more portable approach, use the `pty` module. Availability: Some flavors of UNIX.

kill(*pid*, *sig*)
Kill the process *pid* with signal *sig*. Constants for the specific signals available on the host platform are defined in the `signal` module. Availability: UNIX.

killpg(*pgid*, *sig*)
Kill the process group *pgid* with the signal *sig*. Availability: UNIX. New in version 2.3.

nice(*increment*)
Add *increment* to the process's "niceness". Return the new niceness. Availability: UNIX.

plock(*op*)
Lock program segments into memory. The value of *op* (defined in `<sys/lock.h>`) determines which segments are locked. Availability: UNIX.

popen(...)
popen2(...)
popen3(...)
popen4(...)
Run child processes, returning opened pipes for communications. These functions are described in section 6.1.2.

spawnl(*mode*, *path*, ...)
spawnle(*mode*, *path*, ..., *env*)
spawnlp(*mode*, *file*, ...)
spawnlpe(*mode*, *file*, ..., *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: 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: 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*)
Start a file with its associated application. 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.

`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.

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: 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 UNIX manual page *times*(2) or the corresponding Windows Platform API documentation. Availability: UNIX, Windows.

`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: UNIX.

`waitpid``(`*pid, options*`)`
The details of this function differ on UNIX and Windows.

On UNIX: Wait for completion of a child process given by process id *pid*, and return a tuple containing its process id and exit status indication (encoded as for `wait()`). The semantics of the call are affected by the value of the integer *options*, which should be `0` for normal operation.

If *pid* is greater than `0`, `waitpid()` requests status information for that specific process. If *pid* is `0`, the request is for the status of any child in the process group of the current process. If *pid* is `-1`, the request pertains to any child of the current process. If *pid* is less than `-1`, status is requested for any process in the process group `-`*pid* (the absolute value of *pid*).

On Windows: Wait for completion of a process given by process handle *pid*, and return a tuple containing *pid*, and its exit status shifted left by 8 bits (shifting makes cross-platform use of the function easier). A *pid* less than or equal to `0` has no special meaning on Windows, and raises an exception. The value of integer *options* has no effect. *pid* can refer to any process whose id is known, not necessarily a child process. The `spawn()` functions called with `P_NOWAIT` return suitable process handles.

`WNOHANG`
The option for `waitpid()` to avoid hanging if no child process status is available immediately. Availability: 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: 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: 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: UNIX.

WIFEXITED(*status*)
Returns `True` if the process exited using the *exit*(2) system call, otherwise it returns `False`. Availability: 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: UNIX.

WSTOPSIG(*status*)
Return the signal which caused the process to stop. Availability: UNIX.

WTERMSIG(*status*)
Return the signal which caused the process to exit. Availability: UNIX.

6.1.6 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 in the `confstr_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted. Availability: UNIX.

If the configuration value specified by *name* isn't defined, the empty string 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: 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: 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: 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 the Macintosh. 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 the Macintosh. Also available via `os.path`.

sep
The character used by the operating system to separate pathname components, for example, '`/`' for POSIX

or ':' for the Macintosh. 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 patch 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.

6.2 `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.

`expanduser`(*path*)
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; an initial '~*user*' is looked up in the password directory through the built-in module pwd. If the expansion fails, or if the path does not begin with a tilde, the path is returned unchanged. On the Macintosh, this always returns *path* 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. On the Macintosh, this always returns *path* 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.

getmtime(*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 time of creation 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 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*[, ...]])
Joins one or more path components intelligently. If any component is an absolute path, all previous components 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, e.g. `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.

realpath(*path*)
Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path. Availability: UNIX. 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 objects *fp1* and *fp2* refer to the same file. The two file objects may represent different file descriptors. 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.

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.

6.3 dircache — Cached directory listings

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:

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.

```
>>> 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', 'lib', 'lost+
found', 'mnt', 'proc', 'root', 'sbin', 'tmp', 'usr', 'var', 'vmlinuz']
>>> dircache.annotate('/', a)
>>> a
['bin/', 'boot/', 'cdrom/', 'dev/', 'etc/', 'floppy/', 'home/', 'initrd/', 'lib/
', 'lost+found/', 'mnt/', 'proc/', 'root/', 'sbin/', 'tmp/', 'usr/', 'var/', 'vm
linuz']
```

6.4 `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
Time of last status change (see manual pages 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:

```
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)
```

6.5 statcache — An optimization of os.stat()

Deprecated since release 2.2. Use os.stat() directly instead of using the cache; the cache introduces a very high level of fragility in applications using it and complicates application code with the addition of cache management support.

The statcache module provides a simple optimization to os.stat(): remembering the values of previous invocations.

The statcache module defines the following functions:

stat(*path*)
This is the main module entry-point. Identical for os.stat(), except for remembering the result for future invocations of the function.

The rest of the functions are used to clear the cache, or parts of it.

reset()
Clear the cache: forget all results of previous stat() calls.

forget(*path*)
Forget the result of stat(*path*), if any.

forget_prefix(*prefix*)
Forget all results of stat(*path*) for *path* starting with *prefix*.

forget_dir(*prefix*)
Forget all results of stat(*path*) for *path* a file in the directory *prefix*, including stat(*prefix*).

forget_except_prefix(*prefix*)
Similar to forget_prefix(), but for all *path* values *not* starting with *prefix*.

Example:

```
>>> import os, statcache
>>> statcache.stat('.')
(16893, 2049, 772, 18, 1000, 1000, 2048, 929609777, 929609777, 929609777)
>>> os.stat('.')
(16893, 2049, 772, 18, 1000, 1000, 2048, 929609777, 929609777, 929609777)
```

6.6 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_BAVAIL
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.

6.7 `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*[, *use_statcache*]])
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. Changed in version 2.3: *use_statcache* is obsolete and ignored..

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*[, *use_statcache*]])
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 *errros* 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* and *use_statcache* parameters have the same meanings and default values as for `filecmp.cmp()`.

Example:

```
>>> import filecmp
>>> filecmp.cmp('libundoc.tex', 'libundoc.tex')
True
>>> filecmp.cmp('libundoc.tex', 'lib.tex')
False
```

6.7.1 The `dircmp` class

`dircmp` instances are built using this constructor:

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 subdirctories.

report_full_closure()
Print a comparison between *a* and *b* and common subdirctories (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 lazilly, 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.

6.8 popen2 — Subprocesses with accessible I/O streams

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'.

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.

popen2(*cmd*[, *bufsize*[, *mode*]])
Executes *cmd* as a sub-process. Returns the file objects (*child_stdout*, *child_stdin*).

popen3(*cmd*[, *bufsize*[, *mode*]])
Executes *cmd* as a sub-process. Returns the file objects (*child_stdout*, *child_stdin*, *child_stderr*).

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.

6.8.1 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 6.1.5 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
Where the standard error from the child process goes is *capturestderr* was true for the constructor, or `None`. This will always be `None` for `Popen4` instances.

pid
The process ID of the child process.

6.8.2 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:

```
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:

```
import os
import sys

# note that each of these print statements
# writes a single long string

print >>sys.stderr, 400 * 'this is a test\n'
os.close(sys.stderr.fileno())
print >>sys.stdout, 400 * 'this is another test\n'
```

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.

6.9 `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 meters, 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` (section 5.18):
General calendar related functions.

Module `time` (section 6.10):
Time access and conversions.

6.9.1 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
```

6.9.2 `timedelta` Objects

A `timedelta` object represents a duration, the difference between two dates or times.

class `timedelta`(*days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0*)
All arguments are optional. 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 <=` *microseconds* `< 1000000`
- `0 <=` *seconds* `< 3600*24` (the number of seconds in one day)
- `-999999999 <=` *days* `<= 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,

```
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

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):

Attribute	Value
`days`	Between -999999999 and 999999999 inclusive
`seconds`	Between 0 and 86399 inclusive
`microseconds`	Between 0 and 999999 inclusive

Supported operations:

Operation	Result
t1 `=` *t2* `+` *t3*	Sum of *t2* and *t3*. Afterwards *t1*-*t2* == *t3* and *t1*-*t3* == *t2* are true. (1)
t1 `=` *t2* `-` *t3*	Difference of *t2* and *t3*. Afterwards *t1* == *t2* - *t3* and *t2* == *t1* + *t3* are true. (1)
t1 `=` *t2* `*` *i* or *t1* `=` *i* `*` *t2*	Delta multiplied by an integer or long. Afterwards *t1* // i == *t2* is true, provided `i != 0`. In general, *t1* * i == *t1* * (i-1) + *t1* is true. (1)
t1 `=` *t2* `//` *i*	The floor is computed and the remainder (if any) is thrown away. (3)
`+`*t1*	Returns a `timedelta` object with the same value. (2)
`-`*t1*	equivalent to `timedelta`(*-t1.days*, *-t1.seconds*, *-t1.microseconds*), and to *t1** -1. (1)(4)
`abs(`*t*`)`	equivalent to +*t* when `t.days >= 0`, and to -*t* when `t.days < 0`. (2)

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)`.

6.9.3 `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.

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:

`today`()
Return the current local date. This is equivalent to `date.fromtimestamp(time.time())`.

`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`(*ordinal*)
Return the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. `ValueError` is raised unless `1 <=` *ordinal* `<= date.max.toordinal()`. For any date *d*, `date.fromordinal(`*d*`.toordinal()) ==` *d*.

Class attributes:

`min`
The earliest representable date, `date(MINYEAR, 1, 1)`.

`max`
The latest representable date, `date(MAXYEAR, 12, 31)`.

`resolution`
The smallest possible difference between non-equal date objects, `timedelta(days=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.

Supported operations:

Operation	Result
date2 = *date1* + *timedelta*	*date2* is *timedelta*.days days removed from *date1*. (1)
date2 = *date1* - *timedelta*	Computes *date2* such that *date2* + *timedelta* == *date1*. (2)
timedelta = *date1* - *date2*	(3)
date1<*date2*	*date1* is considered less than *date2* when *date1* precedes *date2* in time. (4)

Notes:

(1) *date2* is moved forward in time if *timedelta*.days > 0, or backward if *timedelta*.days < 0. Afterward *date2* - *date1* == *timedelta*.days. *timedelta*.seconds and *timedelta*.microseconds are ignored. OverflowError is raised if *date2*.year would be smaller than MINYEAR or larger than MAXYEAR.

(2) This isn't quite equivalent to date1 + (-timedelta), because -timedelta in isolation can overflow in cases where date1 - timedelta does not. *timedelta*.seconds and *timedelta*.microseconds are ignored.

(3) This is exact, and cannot overflow. timedelta.seconds and timedelta.microseconds are 0, and date2 + timedelta == date1 after.

(4) In other words, date1 < date2 if and only if *date1*.toordinal() < *date2*.toordinal(). In order to stop comparison from falling back to the default scheme of comparing object addresses, date comparison normally raises TypeError if the other comparand isn't also a date object. However, NotImplemented 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 date 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.

Dates can be used as dictionary keys. In Boolean contexts, all date objects are considered to be true.

Instance methods:

replace(*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(2000, 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.

For example, 2004 begins on a Thursday, so the first week of ISO year 2004 begins on Monday, 29 Dec 2003 and ends on Sunday, 4 Jan 2004, so that `date(2003, 12, 29).isocalendar() == (2004, 1, 1)` and `date(2004, 1, 4).isocalendar() == (2004, 1, 7)`.

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 the section on `strftime()` behavior.

6.9.4 `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=0, minute=0, second=0, microsecond=0, tzinfo=None*)
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=None)()
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 `time.time()` timestamp (for example, this may be possible on platforms supplying the C `gettimeofday()` function).

Else *tz* must be an instance of a class `tzinfo` subclass, and the current date and time are converted to *tz*'s time zone. In this case the result is equivalent to *tz*`.fromutc(datetime.utcnow().replace(tzinfo=`*tz*`))`. See also `today()`, `utcnow()`.

`utcnow``()`
Return the current UTC date and time, with `tzinfo` `None`. This is like `now()`, but returns the current UTC date and time, as a naive `datetime` object. See also `now()`.

`fromtimestamp``(`*timestamp, tz=None*`)`
Return the local date and time corresponding to the POSIX timestamp, such as is returned by `time.time()`. If optional argument *tz* is `None` or not specified, the timestamp is converted to the platform's local date and time, and the returned `datetime` object is naive.

Else *tz* must be an instance of a class `tzinfo` subclass, and the timestamp is converted to *tz*'s time zone. In this case the result is equivalent to *tz*`.fromutc(datetime.utcfromtimestamp(`*timestamp*`).replace(tzinfo=`*tz*`))`.

`fromtimestamp()` may raise `ValueError`, if the timestamp is out of the range of values supported by the platform C `localtime()` or `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 `fromtimestamp()`, and then it's possible to have two timestamps differing by a second that yield identical `datetime` objects. See also `utcfromtimestamp()`.

`utcfromtimestamp``(`*timestamp*`)`
Return the UTC `datetime` corresponding to the POSIX timestamp, with `tzinfo` `None`. This may raise `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.

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:

Operation	Result
datetime2 `=` *datetime1* `+` *timedelta*	(1)
datetime2 `=` *datetime1* `-` *timedelta*	(2)
timedelta `=` *datetime1* `-` *datetime2*	(3)
datetime1 `<` *datetime2*	Compares `datetime` to `datetime`. (4)

(1) 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.

(2) 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.

(3) 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.

(4) *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, `NotImplemented` 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.struct_time((*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: if 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='T'*)
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,

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt): return timedelta(minutes=-399)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

__str__()
For a `datetime` instance *d*, `str(`*d*`)` is equivalent to *d*`.isoformat(' ')`.

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 the section on `strftime()` behavior.

6.9.5 `time` Objects

A time object represents a (local) time of day, independent of any particular day, and subject to adjustment via a `tzinfo` object.

class time(*hour=0, minute=0, second=0, microsecond=0, tzinfo=None*)
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 <= hour < 24`
- `0 <= minute < 60`
- `0 <= second < 60`
- `0 <= microsecond < 1000000`.

If an argument outside those ranges is given, `ValueError` is raised.

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`
In `range(60)`.

`second`
In `range(60)`.

`microsecond`
In `range(1000000)`.

`tzinfo`
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 the section on `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.

6.9.6 `tzinfo` Objects

`tzinfo` is an abstract base clase, 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`.

dst(*self, dt*)
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:

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:

```
  return timedelta(0)   # a fixed-offset class:  doesn't account for DST
```

or

```
  # Code to set dston and dstoff 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:

`fromutc`(*self, 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:

```
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
```

Example `tzinfo` classes:

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)

# A UTC class.

class UTC(tzinfo):
    """UTC"""

    def utcoffset(self, dt):
        return ZERO

    def tzname(self, dt):
        return "UTC"

    def dst(self, dt):
        return ZERO

utc = UTC()

# A class building tzinfo objects for fixed-offset time zones.
# Note that FixedOffset(0, "UTC") is a different way to build a
# UTC tzinfo object.

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
```

```
# 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 Oct.
# 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 cases.
                # It depends on how you want to treat them.  The default
                # fromutc() implementation (called by the default astimezone()
                # implementation) passes a datetime with dt.tzinfo is self.
                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 from
            # 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:

```
      UTC   3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
      EST  22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
      EDT  23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

    start  22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

      end  23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

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 `rzinfo.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)).

6.9.7 `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.

6.10 `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.

Index	Attribute	Values
0	`tm_year`	(for example, 1993)
1	`tm_mon`	range [1,12]
2	`tm_mday`	range [1,31]
3	`tm_hour`	range [0,23]
4	`tm_min`	range [0,59]
5	`tm_sec`	range [0,61]; see **(1)** in `strftime()` description
6	`tm_wday`	range [0,6], Monday is 0
7	`tm_yday`	range [1,366]
8	`tm_isdst`	0, 1 or -1; see below

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 `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 `struct_time`, or having elements of the wrong type, a `TypeError` is raised.

Changed in version 2.2: The time value sequence was changed from a tuple to a `struct_time`, with the addition of attribute names for the fields.

The module defines the following functions and data items:

`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.

`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 `daylight` is nonzero.

`asctime``([`*t*`])`
Convert a tuple or `struct_time` representing a time as returned by `gmtime()` or `localtime()` to a 24-character string of the following form: `'Sun Jun 20 23:21:05 1993'`. If *t* is not provided, the

current time as returned by `localtime()` is used. Locale information is not used by `asctime()`. **Note:** Unlike the C function of the same name, there is no trailing newline. Changed in version 2.1: Allowed *t* to be omitted.

`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.

`ctime``(`[*secs*]`)`
Convert a time expressed in seconds since the epoch to a string representing local time. If *secs* is not provided, the current time as returned by `time()` is used. `ctime(`*secs*`)` is equivalent to `asctime(localtime(`*secs*`))`. Locale information is not used by `ctime()`. Changed in version 2.1: Allowed *secs* to be omitted.

`daylight`
Nonzero if a DST timezone is defined.

`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, 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. Changed in version 2.1: Allowed *secs* to be omitted.

`localtime``(`[*secs*]`)`
Like `gmtime()` but converts to local time. The dst flag is set to `1` when DST applies to the given time. Changed in version 2.1: Allowed *secs* to be omitted.

`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. Changed in version 2.1: Allowed *t* to be omitted.

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:

Directive	Meaning	Notes
%a	Locale's abbreviated weekday name.	
%A	Locale's full weekday name.	
%b	Locale's abbreviated month name.	
%B	Locale's full month name.	
%c	Locale's appropriate date and time representation.	
%d	Day of the month as a decimal number [01,31].	
%H	Hour (24-hour clock) as a decimal number [00,23].	
%I	Hour (12-hour clock) as a decimal number [01,12].	
%j	Day of the year as a decimal number [001,366].	
%m	Month as a decimal number [01,12].	
%M	Minute as a decimal number [00,59].	
%p	Locale's equivalent of either AM or PM.	
%S	Second as a decimal number [00,61].	(1)
%U	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.	
%w	Weekday as a decimal number [0(Sunday),6].	
%W	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.	
%x	Locale's appropriate date representation.	
%X	Locale's appropriate time representation.	
%y	Year without century as a decimal number [00,99].	
%Y	Year with century as a decimal number.	
%Z	Time zone name (no characters if no time zone exists).	
%%	A literal '%' character.	

Notes:

(1) The range really is `0` to `61`; this accounts for leap seconds and the (very rare) double leap seconds.

Here is an example, a format for dates compatible with that specified in the RFC 2822 Internet email standard. [1]

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", 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 `strftime()`; 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 is `(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).

[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.

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)

std offset [dst [offset[,start[/time], end[/time]]]]

Where:

std and dstThree or more alphanumerics giving the timezone abbreviations. These will be propogated into time.tzname

offsetThe 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 east of the Prime Meridian; otherwise, it is west. If no offset follows dst, summmer 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:

J*n*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.

M*m.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'`.

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

See Also:

Module `locale` (section 6.26):
Internationalization services. The locale settings can affect the return values for some of the functions in the `time` module.

Module `calendar` (section 5.18):
General calendar-related functions. `timegm()` is the inverse of `gmtime()` from this module.

6.11 sched — Event scheduler

The `sched` module defines a class which implements a general purpose event scheduler:

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:

```
>>> 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
```

6.11.1 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.

6.12 `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.

6.12.1 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.

6.13 getpass — Portable password input

The getpass module provides two functions:

getpass([*prompt*])
Prompt the user for a password without echoing. The user is prompted using the string *prompt*, which defaults to 'Password: '. Availability: Macintosh, UNIX, Windows.

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.

6.14 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 (section 6.17):
Utilities for working with ASCII characters, regardless of your locale settings.

Module curses.panel (section 6.18):
A panel stack extension that adds depth to curses windows.

Module curses.textpad (section 6.15):
Editable text widget for curses supporting **Emacs**-like bindings.

Module curses.wrapper (section 6.16):
Convenience function to ensure proper terminal setup and resetting on application entry and exit.

Curses Programming with Python
(http://www.python.org/doc/howto/curses/curses.html)
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.

6.14.1 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 cgaracter-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 call 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: BUTTON*n*_PRESSED, BUTTON*n*_RELEASED, BUTTON*n*_CLICKED, BUTTON*n*_DOUBLE_CLICKED, BUTTON*n*_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``(`*yes*`)`
If *yes* is 1, allow 8-bit characters to be input. If *yes* is 0, allow only 7-bit chars.

`mouseinterval``(`*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``(`*mousemask*`)`
Sets the mouse events to be reported, and returns a tuple `(`*availmask*`,` *oldmask*`)`. *availmask* indicates which of the specified mouse events can be reported; on complete failure it returns 0. *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``(`*ms*`)`
Sleep for *ms* milliseconds.

`newpad``(`*nlines, 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.

`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.

`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.

`nocbreak``()`
Leave cbreak mode. Return to normal "cooked" mode with line buffering.

noecho()
Leave echo mode. Echoing of input characters is turned off,

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 `0` 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).

6.14.2 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 then a Python character (a string of length 1). (This note is true whenever the documentation mentions a character.) The builtin `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(*ch*[, *attr*])
Sets the background property of the window to the character *ch*, with attributes *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.

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.

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.

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:

Parameter	Description	Default value
ls	Left side	`ACS_VLINE`
rs	Right side	`ACS_VLINE`
ts	Top	`ACS_HLINE`
bs	Bottom	`ACS_HLINE`
tl	Upper-left corner	`ACS_ULCORNER`
tr	Upper-right corner	`ACS_URCORNER`
bl	Bottom-left corner	`ACS_BLCORNER`
br	Bottom-right corner	`ACS_BRCORNER`

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([*x*, *y*])
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 *ch* with attribute *attr*, and immediately call `refresh()` on the window.

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.

erase()
Clear the window.

getbegyx()
Return a tuple `(`*y*`, `*x*`)` of co-ordinates of upper-left corner.

getch([*x*, *y*])
Get a character. Note that the integer returned does *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.

getkey([*x*, *y*])
Get a character, returning a string instead of an integer, as `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.

getmaxyx()
Return a tuple `(`*y*`, `*x*`)` of the height and width of the window.

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.

getstr([*x*, *y*])
Read a string from the user, with primitive line editing capacity.

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 `refresh()` yourself. However, it may degrade performance considerably, due to repeated calls to wrefresh. This option is disabled by default.

inch([*x*, *y*])
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).

insstr([*y*, *x*,] *str* [, *attr*])
Insert a character string (as many characters as will fit on the line) before the character under the cursor. 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.

scroll([*lines = 1*])
Scroll the screen or scrolling region upward by *lines* lines.

scrollok(*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()`.

setscrreg(*top, bottom*)
Set the scrolling region from line *top* to line *bottom*. All scrolling actions will take place in this region.

standend()
Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

standout()
Turn on attribute *A_STANDOUT*.

subpad([*nlines, ncols,*] *begin_y, begin_x*)
Return a sub-window, whose upper-left corner is at (*begin_y, begin_x*), and whose width/height is *ncols*/*nlines*.

subwin([*nlines, ncols,*] *begin_y, 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 *ch*.

6.14.3 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:

Attribute	Meaning
A_ALTCHARSET	Alternate character set mode.
A_BLINK	Blink mode.
A_BOLD	Bold mode.
A_DIM	Dim mode.
A_NORMAL	Normal attribute.
A_STANDOUT	Standout mode.
A_UNDERLINE	Underline mode.

Keys are referred to by integer constants with names starting with '`KEY_`'. The exact keycaps available are system dependent.

Key constant	Key
KEY_MIN	Minimum key value
KEY_BREAK	Break key (unreliable)
KEY_DOWN	Down-arrow
KEY_UP	Up-arrow
KEY_LEFT	Left-arrow
KEY_RIGHT	Right-arrow
KEY_HOME	Home key (upward+left arrow)
KEY_BACKSPACE	Backspace (unreliable)
KEY_F0	Function keys. Up to 64 function keys are supported.
KEY_F*n*	Value of function key *n*
KEY_DL	Delete line
KEY_IL	Insert line

Key constant	Key
KEY_DC	Delete character
KEY_IC	Insert char or enter insert mode
KEY_EIC	Exit insert char mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backward (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send (unreliable)
KEY_SRESET	Soft (partial) reset (unreliable)
KEY_RESET	Reset or hard reset (unreliable)
KEY_PRINT	Print
KEY_LL	Home down or bottom (lower left)
KEY_A1	Upper left of keypad
KEY_A3	Upper right of keypad
KEY_B2	Center of keypad
KEY_C1	Lower left of keypad
KEY_C3	Lower right of keypad
KEY_BTAB	Back tab
KEY_BEG	Beg (beginning)
KEY_CANCEL	Cancel
KEY_CLOSE	Close
KEY_COMMAND	Cmd (command)
KEY_COPY	Copy
KEY_CREATE	Create
KEY_END	End
KEY_EXIT	Exit
KEY_FIND	Find
KEY_HELP	Help
KEY_MARK	Mark
KEY_MESSAGE	Message
KEY_MOVE	Move
KEY_NEXT	Next
KEY_OPEN	Open
KEY_OPTIONS	Options
KEY_PREVIOUS	Prev (previous)
KEY_REDO	Redo
KEY_REFERENCE	Ref (reference)
KEY_REFRESH	Refresh
KEY_REPLACE	Replace
KEY_RESTART	Restart
KEY_RESUME	Resume
KEY_SAVE	Save
KEY_SBEG	Shifted Beg (beginning)
KEY_SCANCEL	Shifted Cancel
KEY_SCOMMAND	Shifted Command
KEY_SCOPY	Shifted Copy
KEY_SCREATE	Shifted Create
KEY_SDC	Shifted Delete char
KEY_SDL	Shifted Delete line
KEY_SELECT	Select
KEY_SEND	Shifted End

Key constant	Key
KEY_SEOL	Shifted Clear line
KEY_SEXIT	Shifted Dxit
KEY_SFIND	Shifted Find
KEY_SHELP	Shifted Help
KEY_SHOME	Shifted Home
KEY_SIC	Shifted Input
KEY_SLEFT	Shifted Left arrow
KEY_SMESSAGE	Shifted Message
KEY_SMOVE	Shifted Move
KEY_SNEXT	Shifted Next
KEY_SOPTIONS	Shifted Options
KEY_SPREVIOUS	Shifted Prev
KEY_SPRINT	Shifted Print
KEY_SREDO	Shifted Redo
KEY_SREPLACE	Shifted Replace
KEY_SRIGHT	Shifted Right arrow
KEY_SRSUME	Shifted Resume
KEY_SSAVE	Shifted Save
KEY_SSUSPEND	Shifted Suspend
KEY_SUNDO	Shifted Undo
KEY_SUSPEND	Suspend
KEY_UNDO	Undo
KEY_MOUSE	Mouse event has occurred
KEY_RESIZE	Terminal resize event
KEY_MAX	Maximum key value

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 keybboard, 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:

Keycap	Constant
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_NPAGE
Page Down	KEY_PPAGE

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.

ACS code	Meaning
ACS_BBSS	alternate name for upper right corner
ACS_BLOCK	solid square block
ACS_BOARD	board of squares
ACS_BSBS	alternate name for horizontal line
ACS_BSSB	alternate name for upper left corner
ACS_BSSS	alternate name for top tee
ACS_BTEE	bottom tee
ACS_BULLET	bullet

ACS code	Meaning
ACS_CKBOARD	checker board (stipple)
ACS_DARROW	arrow pointing down
ACS_DEGREE	degree symbol
ACS_DIAMOND	diamond
ACS_GEQUAL	greater-than-or-equal-to
ACS_HLINE	horizontal line
ACS_LANTERN	lantern symbol
ACS_LARROW	left arrow
ACS_LEQUAL	less-than-or-equal-to
ACS_LLCORNER	lower left-hand corner
ACS_LRCORNER	lower right-hand corner
ACS_LTEE	left tee
ACS_NEQUAL	not-equal sign
ACS_PI	letter pi
ACS_PLMINUS	plus-or-minus sign
ACS_PLUS	big plus sign
ACS_RARROW	right arrow
ACS_RTEE	right tee
ACS_S1	scan line 1
ACS_S3	scan line 3
ACS_S7	scan line 7
ACS_S9	scan line 9
ACS_SBBS	alternate name for lower right corner
ACS_SBSB	alternate name for vertical line
ACS_SBSS	alternate name for right tee
ACS_SSBB	alternate name for lower left corner
ACS_SSBS	alternate name for bottom tee
ACS_SSSB	alternate name for left tee
ACS_SSSS	alternate name for crossover or big plus
ACS_STERLING	pound sterling
ACS_TTEE	top tee
ACS_UARROW	up arrow
ACS_ULCORNER	upper left corner
ACS_URCORNER	upper right corner
ACS_VLINE	vertical line

The following table lists the predefined colors:

Constant	Color
COLOR_BLACK	Black
COLOR_BLUE	Blue
COLOR_CYAN	Cyan (light greenish blue)
COLOR_GREEN	Green
COLOR_MAGENTA	Magenta (purplish red)
COLOR_RED	Red
COLOR_WHITE	White
COLOR_YELLOW	Yellow

6.15 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.

6.15.1 Textbox objects

You can instantiate a `Textbox` object as follows:

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 containin window, with coordinates `(0, 0)`. The instance's `stripspaces` flag is initially on.

`Textbox` objects have the following methods:

`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.

`do_command`(*ch*)

Process a single command keystroke. Here are the supported special keystrokes:

Keystroke	Action
`Control-A`	Go to left edge of window.
`Control-B`	Cursor left, wrapping to previous line if appropriate.
`Control-D`	Delete character under cursor.
`Control-E`	Go to right edge (stripspaces off) or end of line (stripspaces on).
`Control-F`	Cursor right, wrapping to next line when appropriate.
`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:

Constant	Keystroke
`KEY_LEFT`	`Control-B`
`KEY_RIGHT`	`Control-F`
`KEY_UP`	`Control-P`
`KEY_DOWN`	`Control-N`
`KEY_BACKSPACE`	`Control-h`

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 is gathered.

6.16 `curses.wrapper` — Terminal handler for curses programs

New in version 1.6.

This module supplies one function, `wrapper()`, which runs another function which should be the rest of your curses-using application. If the application raises an exception, `wrapper()` will restore the terminal to a sane state before passing it further up the stack and generating a traceback.

wrapper(*func*, ...)
: Wrapper function that initializes curses and calls another function, *func*, restoring normal keyboard/screen behavior on error. The callable object *func* is then passed the main window 'stdscr' as its first argument, followed by any other arguments passed to `wrapper()`.

Before calling the hook function, `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.

6.17 `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:

Name	Meaning
NUL	
SOH	Start of heading, console interrupt
STX	Start of text
ETX	End of text
EOT	End of transmission
ENQ	Enquiry, goes with ACK flow control
ACK	Acknowledgement
BEL	Bell
BS	Backspace
TAB	Tab
HT	Alias for TAB: “Horizontal tab”
LF	Line feed
NL	Alias for LF: “New line”
VT	Vertical tab
FF	Form feed
CR	Carriage return
SO	Shift-out, begin alternate character set
SI	Shift-in, resume default character set
DLE	Data-link escape
DC1	XON, for flow control
DC2	Device control 2, block-mode flow control
DC3	XOFF, for flow control
DC4	Device control 4
NAK	Negative acknowledgement
SYN	Synchronous idle
ETB	End transmission block
CAN	Cancel
EM	End of medium
SUB	Substitute
ESC	Escape
FS	File separator
GS	Group separator
RS	Record separator, block-mode terminator
US	Unit separator
SP	Space
DEL	Delete

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.

ascii(*c*)
Return the ASCII value corresponding to the low 7 bits of *c*.

ctrl(*c*)
Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

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.

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.

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.

6.18 `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.

6.18.1 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*.

`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.

6.18.2 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.

6.19 `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, it *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.

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:

```
>>> 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')]
>>> args
['a1', 'a2']
```

Using long option names is equally easy:

```
>>> 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'), ('-x',
 '')]
>>> args
['a1', 'a2']
```

In a script, typical usage is something like this:

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    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()
```

6.20 `optparse` — Powerful parser for command line options.

New in version 2.3.

The `optparse` module is a powerful, flexible, extensible, easy-to-use command-line parsing library for Python. Using `optparse`, you can add intelligent, sophisticated handling of command-line options to your scripts with very little overhead.

Here's an example of using `optparse` to add some command-line options to a simple script:

```
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:

```
$ <yourscript> -f outfile --quiet
$ <yourscript> -qfoutfile
$ <yourscript> --file=outfile -q
$ <yourscript> --quiet --file outfile
```

(All of these result in `options.filename == "outfile"` and `options.verbose == False`, just as you might expect.)

Even niftier, 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
  -fFILE, --file=FILE  write report to FILE
  -q, --quiet          don't print status messages to stdout
```

That's just a taste of the flexibility `optparse` gives you in parsing your command-line.

6.20.1 Philosophy

The purpose of `optparse` is to make it very easy to provide the most standard, obvious, straightforward, and user-friendly user interface for UNIX command-line programs. The `optparse` philosophy is heavily influenced by the UNIX and GNU toolkits, and this section is meant to explain that philosophy.

Terminology

First, we need to establish some terminology.

argument
: a chunk of text that a user enters on the command-line, and that the shell passes to `execl()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]`. (`sys.argv[0]` is the name of the program being executed; in the context of parsing arguments, it's not very important.) UNIX shells also use the term "word".

  It is occasionally desirable to use 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 **-** 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**.

  `optparse` does not support these option syntaxes, and it never will. (If you really want to use one of those option syntaxes, you'll have to subclass `OptionParser` and override all the difficult bits. But please don't! `optparse` does things the traditional UNIX/GNU way deliberately; the first three are non-standard anywhere, and the last one makes sense only if you're exclusively targeting MS-DOS/Windows and/or VMS.)

option argument
: an argument that follows an option, is closely associated with that option, and is consumed from the argument list when the option is. Often, option arguments may also be included in the same argument as the option, e.g. :

```
["-f", "foo"]
```

  may be equivalent to:

```
["-ffoo"]
```

  (`optparse` supports this syntax.)

  Some options never take an argument. Some options always take an argument. 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** and **-b** are both options, and **-a** takes an optional argument, how do we interpret **-ab**? `optparse` does not support optional option arguments.

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. The phrase "required option" is an oxymoron; the presence of "required options" in a program is usually a sign of careless user interface design. `optparse` doesn't prevent you from implementing required options, but doesn't give you much help with it either. See "Extending Examples" (section 6.20.5) 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 the **--report** option takes one argument, `/tmp/report.txt` is an option argument. `foo` and `bar` are positional arguments.

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 should be *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 only exceptions I can think of 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. (However, if you insist on adding "required options" to your programs, look in "Extending Examples" (section 6.20.5) for two ways of implementing them with `optparse`.)

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 rely on 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 one file to another, or N files to another directory.

What are positional arguments for?

In case it wasn't clear from the above example: 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, a GUI, or whatever: if you make that many demands on your users, most of them will just 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, checkboxes 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. It's quite easy to overwhelm users (and yourself!) with too much flexibility, so be careful there.

6.20.2 Basic Usage

While `optparse` is quite flexible and powerful, you don't have to jump through hoops or read reams of documentation to get it working in basic cases. This document aims to demonstrate some simple usage patterns that will get you started using `optparse` in your scripts.

To parse a command line with `optparse`, you must create an `OptionParser` instance and populate it. Obviously, you'll have to import the `OptionParser` classes in any script that uses `optparse`:

```
from optparse import OptionParser
```

Early on in the main program, create a parser:

```
parser = OptionParser()
```

Then you can start populating the parser with options. Each option is really a set of synonymous option strings; most commonly, you'll have one short option string and one long option string — e.g. **-f** and **--file**:

```
parser.add_option("-f", "--file", ...)
```

The interesting stuff, of course, is what comes after the option strings. For now, we'll only cover four of the things you can put there: *action*, *type*, *dest* (destination), and *help*.

The *store* action

The action tells `optparse` what to do when it sees one of the option strings for this option on the command-line. For example, the action *store* means: 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, let's fill in the "..." of that last option:

```
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:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

(Note that if you don't pass an argument list to `parse_args()`, it automatically uses `sys.argv[1:]`.)

When `optparse` sees the **-f**, it consumes the next argument—`foo.txt`—and stores it in the `filename` attribute of a special object. That object is the first return value from `parse_args()`, so:

```
print options.filename
```

will print `foo.txt`.

Other option types supported by `optparse` are `int` and `float`. Here's an option that expects an integer argument:

```
parser.add_option("-n", type="int", dest="num")
```

This example doesn't provide a long option, which is perfectly acceptable. It also doesn't specify the action—it defaults to "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).

```
(options, args) = parser.parse_args(["-n42"])
print options.num
```

This prints `42`.

Trying out the "float" type is left as an exercise for the reader.

If you don't specify a type, `optparse` assumes "string". Combined with the fact that the default action is "store",

that means our first example can be a lot shorter:

```
parser.add_option("-f", "--file", dest="filename")
```

If you don't supply a destination, `optparse` figures out a sensible default from the option strings: if the first long option string is **--foo-bar**, then the default destination is `foo_bar`. If there are no long option strings, `optparse` looks at the first short option: the default destination for **-f** is `f`.

Adding types is fairly easy; please refer to section 6.20.5, "Adding new types."

Other *store_** actions

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` sees **-v** on the command line, it sets `options.verbose` to `True`; when it sees **-q**, it sets `options.verbose` to `False`.

Setting 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`. Sometimes, this is just fine (which is why it's the default), but sometimes, you want more control. To address that need, `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:

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Oddly enough, this is exactly equivalent:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Those are equivalent because you're supplying a default value for the option's *destination*, and these two options happen to have the same destination (the `verbose` variable).

Consider this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Again, the default value for `verbose` will be `True`: the last default value supplied for any particular destination

is the one that counts.

Generating help

The last feature that you will use in every script is `optparse`'s ability to generate help messages. All you have to do is supply a *help* argument when you add an option. Let's create a new parser and populate it with user-friendly (documented) options:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--file", dest="filename",
                  metavar="FILE", help="write output to FILE"),
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: one of 'novice', "
                       "'intermediate' [default], 'expert'")
```

If `optparse` encounters either **-h** or **--help** on the command-line, or if you just call `parser.print_help()`, it prints the following to stdout:

```
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)
  -fFILE, --file=FILE  write output to FILE
  -mMODE, --mode=MODE  interaction mode: one of 'novice', 'intermediate'
                       [default], 'expert'
```

There's a lot going on here to help `optparse` generate the best possible help message:

- 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 script, 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:

  ```
  -mMODE, --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:

```
-fFILE, --file=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 formal syntax “-fFILE” 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.

When dealing with many options, it is convenient to group these options for better help output. An `OptionParser` can contain several option groups, each of which can contain several options.

Continuing with the parser defined above, adding an `OptionGroup` to a parser is easy:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

This would result in the following help output:

```
usage:  [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)
  -fFILE, --file=FILE   write output to FILE
  -mMODE, --mode=MODE   interaction mode: one of ’novice’, ’intermediate’
                        [default], ’expert’

  Dangerous Options:
    Caution: use of these options is at your own risk.  It is believed that
    some of them bite.
    -g                  Group option.
```

Print a version number

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`:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

version can contain anything you like; `%prog` is expanded in *version* just as with *usage*. 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, a user might do:

```
$ /usr/bin/foo --version
foo 1.0
```

Error-handling

The one thing you need to know for basic usage is how `optparse` behaves when it encounters an error on the command-line—e.g. **-n 4x** where **-n** is an integer-valued option. In this case, `optparse` prints your usage message to stderr, followed by a useful and human-readable error message. Then it terminates (calls `sys.exit()`) with a non-zero exit status.

If you don't like this, subclass `OptionParser` and override the `error()` method. See section 6.20.5, "Extending `optparse`."

Putting it all together

Here's what `optparse`-based scripts typically look like:

```
from optparse import OptionParser
[...]
def main():
    usage = "usage: \%prog [-f] [-v] [-q] firstarg secondarg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", type="string", 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
    [... go to work ...]

if __name__ == "__main__":
    main()
```

6.20.3 Advanced Usage

Creating and populating the parser

There are several ways to populate the parser with options. One way is to pass a list of `Options` to the `OptionParser` constructor:

```
from optparse import OptionParser, make_option
[...]
parser = OptionParser(option_list=[
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose")])
```

(`make_option()` is a factory function for generating `Option` objects.)

For long option lists, it may be more convenient/readable to create the list separately:

```
option_list = [make_option("-f", "--filename",
                           action="store", type="string", dest="filename"),
               [... more options ...]
               make_option("-q", "--quiet",
                           action="store_false", dest="verbose")]
parser = OptionParser(option_list=option_list)
```

Or, you can use the `add_option()` method of `OptionParser` to add options one-at-a-time:

```
parser = OptionParser()
parser.add_option("-f", "--filename",
                  action="store", type="string", dest="filename")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose")
```

This method makes it easier to track down exceptions raised by the `Option` constructor, which are common because of the complicated interdependencies among the various keyword arguments. (If you get it wrong, `optparse` raises `OptionError`.)

`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 (shown above).

Defining options

Each `Option` instance represents a set of synonymous command-line options, i.e. options that have the same meaning and effect, but different spellings. You can specify any number of short or long option strings, but you must specify at least one option string.

To define an option with only a short option string:

```
make_option("-f", ...)
```

And to define an option with only a long option string:

```
make_option("--foo", ...)
```

The "..." represents a set of keyword arguments that define attributes of the `Option` object. The rules governing which keyword args you must supply for a given `Option` are fairly complicated, but you always have to supply

some. If you get it wrong, `optparse` raises an `OptionError` exception explaining your mistake.

The most important attribute of an option is its action, i.e. what to do when we encounter this option on the command-line. The possible actions are:

Action	Meaning
`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
`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* keywords; see below.)

As you can see, most actions involve storing or updating a value somewhere. `optparse` always creates a particular object (an instance of the `Values` class) specifically for this purpose. Option arguments (and various other values) are stored as attributes of this object, according to the *dest* (destination) argument to `make_option()`/`add_option()`.

For example, when you call:

```
parser.parse_args()
```

one of the first things `optparse` does is create a `values` object:

```
values = Values()
```

If one of the options in this parser is defined with:

```
make_option("-f", "--file", action="store", type="string", dest="filename")
```

and the command-line being parsed includes any of the following:

```
-ffoo
-f foo
--file=foo
--file foo
```

then `optparse`, on seeing the **-f** or **--file** option, will do the equivalent of this:

```
values.filename = "foo"
```

Clearly, the *type* and *dest* arguments are almost as important as *action*. *action* is the only attribute that is meaningful for *all* options, though, so it is the most important.

Option actions

The various option actions all have slightly different requirements and effects. Except for the "help" action, you must supply at least one other keyword argument when creating the `Option`; the exact requirements for each action are listed here.

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 section 6.20.3, "Option types," below.

If *choices* (a sequence of strings) is supplied, 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 strings (e.g., **--foo-bar** becomes `foo_bar`). If there are no long option strings, `optparse` derives a destination from the first short option string (e.g., **-f** becomes `f`).

Example:

```
make_option("-f")
make_option("-p", type="float", nargs=3, dest="point")
```

Given the following command line:

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

`optparse` will set:

```
values.f = "bar.txt"
values.point = (1.0, -3.5, 4.0)
```

(Actually, `values.f` will be set twice, but only the second time is visible in the end.)

store_const

[required: *const*, *dest*]

The *const* value supplied to the `Option` constructor is stored in *dest*.

Example:

```
make_option("-q", "--quiet",
       action="store_const", const=0, dest="verbose"),
make_option("-v", "--verbose",
       action="store_const", const=1, dest="verbose"),
make_option("--noisy",
       action="store_const", const=2, dest="verbose"),
```

If **--noisy** is seen, `optparse` will set:

```
values.verbose = 2
```

store_true

[required: *dest*]

A special case of "store_const" that stores `True` to *dest*.

store_false

[required: *dest*]

Like "store_true", but stores `False`

Example:

```
make_option(None, "--clobber", action="store_true", dest="clobber")
make_option(None, "--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 (i.e. the default is `None`), 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:

```
make_option("-t", "--tracks", action="append", type="int")
```

If **-t3** is seen on the command-line, `optparse` does the equivalent of:

```
values.tracks = []
values.tracks.append(int("3"))
```

If, a little later on, **--tracks=4** is seen, it does:

```
values.tracks.append(int("4"))
```

See "Error handling" (section 6.20.2) for information on how `optparse` deals with something like **--tracks=x**.

count

[required: *dest*]

Increment the integer stored at *dest*. *dest* is set to zero before being incremented the first time (unless you supply a default value).

Example:

```
make_option("-v", action="count", dest="verbosity")
```

The first time **-v** is seen on the command line, `optparse` does the equivalent of:

```
values.verbosity = 0
values.verbosity += 1
```

Every subsequent occurrence of **-v** results in:

```
values.verbosity += 1
```

callback

[required: *callback*; relevant: *type*, *nargs*, *callback_args*, *callback_kwargs*]

Call the function specified by *callback*. The signature of this function should be:

```
func(option : Option,
     opt : string,
     value : any,
     parser : OptionParser,
     *args, **kwargs)
```

Callback options are covered in detail in section 6.20.4, “Callback Options.”

help

[required: none]

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`.

Example:

```
from optparse import Option, OptionParser, SUPPRESS_HELP

usage = "usage: %prog [options]"
parser = OptionParser(usage, option_list=[
  make_option("-h", "--help", action="help"),
  make_option("-v", action="store_true", dest="verbose",
              help="Be moderately verbose")
  make_option("--file", dest="filename",
              help="Input file to read data from"),
  make_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:

```
usage: <yourscript> [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

[required: none]

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.

Option types

`optparse` supports six option types out of the box: *string*, *int*, *long*, *choice*, *float* and *complex*. (Of these, string, int, float, and choice are the most commonly used —long and complex are there mainly for completeness.) It's easy to add new option types by subclassing the `Option` class; see section 6.20.5, “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 are passed to `int()` to convert them to Python integers. If `int()` fails, so will `optparse`, although with a more useful error message. Internally, `optparse` raises `OptionValueError` in `optparse.check_builtin()`; at a higher level (in `OptionParser`), `optparse` catches this exception and terminates your program with a useful error message.

Likewise, float arguments are passed to `float()` for conversion, long arguments to `long()`, and complex arguments to `complex()`. Apart from that, they are handled identically to integer arguments.

Choice options are a subtype of string options. A master list or tuple of choices (strings) must be passed to the option constructor (`make_option()` or `OptionParser.add_option()`) as the *choices* keyword argument. Choice option arguments are compared against this master list in `optparse.check_choice()`, and `OptionValueError` is raised if an unknown string is given.

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*)
: Given an option string such as **-q** or **--verbose**, returns true if the `OptionParser` has an option with that option string.

`get_option`(*opt_str*)
: Returns the `Option` instance that implements the option string you supplied, or `None` if no options implement it.

`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`.

Conflicts between options

If you're not careful, it's easy to define conflicting options:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(This is even easier to do if you've defined your own `OptionParser` subclass with some standard options.)

On the assumption that this is usually a mistake, `optparse` raises an exception (`OptionConflictError`) by default when this happens. Since this is an easily-fixed programming error, you shouldn't try to catch this exception—fix your mistake and get on with life.

Sometimes, you want newer options to deliberately replace the option strings used by older options. You can achieve this by calling:

```
parser.set_conflict_handler("resolve")
```

which instructs `optparse` to resolve option conflicts intelligently.

Here's how it works: every time you add an option, `optparse` checks for conflicts with previously-added options. If it finds any, it invokes the conflict-handling mechanism you specify either to the `OptionParser` constructor:

```
parser = OptionParser(..., conflict_handler="resolve")
```

or via the `set_conflict_handler()` method.

The default conflict-handling mechanism is `error`.

Here's an example: first, define an `OptionParser` set to resolve conflicts intelligently:

```
parser = OptionParser(conflict_handler="resolve")
```

Now add all of our options:

```
parser.add_option("-n", "--dry-run", ..., help="original dry-run option")
...
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 == "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, e.g.:

```
options:
  --dry-run     original dry-run option
  ...
  -n, --noisy   be noisy
```

Note that 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. E.g. if we carry on with our existing `OptionParser`:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the first **-n**/**--dry-run** option is no longer accessible, so `optparse` removes it. If the user asks for help, they'll get something like this:

```
options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

6.20.4 Callback Options

If `optparse`'s built-in actions and types just don't fit the bill for you, but it's not worth extending `optparse` to define your own actions or types, you'll probably need to define a callback option. Defining callback options is quite easy; the tricky part is writing a good callback (the function that is called when `optparse` encounters the option on the command line).

Defining a callback option

As always, you can define a callback option either by directly instantiating the `Option` class, or by using the `add_option()` method of your `OptionParser` object. The only option attribute you must specify is *callback*, the function to call:

```
parser.add_option("-c", callback=my_callback)
```

Note that you supply a function object here—so you must have already defined a function `my_callback()` when you define the callback option. In this simple case, `optparse` knows nothing about the arguments the **-c** option expects to take. Usually, this means that the option doesn't take any 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 document.

There are several other option attributes that you can supply when you define an option attribute:

type
: has its usual meaning: as with the "store" or "append" actions, it instructs `optparse` to consume one argument that must be convertible to *type*. Rather than storing the value(s) anywhere, though, `optparse` converts it to *type* and passes it to your callback function.

nargs
: also has its usual meaning: if it is supplied and '`nargs > 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.

How callbacks are called

All callbacks are called as follows:

```
func(option, opt, value, parser, *args, **kwargs)
```

where

option
: is the `Option` instance that's calling the callback.

opt
: is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, *opt* will be the full, canonical option string—for example, if the user puts **--foo** on the command-line as an abbreviation for **--foobar**, then *opt* 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 (see 6.20.3, "Option types"). 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 it, as instance attributes:

  parser.rargs
  : the current remaining argument list, i.e. with *opt* (and *value*, if any) removed, and only the arguments following them still there. Feel free to modify `parser.rargs`, e.g. by consuming more arguments.

parser.largs
: the current set of leftover arguments, i.e. arguments that have been processed but have not been consumed as options (or arguments to options). Feel free to modify `parser.largs` e.g. by adding more arguments to it.

parser.values
: the object where option values are by default stored. This is useful because it 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 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*.

Since *args* and *kwargs* are optional (they are only passed if you supply *callback_args* and/or *callback_kwargs* when you define your callback option), the minimal callback function is:

```
def my_callback (option, opt, value, parser):
    pass
```

Error handling

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.

Examples

Here's an example of a callback option that takes no arguments, and simply records that the option was seen:

```
def record_foo_seen (option, opt, value, parser):
    parser.saw_foo = 1

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Of course, you could do that with the "store_true" action. 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.

```
def check_order (option, opt, 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")
```

If you want to reuse 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.

```
def check_order (option, opt, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt)
    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='c')
```

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:

```
def check_moon (option, opt, value, parser):
    if is_full_moon():
        raise OptionValueError("%s option invalid when moon full" % opt)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(The definition of `is_full_moon()` is left as an exercise for the reader.)

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 that many arguments.

Here's an example that just emulates the standard "store" action:

```
def store_value (option, opt, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.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. Use your imagination!)

Variable arguments

Things get hairy when you want an option to take a variable number of arguments. For this case, you have to write a callback; `optparse` doesn't provide any built-in capabilities for it. You have to deal with the full-blown syntax for conventional UNIX command-line parsing. (Previously, `optparse` took care of this for you, but I got it wrong. It was fixed at the cost of making this kind of callback more complex.) In particular, callbacks have to worry about bare -- and - arguments; the convention is:

- bare --, if not the argument to some option, causes command-line processing to halt and the -- itself is lost.
- bare - similarly causes command-line processing to halt, but the - itself is kept.
- either -- or - can be option arguments.

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:

```
def varargs (option, opt, 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, rather than as arguments to **-c**. Fixing this is left as an exercise for the reader.

6.20.5 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.

Also, the examples section includes several demonstrations of extending `optparse` in different ways: e.g. a case-insensitive option parser, or two kinds of option parsers that implement "required options".

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:

```
def check_foo (option : Option, opt : string, value : string)
               -> foo
```

You can name it whatever you like, and make it return any type you like. The value returned by a type-checking function will wind up in the `OptionValues` instance returned by `OptionParser.parse_args()`, or be passed to callbacks 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 adds built-in support for complex numbers [purely for completeness], but never mind.)

First, the necessary imports:

```
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it's referred to later (in the `TYPE_CHECKER` class attribute of your `Option` subclass):

```
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:

```
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`:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", action="store", type="complex", dest="c")
```

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:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

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 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.

Some default “store” actions are *store*, *store_const*, *append*, and *count*. The default “typed” actions are *store*, *append*, and *callback*.

When you add an action, you need to decide if it’s a “store” action, a “typed”, neither, or both. Three class attributes of `Option` (or your `Option` subclass) control this:

`ACTIONS`
: All actions must be listed as strings in ACTIONS.

`STORE_ACTIONS`
: “store” actions are additionally listed here.

`TYPED_ACTIONS`
: “typed” actions are additionally listed here.

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",)

    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`.
- `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 `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.

Other reasons to extend `optparse`

Adding new types and new actions are the big, obvious reasons why you might want to extend `optparse`. I can think of at least two other areas to play with.

First, the simple one: `OptionParser` tries to be helpful by calling `sys.exit()` when appropriate, i.e. when there's an error on the command-line or when the user requests help. In the former case, the traditional course of letting the script crash with a traceback is unacceptable; it will make users think there's a bug in your script when they make a command-line error. In the latter case, there's generally not much point in carrying on after printing a help message.

If this behaviour bothers you, it shouldn't be too hard to "fix" it. You'll have to

1. subclass OptionParser and override the error() method
2. subclass Option and override the take_action() method—you'll need to provide your own handling of the "help" action that doesn't call sys.exit()

The second, much more complex, possibility is to override the command-line syntax implemented by `optparse`. In this case, you'd leave the whole machinery of option actions and types alone, but rewrite the code that processes `sys.argv`. You'll need to subclass `OptionParser` in any case; depending on how radical a rewrite you want, you'll probably need to override one or all of `parse_args()`, `_process_long_opt()`, and `_process_short_opts()`.

Both of these are left as an exercise for the reader. I have not tried to implement either myself, since I'm quite happy with `optparse`'s default behaviour (naturally).

Happy hacking, and don't forget: Use the Source, Luke.

Examples

Here are a few examples of extending the `optparse` module.

First, let's change the option-parsing to be case-insensitive:

```
from optparse import Option, OptionParser, _match_abbrev

# This case-insensitive option parser relies on having a
# case-insensitive dictionary type available.  Here's one
# for Python 2.2.  Note that a *real* case-insensitive
# dictionary type would also have to implement __new__(),
# update(), and setdefault() -- but that's not the point
# of this exercise.

class caseless_dict (dict):
    def __setitem__ (self, key, value):
        dict.__setitem__(self, key.lower(), value)

    def __getitem__ (self, key):
```

```
            return dict.__getitem__(self, key.lower())

        def get (self, key, default=None):
            return dict.get(self, key.lower())

        def has_key (self, key):
            return dict.has_key(self, key.lower())


    class CaselessOptionParser (OptionParser):

        def _create_option_list (self):
            self.option_list = []
            self._short_opt = caseless_dict()
            self._long_opt = caseless_dict()
            self._long_opts = []
            self.defaults = {}

        def _match_long_opt (self, opt):
            return _match_abbrev(opt.lower(), self._long_opt.keys())


    if __name__ == "__main__":
        from optik.errors import OptionConflictError

        # test 1: no options to start with
        parser = CaselessOptionParser()
        try:
            parser.add_option("-H", dest="blah")
        except OptionConflictError:
            print "ok: got OptionConflictError for -H"
        else:
            print "not ok: no conflict between -h and -H"

        parser.add_option("-f", "--file", dest="file")
        #print 'parser.get_option("-f")'
        #print 'parser.get_option("-F")'
        #print 'parser.get_option("--file")'
        #print 'parser.get_option("--fIlE")'
        (options, args) = parser.parse_args(["--FiLe", "foo"])
        assert options.file == "foo", options.file
        print "ok: case insensitive long options work"

        (options, args) = parser.parse_args(["-F", "bar"])
        assert options.file == "bar", options.file
        print "ok: case insensitive short options work"
```

And two ways of implementing “required options” with `optparse`.

Version 1: Add a method to `OptionParser` which applications must call after parsing arguments:

_1.py

Version 2: Extend `Option` and add a `required` attribute; extend `OptionParser` to ensure that required options are present after parsing:

_2.py

6.21 `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`(\[*suffix* \] \[, *prefix* \] \[, *dir* \] \[, *text=False* \])
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 Macintosh, the 'Temporary Items' folder.
   - 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.

6.22 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

EACCES
Permission denied

EFAULT
Bad address

ENOTBLK
Block device required

EBUSY
Device or resource busy

EEXIST
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

ENOBUFS
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

6.23 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*)
Returns 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.

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.

```
>>> 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 (section 6.24):
Shell-style filename (not path) expansion

6.24 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:

Pattern	Meaning
*	matches everything
?	matches any single character
[*seq*]	matches any character in *seq*
[!*seq*]	matches any character not in *seq*

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` (section 6.23):
UNIX shell-style path expansion.

6.25 `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*. If *dst* exists, it will be replaced, otherwise it will be created. Special files such as character or block devices and pipes cannot not be copied with this function. *src* and *dst* are path names given as strings.

copyfileobj(*fsrc, fdst*[, *length*])
Copy the contents of the file-like object *fsrc* 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. 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.

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.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.

6.25.1 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.

```
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', str(why))
```

6.26 `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

```
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:

Key	Category	Meaning
`LC_NUMERIC`	`'decimal_point'`	Decimal point character.
	`'grouping'`	Sequence of numbers specifying which relative positions the `'thousands_sep'` is expected. If the sequence is terminated with `CHAR_MAX`, no further grouping is performed. If the sequence terminates with a `0`, the last group size is repeatedly used.
	`'thousands_sep'`	Character used between groups.
`LC_MONETARY`	`'int_curr_symbol'`	International currency symbol.
	`'currency_symbol'`	Local currency symbol.
	`'mon_decimal_point'`	Decimal point used for monetary values.
	`'mon_thousands_sep'`	Group separator used for monetary values.
	`'mon_grouping'`	Equivalent to `'grouping'`, used for monetary values.
	`'positive_sign'`	Symbol used to annotate a positive monetary value.
	`'negative_sign'`	Symbol used to annotate a nnegative monetary value.
	`'frac_digits'`	Number of fractional digits used in local formatting of monetary values.
	`'int_frac_digits'`	Number of fractional digits used in international formatting of monetary values.

The possible values for `'p_sign_posn'` and `'n_sign_posn'` are given below.

Value	Explanation
`0`	Currency and value are surrounded by parentheses.
`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.
`LC_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*])
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.

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:

```
>>> import locale
>>> loc = locale.setlocale(locale.LC_ALL) # get current locale
>>> locale.setlocale(locale.LC_ALL, 'de_DE') # use German locale; name might vary with platf
>>> locale.strcoll('f\xe4n', 'foo') # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '') # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C') # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc) # restore saved locale
```

6.26.1 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()`.

6.26.2 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 is embedded in an application, if the application sets the locale to something specific before initializing Python, that is generally okay, and Python will use whatever locale is set, *except* that the `LC_NUMERIC` locale should always be 'C'.

The `setlocale()` function in the `locale` module gives the Python programmer the impression that you can manipulate the `LC_NUMERIC` locale setting, but this not the case at the C level: C code will always find that the `LC_NUMERIC` locale setting is 'C'. This is because too much would break when the decimal point character is set to something else than a period (e.g. the Python parser would break). Caveat: threads that run without holding Python's global interpreter lock may occasionally find that the numeric locale setting differs; this is because the only portable way to implement this feature is to set the numeric locale settings to what the user requests, extract the relevant characteristics, and then restore the 'C' numeric locale.

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.

6.26.3 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()`, and `bindtextdomain()`. 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 `cdgettext()`. For these applications, it may be necessary to bind the text domain, so that the libraries can properly locate their message catalogs.

6.27 `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.

6.27.1 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`(*domain*[, *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.[2]

`textdomain`([*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`(*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).

`dgettext`(*domain*, *message*)
Like `gettext()`, but look the message up in the specified *domain*.

`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.

`dngettext`(*domain*, *singular*, *plural*, *n*)
Like `ngettext()`, but look the message up in the specified *domain*.

New in version 2.3.

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:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print _('This is a translatable string.')
```

6.27.2 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

[2] 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.

either standard 8-bit strings or Unicode strings. Translations instances 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.[3] 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*]]]])

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 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.

`install`(*domain*[, *localedir*[, *unicode*]])

This installs the function _ in Python's builtin namespace, based on *domain*, and *localedir* which are passed to the function `translation()`. The *unicode* flag is passed to the resulting 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.

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

[3] See the footnote for `bindtextdomain()` above.

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.

ugettext(*message*)
If a fallback has been set, forward `ugettext` 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.

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.

install([*unicode*])
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.

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:

```
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.

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.

`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.

`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:

```
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) % {'n': n}
```

  New in version 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.

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:

```
import gettext
cat = gettext.Catalog(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.

6.27.3 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 `_('...')` — that is, a call to the function `_()`. For example:

```
filename = 'mylog.txt'
message = _('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**[4] 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
```

[4] 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://www.iro.umontreal.ca/contrib/po-utils/HTML/.

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**[5] 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 your entire application or a single module.

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:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

If your translators were providing you with Unicode strings in their '.po' files, you'd instead do:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.ugettext
```

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:

```
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:

```
import gettext
gettext.install('myapplication', '/usr/share/locale', unicode=1)
```

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:

[5] **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.

```
import gettext

lang1 = gettext.translation(languages=['en'])
lang2 = gettext.translation(languages=['fr'])
lang3 = gettext.translation(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()
```

Deferred translations

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:

```
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:

```
def _(message): return message

animals = [_('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 **pygettext** program, since it is not a string.

Another way to handle this is with the following example:

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
   N_('rat'),
   N_('penguin'),
   N_('python'),
   ]

# ...
for a in animals:
    print _(a)
```

In this case, you are marking translatable strings with the function `N_()`,[6] 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 `N_()`. **pygettext** and **xpot** both support this through the use of command line switches.

6.27.4 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

6.28 `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.

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

[6] The choice of `N_()` here is totally arbitrary; it could have just as easily been `MarkThisStringForTranslation()`.

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, an `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. `RotatingFileHandler` instances send error messages to disk files, with support for maximum log file sizes and log file rotation.
4. `SocketHandler` instances send error messages to TCP/IP sockets.
5. `DatagramHandler` instances send error messages to UDP sockets.
6. `SMTPHandler` instances send error messages to a designated email address.
7. `SysLogHandler` instances send error messages to a UNIX syslog daemon, possibly on a remote machine.
8. `NTEventLogHandler` instances send error messages to a Windows NT/2000/XP event log.
9. `MemoryHandler` instances send error messages to a buffer in memory, which is flushed whenever specific criteria are met.
10. `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.

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.

debug(*msg*[, **args*[, ***kwargs*]])
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*. The only keyword argument in *kwargs* which is inspected is *exc_info* which, if it does not evaluate as false, causes exception information (via a call to sys.exc_info()) to be added to the logging message.

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().

error(*msg*[, **args*[, ***kwargs*]])
Logs a message with level ERROR 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.

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. 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()
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.

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.

6.28.1 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 in the root logger, or delegation to the parent in non-root loggers).

`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 `setLevel()`, it is returned. Otherwise, the hierarchy is traversed towards the root until a value other than `NOTSET` is found, and that value is returned.

`debug`(*msg*[, **args*[, ***kwargs*]])
Logs a message with level `DEBUG` on this logger. The *msg* is the message format string, and the *args* are the arguments which are merged into *msg*. The only keyword argument in *kwargs* which is inspected is *exc_info* which, if it does not evaluate as false, causes exception information (via a call to `sys.exc_info()`) to be added to the logging message.

`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 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*)
This is a factory method which can be overridden in subclasses to create specialized `LogRecord` instances.

6.28.2 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__()`.

__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.

createLock()
Initializes a thread lock which can be used to serialize access to underlying I/O functionality which may not be threadsafe.

acquire()
Acquires the thread lock created with `createLock()`.

release()
Releases the thread lock acquired with `acquire()`.

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).

setFormatter(*form*)
Sets the `Formatter` for this handler to *form*.

addFilter(*filt*)
Adds the specified filter *filt* to this handler.

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()
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.

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`.

StreamHandler

The `StreamHandler` class 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).

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.

FileHandler

The `FileHandler` class sends logging output to a disk file. It inherits the output functionality from `StreamHandler`.

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.

`close`()
Closes the file.

`emit`(*record*)
Outputs the record to the file.

RotatingFileHandler

The `RotatingFileHandler` class supports rotation of disk log files.

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 in `setRollover()`.

SocketHandler

The `SocketHandler` class sends logging output to a network socket. The base class uses a TCP socket.

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.

DatagramHandler

The `DatagramHandler` class 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.

SysLogHandler

The `SysLogHandler` class supports sending logging messages to a remote or local UNIX syslog.

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.

NTEventLogHandler

The `NTEventLogHandler` class 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'.

SMTPHandler

The `SMTPHandler` class supports sending logging messages to an email address via SMTP.

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 without domain names (That's what the *mailhost* is for). 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.

emit(*record*)

Formats the record and sends it to the specified addressees.

getSubject(*record*)
If you want to specify a subject line which is record-dependent, override this method.

MemoryHandler

The `MemoryHandler` 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.

emit(*record*)
Appends the record to the buffer. If `shouldFlush()` returns true, calls `flush()` to process the buffer.

flush()
You can override this to implement custom flushing behavior. This version just zaps the buffer to empty.

shouldFlush(*record*)
Returns true if the buffer is up to capacity. This method can be overridden to implement custom flushing strategies.

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.

HTTPHandler

The `HTTPHandler` class 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. If no *method* is specified, '`GET`' is used.

emit(*record*)
Sends the record to the Web server as an URL-encoded dictionary.

6.28.3 Formatter Objects

`Formatter`s 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 2.2.6, "String Formatting Operations," for more information on string formatting.

Currently, the useful mapping keys in a LogRecord are:

Format	Description
`%(name)s`	Name of the logger (logging channel).
`%(levelno)s`	Numeric logging level for the message (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`).
`%(levelname)s`	Text logging level for the message (`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`).
`%(pathname)s`	Full pathname of the source file where the logging call was issued (if available).
`%(filename)s`	Filename portion of pathname.
`%(module)s`	Module (name portion of filename).
`%(lineno)d`	Source line number where the logging call was issued (if available).
`%(created)f`	Time when the LogRecord was created (as returned by `time.time()`).
`%(asctime)s`	Human-readable time when the LogRecord was created. By default this is of the form "2003-07-08 16:49:4
`%(msecs)d`	Millisecond portion of the time when the `LogRecord` was created.
`%(thread)d`	Thread ID (if available).
`%(process)d`	Process ID (if available).
`%(message)s`	The logged message, computed as `msg %args`.

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 `sys.exc_info()`) as a string. This default implementation just uses `traceback.print_exception()`. The resulting string is returned.

6.28.4 Filter Objects

`Filter`s can be used by `Handler`s and `Logger`s 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.

6.28.5 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.

LogRecord has no methods; it's just a repository for information about the logging event. The only reason it's a class rather than a dictionary is to facilitate extension.

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).

6.28.6 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 though 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.

6.28.7 Configuration

Configuration functions

The following functions allow the logging module to be configured. Before they can be used, you must import `logging.config`. Their use is optional — you can configure the logging module entirely 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()`.

`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()`.

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 eval()uated 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, for example, 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.LOG_USER)

[handler_hand06]
class=NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')

[handler_hand08]
class=MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=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=
```

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`.

6.28.8 Using the logging package

Basic example - log to a file

Here’s a simple logging example that just logs to a file. In order, it creates a `Logger` instance, then a `FileHandler` and a `Formatter`. It attaches the `Formatter` to the `FileHandler`, then the `FileHandler` to the `Logger`. Finally, it sets a debug level for the logger.

```
import logging
logger = logging.getLogger('myapp')
hdlr = logging.FileHandler('/var/tmp/myapp.log')
formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
hdlr.setFormatter(formatter)
logger.addHandler(hdlr)
logger.setLevel(logging.WARNING)
```

We can use this logger object now to write entries to the log file:

```
logger.error('We have a problem')
logger.info('While this is just chatty')
```

If we look in the file that was created, we’ll see something like this:

```
2003-07-08 16:49:45,896 ERROR We have a problem
```

The info message was not written to the file - we called the `setLevel` method to say we only wanted `WARNING` or worse, so the info message is discarded.

The timestamp is of the form “year-month-day hour:minutes:seconds,milliseconds.” Note that despite the three digits of precision in the milliseconds field, not all systems provide time with this much precision.

CHAPTER

SEVEN

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:

signal	Set handlers for asynchronous events.
socket	Low-level networking interface.
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.
Queue	A synchronized queue class.
mmap	Interface to memory-mapped files for UNIX and Windows.
anydbm	Generic interface to DBM-style database modules.
dbhash	DBM-style interface to the BSD database library.
whichdb	Guess which DBM-style module created a given database.
bsddb	Interface to Berkeley DB database library
dumbdbm	Portable implementation of the simple DBM interface.
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.
readline	GNU readline support for Python.
rlcompleter	Python identifier completion for the GNU readline library.

7.1 `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 id 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; see the reference manual for a description of frame objects).

7.1.1 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.

```
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
```

7.2 `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.

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.

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 `hstrerror()` C function.

exception `gaierror`
: This exception is raised for address-related errors, for `getaddrinfo()` and `getnameinfo()`. The accompanying value is a pair `(`*error*`,` *string*`)` representing an error returned by a library call. *string* represents the description of *error*, as returned by the `gai_strerror()` C function.

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 “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 `socket()`. If the `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 `socket()`. (Only `SOCK_STREAM` and `SOCK_DGRAM` appear to be generally useful.)

`SO_*`
`SOMAXCONN`
`MSG_*`
`SOL_*`
`IPPROTO_*`
`IPPORT_*`
`INADDR_*`

IP_*
IPV6_*
EAI_*
AI_*
NI_*
TCP_*

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*, *canonname*, *sockaddr*)

family, *socktype*, *proto* are all integer and are meant to be passed to the `socket()` function. *canonname* 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 `httplib` 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 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 `gethostbyaddr(gethostname())` (see below).

gethostbyaddr(*ip_address*)

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/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 `getfqdn()`. `gethostbyaddr`

supports both IPv4 and IPv6.

getnameinfo(*sockaddr, flags*)
Translate a socket address *sockaddr* into a 2-tuple `(`*host*`,` *port*`)`. Depending on the settings of *flags*, the result can contain a fully-qualified domain name or numeric address representation in *host*. Similarly, *port* can contain a string port name or a numeric port number. New in version 2.2.

getprotobyname(*protocolname*)
Translate an Internet protocol name (for example, `'icmp'`) to a constant suitable for passing as the (optional) third argument to the `socket()` function. This is usually only needed for sockets opened in "raw" mode (`SOCK_RAW`); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

getservbyname(*servicename, protocolname*)
Translate an Internet service name and protocol name to a port number for that service. The protocol name should be `'tcp'` or `'udp'`.

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`, `AF_INET6` or `AF_UNIX`. The socket type should be `SOCK_STREAM`, `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!

fromfd(*fd, family, type*[, *proto*])
Build a socket object from an existing file descriptor (an integer as returned by a file object's `fileno()` method). 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.

`getdefaulttimeout`()
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` (section 11.15):
Classes that simplify writing network servers.

7.2.1 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 be available in Python 2.0.

`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 be available in Python 2.0 and later.

`fileno``()`
Return the socket's file descriptor (a small integer). This is useful with `select.select()`.

`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 2.2.8, "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 should 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.

`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()
Returns 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 should 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.

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 `0`, further receives are disallowed. If *how* is `1`, further sends are disallowed. If *how* is `2`, further sends and receives are disallowed.

Note that there are no methods `read()` or `write()`; use `recv()` and `send()` without *flags* argument instead.

7.2.2 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.

7.2.3 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.

```
# 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', '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.

```
# 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, 0, socket.AI
    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'
    sys.exit(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
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)
    except socket.error, msg:
s.close()
s = None
continue
    break
if s is None:
    print 'could not open socket'
    sys.exit(1)
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', 'data'
```

7.3 `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 7.3.1 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 lists of 'waitable objects': either integers representing file descriptors or objects with a parameterless method named `fileno()` returning such an integer. The three lists of waitable objects are for input, output and 'exceptional conditions', respectively. Empty lists are allowed, but acceptance of three empty lists 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 lists 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 desciptors that don't originate from WinSock.

7.3.1 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(highest file descriptor), while `poll()` is O(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.

Constant	Meaning
POLLIN	There is data to read
POLLPRI	There is urgent data to read
POLLOUT	Ready for output: writing will not block
POLLERR	Error condition of some sort
POLLHUP	Hung up
POLLNVAL	Invalid request: descriptor not open

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, *fd* can be an integer or an object with a fileno() method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a KeyError exception to be raised.

poll([*timeout*])
Polls the set of registered file descriptors, and returns a possibly-empty list containing (*fd*, *event*) 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor — POLLIN for waiting input, 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 *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, negative, or None, the call will block until there is an event for this poll object.

7.4 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 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.

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), and returns `None`. 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. If an argument is present, 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.

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.

7.5 `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.

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.

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.

7.5.1 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. 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.

  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.

7.5.2 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.

7.5.3 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:

```
# 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.

7.5.4 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 `acquire()` and `release()`).

A semaphore manages an internal counter which is decremented by each `acquire()` call and incremented by each `release()` call. The counter can never go below zero; when `acquire()` finds that it is zero, it blocks, waiting until some other thread calls `release()`.

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 `release()` to make it larger than zero. This is done with proper interlocking so that if multiple `acquire()` calls are blocked, `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 *blocking* set to true, do the same thing as when called without arguments, and return true.

When invoked with *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 zero again, wake up that thread.

`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:

```
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:

```
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.

7.5.5 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.

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).

7.5.6 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=(), kwargs={}*)
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 `join()` method is called terminates – either normally or through an unhandled exception – 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).

A thread can be `join()`ed many times.

A thread cannot join itself because this would cause a deadlock.

It is an error to attempt to `join()` a thread before it has been started.

`getName`()
Return the thread's name.

`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.

`isAlive`()
Return whether the thread is alive.

Roughly, a thread is alive from the moment the `start()` method returns until its `run()` method terminates.

`isDaemon`()
Return the thread's daemon flag.

`setDaemon`(*daemonic*)
Set the thread's daemon flag to the Boolean value *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.

7.5.7 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:

```
def hello():
    print "hello, world"

t = Timer(30.0, hello)
t.start() # after 30 seconds, "hello, world" will be printed
```

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.

7.6 `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:

```
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.

7.7 `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:

```
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.

7.8 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.

See Also:

Module bisect (section 5.10):
PriorityQueue example using the Queue class

The Queue module defines the following class and exception:

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 or locked.

exception Full
Exception raised when non-blocking put() (or put_nowait()) is called on a Queue object which is full or locked.

7.8.1 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. Becauseof 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 `Empty` exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the `Empty` exception (*timeout* is ignored in that case).

New in version 2.3: the timeout parameter.

get_nowait()
Equivalent to `get(False)`.

7.9 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. Assigment to an `ACCESS_COPY` memory map affects memory but does not update the underlying file.

mmap(*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 `0`, the maximum length of the map will be the current size of the file when `mmap()` is called.

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.

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*)
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.

7.10 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.

See Also:

Module `dbhash` (section 7.11):
BSD `db` database interface.

Module `dbm` (section 8.6):
Standard UNIX database interface.

Module `dumbdbm` (section 7.14):
Portable implementation of the `dbm` interface.

Module `gdbm` (section 8.7):
GNU database interface, based on the `dbm` interface.

Module `shelve` (section 3.17):
General object persistence built on top of the Python `dbm` interface.

Module `whichdb` (section 7.12):
Utility module used to determine the type of an existing database.

7.11 dbhash — DBM-style interface to the BSD database library

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` (section 7.10):
Generic interface to `dbm`-style databases.

Module `bsddb` (section 7.13):
Lower-level interface to the BSD `db` library.

Module `whichdb` (section 7.12):
Utility module used to determine the type of an existing database.

7.11.1 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:

```
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.

7.12 `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'`.

7.13 `bsddb` — Interface to Berkeley DB library

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.

Starting with Python 2.3 the `bsddb` module requires the Berkeley DB library version 3.1 or later (it is known to work with 3.1 thru 4.1 at the time of this writing).

See Also:

http://pybsddb.sourceforge.net/
Website with documentation for the new python Berkeley DB interface that closely mirrors the sleepycat object oriented interface provided in Berkeley DB 3 and 4.

http://www.sleepycat.com/
Sleepycat Software produces the modern Berkeley DB library.

The following is a description of the legacy `bsddb` interface compatible with the old python bsddb module. For details about the more modern Db and DbEnv object oriented interface see the above mentioned pybsddb URL.

Warning: This legacy interface is not thread safe in python 2.3.x or earlier. Data corruption, core dumps or deadlocks may occur if you attempt multi-threaded access. You must use the modern pybsddb interface linked to above if you need multi-threaded or multi-process database access.

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.

`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, default), '`w`' (read-write) , '`c`' (read-write - create if necessary) 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.

`btopen`(*filename*[, *flag*[, *mode*[, *btflags*[, *cachesize*[, *maxkeypage*[, *minkeypage*[, *psize*[, *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, default), '`w`' (read-write), '`c`' (read-write - create if necessary) 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.

`rnopen`(*filename*[, *flag*[, *mode*[, *rnflags*[, *cachesize*[, *psize*[, *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, default), '`w`' (read-write), '`c`' (read-write - create if necessary) 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.

See Also:

Module `dbhash` (section 7.11):
DBM-style interface to the `bsddb`

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.

7.13.1 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 mapping 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.

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()).

sync()
Synchronize the database on disk.

Example:

```
>>> 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
```

7.14 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 (section 7.10):
Generic interface to dbm-style databases.

Module dbm (section 8.6):
Similar interface to the DBM/NDBM library.

Module gdbm (section 8.7):
Similar interface to the GNU GDBM library.

Module shelve (section 3.17):
Persistence module which stores non-string data.

Module whichdb (section 7.12):
Utility module used to determine the type of an existing database.

7.14.1 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.

7.15 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.gzip.org/zlib/. Version 1.1.3 is the most recent version as of September 2000; use a later version if one is available. There are known incompatibilities between the Python module and earlier versions of the zlib library.

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 and are used to allow partial error recovery on decompression, 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.

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()
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.

See Also:

Module `gzip` (section 7.16):
Reading and writing **gzip**-format files.

http://www.gzip.org/zlib/
The zlib library home page.

7.16 `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` (section 7.15):
The basic data compression module needed to support the **gzip** file format.

7.17 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;

7.17.1 (De)compression of files

Handling of compressed files is offered by the `BZ2File` class.

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 `>= 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.

7.17.2 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.

7.17.3 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.

7.18 `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.

The available attributes of this module are:

exception `error`
The error raised for bad ZIP files.

class `ZipFile`
The class for reading and writing ZIP files. See "*ZipFile Objects*" (section 7.18.1) for constructor details.

class `PyZipFile`
Class for creating ZIP archives containing Python libraries.

class `ZipInfo`([*filename*[, *date_time*]])
Class used the represent infomation 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 7.18.3, "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
(http://www.pkware.com/appnote.html)
Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

Info-ZIP Home Page
(http://www.info-zip.org/pub/infozip/)
Information about the Info-ZIP project's ZIP archive programs and development libraries.

7.18.1 ZipFile Objects

class `ZipFile`(*file*[, *mode*[, *compression*]])
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`.

`close``()`
Close the archive file. You must call `close()` before exiting your program or essential records will not be written.

`getinfo``(`*name*`)`
Return a `ZipInfo` object with information about the archive member *name*.

`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. 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*). 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'`.

`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`.

7.18.2 PyZipFile Objects

The `PyZipFile` constructor takes the same parameters as the `ZipFile` constructor. Instances have one method in addition to those of `ZipFile` objects.

`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. *basename* is intended for internal use only. The `writepy()` method makes archives with file names like this:

```
string.pyc                              # Top level name
test/__init__.pyc                       # Package directory
test/testall.pyc                        # Module test.testall
test/bogus/__init__.pyc                 # Subpackage directory
test/bogus/myfile.pyc                   # Submodule test.bogus.myfile
```

7.18.3 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:

Index	Value
`0`	Year
`1`	Month (one-based)
`2`	Day of month (one-based)
`3`	Hours (zero-based)
`4`	Minutes (zero-based)
`5`	Seconds (zero-based)

`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.

`file_offset`

Byte offset to the start of the file data.

`CRC`
CRC-32 of the uncompressed file.

`compress_size`
Size of the compressed data.

`file_size`
Size of the uncompressed file.

7.19 `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 7.19.1).

mode has to be a string of the form `'filemode[:compression]'`, it defaults to `'r'`. Here is a full list of mode combinations:

mode	action
`'r'`	Open for reading with transparent compression (recommended).
`'r:'`	Open for reading exclusively without compression.
`'r:gz'`	Open for reading with gzip compression.
`'r:bz2'`	Open for reading with bzip2 compression.
`'a' or 'a:'`	Open for appending with no compression.
`'w' or 'w:'`	Open for uncompressed writing.
`'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()` resp. `write()` method. *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 7.19.3). The currently possible modes:

mode	action
`'r|'`	Open a *stream* of uncompressed tar blocks for reading.
`'r|gz'`	Open a gzip compressed *stream* for reading.
`'r|bz2'`	Open a bzip2 compressed *stream* for reading.
`'w|'`	Open an uncompressed *stream* for writing.
`'w|gz'`	Open an gzip compressed *stream* for writing.
`'w|bz2'`	Open an bzip2 compressed *stream* for writing.

class `TarFile`
Class for reading and writing tar archives. Do not use this class directly, better use `open()` instead. See *TarFile Objects* (section 7.19.1).

`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 `zipfile` 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` (section **??**):
Documentation of the `zipfile` standard module.

GNU tar manual, Standard Section
(http://www.gnu.org/manual/tar/html_chapter/tar_8.html#SEC118)
Documentation for tar archive files, including GNU tar extensions.

7.19.1 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 7.19.2) 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 7.19 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 occurence 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`, an `"ls -l"`-like output 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.

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*.

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()`.

add(*name*[, *arcname*[, *recursive=True*]])
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`.

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.

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.

close()
Close the `TarFile`. In write-mode, two finishing zero blocks are appended to the archive.

posix=True
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 and linknames to 100 characters. A `ValueError` is raised, if a pathname exceeds this limit. If `False`, create a GNU tar compatible archive. It will not be POSIX compliant, but can store pathnames of unlimited length.

dereference=False
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 links.

ignore_zeros=False
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.

debug=0
To be set from `0`(no debug messages) up to `3`(all debug messages). The messages are written to `sys.stdout`.

errorlevel=0

If `0`, 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.

7.19.2 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()`.

class `TarInfo`(`[`*name*`]`)
Create a `TarInfo` object.

`frombuf`()
Create and return a `TarInfo` object from a string buffer.

`tobuf`()
Create a string buffer from a `TarInfo` object.

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, CONTTYPE, 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, gid`
User and group ID of who originally stored this member.

`uname, gname`
User and 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.

7.19.3 Examples

How to create an uncompressed tar archive from a list of filenames:

```
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:

```
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:

```
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()
```

7.20 `readline` — GNU readline interface

The `readline` module defines a number of functions used either directly or from the `rlcompleter` module to facilitate completion and history file read and write from the Python interpreter.

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'.

`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.

`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` (section 7.21):
Completion of Python identifiers at the interactive prompt.

7.20.1 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.

```
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
```

7.21 `rlcompleter` — Completion function for GNU readline

The `rlcompleter` module defines a completion function for the `readline` module by completing valid Python identifiers and keywords.

This module is UNIX-specific due to its dependence on the `readline` module.

The `rlcompleter` module defines the `Completer` class.

Example:

```
>>> 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
>>> readline.
```

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")
```

7.21.1 Completer Objects

Completer objects have the following method:

`complete`(*text*, *state*)
Return the *state*th 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.

CHAPTER

EIGHT

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).
grp	The group database (getgrnam() and friends).
crypt	The crypt() function used to check UNIX passwords.
dl	Call C functions in shared objects.
dbm	The standard "database" interface, based on ndbm.
gdbm	GNU's reinterpretation of dbm.
termios	POSIX style tty control.
TERMIOS	Symbolic constants required to use the termios module.
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.

8.1 posix — The most common POSIX system calls

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.

8.1.1 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 $CFLAGS" \
        ./configure
```

8.1.2 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.

8.2 pwd — The password database

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>`):

Index	Attribute	Meaning
0	`pw_name`	Login name
1	`pw_passwd`	Optional encrypted password
2	`pw_uid`	Numerical user ID
3	`pw_gid`	Numerical group ID
4	`pw_gecos`	User name or comment field
5	`pw_dir`	User home directory
6	`pw_shell`	User command interpreter

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 field `pw_passwd` only contains a asterisk (`'*'`) or the letter ‘`x`’ where the encrypted password is stored in a file ‘/etc/shadow’ which is not world readable.

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 (section 8.3):
An interface to the group database, similar to this.

8.3 grp — The group database

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>):

Index	Attribute	Meaning
0	gr_name	the name of the group
1	gr_passwd	the (encrypted) group password; often empty
2	gr_gid	the numerical group ID
3	gr_mem	all the group member's user names

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 (section 8.2):
An interface to the user database, similar to this.

8.4 crypt — Function to check UNIX passwords

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.

crypt(*word*, *salt*)
word will usually be a user's password as typed at a prompt or in a graphical interface. *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 *salt* must be in the set ⌜[./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).

A simple example illustrating typical use:

```
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[:2]) == cryptedpasswd
    else:
        return 1
```

8.5 dl — Call C functions in shared objects

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.

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:

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:

```
>>> 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.

8.5.1 Dl Objects

Dl 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:

```
>>> 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.

8.6 dbm — Simple "database" interface

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:

Value	Meaning
`'r'`	Open existing database for reading only (default)
`'w'`	Open existing database for reading and writing
`'c'`	Open database for reading and writing, creating it if it doesn't exist
`'n'`	Always create a new, empty database, open for reading and writing

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` (section 7.10):
Generic interface to `dbm`-style databases.

Module `gdbm` (section 8.7):
Similar interface to the GNU GDBM library.

Module `whichdb` (section 7.12):
Utility module used to determine the type of an existing database.

8.7 gdbm — GNU's reinterpretation of dbm

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 syncronized.
- `'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 `gdbm`. The module constant `open_flags` is a string of supported flag characters. The exception `error` is raised if an invalid flag is specified.

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`.

In addition to the dictionary-like methods, `gdbm` objects have the following methods:

`firstkey`()
It's possible to loop over every key in the database using this method and the `nextkey()` method. The traversal is ordered by `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 *key* in the traversal. The following code prints every key in the database `db`, without having to create a list in memory that contains them all:

```
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 `gdbm` file, this routine will reorganize the database. `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 (section 7.10):
Generic interface to dbm-style databases.

Module whichdb (section 7.12):
Utility module used to determine the type of an existing database.

8.8 termios — POSIX style tty control

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: TCIFLUSH for the input queue, TCOFLUSH for the output queue, or TCIOFLUSH for both queues.

tcflow(*fd*, *action*)
Suspend or resume input or output on file descriptor *fd*. The *action* argument can be TCOOFF to suspend output, TCOON to restart output, TCIOFF to suspend input, or TCION to restart input.

See Also:

Module tty (section 8.10):
Convenience functions for common terminal control operations.

8.8.1 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:

```
def getpass(prompt = "Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = raw_input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

8.9 `TERMIOS` — Constants used with the `termios` module

Deprecated since release 2.1. Import needed constants from `termios` instead.

This module defines the symbolic constants required to use the `termios` module (see the previous section). See the POSIX or UNIX manual pages for a list of those constants.

8.10 `tty` — Terminal control functions

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.TCAFLUSH`, 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.TCAFLUSH`, and is passed to `termios.tcsetattr()`.

See Also:

Module `termios` (section 8.8):
: Low-level terminal control interface.

Module `TERMIOS` (section 8.9):
: Constants useful for terminal control operations.

8.11 `pty` — Pseudo-terminal utilities

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 (*pid*, *fd*). Note that the child gets *pid* 0, and the *fd* is *invalid*. The parent's return value is the *pid* of the child, and *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 `os.openpty()` if possible, or emulation code for SGI and generic UNIX systems. Return a pair of file descriptors (*master*, *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 *master_read* and *stdin_read* should be functions which read from a file-descriptor. The defaults try to read 1024 bytes each time they are called.

8.12 `fcntl` — The `fcntl()` and `ioctl()` system calls

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 *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, which provides a `fileno()` which returns a genuine file descriptor.

The module defines the following functions:

fcntl(*fd*, *op*[, *arg*])
Perform the requested operation on file descriptor *fd* (file objects providing a `fileno()` method are accepted as well). The operation is defined by *op* and is operating system dependent. These codes are also found in the `fcntl` module. The argument *arg* is optional, and defaults to the integer value `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 `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 *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.

ioctl(*fd*, *op*[, *arg*[, *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 longer than 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 in 2.3 it defaults to false. This is planned to change over the next few Python versions: in 2.4 failing to supply *mutate_flag* will get a warning but the same behavior and in versions later than 2.5 it will default to true.

An example:

```
>>> import array, fnctl, 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*, [*len*, [*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):

```
import struct, fcntl

file = open(...)
rv = fcntl(file, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(file, 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 `flock()` call may be better.

See Also:

Module `os` (section 6.1):
The `os.open` function supports locking flags and is available on a wider variety of platforms than the `fcntl.lockf` and `fcntl.flock` functions, providing a more platform-independent file locking facility.

8.13 `pipes` — Interface to shell pipelines

The `pipes` module defines a class to abstract the concept of a *pipeline* — a sequence of convertors 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:

class `Template()`
An abstraction of a pipeline.

Example:

```
>>> 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'
```

8.13.1 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 commands reads a given file on the command line) or `'.'` (which means the commands 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.

8.14 posixfile — File-like objects with locking support

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:

Format	Meaning
'u'	unlock the specified region
'r'	request a read lock for the specified section
'w'	request a write lock for the specified section

In addition the following modifiers can be added to the format:

Modifier	Meaning	Notes
'|'	wait until the lock has been granted	
'?'	return the first lock conflicting with the requested lock, or `None` if there is no conflict.	(1)

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:

Format	Meaning
'a'	append only flag
'c'	close on exec flag
'n'	no delay flag (also called non-blocking flag)
's'	synchronization flag

In addition the following modifiers can be added to the format:

Modifier	Meaning	Notes
'!'	turn the specified flags 'off', instead of the default 'on'	(1)
'='	replace the flags, instead of the default 'OR' operation	(1)
'?'	return a string in which the characters represent the flags that are set.	(2)

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:

```
import posixfile

file = posixfile.open('/tmp/test', 'w')
file.lock('w|')
...
file.lock('u')
file.close()
```

8.15 `resource` — Resource usage information

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.

8.15.1 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 underyling 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 underyling 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.)

`RLIMIT_FSIZE`
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.

8.15.2 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:

Index	Field	Resource
0	ru_utime	time in user mode (float)
1	ru_stime	time in system mode (float)
2	ru_maxrss	maximum resident set size
3	ru_ixrss	shared memory size
4	ru_idrss	unshared memory size
5	ru_isrss	unshared stack size
6	ru_minflt	page faults not requiring I/O
7	ru_majflt	page faults requiring I/O
8	ru_nswap	number of swap outs
9	ru_inblock	block input operations
10	ru_oublock	block output operations
11	ru_msgsnd	messages sent
12	ru_msgrcv	messages received
13	ru_nsignals	signals received
14	ru_nvcsw	voluntary context switches
15	ru_nivcsw	involuntary context switches

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.

8.16 `nis` — Interface to Sun's NIS (Yellow Pages)

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*)
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.

`cat`(*mapname*)
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.

`maps`()
Return a list of all valid maps.

The `nis` module defines the following exception:

exception `error`
An error raised when a NIS function returns an error code.

8.17 `syslog` — UNIX syslog library routines

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 bit field - 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 *pri*. The function `LOG_UPTO(`*pri*`)` calculates the mask for all priorities up to and including *pri*.

The module defines the following constants:

Priority levels (high to low): `LOG_EMERG`, `LOG_ALERT`, `LOG_CRIT`, `LOG_ERR`, `LOG_WARNING`, `LOG_NOTICE`, `LOG_INFO`, `LOG_DEBUG`.

Facilities: `LOG_KERN`, `LOG_USER`, `LOG_MAIL`, `LOG_DAEMON`, `LOG_AUTH`, `LOG_LPR`, `LOG_NEWS`, `LOG_UUCP`, `LOG_CRON` and `LOG_LOCAL0` to `LOG_LOCAL7`.

Log options: `LOG_PID`, `LOG_CONS`, `LOG_NDELAY`, `LOG_NOWAIT` and `LOG_PERROR` if defined in `<syslog.h>`.

8.18 commands — Utilities for running commands

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 -ld` *file*' as a string. This function uses the `getoutput()` function, and properly escapes backslashes and dollar signs in the argument.

Example:

```
>>> 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'
```

CHAPTER

NINE

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:

```
>>> 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:

```
python /usr/local/lib/python1.5/pdb.py myscript.py
```

Typical usage to inspect a crashed program is:

```
>>> 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`.

9.1 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 an newer frame).

u(p) Move the current frame one level up in the stack trace (to a 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.

cl(ear) [*bpnumber* [*bpnumber ...*]] With a space separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks (but first ask confirmation).

disable [*bpnumber* [*bpnumber ...*]] 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.

enable [*bpnumber* [*bpnumber ...*]] Enables the breakpoints specified.

ignore *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.

condition *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.

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(eturn) 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 exception 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.

9.2 How It Works

Some changes were made to the interpreter:

- `sys.settrace(`*func*`)` sets the global trace function
- there can also 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'` or `'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.

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*.

CHAPTER

TEN

The Python Profiler

Written by James Roskind.[1]

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.

10.1 Introduction to the profiler

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.

10.2 How Is This Profiler Different From The Old Profiler?

(This section is of historical importance only; the old profiler discussed here was last seen in Python 1.1.)

The big changes from old profiling module are that you get more information, and you pay less CPU time. It's not a trade-off, it's a trade-up.

To be specific:

Bugs removed: Local stack frame is no longer molested, execution time is now charged to correct functions.

[1] Updated and converted to LaTeX by Guido van Rossum. The references to the old profiler are left in the text, although it no longer exists.

Accuracy increased: Profiler execution time is no longer charged to user's code, calibration for platform is supported, file reads are not done *by* profiler *during* profiling (and charged to user's code!).

Speed increased: Overhead CPU cost was reduced by more than a factor of two (perhaps a factor of five), lightweight profiler module is all that must be loaded, and the report generating module (`pstats`) is not needed during profiling.

Recursive functions support: Cumulative times in recursive functions are correctly calculated; recursive entries are counted.

Large growth in report generating UI: Distinct profiles runs can be added together forming a comprehensive report; functions that import statistics take arbitrary lists of files; sorting criteria is now based on keywords (instead of 4 integer options); reports shows what functions were profiled as well as what profile file was referenced; output format has been improved.

10.3 Instant Users 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:

```
import profile
profile.run('foo()')
```

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:

```
import profile
profile.run('foo()', 'fooprof')
```

The file 'profile.py' can also be invoked as a script to profile another script. For example:

```
python /usr/local/lib/python1.5/profile.py myscript.py
```

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:

```
import pstats
p = 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 `profile.run()` above, what was printed was the result of three method calls:

```
p.strip_dirs().sort_stats(-1).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 (this is to comply with the semantics of the old profiler). The third method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats('name')
p.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.sort_stats('cumulative').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.sort_stats('time').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.sort_stats('file').print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods ('cause 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.

10.4 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.

10.5 Reference Manual

The primary entry point for the profiler is the global function `profile.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`(*string*[, *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:

```
        main()
        2706 function calls (2004 primitive calls) in 4.504 CPU seconds

    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 this profile was generated by the call:
`profile.run('main()')`, and hence the exec'ed string is `'main()'`. The second 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.

Analysis of the profiler data is done using this class from the `pstats` module:

class `Stats`(*filename*[, ...])

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.

The file selected by the above constructor must have been created by the corresponding version of `profile`. 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 (such as the old system profiler).

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.

10.5.1 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()`. 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` class. 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 the 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:

Valid Arg	Meaning
`'calls'`	call count
`'cumulative'`	cumulative time
`'file'`	file name
`'module'`	file name
`'pcalls'`	primitive call count
`'line'`	line number
`'name'`	function name
`'nfl'`	name/file/line
`'stdname'`	standard name
`'time'`	internal time

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 compatibility with the old profiler, 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. This method is provided primarily for compatibility with the old profiler. Its utility is questionable now that 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. For convenience, a number is shown in parentheses after each caller to show how many times this specific call was made. A second non-parenthesized number is the cumulative time spent in the function at the right.

`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.

`ignore``()`

Deprecated since release 1.5.1. This is not needed in modern versions of Python.[2]

[2]This was once necessary, when Python would print any unused expression result that was not `None`. The method is still defined for backward compatibility.

10.6 Limitations

There are two fundamental limitations on this profiler. The first is that it relies on the Python interpreter to dispatch *call*, *return*, and *exception* events. Compiled C code does not get interpreted, and hence is "invisible" to the profiler. All time spent in C code (including built-in functions) will be charged to the Python function that invoked the C code. If the C code calls out to some native Python code, then those calls will be profiled properly.

The second 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. This profiler 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.

10.7 Calibration

The profiler 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).

```
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:[3]

[3] 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.

```
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.

10.8 Extensions — Deriving Better Profilers

The `Profile` class of module `profile` was 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 module `profile` 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:

```
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func()`. The function 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.

10.9 `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.

class `Profile`(*logfile*[, *lineevents*=*0*[, *linetimings*=*1*]])

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).

10.9.1 Profile Objects

Profile objects have the following methods:

addinfo(*key, value*)
Add an arbitrary labelled value to the profile output.

close()
Close the logfile and terminate the profiler.

fileno()
Return the file descriptor of the profiler's log file.

run(*cmd*)
Profile an `exec`-compatible string in the script environment. The globals from the `__main__` module are used as both the globals and locals for the script.

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 propogate cleanly, while ensuring that profiling is disabled on the way out.

runctx(*cmd, globals, locals*)
Evaluate an `exec`-compatible string in a specific environment. The string is compiled before profiling begins.

start()
Start the profiler.

stop()
Stop the profiler.

10.9.2 Using hotshot data

New in version 2.2.

This module loads hotshot profiling data into the standard `pstats` Stats objects.

load(*filename*)
Load hotshot data from *filename*. Returns an instance of the `pstats.Stats` class.

See Also:

Module `profile` (section 10.5):
The `profile` module's `Stats` class

10.9.3 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)
 .
 .
 .
```

10.10 `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.

10.10.1 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 `time.clock()` to measure CPU time.

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.

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 **-O** option for the older versions to avoid timing `SET_LINENO` instructions.

10.10.2 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.

```
% timeit.py 'try:' '  str.__nonzero__' 'except AttributeError:' '  pass'
100000 loops, best of 3: 15.7 usec per loop
% timeit.py 'if hasattr(str, "__nonzero__"): pass'
100000 loops, best of 3: 4.26 usec per loop
% timeit.py 'try:' '  int.__nonzero__' 'except AttributeError:' '  pass'
1000000 loops, best of 3: 1.43 usec per loop
% timeit.py 'if hasattr(int, "__nonzero__"): pass'
100000 loops, best of 3: 2.23 usec per loop


>>> import timeit
>>> s = """\
... try:
...     str.__nonzero__
... except AttributeError:
...     pass
... """
>>> t = timeit.Timer(stmt=s)
>>> print "%.2f usec/pass" % (1000000 * t.timeit(number=100000)/100000)
17.09 usec/pass
>>> s = """\
... if hasattr(str, '__nonzero__'): pass
... """
>>> t = timeit.Timer(stmt=s)
>>> print "%.2f usec/pass" % (1000000 * t.timeit(number=100000)/100000)
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)/100000)
1.97 usec/pass
>>> s = """\
... if hasattr(int, '__nonzero__'): pass
... """
>>> t = timeit.Timer(stmt=s)
>>> print "%.2f usec/pass" % (1000000 * t.timeit(number=100000)/100000)
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:

```
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()
```

CHAPTER

ELEVEN

Internet Protocols and Support

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.
`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).
`telnetlib`	Telnet client class.
`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.
`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.
`asyncore`	A base class for developing asynchronous socket handling services.
`asynchat`	Support for asynchronous command/response protocols.

11.1 `webbrowser` — Convenient Web-browser controller

The `webbrowser` module provides a very high-level interface to allow displaying Web-based documents to users. The controller objects are easy to use and are platform-independent. 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.

Under UNIX, if the environment variable BROWSER exists, it is interpreted to override the platform default list of browsers, as a colon-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 the `%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 X11 browsers are available on UNIX, the controlling process will not wait for the user to finish with the browser, but allow the browser to maintain its own window on the display.

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 true, a new browser window 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).

`open_new`(*url*)
Open *url* in a new window of the default browser, if possible, otherwise, open *url* in the only browser window.

`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.

Type Name	Class Name	Notes
`'mozilla'`	`Netscape('mozilla')`	
`'netscape'`	`Netscape('netscape')`	
`'mosaic'`	`GenericBrowser('mosaic %s &')`	
`'kfm'`	`Konqueror()`	(1)
`'grail'`	`Grail()`	
`'links'`	`GenericBrowser('links %s')`	
`'lynx'`	`GenericBrowser('lynx %s')`	
`'w3m'`	`GenericBrowser('w3m %s')`	
`'windows-default'`	`WindowsDefault`	(2)
`'internet-config'`	`InternetConfig`	(3)

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; requires the common extension modules `win32api` and `win32con`.

(3) Only on MacOS platforms; requires the standard MacPython `ic` module, described in the *Macintosh Library Modules* manual.

11.1.1 Browser Controller Objects

Browser controllers provide two methods which parallel two of the module-level convenience functions:

open(*url*[, *new*])
Display *url* using the browser handled by this controller. If *new* is true, a new browser window 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.

11.2 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.

11.2.1 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 — Grail 0.3 and Netscape 2.0 do).

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:

```
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:

```
print "<TITLE>CGI script output</TITLE>"
print "<H1>This is my first CGI script</H1>"
print "Hello, world!"
```

11.2.2 Using the cgi module

Begin by writing '`import cgi`'. Do not use '`from cgi import *`' — 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:

```
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:

```
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:

```
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 `isinstance()` built-in function to determine whether you have a single instance or a list of instances. For example, this code concatenates any number of username fields, separated by commas:

```
value = form.getvalue("username", "")
if isinstance(value, list):
    # Multiple username fields specified
    usernames = ",".join(value)
else:
    # Single or no username field specified
    usernames = 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:

```
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`.

11.2.3 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:

```
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:

```
<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:

```
user = form.getvalue("user").toupper()
```

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*])
: Thin 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.[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:

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").toupper()    # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

11.2.4 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.

11.2.5 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.

[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.

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 double-quote 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.

11.2.6 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.

11.2.7 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).

11.2.8 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.

11.2.9 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:

```
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):

```
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.

11.2.10 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 every user on the system.
- 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.

11.3 cgitb — Traceback manager for CGI scripts

New in version 2.2.

The cgitb module provides a special exception handler for Python scripts. (It's 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().

11.4 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:

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, or if the server returns an error code, 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.

Except for the info() and geturl() methods, these methods have the same interface as for file objects — see section 2.2.8 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 `mimetools` 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):

```
% 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:

```
# Use http://www.someproxy.com:3128 for http proxying
proxies = 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.

`_urlopener`
The public functions `urlopen()` and `urlretrieve()` 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:

```
import urllib

class AppURLopener(urllib.FancyURLopener):
    def __init__(self, *args):
        self.version = "App/1.7"
        urllib.FancyURLopener.__init__(self, *args)

urllib._urlopener = AppURLopener()
```

`urlcleanup``()`
Clear the cache that may have been built up by previous calls to `urlretrieve()`.

`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 instance attribute `version` to an appropriate string value before the `open()` method is called.

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*, are used for authentication with the 'https:' scheme. The keywords *key_file* and *cert_file* are supported; both are needed to actually retrieve a resource at an 'https:' URL.

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.

Note: According to the letter of RFC 2616, 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.

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 (e.g. 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.
- 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 unparse URL strings, the recommended interface for URL manipulation is in module `urlparse`.

11.4.1 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 `mimetools.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:

`prompt_user_passwd`(*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, (*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.

11.4.2 Examples

Here is an example session that uses the 'GET' method to retrieve a URL containing parameters:

```
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musi-cal.com/cgi-bin/query?%s" % params)
>>> print f.read()
```

The following example uses the 'POST' method instead:

```
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musi-cal.com/cgi-bin/query", params)
>>> print f.read()
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings:

```
>>> 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:

```
>>> import urllib
>>> opener = urllib.FancyURLopener({})
>>> f = opener.open("http://www.python.org/")
>>> f.read()
```

11.5 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 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 (currently the code checks that it really is a Request instance, or an instance of a subclass of Request).

data should be a string, which specifies additional data to send to the server. In HTTP requests, which are the only ones that support *data*, it should be a buffer in the format of application/x-www-form-urlencoded, for example one returned from urllib.urlencode().

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.

install_opener(*opener*)
Install an OpenerDirector instance as the default opener. 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. *handler*s 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 *handler*s, unless the *handler*s contain them, instances of them or subclasses of them: ProxyHandler, UnknownHandler, HTTPHandler, HTTPDefaultErrorHandler, HTTPRedirectHandler, FTPHandler, FileHandler

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. Besides ProxyHandler, which has handler_order of 100, all handlers currently have it set to 500.

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*]])
This class is an abstraction of a URL request.

url should be a string which is a valid URL. For a description of *data* see the `add_data()` description. *headers* should be a dictionary, and will be treated as if `add_header()` was called with each key and value as arguments.

class `OpenerDirector`()
The `OpenerDirector` class opens URLs via `BaseHandler`s 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 `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 11.5.6 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 11.5.6 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 11.5.6 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 11.5.6 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 11.5.6 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 11.5.6 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.

11.5.1 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 an application/x-www-form-encoded buffer, 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 takes one of the values ("GET", "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.

`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.

11.5.2 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.

- *protocol*`_open()` — signal that the handler knows how to open *protocol* URLs.
- *protocol*`_error_`*type*`()` — signal that the handler knows how to handle *type* errors from *protocol*.

close()
Explicitly break cycles, and delete all the handlers. Because the `OpenerDirector` needs to know the registered handlers, and a handler needs to know who the `OpenerDirector` who called it is, there is a reference cycle. Even though recent versions of Python have cycle-collection, it is sometimes preferable to explicitly break the cycles.

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 default installed `OpenerDirector`).

error(*proto*[, *arg*[, ...]])
Handle an error in a 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()`.

11.5.3 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`:

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().

11.5.4 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.

11.5.5 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.

11.5.6 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 sequene 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.

11.5.7 AbstractBasicAuthHandler Objects

handle_authentication_request(*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* is the host to authenticate to, *req* should be the (failed) Request object, and *headers* should be the error headers.

11.5.8 HTTPBasicAuthHandler Objects

http_error_401(*req, fp, code, msg, hdrs*)
Retry the request with authentication information, if available.

11.5.9 ProxyBasicAuthHandler Objects

http_error_407(*req, fp, code, msg, hdrs*)
Retry the request with authentication information, if available.

11.5.10 AbstractDigestAuthHandler Objects

handle_authentication_request(*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.

11.5.11 HTTPDigestAuthHandler Objects

http_error_401(*req, fp, code, msg, hdrs*)
Retry the request with authentication information, if available.

11.5.12 ProxyDigestAuthHandler Objects

http_error_407(*req, fp, code, msg, hdrs*)
Retry the request with authentication information, if available.

11.5.13 HTTPHandler Objects

http_open(*req*)
Send an HTTP request, which can be either GET or POST, depending on *req*.has_data().

11.5.14 HTTPSHandler Objects

https_open(*req*)
Send an HTTPS request, which can be either GET or POST, depending on *req*.has_data().

11.5.15 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`.

11.5.16 FTPHandler Objects

ftp_open(*req*)
Open the FTP file indicated by *req*. The login is always done with empty username and password.

11.5.17 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*.

11.5.18 GopherHandler Objects

gopher_open(*req*)
Open the gopher resource indicated by *req*.

11.5.19 UnknownHandler Objects

unknown_open()
Raise a `URLError` exception.

11.5.20 Examples

This example gets the python.org main page and displays the first 100 bytes of it:

```
>>> 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:

```
>>> 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\n\nGot Data: "%s"' %
data
```

11.6 `httplib` — HTTP protocol client

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 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`).

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:

```
>>> 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.

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.

11.6.1 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.

`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.

`send`(*data*)
Send data to the server. This should be used directly only after the `endheaders()` method has been called and before `getreply()` has been called.

`putrequest`(*request, selector*)
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`).

`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.

11.6.2 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.

`msg`
A `mimetools.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.

11.6.3 Examples

Here is an example session that uses the '`GET`' method:

```
>>> 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:

```
>>> 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()
```

11.7 `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:

```
>>> 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` (section 12.18):
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.

11.7.1 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.

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 reponds 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).

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).

11.8 gopherlib — Gopher protocol client

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.

11.9 poplib — POP3 protocol client

This module defines a class, POP3, which encapsulates a connection to an POP3 server and implements the protocol as defined in RFC 1725. The POP3 class supports both the minimal and optional command sets.

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.

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 (section 11.10):
The standard Python IMAP module.

Frequently Asked Questions About Fetchmail
(http://www.tuxedo.org/~esr/fetchmail/fetchmail-FAQ.html)
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.

11.9.1 POP3 Objects

All POP3 commands are represented by methods of the same name, in lower-case; 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', ...]`). 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*).

11.9.2 POP3 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
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.

11.10 `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:

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:

exception `IMAP4.error`
Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

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.

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/).

11.10.1 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).

An IMAP4 instance has the following methods:

append(*mailbox*, *flags*, *date_time*, *message*)
Append message to named mailbox.

authenticate(*func*)
Authenticate command — requires response processing. This is currently unimplemented, and raises an exception.

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*.

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.

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.

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. Returned data contains a space separated list of matching message numbers. *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:

```
# M is a connected IMAP4 instance...
msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
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.

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.

subscribe(*mailbox*)
Subscribe to new mailbox.

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 SSLObject instance 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.

11.10.2 IMAP4 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
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.logout()
```

11.11 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:

```
>>> 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 Modules
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):

```
>>> 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:

class NNTP(*host*[, *port* [, *user*[, *password* [, *readermode*]]]])
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', 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.

class NNTPError()
Derived from the standard exception Exception, this is the base class for all exceptions raised by the nntplib module.

class NNTPReplyError()
Exception raised when an unexpected reply is received from the server. For backwards compatibility, the exception error_reply is equivalent to this class.

class 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.

class 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.

class 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.

class NNTPDataError()
Exception raised when there is some error in the response data. For backwards compatibility, the exception error_data is equivalent to this class.

11.11.1 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 *'yymmdd'* 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 article 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.

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, *first* is the first article number in the group, *last* is the last article number in the group, and *name* is the group name. The numbers are returned as strings.

help([*file*])
Send a 'HELP' command. Return a pair (*response*, *list*) where *list* is a list of help strings. If the *file* parameter is supplied, then the output of the 'HELP' 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.

stat(*id*)
Send a 'STAT' command, where *id* is the message id (enclosed in '<' and '>') or an article number (as a string). Return a triple (*response*, *number*, *id*) where *number* is the article number (as a string) and *id* is the article id (enclosed in '<' and '>').

next()
Send a 'NEXT' command. Return as for `stat()`.

last()
Send a 'LAST' command. Return as for `stat()`.

head(*id*)
Send a 'HEAD' command, where *id* has the same meaning as for `stat()`. Return a tuple (*response*, *number*, *id*, *list*) where the first three are the same as for `stat()`, and *list* is a list of the article's headers (an uninterpreted list of lines, without trailing newlines).

body(*id*,[*file*])
Send a 'BODY' command, where *id* has the same meaning as for `stat()`. If the *file* parameter is supplied, then the body 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 body. Return as for `head()`. If *file* is supplied, then the returned *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 id (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. 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 (*response*, *list*), where *list* is a list of tuples containing (*name*, *title*). If the *file* parameter is supplied, then the output of the 'XGTITLE' 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. This is an optional NNTP extension, and may not be supported by all servers.

xover(*start, end,* [*file*])
Return a pair (*resp*, *list*). *list* is a list of tuples, one for each article in the range delimited by the *start* and *end* article numbers. Each tuple is of the form (*article number*, *subject*, *poster*, *date*, *id*, *references*, *size*, *lines*). If the *file* parameter is supplied, then the output of the 'XOVER' 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. This is an optional NNTP extension, and may not be supported by all servers.

xpath(*id*)
Return a pair (*resp*, *path*), where *path* is the directory path to the article with message ID *id*. This is an optional NNTP extension, and may not be supported by all servers.

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.

11.12 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*"
Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

RFC 1869, "*SMTP Service Extensions*"
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.

11.12.1 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 `1` if *name* is in the set of SMTP service extensions returned by the server, `0` 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, 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.

11.12.2 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.

```
import smtplib

def prompt(prompt):
    return raw_input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs  = prompt("To: ").split()
print "Enter message, end with ^D (Unix) or ^Z (Windows):"

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\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()
```

11.13 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.

11.13.1 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.

11.13.2 Telnet Example

A simple example illustrating typical use:

```
import getpass
import sys
import telnetlib

HOST = "localhost"
user = raw_input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.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()
```

11.14 `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 defines the following functions:

`urlparse`(*urlstring*[, *default_scheme*[, *allow_fragments*]])

Parse a URL into 6 components, returning a 6-tuple: (addressing scheme, network location, path, parameters, query, fragment identifier). 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 (e.g. the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the tuple items, except for a leading slash in the *path* component, which is retained if present.

Example:

```
urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
```

yields the tuple

```
('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 string does not specify one. The default value for this argument is the empty string.

If the *allow_fragments* argument is zero, fragment identifiers are not allowed, even if the URL's addressing scheme normally does support them. The default value for this argument is `1`.

`urlunparse`(*tuple*)

Construct a URL string from a tuple as returned by `urlparse()`. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had redundant delimiters, e.g. a ? with an empty query (the draft states that these are equivalent).

urlsplit(*urlstring*[, *default_scheme*[, *allow_fragments*]])
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). 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). New in version 2.2.

urlunsplit(*tuple*)
Combine the elements of a tuple as returned by `urlsplit()` into a complete URL as a string. 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.

Example:

```
urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
```

yields the string

```
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The *allow_fragments* argument has the same meaning 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).

11.15 `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:

`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. 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 return true.

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. 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 mix-in request handler classes StreamRequestHandler or DatagramRequestHandler, which override the setup() and finish() methods, and provides 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.

11.16 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:

```
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 han-

dler'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`
Specifies 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 `mimetools.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 MIME 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 MIME 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.

`date_time_string``()`
Returns the current date and time, formatted for a message header.

`log_data_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` (section 11.18):
Extended request handler that supports CGI scripts.

Module `SimpleHTTPServer` (section 11.17):
Basic request handler that limits response to files actually under the document root.

11.17 SimpleHTTPServer — Simple HTTP request handler

The SimpleHTTPServer module defines a request-handler class, interface compatible with BaseHTTPServer.BaseHTTPRequestHandler which serves files only from a base directory.

The SimpleHTTPServer module defines the following class:

class SimpleHTTPRequestHandler(*request, client_address, server*)
This class is used, to serve files from current directory and below, directly mapping the directory structure to HTTP requests.

A lot of the work is done by the base class BaseHTTPServer.BaseHTTPRequestHandler, such as parsing the request. 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. Default is signified by an empty string, and is considered to be text/plain. 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 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, a 403 respond is output, followed by the explanation 'Directory listing not supported'. Any IOError exception in opening the requested file, is mapped to a 404, 'File not found' error. Otherwise, the content type is guessed using the *extensions_map* variable.

A 'Content-type:' with the guessed content type is output, and then a blank line, signifying end of headers, and then the contents of the file. The file is always opened in binary mode.

For example usage, see the implementation of the test() function.

See Also:

Module BaseHTTPServer (section 11.16):
Base class implementation for Web server and request handler.

11.18 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.

The CGIHTTPServer module defines the following class:

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:

`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` (section 11.16):
Base class implementation for Web server and request handler.

11.19 `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:

RFC 2109, "*HTTP State Management Mechanism*"
This is the state management specification implemented by this module.

11.19.1 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 a newline.

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 `Morsel`s. If it is a dictionary, it is equivalent to:

```
for k, v in rawdata.items():
    cookie[k] = v
```

11.19.2 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()`.

11.19.3 Example

The following example demonstrates how to use the `Cookie` module.

```
>>> 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 header)
>>> print C
Set-Cookie: vienna=finger;
Set-Cookie: chips=ahoy;
>>> C = Cookie.SmartCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\\012;";')
>>> print C
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;";
>>> C = Cookie.SmartCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print C
Set-Cookie: oreo=doublestuff; Path=/;
>>> C = Cookie.SmartCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = Cookie.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print C
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;
```

11.20 `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.

class `ServerProxy`(*uri*[, *transport*[, *encoding*[, *verbose*[, *allow_none*]]]])

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.html for a description.

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):

Name	Meaning
`boolean`	The `True` and `False` constants
`integers`	Pass in directly
`floating-point numbers`	Pass in directly
`strings`	Pass in directly
`arrays`	Any Python sequence type containing conformable elements. Arrays are returned as list
`structures`	A Python dictionary. Keys must be strings, values may be any conformable type.
`dates`	in seconds since the epoch; pass in an instance of the `DateTime` wrapper class
`binary data`	pass in an instance of the `Binary` wrapper class

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. Note that even though starting with Python 2.2 you can subclass builtin types, the xmlrpclib 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`.

See Also:

XML-RPC HOWTO
(http://xmlrpc-c.sourceforge.net/xmlrpc-howto/xmlrpc-howto.html)

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

(http://xmlrpc-c.sourceforge.net/hacks.php)
Extensions for various open-source libraries to support instrospection and multicall.

11.20.1 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.

11.20.2 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.

11.20.3 DateTime Objects

This class may initialized from date in seconds since the epoch, a time tuple, or an ISO 8601 time/date string. 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.

11.20.4 Binary Objects

This class may initialized from string data (which may include NULs). The primary acess 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:

`decode`(*string*)
Accept a base64 string and decode it as the instance's new data.

`encode`(*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.

11.20.5 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.

11.20.6 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.

11.20.7 Convenience Functions

`boolean`(*value*)
Convert any Python value to one of the XML-RPC Boolean constants, `True` or `False`.

`binary`(*data*)
Trivially convert any Python string to a `Binary` object.

11.20.8 Example of Client Usage

```
# simple test program (from the XML-RPC specification)

# 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
```

11.21 `SimpleXMLRPCServer` — Basic XML-RPC server

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`.

class `SimpleXMLRPCServer`(*addr*[, *requestHandler*[, *logRequests*]])
Create a new server instance. 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. This class provides methods for registration of functions that can be called by the XML-RPC protocol.

class `CGIXMLRPCRequestHandler`()
Create a new instance to handle XML-RPC requests in a CGI environment. New in version 2.3.

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.

11.21.1 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*)
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`()
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.

Example:

```
class MyFuncs:
    def div(self, x, y) : return div(x,y)


server = SimpleXMLRPCServer(("localhost", 8000))
server.register_function(pow)
server.register_function(lambda x,y: x+y, 'add')
server.register_introspection_functions()
server.register_instance(MyFuncs())
server.serve_forever()
```

11.21.2 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:

```
class MyFuncs:
    def div(self, x, y) : return div(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()
```

11.22 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.

class DocXMLRPCServer(*addr*[, *requestHandler*[, *logRequests*]])
Create a new server instance. All parameters have the same meaning as for SimpleXMLRPCServer.SimpleXMLRPCServer; *requestHandler* defaults to DocXMLRPCRequestHandler.

class DocCGIXMLRPCRequestHandler()
Create a new instance to handle XML-RPC requests in a CGI environment.

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.

11.22.1 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.

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.

11.22.2 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.

11.23 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 `asynchat` module is invaluable.

The basic idea behind both modules is to create one or more network *channels*, instances of class `asyncore.dispatcher` and `asynchat.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*]]]`)`

Enter a polling loop that only terminates after all open channels have been closed. All arguments are optional. The *timeout* argument sets the timeout parameter for the appropriate `select()` or `poll()` call, measured in seconds; the default is 30 seconds. The *use_poll* parameter, if true, indicates that `poll()` should be used in preference to `select()` (the default is `False`). The *map* parameter is a dictionary whose items are the channels to watch. As channels are closed they are deleted from their map. If *map* is omitted, a global map is used (this map is updated by the default class `__init__()` – make sure you extend, rather than override, `__init__()` if you want to retain this behavior).

Channels (instances of `asyncore.dispatcher`, `asynchat.async_chat` and subclasses thereof) can freely be mixed in the map.

class `dispatcher``()`

The `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.

`ac_in_buffer_size`

The asynchronous input buffer size (default `4096`).

`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:

Event	Description
`handle_connect()`	Implied by the first write event
`handle_close()`	Implied by a read event with no data available
`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:

```
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.

11.23.1 asyncore Example basic HTTP client

As a basic example, below is a very basic HTTP client that uses the `dispatcher` class to implement its socket handling:

```
class http_client(asyncore.dispatcher):
    def __init__(self, host,path):
        asyncore.dispatcher.__init__(self)
        self.path = path
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connect( (host, 80) )
        self.buffer = 'GET %s HTTP/1.0\r\n\r\n' % self.path

    def handle_connect(self):
        pass

    def handle_read(self):
        data = self.recv(8192)
        print data

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]
```

11.24 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 `asynchat.async_chat`, can freely be mixed in the channel map. Typically an `asyncore.dispatcher` server channel generates new `asynchat.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.

term	Description
string	Will call `found_terminator()` when the string is found in the input stream
integer	Will call `found_terminator()` when the indicated number of characters have been received
`None`	The channel continues to collect data forever

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.

11.24.1 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*.

11.24.2 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.

```
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, 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()
```

CHAPTER

TWELVE

Internet Data Handling

This chapter describes modules which support handling data formats commonly used on the Internet.

formatter	Generic output formatter and device interface.
email.Iterators	Iterate over a message object tree.
mailcap	Mailcap file handling.
mailbox	Read various mailbox formats.
mhlib	Manipulate MH mailboxes from Python.
mimetools	Tools for parsing MIME-style message bodies.
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	Encode and decode files using the MIME base64 data.
binascii	Tools for converting between binary and various ASCII-encoded binary representations.
binhex	Encode and decode files in binhex4 format.
quopri	Encode and decode files using the MIME quoted-printable encoding.
uu	Encode and decode files in uuencode format.
xdrlib	Encoders and decoders for the External Data Representation (XDR).
netrc	Loading of '.netrc' files.
robotparser	Loads a 'robots.txt' file and answers questions about fetchability of other URLs.
csv	Write and read tabular data to and from delimited files.

12.1 `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.

12.1.1 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_flowing_data(*data*)
Provide data which should be formatted with collapsed whitespace. Whitespace 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.

12.1.2 Formatter Implementations

Two implementations of formatter objects are provided by this module. Most applications may use one of these classes without modification or subclassing.

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.

12.1.3 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.

12.1.4 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.

12.2 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 RFC 2822-based message documents. It subsumes most of the functionality in several older standard modules such as rfc822, mimetools, 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) servers; that is the function of the smtplib module. 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 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 rearrange 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 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 (section 11.12):
SMTP protocol client

12.2.1 Representing an email message

The central class in the email package is the Message class; 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:

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 more flexibility, instantiate a `Generator` instance and use its `flatten()` method directly. For example:

```
from cStringIO import StringIO
from email.Generator import Generator
fp = StringIO()
g = Generator(mangle_from_=False, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

__str__()
Equivalent to `as_string(unixfrom=True)`.

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.

set_unixfrom(*unixfrom*)
Set the message's envelope header to *unixfrom*, which should be a string.

get_unixfrom()
Return the message's envelope header. Defaults to `None` if the envelope header was never set.

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.

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.:

```
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.:

```
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`.

Your application should be prepared to deal with 3-tuple return values, and can convert the parameter to a Unicode string like so:

```
param = msg.get_param('foo')
if isinstance(param, tuple):
    param = unicode(param[2], param[0] or 'us-ascii')
```

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*[, *requote*[, *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, or *failobj* if either the header is missing, or has no `filename` parameter. 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.

get_charsets([*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:

```
>>> for part in msg.walk():
>>>     print part.get_content_type()
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
```

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.

One note: when generating the flat text for a multipart message that has no *epilogue* (using the standard Generator class), no newline is added after the closing boundary line. If the message object has an *epilogue* and its value does not start with a newline, a newline is printed after the closing boundary. This

seems a little clumsy, but it makes the most practical sense. The upshot is that if you want to ensure that a newline get printed after your closing multipart boundary, set the *epilogue* to the empty string.

Deprecated methods

The following methods are deprecated in `email` version 2. They are documented here for completeness.

`add_payload`(*payload*)
Add *payload* to the message object's existing payload. If, prior to calling this method, the object's payload was `None` (i.e. never before set), then after this method is called, the payload will be the argument *payload*.

If the object's payload was already a list (i.e. `is_multipart()` returns 1), then *payload* is appended to the end of the existing payload list.

For any other type of existing payload, `add_payload()` will transform the new payload into a list consisting of the old payload and *payload*, but only if the document is already a MIME multipart document. This condition is satisfied if the message's Content-Type: header's main type is either multipart, or there is no Content-Type: header. In any other situation, `MultipartConversionError` is raised.

Deprecated since release 2.2.2. Use the `attach()` method instead.

`get_type`([*failobj*])
Return the message's content type, as a string of the form maintype/subtype as taken from the Content-Type: header. The returned string is coerced to lowercase.

If there is no Content-Type: header in the message, *failobj* is returned (defaults to `None`).

Deprecated since release 2.2.2. Use the `get_content_type()` method instead.

`get_main_type`([*failobj*])
Return the message's *main* content type. This essentially returns the *maintype* part of the string returned by `get_type()`, with the same semantics for *failobj*.

Deprecated since release 2.2.2. Use the `get_content_maintype()` method instead.

`get_subtype`([*failobj*])
Return the message's sub-content type. This essentially returns the *subtype* part of the string returned by `get_type()`, with the same semantics for *failobj*.

Deprecated since release 2.2.2. Use the `get_content_subtype()` method instead.

12.2.2 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.

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.

The primary parser class is `Parser` which parses both the headers and the payload of the message. In the case of multipart messages, it will recursively parse the body of the container message. Two modes of parsing are supported, *strict* parsing, which will usually reject any non-RFC compliant message, and *lax* parsing, which attempts to adjust for common MIME formatting problems.

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.

Parser class API

class `Parser`(`[`*_class*`[`, *strict*`]]`)

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 specifies whether strict or lax parsing should be performed. Normally, when things like MIME terminating boundaries are missing, or when messages contain other formatting problems, the `Parser` will raise a `MessageParseError`. However, when lax parsing is enabled, the `Parser` will attempt to work around such broken formatting to produce a usable message structure (this doesn't mean `MessageParseError`s are never raised; some ill-formatted messages just can't be parsed). The *strict* flag defaults to `False` since lax parsing usually provides the most convenient behavior.

Changed in version 2.2.2: The *strict* flag was added.

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 *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 *headersonly* is as with the `parse()` method.

Changed in version 2.2.2: The *headersonly* flag was added.

`parsestr`(*text*`[`, *headersonly*`]`)

Similar to the `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 *text* in a `StringIO` instance first and calling `parse()`.

Optional *headersonly* is a flag specifying whether to stop parsing after reading the headers or not. The default is `False`, meaning it parses the entire contents of the file.

Changed in version 2.2.2: The *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 `email` package namespace.

`message_from_string`(*s*`[`, *_class*`[`, *strict*`]]`)

Return a message object structure from a string. This is exactly equivalent to `Parser().parsestr(s)`. Optional *_class* and *strict* are interpreted as with the `Parser` class constructor.

Changed in version 2.2.2: The *strict* flag was added.

`message_from_file`(*fp*`[`, *_class*`[`, *strict*`]]`)

Return a message object structure tree from an open file object. This is exactly equivalent to `Parser().parse(fp)`. Optional *_class* and *strict* are interpreted as with the `Parser` class constructor.

Changed in version 2.2.2: The *strict* flag was added.

Here's an example of how you might use this at an interactive Python prompt:

```
>>> import email
>>> msg = email.message_from_string(myString)
```

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/deliver-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.

12.2.3 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:

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 broken on semicolons and continued as per RFC 2822. If no semicolon is found, then the header is left alone. Set to zero to disable wrapping headers. 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.

Deprecated methods

The following methods are deprecated in `email` version 2. They are documented here for completeness.

`__call__`(*msg*[, *unixfrom*])

This method is identical to the `flatten()` method.

Deprecated since release 2.2.2. Use the `flatten()` method instead.

12.2.4 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. Each of these classes should be imported from a module with the same name as the class, from within the `email` package. E.g.:

```
import email.MIMEImage.MIMEImage
```

or

```
from email.MIMEText import MIMEText
```

Here are the classes:

class `MIMEBase`(*_maintype*, *_subtype*, ***_params*)

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`()

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*]]]])

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 `MIMEAudio`(*_audiodata*[, *_subtype*[, *_encoder*[, ***_params*]]])

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*]]])

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*])

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*[, *_encoder*]]])

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.

Deprecated since release 2.2.2. The *_encoding* argument has been deprecated. Encoding now happens implicitly based on the *_charset* argument.

12.2.5 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 transfered 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. 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:

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.

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. `isinstance(s, str)` is true), then *charset* is the encoding of that byte string, and a `UnicodeError` will be raised if the string cannot be decoded with that character set.

If *s* is a Unicode string, then *charset* is a hint specifying the character set of the characters in the string. In this case, when producing an RFC 2822-compliant header using RFC 2047 rules, the Unicode string will be encoded using the following charsets in order: `us-ascii`, the *charset* hint, `utf-8`. The first character set to not provoke a `UnicodeError` is used.

Optional *errors* is passed through to any `unicode()` or `ustr.encode()` call, and defaults to "strict".

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 *splitchars* is a string containing characters to split long ASCII lines on, in rough support of RFC 2822's *highest level syntactic breaks*. This doesn't affect RFC 2047 encoded lines.

The `Header` class also provides a number of methods to support standard operators and built-in functions.

__**str**__()
A synonym for `Header.encode()`. Useful for `str(aHeader)`.

__**unicode**__()
A helper for the built-in `unicode()` function. Returns the header as a Unicode string.

__**eq**__(*other*)
This method allows you to compare two `Header` instances for equality.

__**ne**__(*other*)
This method allows you to compare two `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
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[('p\\xf6stal', 'iso-8859-1')]
```

make_header(*decoded_seq*[, *maxlinelen*[, *header_name*[, *continuation_ws*]]])
Create a `Header` instance from a sequence of pairs as returned by `decode_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.

12.2.6 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.

New in version 2.2.2.

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.

12.2.7 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 `MIMEImage` and `MIMEText` 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`[1]. 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.

12.2.8 Exception 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

[1] Note that encoding with `encode_quopri()` also encodes all tabs and space characters in the data.

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`).

12.2.9 Miscellaneous utilities

There are several useful utilities provided with the `email` package.

`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:

```
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. 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)[2]. 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. It 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*]])
Returns a date string as per RFC 2822, 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.

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*.

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)`.

The following functions have been deprecated:

dump_address_pair(*pair*)
Deprecated since release 2.2.2. Use `formataddr()` instead.

decode(*s*)
Deprecated since release 2.2.2. Use `Header.decode_header()` instead.

encode(*s*[, *charset*[, *encoding*]])
Deprecated since release 2.2.2. Use `Header.encode()` instead.

12.2.10 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

[2]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.

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:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
    text/plain
    text/plain
    multipart/digest
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
    text/plain
```

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.

12.2.11 Differences from `email` v1 (up to Python 2.2.1)

Version 1 of the `email` package was bundled with Python releases up to Python 2.2.1. Version 2 was developed for the Python 2.3 release, and backported to Python 2.2.2. It was also available as a separate distutils based package. `email` version 2 is almost entirely backward compatible with version 1, with the following differences:

- 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 `MIMEMultipart` 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.

12.2.12 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[3].

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.

[3] Delivery Status Notifications (DSN) are defined in RFC 1894.

12.2.13 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:

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.MIMEText import MIMEText

# 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()
```

Here's an example of how to send a MIME message containing a bunch of family pictures that may be residing in a directory:

```
# Import smtplib for the actual sending function
import smtplib

# Here are the email pacakge modules we'll need
from email.MIMEImage import MIMEImage
from email.MIMEMultipart import MIMEMultipart

COMMASPACE = ', '

# Create the container (outer) email message.
msg = MIMEMultipart()
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'
# Guarantees the message ends in a newline
msg.epilogue = ''

# 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 automatically
    # 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()
```

Here's an example of how to send the entire contents of a directory as an email message: [4]

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

    """Send the contents of a directory as a MIME message.

    Usage: dirmail [options] from to [to ...]*

    Options:
        -h / --help
            Print this message and exit.

        -d directory
        --directory=directory
            Mail the contents of the specified directory, otherwise use the
            current directory.  Only the regular files in the directory are sent,
            and we don't recurse to subdirectories.

    'from' is the email address of the sender of the message.

    'to' is the email address of the recipient of the message, and multiple
    recipients may be given.

    The email is sent by forwarding to your local SMTP server, which then does the
    normal delivery process.  Your local machine must be running an SMTP server.
    """

    import sys
    import os
    import getopt
    import smtplib
    # For guessing MIME type based on file name extension
    import mimetypes

    from email import Encoders
    from email.Message import Message
    from email.MIMEAudio import MIMEAudio
    from email.MIMEMultipart import MIMEMultipart
    from email.MIMEImage import MIMEImage
    from email.MIMEText import MIMEText

    COMMASPACE = ', '


    def usage(code, msg=''):
        print >> sys.stderr, __doc__
        if msg:
            print >> sys.stderr, msg
        sys.exit(code)


    def main():
        try:
            opts, args = getopt.getopt(sys.argv[1:], 'hd:', ['help', 'directory='])
```

[4] Thanks to Matthew Dixon Cowles for the original inspiration and examples.

```
except getopt.error, msg:
    usage(1, msg)

dir = os.curdir
for opt, arg in opts:
    if opt in ('-h', '--help'):
        usage(0)
    elif opt in ('-d', '--directory'):
        dir = arg

if len(args) < 2:
    usage(1)

sender = args[0]
recips = args[1:]

# Create the enclosing (outer) message
outer = MIMEMultipart()
outer['Subject'] = 'Contents of directory %s' % os.path.abspath(dir)
outer['To'] = COMMASPACE.join(recips)
outer['From'] = sender
outer.preamble = 'You will not see this in a MIME-aware mail reader.\n'
# To guarantee the message ends with a newline
outer.epilogue = ''

for filename in os.listdir(dir):
    path = os.path.join(dir, filename)
    if not os.path.isfile(path):
        continue
    # Guess the content type based on the file's extension.  Encoding
    # will be ignored, although we should check for simple things like
    # 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 (compressed), so
        # use a generic bag-of-bits type.
        ctype = 'application/octet-stream'
    maintype, subtype = ctype.split('/', 1)
    if maintype == 'text':
        fp = open(path)
        # Note: we should handle calculating the charset
        msg = MIMEText(fp.read(), _subtype=subtype)
        fp.close()
    elif maintype == 'image':
        fp = open(path, 'rb')
        msg = MIMEImage(fp.read(), _subtype=subtype)
        fp.close()
    elif maintype == 'audio':
        fp = open(path, 'rb')
        msg = MIMEAudio(fp.read(), _subtype=subtype)
        fp.close()
    else:
        fp = open(path, 'rb')
        msg = MIMEBase(maintype, subtype)
        msg.set_payload(fp.read())
        fp.close()
        # Encode the payload using Base64
        Encoders.encode_base64(msg)
    # Set the filename parameter
    msg.add_header('Content-Disposition', 'attachment', filename=filename)
    outer.attach(msg)

# Now send the message
```

```
    s = smtplib.SMTP()
    s.connect()
    s.sendmail(sender, recips, outer.as_string())
    s.close()


if __name__ == '__main__':
    main()
```

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.

Usage: unpackmail [options] msgfile

Options:
    -h / --help
        Print this message and exit.

    -d directory
    --directory=directory
        Unpack the MIME message into the named directory, which will be
        created if it doesn't already exist.

msgfile is the path to the file containing the MIME message.
"""

import sys
import os
import getopt
import errno
import mimetypes
import email


def usage(code, msg=''):
    print >> sys.stderr, __doc__
    if msg:
        print >> sys.stderr, msg
    sys.exit(code)


def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], 'hd:', ['help', 'directory='])
    except getopt.error, msg:
        usage(1, msg)

    dir = os.curdir
    for opt, arg in opts:
        if opt in ('-h', '--help'):
            usage(0)
        elif opt in ('-d', '--directory'):
            dir = arg

    try:
        msgfile = args[0]
    except IndexError:
        usage(1)

    try:
        os.mkdir(dir)
```

```
        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 that an
            # 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(dir, filename), 'wb')
            fp.write(part.get_payload(decode=1))
            fp.close()


    if __name__ == '__main__':
        main()
```

12.3 `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.

`findmatch`(*caps*, *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:

```
>>> import mailcap
>>> d=mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='/tmp/tmp1223')
('xmpeg /tmp/tmp1223', {'view': 'xmpeg %s'})
```

12.4 `mailbox` — Read various mailbox formats

This module defines a number of classes that allow easy and uniform access to mail messages in a (UNIX) mailbox.

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).

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 *dirname*. *factory* is as with the `UnixMailbox` class.

class `Maildir`(*dirname*[, *factory*])
Access a Qmail mail directory. All new and current mail for the mailbox specified by *dirname* is made available. *factory* is as with the `UnixMailbox` class.

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 *original* headers and the *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 `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'`. *factory* is as with the `UnixMailbox` class.

Note that because the `rfc822` module is deprecated, it is recommended that you use the `email` package to create message objects from a mailbox. (The default can't be changed for backwards compatibility reasons.) The safest way to do this is with bit of code:

```
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)
```

The above wrapper is defensive against ill-formed MIME messages in the mailbox, but you have to be prepared to receive the empty string from the mailbox's `next()` method. On the other hand, if you know your mailbox contains only well-formed MIME messages, you can simplify this to:

```
import email
import mailbox

mbox = mailbox.UnixMailbox(fp, email.message_from_file)
```

See Also:

mbox - file containing mail messages
(http://www.qmail.org/man/man5/mbox.html)

Description of the traditional "mbox" mailbox format.

maildir - directory for incoming mail messages
(http://www.qmail.org/man/man5/maildir.html)

Description of the "maildir" mailbox format.

Configuring Netscape Mail on UNIX*: Why the Content-Length Format is Bad*
(http://home.netscape.com/eng/mozilla/2.0/relnotes/demo/content-length.html)

A description of problems with relying on the Content-Length: header for messages stored in mailbox files.

12.4.1 Mailbox Objects

All implementations of mailbox objects are iterable objects, and have one externally visible method. This method is used by iterators created from mailbox objects and may also be used directly.

`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 sim-

ulating 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`.

12.5 `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.

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`.

12.5.1 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.

12.5.2 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.

12.5.3 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).

12.6 `mimetools` — 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` (section 12.2):
Comprehensive email handling package; supercedes the `mimetools` module.

Module `rfc822` (section 12.11):
Provides the base class for `mimetools.Message`.

Module `multifile` (section 12.10):
Support for reading files which contain distinct parts, such as MIME data.

http://www.cs.uu.nl/wais/html/na-dir/mail/mime-faq/.html
The MIME Frequently Asked Questions document. For an overview of MIME, see the answer to question 1.1 in Part 1 of this document.

12.6.1 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.

12.7 `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 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.

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.

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.

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.

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:

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.

12.7.1 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.

12.8 `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.

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.

12.8.1 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.

12.9 `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 *decode_base64* argument is provided and tests true, any parts that are coded in the base64 encoding are decoded as well.

mime_decode_header(*line*)
: Return a decoded version of the encoded header line in *line*. This only supports the ISO 8859-1 charset (Latin-1).

mime_encode_header(*line*)
: Return a MIME-encoded version of the header line in *line*.

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 `MAXLEN` characters (default value 200).

CHARSET
: When not specified in the mail headers, a character set must be filled in. The string used is stored in `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:

```
mimify.py -e [-l length] [infile [outfile]]
mimify.py -d [-b] [infile [outfile]]
```

to encode (mimify) and decode (unmimify) respectively. *infile* defaults to standard input, *outfile* defaults to standard output. The same file can be specified for input and output.

If the **-l** option is given when encoding, if there are any lines longer than the specified *length*, the containing part will be encoded.

If the **-b** option is given when decoding, any base64 parts will be decoded as well.

See Also:

Module `quopri` (section 12.15):
: Encode and decode MIME quoted-printable files.

12.10 `multifile` — Support for files containing distinct parts

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.

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` (section 12.2):
Comprehensive email handling package; supercedes the `multifile` module.

12.10.1 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 an appropriately decorated version of this boundary is found as an input line, it will be interpreted as a section-divider or end-marker. 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.

12.10.2 `MultiFile` Example

```
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()
```

12.11 `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.[5] Such messages consist of a collection of message headers, and a message body. This module also

[5]This module originally conformed to RFC 822, hence the name. Since then, RFC 2822 has been released as an update to RFC 822. This

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. *m*`['From']`, *m*`['from']` and *m*`['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

module should be considered RFC 2822-conformant, especially in cases where the syntax or semantics have changed since RFC 822.

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` (section 12.2):
Comprehensive email handling package; supercedes the `rfc822` module.

Module `mailbox` (section 12.4):
Classes to read various mailbox formats produced by end-user mail programs.

Module `mimetools` (section 12.6):
Subclass of `rfc822.Message` that handles MIME encoded messages.

12.11.1 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.

12.11.2 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).

12.12 `base64` — Encode and decode MIME base64 data

This module performs base64 encoding and decoding of arbitrary binary strings into text strings that can be safely sent by email or included as part of an HTTP POST request. The encoding scheme is defined in 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") and is used for MIME email and various other Internet-related applications; it is not the same as the output produced by the **uuencode** program. For example, the string `'www.python.org'` is encoded as the string `'d3d3LnB5dGhvbi5vcmc=\n'`.

`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'`).

See Also:

Module `binascii` (section 12.13):
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 In*
Section 5.2, "Base64 Content-Transfer-Encoding," provides the definition of the base64 encoding.

12.13 `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` or `binhex` instead, this module solely exists because bit-manipulation of large amounts of data is slow in Python.

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 *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` (section 12.12):
Support for base64 encoding used in MIME email messages.

Module `binhex` (section 12.14):
Support for the binhex format used on the Macintosh.

Module `uu` (section 12.16):
Support for UU encoding used on UNIX.

Module `quopri` (section 12.15):
Support for quoted-printable encoding used in MIME email messages.

12.14 `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` (section 12.13):
Support module containing ASCII-to-binary and binary-to-ASCII conversions.

12.14.1 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.

12.15 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.

encodestring(*s*[, *quotetabs*])
Like encode(), except that it accepts a source string and returns the corresponding encoded string. *quotetabs* is optional (defaulting to 0), and is passed straight through to encode().

See Also:

Module mimify (section 12.9):
General utilities for processing of MIME messages.

Module base64 (section 12.12):
Encode and decode MIME base64 data

12.16 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*]])
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.

exception Error()

Subclass of `Exception`, this can be raised by `uu.decode()` under various situations, such as described above, but also including a badly formated header, or truncated input file.

See Also:

Module `binascii` (section 12.13):
Support module containing ASCII-to-binary and binary-to-ASCII conversions.

12.17 `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.

12.17.1 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.

12.17.2 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.

12.17.3 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:

```
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError, instance:
    print 'packing the double failed:', instance.msg
```

12.18 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.

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.

12.18.1 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.

12.19 `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.

```
>>> 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
```

12.20 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 generally be printable ASCII to be safe. 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.

12.20.1 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. 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 12.20.2, "Dialects and Formatting Parameters" for details of these parameters.

  All data read are returned as strings. No automatic data type conversion is performed.

`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 12.20.2, "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.

`register_dialect`(*name, dialect*)
Associate *dialect* with *name*. *dialect* must be a subclass of `csv.Dialect`. *name* must be a string or Unicode object.

`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.

`get_dialect`(*name*)
Return the dialect associated with *name*. An `Error` is raised if *name* is not a registered dialect name.

`list_dialects`()
Return the names of all registered dialects.

The `csv` module defines the following classes:

class `DictReader`(*csvfile, fieldnames*[, *restkey=None*[, *restval=None*[, *dialect='excel'*[, *fmtparam*]]]])
Create an object which operates like a regular reader but maps the information read into a dict whose keys are given by the *fieldnames* parameter. 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. All other parameters are interpreted as for `reader` objects.

class `DictWriter`(*csvfile, fieldnames*[, *restval=""*[, *extrasaction='raise'*[, *dialect='excel'*[, *fmtparam*]]]])
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. All other parameters are interpreted as for `writer` objects.

class `Dialect`
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` instance.

class `Sniffer`()
The `Sniffer` class is used to deduce the format of a CSV file.

The `Sniffer` class provides a single method:

`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 the current *delimiter* or begin with the current *quotechar*.

QUOTE_NONNUMERIC
Instructs `writer` objects to quote all non-numeric fields.

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. When `QUOTE_NONE` is in effect, it is an error not to have a single-character *escapechar* defined, even if no data to be written contains the *delimiter* character.

The `csv` module defines the following exception:

exception `Error`
Raised by any of the functions when an error is detected.

12.20.2 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 *quotechar* appearing inside a field should be themselves be quoted. When `True`, the character is doubledd. When `False`, the *escapechar* must be a one-character string which is used as a prefix to the *quotechar*. It defaults to `True`.

`escapechar`
A one-character string used to escape the *delimiter* if *quoting* is set to `QUOTE_NONE`. It defaults to `None`.

`lineterminator`
The string used to terminate lines in the CSV file. It defaults to `'\r\n'`.

`quotechar`
A one-character string used to quote elements containing the *delimiter* or which start with the *quotechar*. It defaults to `'"'`.

`quoting`
Controls when quotes should be generated by the writer. It can take on any of the `QUOTE_*` constants (see section 12.20.1) and defaults to `QUOTE_MINIMAL`.

`skipinitialspace`
When `True`, whitespace immediately following the *delimiter* is ignored. The default is `False`.

12.20.3 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.

12.20.4 Writer Objects

Writer objects (`DictWriter` instances and objects returned by the `writer()` function) have the following public methods:

`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 to the writer's file object, formatted according to the current dialect.

12.20.5 Examples

The "Hello, world" of csv reading is

```
import csv
reader = csv.reader(file("some.csv"))
for row in reader:
    print row
```

The corresponding simplest possible writing example is

```
import csv
writer = csv.writer(file("some.csv", "w"))
for row in someiterable:
    writer.writerow(row)
```

CHAPTER

THIRTEEN

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.
`sgmllib`	Only as much of an SGML parser as needed to parse HTML.
`htmllib`	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.
`xml.sax.saxutils`	Convenience functions and classes for use with SAX.
`xml.sax.xmlreader`	Interface which SAX-compliant XML parsers must implement.
`xmllib`	A parser for XML documents.

See Also:

Python/XML Libraries
(http://pyxml.sourceforge.net/)
Home page for the PyXML package, containing an extension of `xml` package bundled with Python.

13.1 `HTMLParser` — Simple HTML and XHTML parser

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 `sgmllib`.

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.

`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 pro-

cessing 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 instruction. An XHTML processing instruction using the trailing '`?`' will cause the '`?`' to be included in *data*.

13.1.1 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:

```
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
```

13.2 `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 '`--`'.

`SGMLParser` instances have the following interface 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 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 'unknown_starttag('a', [('href', 'http://www.cwi.nl/')])'. The base implementation simply calls *method* with *attributes* as the only argument.

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*;'. In the base implementation, *ref* must be a decimal number in the range 0-255. It translates the character to ASCII and calls the method handle_data() with the character as argument. If *ref* is invalid or out of range, the method unknown_charref(*ref*) is called to handle the error. A subclass must override this method to provide support for named character entities.

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 looks for *ref* in the instance (or class) variable entitydefs which should be a mapping from entity names to corresponding translations. If a translation is found, it calls the method handle_data() with the translation; otherwise, it calls the method unknown_entityref(*ref*). The default entitydefs defines translations for &, &apos, >, <, and ".

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* 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* that does not come with a matching closing tag. The *attributes* argument has the same meaning as described for handle_starttag() above.

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 start_*tag*() are pushed on this stack. Definition of an end_*tag*() method is optional for these tags. For tags processed by do_*tag*() or by unknown_tag(), no end_*tag*() method must be defined; if defined, it will not be used. If both start_*tag*() and do_*tag*() methods exist for a tag, the start_*tag*() method takes precedence.

13.3 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 tags, these are processed immediately; incomplete elements 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:

  ```
  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 single class:

class `HTMLParser`(*formatter***)**
This is the basic HTML parser class. It supports all entity names required by the HTML 2.0 specification (RFC 1866). It also defines handlers for all HTML 2.0 and many HTML 3.0 and 3.2 elements.

See Also:

Module `formatter` (section 12.1):
Interface definition for transforming an abstract flow of formatting events into specific output events on writer objects.

Module `HTMLParser` (section 13.1):
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` (section 13.4):
Definition of replacement text for HTML 2.0 entities.

Module `sgmllib` (section 13.2):
Base class for `HTMLParser`.

13.3.1 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`()
Begins 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.

13.4 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.

13.5 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 13.5.2, “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
(http://www.libexpat.org/)
Home page of the Expat project.

13.5.1 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.

`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 0, 8-bit strings containing UTF-8 encoded data will be passed to the handlers. 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.

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 delaration. 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 paremeter 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 `StartCdataSectionHandler` 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.

13.5.2 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.

13.5.3 Example

The following program defines three handlers that just print out their arguments.

```
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
```

13.5.4 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*`.

13.5.5 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

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.

13.6 `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 13.6.3, "Conformance," for a detailed discussion of mapping requirements.

See Also:

Document Object Model (DOM) Level 2 Specification
(http://www.w3.org/TR/DOM-Level-2-Core/)
The W3C recommendation upon which the Python DOM API is based.

Document Object Model (DOM) Level 1 Specification
(http://www.w3.org/TR/REC-DOM-Level-1/)
The W3C recommendation for the DOM supported by `xml.dom.minidom`.

PyXML
(http://pyxml.sourceforge.net)
: Users that require a full-featured implementation of DOM should use the PyXML package.

CORBA Scripting with Python
(http://cgi.omg.org/cgi-bin/doc?orbos/99-08-02.pdf)
: This specifies the mapping from OMG IDL to Python.

13.6.1 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.

13.6.2 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.

Interface	Section	Purpose
`DOMImplementation`	13.6.2	Interface to the underlying implementation.
`Node`	13.6.2	Base interface for most objects in a document.
`NodeList`	13.6.2	Interface for a sequence of nodes.
`DocumentType`	13.6.2	Information about the declarations needed to process a document.
`Document`	13.6.2	Object which represents an entire document.
`Element`	13.6.2	Element nodes in the document hierarchy.
`Attr`	13.6.2	Attribute value nodes on element nodes.
`Comment`	13.6.2	Representation of comments in the source document.
`Text`	13.6.2	Nodes containing textual content from the document.
`ProcessingInstruction`	13.6.2	Processing instruction representation.

An additional section describes the exceptions defined for working with the DOM in Python.

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*)

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.

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.

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.

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.

Document Objects

A `Document` represents an entire XML document, including its constituent elements, attributes, processing instructions, comments etc. Remeber 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.

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.

getAttribute(*attname*)
Return an attribute value as a string.

getAttributeNode(*attrname*)
Return the `Attr` node for the attribute named by *attrname*.

getAttributeNS(*namespaceURI, localName*)
Return an attribute value as a string, given a *namespaceURI* and *localName*.

getAttributeNodeNS(*namespaceURI, localName*)
Return an attribute value as a node, given a *namespaceURI* and *localName*.

removeAttribute(*attname*)
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(*attname, value*)
Set an attribute value from a string.

setAttributeNode(*newAttr*)
Add a new attibute 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 attibute 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.

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.

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` attribbute.

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.

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.

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.

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.

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 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:

Constant	Exception
`DOMSTRING_SIZE_ERR`	`DomstringSizeErr`
`HIERARCHY_REQUEST_ERR`	`HierarchyRequestErr`
`INDEX_SIZE_ERR`	`IndexSizeErr`
`INUSE_ATTRIBUTE_ERR`	`InuseAttributeErr`
`INVALID_ACCESS_ERR`	`InvalidAccessErr`
`INVALID_CHARACTER_ERR`	`InvalidCharacterErr`
`INVALID_MODIFICATION_ERR`	`InvalidModificationErr`
`INVALID_STATE_ERR`	`InvalidStateErr`
`NAMESPACE_ERR`	`NamespaceErr`
`NOT_FOUND_ERR`	`NotFoundErr`
`NOT_SUPPORTED_ERR`	`NotSupportedErr`
`NO_DATA_ALLOWED_ERR`	`NoDataAllowedErr`
`NO_MODIFICATION_ALLOWED_ERR`	`NoModificationAllowedErr`
`SYNTAX_ERR`	`SyntaxErr`
`WRONG_DOCUMENT_ERR`	`WrongDocumentErr`

13.6.3 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.

Type Mapping

The primitive IDL types used in the DOM specification are mapped to Python types according to the following table.

IDL Type	Python Type
`boolean`	`IntegerType` (with a value of `0` or `1`)
`int`	`IntegerType`
`long int`	`IntegerType`
`unsigned int`	`IntegerType`

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.

Accessor Methods

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

```
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.

Additionally, the 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.

13.7 `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:

```
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:

```
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:

```
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:

```
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 descendents are essentially useless.

See Also:

Document Object Model (DOM) Level 1 Specification
(http://www.w3.org/TR/REC-DOM-Level-1/)
The W3C recommendation for the DOM supported by `xml.dom.minidom`.

13.7.1 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*`)`
Write XML to the writer object. The writer should have a `write()` method which matches that of the file object interface.

New in version 2.1: To support pretty output, new keyword parameters indent, addindent, and newl have

been added.

New 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.

New in version 2.3: the *encoding* argument.

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”.

`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.

New 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.

13.7.2 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.

```
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)
```

13.7.3 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.

13.8 `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.

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 *busize* 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.

13.8.1 DOMEventStream Objects

`getEvent``()`
...

`expandNode``(`*node*`)`
...

`reset``()`
...

13.9 `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
(http://www.saxproject.org/)
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` (section 13.10):
Definitions of the interfaces for application-provided objects.

Module `xml.sax.saxutils` (section 13.11):
Convenience functions for use in SAX applications.

Module `xml.sax.xmlreader` (section 13.12):
Definitions of the interfaces for parser-provided objects.

13.9.1 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`.

13.10 `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`, 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.

13.10.1 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`()
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 `feature_external_ges` and the `feature_external_pes` properties.

13.10.2 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.

13.10.3 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*.

13.10.4 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.

13.11 `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 '`&`', '`<`', and '`>`' 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 doule-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.

13.12 `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.

class XMLReader()
Base class which can be inherited by SAX parsers.

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.

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 13.12.5). 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 13.12.6).

13.12.1 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 `ErrorHandler`.

`setErrorHandler`(*handler*)
Set the current error handler. If no `ErrorHandler` 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.

13.12.2 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.

13.12.3 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.

13.12.4 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.

13.12.5 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*.

13.12.6 The AttributesNS Interface

This interface is a subtype of the Attributes interface (see section 13.12.5). 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.

13.13 xmllib — A parser for XML documents

Deprecated since release 2.0. Use xml.sax instead. The newer XML package includes full support for XML 1.0.

Changed in version 1.5.2: Added namespace support.

This module defines a class XMLParser which serves as the basis for parsing text files formatted in XML (Extensible Markup Language).

class XMLParser()
The XMLParser class must be instantiated without arguments.[1]

This class provides the following interface methods and instance variables:

attributes
A mapping of element names to mappings. The latter mapping maps attribute names that are valid for the element to the default value of the attribute, or if there is no default to None. The default value is the empty dictionary. This variable is meant to be overridden, not extended since the default is shared by all instances of XMLParser.

elements
A mapping of element names to tuples. The tuples contain a function for handling the start and end tag respectively of the element, or None if the method unknown_starttag() or unknown_endtag() is to be called. The default value is the empty dictionary. This variable is meant to be overridden, not extended since the default is shared by all instances of XMLParser.

entitydefs
A mapping of entitynames to their values. The default value contains definitions for 'lt', 'gt', 'amp', 'quot', and 'apos'.

reset()
Reset the instance. Loses all unprocessed data. This is called implicitly at the instantiation time.

setnomoretags()
Stop processing tags. Treat all following input as literal input (CDATA).

setliteral()
Enter literal mode (CDATA mode). This mode is automatically exited when the close tag matching the last unclosed open tag is encountered.

feed(*data*)
Feed some text to the parser. It is processed insofar as it consists of complete tags; 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

[1] Actually, a number of keyword arguments are recognized which influence the parser to accept certain non-standard constructs. The following keyword arguments are currently recognized. The defaults for all of these is 0 (false) except for the last one for which the default is 1 (true). *accept_unquoted_attributes* (accept certain attribute values without requiring quotes), *accept_missing_endtag_name* (accept end tags that look like </>), *map_case* (map upper case to lower case in tags and attributes), *accept_utf8* (allow UTF-8 characters in input; this is required according to the XML standard, but Python does not as yet deal properly with these characters, so this is not the default), *translate_attribute_references* (don't attempt to translate character and entity references in attribute values).

redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call `close()`.

`translate_references`(*data*)
Translate all entity and character references in *data* and return the translated string.

`getnamespace`()
Return a mapping of namespace abbreviations to namespace URIs that are currently in effect.

`handle_xml`(*encoding, standalone*)
This method is called when the '`<?xml ...?>`' tag is processed. The arguments are the values of the encoding and standalone attributes in the tag. Both encoding and standalone are optional. The values passed to `handle_xml()` default to `None` and the string `'no'` respectively.

`handle_doctype`(*tag, pubid, syslit, data*)
This method is called when the '`<!DOCTYPE...>`' declaration is processed. The arguments are the tag name of the root element, the Formal Public Identifier (or `None` if not specified), the system identifier, and the uninterpreted contents of the internal DTD subset as a string (or `None` if not present).

`handle_starttag`(*tag, method, attributes*)
This method is called to handle start tags for which a start tag handler is defined in the instance variable `elements`. The *tag* argument is the name of the tag, and the *method* argument is the function (method) which should be used to support semantic interpretation of the start tag. The *attributes* argument is a dictionary of attributes, the key being the *name* and the value being the *value* of the attribute found inside the tag's <> brackets. Character and entity references in the *value* have been interpreted. For instance, for the start tag `<A HREF="http://www.cwi.nl/">`, this method would be called as `handle_starttag('A', self.elements['A'][0], {'HREF': 'http://www.cwi.nl/'})`. The base implementation simply calls *method* with *attributes* as the only argument.

`handle_endtag`(*tag, method*)
This method is called to handle endtags for which an end tag handler is defined in the instance variable `elements`. The *tag* argument is the name of the tag, and the *method* argument is the function (method) which should be used to support semantic interpretation of the end tag. For instance, for the endtag `</A>`, this method would be called as `handle_endtag('A', self.elements['A'][1])`. 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*`;`'. *ref* can either be a decimal number, or a hexadecimal number when preceded by an '`x`'. In the base implementation, *ref* must be a number in the range 0-255. It translates the character to ASCII and calls the method `handle_data()` with the character as argument. If *ref* is invalid or out of range, the method `unknown_charref(`*ref*`)` is called to handle the error. A subclass must override this method to provide support for character references outside of the ASCII range.

`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_cdata`(*data*)
This method is called when a CDATA element is encountered. The *data* argument is a string containing the text between the '`<![CDATA[`' and '`]]>`' delimiters, but not the delimiters themselves. For example, the entity '`<![CDATA[text]]>`' will cause this method to be called with the argument `'text'`. The default method does nothing, and is intended to be overridden.

`handle_proc`(*name, data*)
This method is called when a processing instruction (PI) is encountered. The *name* is the PI target, and the *data* argument is a string containing the text between the PI target and the closing delimiter, but not the delimiter itself. For example, the instruction '`<?XML text?>`' will cause this method to be called with

the arguments 'XML' and 'text'. The default method does nothing. Note that if a document starts with '<?xml ..?>', handle_xml() is called to handle it.

handle_special(*data*)
This method is called when a declaration is encountered. The *data* argument is a string containing the text between the '<!' and '>' delimiters, but not the delimiters themselves. For example, the entity declaration '<!ENTITY text>' will cause this method to be called with the argument 'ENTITY text'. The default method does nothing. Note that '<!DOCTYPE ...>' is handled separately if it is located at the start of the document.

syntax_error(*message*)
This method is called when a syntax error is encountered. The *message* is a description of what was wrong. The default method raises a RuntimeError exception. If this method is overridden, it is permissible for it to return. This method is only called when the error can be recovered from. Unrecoverable errors raise a RuntimeError without first calling syntax_error().

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. 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 calls syntax_error() to signal an error.

See Also:

Extensible Markup Language (XML) 1.0
(http://www.w3.org/TR/REC-xml)
The XML specification, published by the World Wide Web Consortium (W3C), defines the syntax and processor requirements for XML. References to additional material on XML, including translations of the specification, are available at http://www.w3.org/XML/.

Python and XML Processing
(http://www.python.org/topics/xml/)
The Python XML Topic Guide provides a great deal of information on using XML from Python and links to other sources of information on XML.

SIG for XML Processing in Python
(http://www.python.org/sigs/xml-sig/)
The Python XML Special Interest Group is developing substantial support for processing XML from Python.

13.13.1 XML Namespaces

This module has support for XML namespaces as defined in the XML Namespaces proposed recommendation.

Tag and attribute names that are defined in an XML namespace are handled as if the name of the tag or element consisted of the namespace (the URL that defines the namespace) followed by a space and the name of the tag or attribute. For instance, the tag <html xmlns='http://www.w3.org/TR/REC-html40'> is treated as if the tag name was 'http://www.w3.org/TR/REC-html40 html', and the tag <html:a href='http://frob.com'> inside the above mentioned element is treated as if the tag name were 'http://www.w3.org/TR/REC-html40 a' and the attribute name as if it were 'http://www.w3.org/TR/REC-html40 href'.

An older draft of the XML Namespaces proposal is also recognized, but triggers a warning.

See Also:

Namespaces in XML
(http://www.w3.org/TR/REC-xml-names/)
: This World Wide Web Consortium recommendation describes the proper syntax and processing requirements for namespaces in XML.

CHAPTER

FOURTEEN

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.
colorsys	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.

14.1 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 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*.

adpcm32lin(*adpcmfragment, width, state*)
Decode an alternative 3-bit ADPCM code. See lin2adpcm3() for details.

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 `rms(add(`*fragment*`, mul(`*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 `len(`*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 `rms(fragment[i*2:(i+length)*2])` is maximal. The fragments should both contain 2-byte samples.

The routine takes time proportional to `len(`*fragment*`)`.

getsample(*fragment, width, index*)
Return the value of sample *index* from the fragment.

lin2lin(*fragment, width, newwidth*)
Convert samples between 1-, 2- and 4-byte formats.

lin2adpcm(*fragment, width, 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.

state is a tuple containing the state of the coder. The coder returns a tuple `(`*adpcmfrag*`, `*newstate*`)`, and the *newstate* should be passed to the next call of `lin2adpcm()`. In the initial call, `None` can be passed as the state. *adpcmfrag* is the ADPCM coded fragment packed 2 4-bit values per byte.

lin2adpcm3(*fragment, width, state*)
This is an alternative ADPCM coder that uses only 3 bits per sample. It is not compatible with the Intel/DVI ADPCM coder and its output is not packed (due to laziness on the side of the author). Its use is discouraged.

lin2ulaw(*fragment, 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(*fragment, width*)
Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.

max(*fragment, width*)
Return the maximum of the *absolute value* of all samples in a fragment.

maxpp(*fragment, width*)
Return the maximum peak-peak value in the sound fragment.

mul(*fragment, width, factor*)
Return a fragment that has all samples in the original fragment multiplied by the floating-point value *factor*. Overflow is silently ignored.

ratecv(*fragment, width, nchannels, inrate, outrate, state*[, *weightA*[, *weightB*]])
Convert the frame rate of the input fragment.

state is a tuple containing the state of the converter. The converter returns a tuple (*newfragment*, *newstate*), and *newstate* should be passed to the next call of `ratecv()`. The initial call should pass `None` as the state.

The *weightA* and *weightB* arguments are parameters for a simple digital filter and default to `1` and `0` respectively.

`reverse`(*fragment, width*)
Reverse the samples in a fragment and returns the modified fragment.

`rms`(*fragment, 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.

`tomono`(*fragment, width, lfactor, rfactor*)
Convert a stereo fragment to a mono fragment. The left channel is multiplied by *lfactor* and the right channel by *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:

```
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) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

14.2 `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 `mono2grey()`.

`dither2mono`(*image, width, height*)
Convert an 8-bit greyscale image to a 1-bit monochrome image using a (simple-minded) dithering algorithm.

`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 *p0* on output and all one-value input pixels get value *p1* on output. To convert a monochrome black-and-white image to greyscale pass the values `0` and `255` respectively.

`grey2grey4`(*image, width, height*)
Convert an 8-bit greyscale image to a 4-bit greyscale image without dithering.

`grey2grey2`(*image, width, height*)
Convert an 8-bit greyscale image to a 2-bit greyscale image without dithering.

`dither2grey2`(*image, width, height*)

Convert an 8-bit greyscale image to a 2-bit greyscale image with dithering. As for `dither2mono()`, the dithering algorithm is currently very simple.

`grey42grey`(*image, width, height*)
Convert a 4-bit greyscale image to an 8-bit greyscale image.

`grey22grey`(*image, width, height*)
Convert a 2-bit greyscale image to an 8-bit greyscale image.

14.3 `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*()` methods can only be called after the corresponding `set*()` methods 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.

14.4 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:

Field	Contents
magic word	The four bytes '.snd'.
header size	Size of the header, including info, in bytes.
data size	Physical size of the data, in bytes.
encoding	Indicates how the audio samples are encoded.
sample rate	The sampling rate.
# of channels	The number of channels in the samples.
info	ASCII string giving a description of the audio file (padded with null bytes).

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.

14.4.1 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.

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.

setpos(*pos*)
Set the file pointer to the specified position. Only values returned from `tell()` should be used for *pos*.

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.

getmarkers()
Returns `None`.

getmark(*id*)
Raise an error.

14.4.2 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 *nframes* is correct, and close the file.

This method is called upon deletion.

Note that it is invalid to set any parameters after calling `writeframes()` or `writeframesraw()`.

14.5 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:

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).

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.

14.5.1 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.

14.5.2 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`.

14.6 chunk — Read IFF chunked data

This module provides an interface for reading files that use EA IFF 85 chunks.[1] 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:

Offset	Length	Contents
0	4	Chunk ID
4	4	Size of chunk in big-endian byte order, not including the header
8	*n*	Data bytes, where *n* is the size given in the preceding field
8 + *n*	0 or 1	Pad byte needed if *n* is odd and chunk alignment is used

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.

[1] "EA IFF 85" Standard for Interchange Format Files, Jerry Morrison, Electronic Arts, January 1985.

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.

14.7 `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.inforamp.net/%7epoynton/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)
```

14.8 rgbimg — Read and write “SGI RGB” files

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.

14.9 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:

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():

Value	Image format
'rgb'	SGI ImgLib Files
'gif'	GIF 87a and 89a Files
'pbm'	Portable Bitmap Files
'pgm'	Portable Graymap Files
'ppm'	Portable Pixmap Files
'tiff'	TIFF Files
'rast'	Sun Raster Files
'xbm'	X Bitmap Files
'jpeg'	JPEG data in JFIF format
'bmp'	BMP files
'png'	Portable Network Graphics

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:

```
>>> import imghdr
>>> imghdr.what('/tmp/bass.gif')
'gif'
```

14.10 `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`.

14.11 `ossaudiodev` — Access to OSS-compatible audio devices

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
(http://www.opensound.com/pguide/oss.pdf)
: 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'.

14.11.1 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:

`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. On a typical Linux system, these formats are:

Format	Description
AFMT_MU_LAW	a logarithmic encoding (used by Sun .au files and /dev/audio)
AFMT_A_LAW	a logarithmic encoding
AFMT_IMA_ADPCM	a 4:1 compressed format defined by the Interactive Multimedia Association
AFMT_U8	Unsigned, 8-bit audio
AFMT_S16_LE	Unsigned, 16-bit audio, little-endian byte order (as used by Intel processors)
AFMT_S16_BE	Unsigned, 16-bit audio, big-endian byte order (as used by 68k, PowerPC, Sparc)
AFMT_S8	Signed, 8 bit audio
AFMT_U16_LE	Signed, 16-bit little-endian audio
AFMT_U16_BE	Signed, 16-bit big-endian audio

Most systems support only a subset of these formats. Many 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:

Rate	Description
8000	default rate for /dev/audio
11025	speech recording
22050	
44100	CD quality audio (at 16 bits/sample and 2 channels)
96000	DVD quality audio (at 24 bits/sample)

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=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 valus of setfmt(), channels(), and speed()).

For example,

```
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

is equivalent to

```
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.

14.11.2 Mixer Device Objects

File-like interface

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.

Mixer interface

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 channels—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:

```
mixer=ossaudiodev.openmixer()
if mixer.channels() & (1 << ossaudiodev.SOUND_MIXER_PCM):
# PCM is supported
<code>
```

For most purposes, the SOUND_MIXER_VOLUME (Master volume) and SOUND_MIXER_PCM channels should suffice—but code that uses the mixer should be flexible when it comes to choosing sound channels. On the Gravis Ultrasound, for example, SOUND_MIXER_VOLUME does not exist.

stereocontrols()
Returns a bitmask indicating stereo mixer channels. If a bit is set, the corresponding channel is stereo; if it is unset, the channel is either monophonic or not supported by the mixer (use in combination with channels() to determine which).

See the code example for the channels() 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(*channel*)
Returns the volume of a given mixer channel. 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 channel is monophonic, a 2-tuple is still returned, but both channel volumes are the same.

If an unknown channel is specified, `error` is raised.

`set`(*channel, (left, right)*)
Sets the volume for a given mixer channel 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 `IOError` if an invalid mixer channel was specified; `TypeError` if the argument format was incorrect, and `error` if the specified volumes were out-of-range.

`get_recsrc`()
This method returns a bitmask indicating which channel or channels 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:

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```

CHAPTER

FIFTEEN

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:

hmac	Keyed-Hashing for Message Authentication (HMAC) implementation for Python.
md5	RSA's MD5 message digest algorithm.
sha	NIST's secure hash algorithm, SHA.
mpz	Interface to the GNU MP library for arbitrary precision arithmetic.
rotor	Enigma-like encryption and decryption.

Hardcore cypherpunks will probably find the cryptographic modules written by A.M. Kuchling of further interest; the package adds built-in modules for DES and IDEA encryption, provides a Python module for reading and decrypting PGP files, and then some. These modules are not distributed with Python but available separately. See the URL http://www.amk.ca/python/code/crypto.html for more information.

15.1 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 module for the HMAC object to use. It defaults to the `md5` module.

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 is a 16-byte string (for `md5`) or a 20-byte string (for `sha`) which may contain non-ASCII characters, including NUL bytes.

hexdigest()
Like `digest()` except the digest is returned as a string of length 32 for `md5` (40 for `sha`), 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.

15.2 md5 — MD5 message digest algorithm

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'`:

```
>>> import md5
>>> m = md5.new()
>>> 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:

```
>>> md5.new("Nobody inspects the spammish repetition").digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\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`.

md5 objects support the following methods:

`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` (section 15.3):
Similar module implementing the Secure Hash Algorithm (SHA). The SHA algorithm is considered a more secure hash.

15.3 sha — SHA-1 message digest algorithm

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
(http://csrc.nist.gov/publications/fips/fips180-1/fip180-1.txt)
The Secure Hash Algorithm is defined by NIST document FIPS PUB 180-1: *Secure Hash Standard*, published in April of 1995. It is available online as plain text (at least one diagram was omitted) and as PDF at http://csrc.nist.gov/publications/fips/fips180-1/fip180-1.pdf.

Cryptographic Toolkit (Secure Hashing)
(http://csrc.nist.gov/encryption/tkhash.html)
Links from NIST to various information on secure hashing.

15.4 mpz — GNU arbitrary magnitude integers

Deprecated since release 2.2. See the references at the end of this section for information about packages which provide similar functionality. This module will be removed in Python 2.3.

This is an optional module. It is only available when Python is configured to include it, which requires that the GNU MP software is installed.

This module implements the interface to part of the GNU MP library, which defines arbitrary precision integer and rational number arithmetic routines. Only the interfaces to the *integer* (`mpz_*()`) routines are provided. If not stated otherwise, the description in the GNU MP documentation can be applied.

Support for rational numbers can be implemented in Python. For an example, see the `Rat` module, provided as 'Demos/classes/Rat.py' in the Python source distribution.

In general, *mpz*-numbers can be used just like other standard Python numbers, e.g., you can use the built-in operators like `+`, `*`, etc., as well as the standard built-in functions like `abs()`, `int()`, ..., `divmod()`, `pow()`. **Please note:** the *bitwise-xor* operation has been implemented as a bunch of *and*s, *invert*s and *or*s, because the

library lacks an `mpz_xor()` function, and I didn't need one.

You create an mpz-number by calling the function `mpz()` (see below for an exact description). An mpz-number is printed like this: `mpz(`*value*`)`.

`mpz`(*value*)
Create a new mpz-number. *value* can be an integer, a long, another mpz-number, or even a string. If it is a string, it is interpreted as an array of radix-256 digits, least significant digit first, resulting in a positive number. See also the `binary()` method, described below.

`MPZType`
The type of the objects returned by `mpz()` and most other functions in this module.

A number of *extra* functions are defined in this module. Non mpz-arguments are converted to mpz-values first, and the functions return mpz-numbers.

`powm`(*base, exponent, modulus*)
Return `pow(`*base*`, `*exponent*`) % `*modulus*. If *exponent* `== 0`, return `mpz(1)`. In contrast to the C library function, this version can handle negative exponents.

`gcd`(*op1, op2*)
Return the greatest common divisor of *op1* and *op2*.

`gcdext`(*a, b*)
Return a tuple `(`*g*`, `*s*`, `*t*`)`, such that *a*`*`*s* `+` *b*`*`*t* `==` *g* `==` `gcd(`*a*`, `*b*`)`.

`sqrt`(*op*)
Return the square root of *op*. The result is rounded towards zero.

`sqrtrem`(*op*)
Return a tuple `(`*root*`, `*remainder*`)`, such that *root*`*`*root* `+` *remainder* `==` *op*.

`divm`(*numerator, denominator, modulus*)
Returns a number *q* such that *q* `*` *denominator* `%` *modulus* `==` *numerator*. One could also implement this function in Python, using `gcdext()`.

An mpz-number has one method:

`binary`()
Convert this mpz-number to a binary string, where the number has been stored as an array of radix-256 digits, least significant digit first.

The mpz-number must have a value greater than or equal to zero, otherwise `ValueError` will be raised.

See Also:

General Multiprecision Python
(http://gmpy.sourceforge.net/)
This project is building new numeric types to allow arbitrary-precision arithmetic in Python. Their first efforts are also based on the GNU MP library.

mxNumber — Extended Numeric Types for Python
(http://www.egenix.com/files/python/mxNumber.html)
Another wrapper around the GNU MP library, including a port of that library to Windows.

15.5 `rotor` — Enigma-like encryption and decryption

Deprecated since release 2.3. The encryption algorithm is insecure.

This module implements a rotor-based encryption algorithm, contributed by Lance Ellinghouse. The design is derived from the Enigma device, a machine used during World War II to encipher messages. A rotor is simply a permutation. For example, if the character 'A' is the origin of the rotor, then a given rotor might map 'A' to 'L', 'B' to 'Z', 'C' to 'G', and so on. To encrypt, we choose several different rotors, and set the origins of the rotors to known positions; their initial position is the ciphering key. To encipher a character, we permute the original character by the first rotor, and then apply the second rotor's permutation to the result. We continue until we've applied all the rotors; the resulting character is our ciphertext. We then change the origin of the final rotor by one

position, from ‘A’ to ‘B’; if the final rotor has made a complete revolution, then we rotate the next-to-last rotor by one position, and apply the same procedure recursively. In other words, after enciphering one character, we advance the rotors in the same fashion as a car’s odometer. Decoding works in the same way, except we reverse the permutations and apply them in the opposite order.

The available functions in this module are:

newrotor(*key*[, *numrotors*])
: Return a rotor object. *key* is a string containing the encryption key for the object; it can contain arbitrary binary data but not null bytes. The key will be used to randomly generate the rotor permutations and their initial positions. *numrotors* is the number of rotor permutations in the returned object; if it is omitted, a default value of 6 will be used.

Rotor objects have the following methods:

setkey(*key*)
: Sets the rotor’s key to *key*. The key should not contain null bytes.

encrypt(*plaintext*)
: Reset the rotor object to its initial state and encrypt *plaintext*, returning a string containing the ciphertext. The ciphertext is always the same length as the original plaintext.

encryptmore(*plaintext*)
: Encrypt *plaintext* without resetting the rotor object, and return a string containing the ciphertext.

decrypt(*ciphertext*)
: Reset the rotor object to its initial state and decrypt *ciphertext*, returning a string containing the plaintext. The plaintext string will always be the same length as the ciphertext.

decryptmore(*ciphertext*)
: Decrypt *ciphertext* without resetting the rotor object, and return a string containing the plaintext.

An example usage:

```
>>> import rotor
>>> rt = rotor.newrotor('key', 12)
>>> rt.encrypt('bar')
'\xab4\xf3'
>>> rt.encryptmore('bar')
'\xef\xfd$'
>>> rt.encrypt('bar')
'\xab4\xf3'
>>> rt.decrypt('\xab4\xf3')
'bar'
>>> rt.decryptmore('\xef\xfd$')
'bar'
>>> rt.decrypt('\xef\xfd$')
'l(\xcd'
>>> del rt
```

The module’s code is not an exact simulation of the original Enigma device; it implements the rotor encryption scheme differently from the original. The most important difference is that in the original Enigma, there were only 5 or 6 different rotors in existence, and they were applied twice to each character; the cipher key was the order in which they were placed in the machine. The Python `rotor` module uses the supplied key to initialize a random number generator; the rotor permutations and their initial positions are then randomly generated. The original device only enciphered the letters of the alphabet, while this module can handle any 8-bit binary data; it also produces binary output. This module can also operate with an arbitrary number of rotors.

The original Enigma cipher was broken in 1944. The version implemented here is probably a good deal more difficult to crack (especially if you use many rotors), but it won’t be impossible for a truly skillful and determined attacker to break the cipher. So if you want to keep the NSA out of your files, this rotor cipher may well be unsafe, but for discouraging casual snooping through your files, it will probably be just fine, and may be somewhat safer than using the UNIX **crypt** command.

CHAPTER

SIXTEEN

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, but is however the most commonly used one; see section **??**, "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.

16.1 `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
(http://www.python.org/topics/tkinter/)
: 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
(http://www.pythonware.com/library/an-introduction-to-tkinter.htm)
: Fredrik Lundh's on-line reference material.

Tkinter reference: a GUI for Python
(http://www.nmt.edu/tcc/help/pubs/lang.html)
: On-line reference material.

Tkinter for JPython
(http://jtkinter.sourceforge.net)
: The Jython interface to Tkinter.

Python and Tkinter Programming
(http://www.amazon.com/exec/obidos/ASIN/1884777813)
: The book by John Grayson (ISBN 1-884777-81-3).

16.1.1 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'*)
The `Tk` class is instantiated without arguments. This creates a toplevel widget of Tk which usually is the main window of an appliation. Each instance has its own associated Tcl interpreter.

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.

16.1.2 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.

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 `mann` 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
(http://tcl.activestate.com/)
The Tk/Tcl development is largely taking place at ActiveState.

Tcl and the Tk Toolkit
(http://www.amazon.com/exec/obidos/ASIN/020163337X)
The book by John Ousterhout, the inventor of Tcl .

Practical Programming in Tcl and Tk
(http://www.amazon.com/exec/obidos/ASIN/0130220280)
Brent Welch's encyclopedic book.

```
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()

app = Application()
app.mainloop()
```

16.1.3 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 16.1.4 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"
   ^       ^     \_____________________/
   |       |                |
 class    new            options
command  widget  (-opt val -opt val ...)
```

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 (disabling 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.

16.1.4 Mapping Basic Tk into Tkinter

Class commands in Tk correspond to class constructors in Tkinter.

```
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.

```
button .panel.fred          =====>  fred = Button(panel)
```

The configuration options in Tk are given in lists of hyphened 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 16.1.6 on setting options.

```
button .fred -fg red        =====>  fred = Button(panel, fg = "red")
.fred configure -fg red     =====>  fred["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.

```
.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.

```
pack .fred -side left       =====>  fred.pack(side = "left")
```

16.1.5 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.

16.1.6 Handy Reference

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 subesequent 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 ideosyncratic 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')`).

Index	Meaning	Example
0	option name	`'relief'`
1	option name for database lookup	`'relief'`
2	option class for database lookup	`'Relief'`
3	default value	`'raised'`
4	current value	`'groove'`

Example:

```
>>> 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.

The Packer

The packer is one of Tk’s geometry-management mechanisms. See also *the Packer class interface*.

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 accomodate 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:

```
fred.pack()                     # defaults to side = "top"
fred.pack(side = "left")
fred.pack(expand = 1)
```

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'`.

Coupling Widget Variables

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:

```
class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        self.button.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()
```

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:

```
import Tkinter
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()
```

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:

```
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" `"#RRRGGGBBB"`, or 16 bit `"#RRRRGGGGBBBB"` 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 centimeters, `i` for inches, `m` for millimeters, `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.

geometry This is a string of the form '*width*`x`*height*', 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"`.

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 accesed in the `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.

Tk	Tkinter Event Field	Tk	Tkinter Event Field
%f	focus	%A	char
%h	height	%E	send_event
%k	keycode	%K	keysym
%s	state	%N	keysym_num
%t	time	%T	type
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

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.

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.

16.2 `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
(http://tix.sourceforge.net/)
The home page for `Tix`. This includes links to additional documentation and downloads.

Tix Man Pages
(http://tix.sourceforge.net/dist/current/man/)
On-line version of the man pages and reference material.

Tix Programming Guide
(http://tix.sourceforge.net/dist/current/docs/tix-book/tix.book.html)
On-line version of the programmer's reference material.

Tix Development Applications
(http://tix.sourceforge.net/Tide/)
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.

16.2.1 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]" Tix]
```

16.2.2 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.

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 subwdget 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 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.

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 an 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.

Hierachical 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 hierachy.

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 hierachical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.

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.

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.

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 simutaneously in a Tk `Button` widget.

Miscellaneous Widgets

class `InputOnly`()
The InputOnly widgets are to accept inputs from the user, which can be done with the `bind` command (UNIX only).

Form Geometry Manager

In addition, `Tix` augments `Tkinter` by providing:

class `Form`()
The Form geometry manager based on attachment rules for all Tk widgets.

16.2.3 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 applications 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 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 specified 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 manitained 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 has imported and inited, 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.

16.3 `ScrolledText` — Scrolled Text Widget

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.

16.4 `turtle` — Turtle graphics for 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.

`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.

forward(*distance*)
Go forward *distance* steps.

backward(*distance*)
Go backward *distance* steps.

left(*angle*)
Turn left *angle* units. Units are by default degrees, but can be set via the `degrees()` and `radians()` functions.

right(*angle*)
Turn right *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 up — draw when moving.

width(*width*)
Set the line width to *width*.

color(*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(*text*[, *move*])
Write *text* at the current pen position. If *move* is true, the pen is moved to the bottom-right corner of the text. By default, *move* is false.

fill(*flag*)
The complete specifications are rather complex, but the recommended usage is: call `fill(1)` before drawing a path you want to fill, and call `fill(0)` when you finish to draw the path.

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.

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 `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.

16.4.1 Pen and RawPen Objects

`Pen` and `RawPen` objects have all the global functions described above, except for `demo()` as methods, which manipulate the given pen.

The only method which is more powerful as a method is `degrees()`.

`degrees`([*fullcircle*])
fullcircle is by default 360. This can cause the pen to have any angular units whatever: give *fullcircle* 2*π for radians, or 400 for gradians.

16.5 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)

16.5.1 Menus

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)

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 Turns *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

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).

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

16.5.2 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`

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.

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

16.5.3 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

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`.

16.6 Other Graphical User Interface Packages

There are an number of extension widget sets to `Tkinter`.

Python megawidgets
(http://pmw.sourceforge.net/)
: 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)
(http://tkinter.effbot.org/)
: 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.

Tk is not the only GUI for Python, but is however the most commonly used one.

wxWindows
(http://www.wxwindows.org)
: is a GUI toolkit that combines the most attractive attributes of Qt, Tk, Motif, and GTK+ in one powerful and efficient package. It is implemented in C++. wxWindows supports two flavors of UNIX implementation: GTK+ and Motif, and under Windows, it has a standard Microsoft Foundation Classes (MFC) appearance, because it uses Win32 widgets. There is a Python class wrapper, independent of Tkinter.

  wxWindows is much richer in widgets than `Tkinter`, with its help system, sophisticated HTML and image viewers, and other specialized widgets, extensive documentation, and printing capabilities.

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
(http://www.riverbankcomputing.co.uk/pykde/index.php)

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.

FXPy
(http://fxpy.sourceforge.net/)

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
(http://www.daa.com.au/~james/software/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.

CHAPTER

SEVENTEEN

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 `rexec` 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
(http://grail.sourceforge.net/)
: 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.

17.1 `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
(http://grail.sourceforge.net/)
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.

17.1.1 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(*modulename*[, *globals*[, *locals*[, *fromlist*]]])
Import the module *modulename*, 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(*modulename*[, *globals*[, *locals*[, *fromlist*]]])
Import the module *modulename*, 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*.

17.1.2 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', 'rotor', '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`, `PY_COMPILED`, and `C_EXTENSION`. The value for `RExec` is `(C_EXTENSION, PY_SOURCE)`. Adding `PY_COMPILED` 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.

17.1.3 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:

```
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.

17.2 `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 `rexec` 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.

CHAPTER

EIGHTEEN

Python Language Services

Python provides a number of modules to assist in working with the Python language. These module 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.
tokenize	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.
distutils	Support for building and installing Python modules into an existing Python installation.

18.1 `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 `symbol`. 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` (section 18.2):
Useful constants representing internal nodes of the parse tree.

Module `token` (section 18.3):
Useful constants representing leaf nodes of the parse tree and functions for testing node values.

18.1.1 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.

18.1.2 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.

`ast2list`(*ast*[, *line_info*])
This function accepts an AST object from the caller in *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, `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 *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 *ends*. This information is omitted if the flag is false or omitted.

`ast2tuple`(*ast*[, *line_info*])
This function accepts an AST object from the caller in *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 `ast2list()`.

If *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.

`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 `exec` statement or a call to the built-in `eval()` function. This function provides the interface to the compiler, passing the internal parse tree from *ast* to the parser, using the source file name specified by the *filename* parameter. The default value supplied for *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 `SyntaxError` caused by the parse tree for `del f(0)`: this statement is considered legal within the formal grammar for Python but is not a legal language construct. The `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 `parser` module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.

18.1.3 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.

18.1.4 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.

18.1.5 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*`)`.

18.1.6 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.

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

```
>>> 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:

```
>>> 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:

```
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()
```

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'.)

```
"""Some documentation.
"""
```

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.

```
>>> 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.\n"""')))))))))))))))),
   (4, ''))),
 (4, ''),
 (0, ''))
```

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'.)

```
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’.)

```
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 `match()` function with this pattern, extracting the module docstring from the parse tree created previously is easy:

```
>>> 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 `stmt` node in a code block (`file_input` or `suite` node types). A module consists of a single `file_input` node, and class and function definitions each contain exactly one `suite` node. Classes and functions are readily identified as subtrees of code block nodes which start with `(stmt, (compound_stmt, (classdef, ...` or `(stmt, (compound_stmt, (funcdef, ...`. Note that these subtrees cannot be matched by `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':

```
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

```
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)
    return ModuleInfo(ast.totuple(), basename)
```

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.

18.2 `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` (section 18.1):
The second example for the `parser` module shows how to use the `symbol` module.

18.3 `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` (section 18.1):
The second example for the `parser` module shows how to use the `symbol` module.

18.4 `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.

18.5 `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 argment, *readline*, which must be a callable object which provides the same interface as the `readline()` method of built-in file objects (see section 2.2.8). 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 2.2.8). Each call to the function should return one line of input as a string.

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.

18.6 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.

`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 `tokeneater()` if detecting an ambiguous indent. Captured and handled in `check()`.

`tokeneater`(*type, token, start, end, line*)
This function is used by `check()` as a callback parameter to the function `tokenize.tokenize()`.

See Also:

Module `tokenize` (section 18.5):
Lexical scanner for Python source code.

18.7 pyclbr — Python class browser support

The `pyclbr` can be used to determine some limited information about the classes and methods 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 from an imported 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.

18.7.1 Class Descriptor Objects

The class descriptor objects used as values in the dictionary returned by `readmodule()` 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`.

18.8 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* = True, a PyCompileError is raised when an error is encountered while compiling *file*. If *doraise* = 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` (section 18.9):
Utilities to compile all Python source files in a directory tree.

18.9 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.

See Also:

Module `py_compile` (section 18.8):
Byte-compile a single source file.

18.10 `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`:

```
def myfunc(alist):
    return len(alist)
```

the following command can be used to get the disassembly of `myfunc()`:

```
>>> dis.dis(myfunc)
  2           0 LOAD_GLOBAL              0 (len)
              3 LOAD_FAST                0 (alist)
              6 CALL_FUNCTION            1
              9 RETURN_VALUE
             10 LOAD_CONST               0 (None)
             13 RETURN_VALUE
```

(The "2" is a line number).

The `dis` module defines the following functions and constants:

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.

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.

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.

18.10.1 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.

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 `TOS = TOS1[TOS]`.

BINARY_LSHIFT
Implements `TOS = TOS1 << TOS`.

BINARY_RSHIFT
Implements `TOS = TOS1 >> TOS`.

BINARY_AND
Implements `TOS = TOS1 & TOS`.

BINARY_XOR
Implements `TOS = TOS1 ^ TOS`.

BINARY_OR
Implements `TOS = TOS1 | 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 `TOS = TOS1 ** TOS`.

INPLACE_MULTIPLY
Implements in-place `TOS = TOS1 * TOS`.

INPLACE_DIVIDE
Implements in-place `TOS = TOS1 / TOS` when `from __future__ import division` is not in effect.

INPLACE_FLOOR_DIVIDE
Implements in-place `TOS = TOS1 // TOS`.

INPLACE_TRUE_DIVIDE
Implements in-place `TOS = TOS1 / TOS` when `from __future__ import division` is in effect.

INPLACE_MODULO
Implements in-place `TOS = TOS1 % TOS`.

INPLACE_ADD
Implements in-place `TOS = TOS1 + TOS`.

INPLACE_SUBTRACT
Implements in-place `TOS = TOS1 - TOS`.

INPLACE_LSHIFT
Implements in-place `TOS = TOS1 << TOS`.

INPLACE_RSHIFT
Implements in-place `TOS = TOS1 >> TOS`.

INPLACE_AND
Implements in-place `TOS = TOS1 & TOS`.

INPLACE_XOR
Implements in-place `TOS = TOS1 ^ TOS`.

INPLACE_OR
Implements in-place `TOS = TOS1 | TOS`.

The slice opcodes take up to three parameters.

SLICE+0
Implements `TOS = TOS[:]`.

SLICE+1
Implements `TOS = TOS1[TOS:]`.

SLICE+2
Implements `TOS = TOS1[:TOS]`.

SLICE+3
Implements `TOS = TOS2[TOS1: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).

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.

18.11 `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*. 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
(../dist/dist.html)
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 installaion.

Installing Python Modules
(../inst/inst.html)
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.

CHAPTER

NINETEEN

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 `compiler` package is a Python source to bytecode translator written in Python. It uses the built-in parser and standard `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 `compiler` package work. It blends reference material with a tutorial.

The following modules are part of the `compiler` package:

19.1 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=None, dont_inherit=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()`. This function is a replacement for the built-in `compile()` function.

  The *filename* will be used for run-time error messages.

  The *mode* must be 'exec' to compile a module, 'single' to compile a single (interactive) statement, or 'eval' to compile an expression.

  The *flags* and *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: `ast`, `consts`, `future`, `misc`, `pyassem`, `pycodegen`, `symbols`, `transformer`, and `visitor`.

19.2 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 get the first phases of error checking for free. It implements the second phase itself, and that implement 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.

19.3 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.

19.3.1 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.

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:

`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.

`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.

```
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()`.

Node type	Attribute	Value
`Add`	`left`	left operand
	`right`	right operand
`And`	`nodes`	list of operands
`AssAttr`		*attribute as target of assignment*
	`expr`	expression on the left-hand side of the dot
	`attrname`	the attribute name, a string
	`flags`	XXX
`AssList`	`nodes`	list of list elements being assigned to
`AssName`	`name`	name being assigned to
	`flags`	XXX
`AssTuple`	`nodes`	list of tuple elements being assigned to
`Assert`	`test`	the expression to be tested
	`fail`	the value of the `AssertionError`
`Assign`	`nodes`	a list of assignment targets, one per equal sign
	`expr`	the value being assigned
`AugAssign`	`node`	
	`op`	
	`expr`	
`Backquote`	`expr`	
`Bitand`	`nodes`	
`Bitor`	`nodes`	
`Bitxor`	`nodes`	
`Break`		
`CallFunc`	`node`	expression for the callee
	`args`	a list of arguments
	`star_args`	the extended *-arg value
	`dstar_args`	the extended **-arg value
`Class`	`name`	the name of the class, a string

Node type	Attribute	Value
	bases	a list of base classes
	doc	doc string, a string or None
	code	the body of the class statement
Compare	expr	
	ops	
Const	value	
Continue		
Dict	items	
Discard	expr	
Div	left	
	right	
Ellipsis		
Exec	expr	
	locals	
	globals	
For	assign	
	list	
	body	
	else_	
From	modname	
	names	
Function	name	name used in def, a string
	argnames	list of argument names, as strings
	defaults	list of default values
	flags	xxx
	doc	doc string, a string or None
	code	the body of the function
Getattr	expr	
	attrname	
Global	names	
If	tests	
	else_	
Import	names	
Invert	expr	
Keyword	name	
	expr	
Lambda	argnames	
	defaults	
	flags	
	code	
LeftShift	left	
	right	
List	nodes	
ListComp	expr	
	quals	
ListCompFor	assign	
	list	
	ifs	
ListCompIf	test	
Mod	left	
	right	
Module	doc	doc string, a string or None
	node	body of the module, a Stmt
Mul	left	
	right	
Name	name	

Node type	Attribute	Value
Not	expr	
Or	nodes	
Pass		
Power	left	
	right	
Print	nodes	
	dest	
Printnl	nodes	
	dest	
Raise	expr1	
	expr2	
	expr3	
Return	value	
RightShift	left	
	right	
Slice	expr	
	flags	
	lower	
	upper	
Sliceobj	nodes	list of statements
Stmt	nodes	
Sub	left	
	right	
Subscript	expr	
	flags	
	subs	
TryExcept	body	
	handlers	
	else_	
TryFinally	body	
	final	
Tuple	nodes	
UnaryAdd	expr	
UnarySub	expr	
While	test	
	body	
	else_	
Yield	value	

19.3.2 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. `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.

19.3.3 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.

```
>>> import compiler
>>> mod = compiler.parseFile("/tmp/doublelib.py")
>>> mod
Module('This is an example module.\n\nThis is the docstring.\n',
       Stmt([Function('double', ['x'], [], 0, 'Return twice the argument',
       Stmt([Return(Mul((Name('x'), Const(2))))]))]))
>>> from compiler.ast import *
>>> Module('This is an example module.\n\nThis is the docstring.\n',
...    Stmt([Function('double', ['x'], [], 0, 'Return twice the argument',
...    Stmt([Return(Mul((Name('x'), Const(2))))]))]))
Module('This is an example module.\n\nThis is the docstring.\n',
       Stmt([Function('double', ['x'], [], 0, 'Return twice the argument',
       Stmt([Return(Mul((Name('x'), Const(2))))]))]))
>>> mod.doc
'This is an example module.\n\nThis is the docstring.\n'
>>> for node in mod.node.nodes:
...     print node
...
Function('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))))])
```

19.4 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 elminiate 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.

`walk`(*tree, visitor*[, *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

default(*node*[, ...])

dispatch(*node*[, ...])

preorder(*tree*, *visitor*)

19.5 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.

CHAPTER

TWENTY

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.
fl	FORMS library for applications with graphical user interfaces.
FL	Constants used with the fl 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.

20.1 al — Audio functions on the SGI

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 ALqueryparams().

getparams(*device, list*)
The *device* argument is an integer. The list argument is a list such as returned by `queryparams()`; it is modified in place (!).

setparams(*device, list*)
The *device* argument is an integer. The *list* argument is a list such as returned by `queryparams()`.

20.1.1 Configuration Objects

Configuration objects returned by `newconfig()` have the following methods:

getqueuesize()
Return the queue size.

setqueuesize(*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.

20.1.2 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.

20.2 AL — Constants used with the al module

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:

```
import al
from AL import *
```

20.3 cd — CD-ROM access on SGI systems

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*) triple 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 omited, 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).

20.3.1 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 `readda()` 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*, *ttime*, *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; *ttime* 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 `msftoframe()` 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.

20.3.2 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:

Type	Value
`audio`	String which can be passed unmodified to `al.writesamps()`.
`pnum`	Integer giving the program (track) number.
`index`	Integer giving the index number.
`ptime`	Tuple consisting of the program time in minutes, seconds, and frames.
`atime`	Tuple consisting of the absolute time in minutes, seconds, and frames.
`catalog`	String of 13 characters, giving the catalog number of the CD.
`ident`	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.
`control`	Integer giving the control bits from the CD subcode data

`deleteparser`()
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.

20.4 `fl` — FORMS library for graphical user interfaces

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 '`fl_`' from their name. Constants used by the library are defined in module FL 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 `fl_addto_form()` and `fl_end_form()`, and the equivalent of `fl_bgn_form()` is called `fl.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 `fl` implies a call to the GL function `foreground()` and to the FORMS routine `fl_init()`.

20.4.1 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`()
`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 han-

dle 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()`.

20.4.2 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.

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.

end_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:

Name	C Type	Meaning
`window`	int (read-only)	GL window id
`w`	float	form width
`h`	float	form height
`x`	float	form x origin
`y`	float	form y origin
`deactivated`	int	nonzero if form is deactivated
`visible`	int	nonzero if form is visible
`frozen`	int	nonzero if form is frozen
`doublebuf`	int	nonzero if double buffering on

20.4.3 FORMS Objects

Besides methods specific to particular kinds of FORMS objects, all FORMS objects also have the following methods:

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.

delete_object()
Delete the object.

show_object()
Show the object.

hide_object()
Hide the object.

redraw_object()
Redraw the object.

freeze_object()
Freeze the object.

unfreeze_object()
Unfreeze the object.

FORMS objects have these data attributes; see the FORMS documentation:

Name	C Type	Meaning
objclass	int (read-only)	object class
type	int (read-only)	object type
boxtype	int	box type
x	float	x origin
y	float	y origin
w	float	width
h	float	height
col1	int	primary color
col2	int	secondary color
align	int	alignment
lcol	int	label color
lsize	float	label font size
label	string	label string
lstyle	int	label style
pushed	int (read-only)	(see FORMS docs)
focus	int (read-only)	(see FORMS docs)
belowmouse	int (read-only)	(see FORMS docs)
frozen	int (read-only)	(see FORMS docs)
active	int (read-only)	(see FORMS docs)
input	int (read-only)	(see FORMS docs)
visible	int (read-only)	(see FORMS docs)
radio	int (read-only)	(see FORMS docs)
automatic	int (read-only)	(see FORMS docs)

20.5 FL — Constants used with the fl module

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:

```
import fl
from FL import *
```

20.6 flp — Functions for loading stored FORMS designs

This module defines functions that can read form definitions created by the 'form designer' (**fdesign**) program that comes with the FORMS library (see module fl above).

For now, see the file 'flp.doc' in the Python library source directory for a description.

XXX A complete description should be inserted here!

20.7 fm — *Font Manager* interface

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*`)`.

20.8 `gl` — *Graphics Library* interface

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

  ```
  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.

nvarray()
: 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 `nvarray()` 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: `[len(`*s_k*`) - `*s_ord*`]`, `[len(`*t_k*`) - `*t_ord*`]`.

nurbscurve(*knots, ctlpoints, order, type*)
: Defines a nurbs curve. The length of ctlpoints is `len(`*knots*`) - `*order*.

pwlcurve(*points, type*)
: Defines a piecewise-linear curve. *points* is a list of points. *type* must be `N_ST`.

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:

```
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
(http://pyopengl.sourceforge.net/)
An interface to OpenGL is also available; see information about the **PyOpenGL** project online at http://pyopengl.sourceforge.net/. This may be a better option if support for SGI hardware from before about 1996 is not required.

20.9 `DEVICE` — Constants used with the `gl` module

This modules 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.

20.10 `GL` — Constants used with the `gl` module

This module contains constants used by the Silicon Graphics *Graphics Library* from the C header file `<gl/gl.h>`. Read the module source file for details.

20.11 `imgfile` — Support for SGI imglib files

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 smoothen 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()`.

20.12 `jpeg` — Read and write JPEG files

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:

Option	Effect
`'forcegray'`	Force output to be grayscale, even if input is RGB.
`'quality'`	Set the quality of the compressed image to a value between `0` and `100` (default is `75`). This only affects compression.
`'optimize'`	Perform Huffman table optimization. Takes longer, but results in smaller compressed image. This only affects compression.
`'smooth'`	Perform inter-block smoothing on uncompressed image. Only useful for low-quality images. This only affects decompression.

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 (http://www.w3.org/Graphics/JPEG/itu-t81.pdf)
The ISO standard for JPEG is also published as ITU T.81. This is available online in PDF form.

CHAPTER

TWENTYONE

SunOS Specific Services

The modules described in this chapter provide interfaces to features that are unique to the SunOS operating system (versions 4 and 5; the latter is also known as Solaris version 2).

21.1 `sunaudiodev` — Access to Sun audio hardware

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.

21.1.1 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:

```
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)
```

21.2 SUNAUDIODEV — Constants used with sunaudiodev

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.

CHAPTER

TWENTYTWO

MS Windows Specific Services

This chapter describes modules that are only available on MS Windows platforms.

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.

22.1 msvcrt – Useful routines from the MS VC++ runtime

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.

22.1.1 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.

22.1.2 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()`.

22.1.3 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`.

22.2 _winreg – Windows registry access

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 the `handle.Close()` 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:

Index	Meaning
`0`	A string that identifies the value name
`1`	An object that holds the value data, and whose type depends on the underlying registry type
`2`	An integer that identifies the type of the value data

`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 *fileName* 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:

Index	Meaning
0	An integer giving the number of sub keys this key has.
1	An integer giving the number of values this key has.
2	A long integer giving when the key was last modified (if available) as 100's of nanoseconds since Jan 1, 1600.

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:

Index	Meaning
0	The value of the registry item.
1	An integer giving the registry type for this value.

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:

Constant	Meaning
`REG_BINARY`	Binary data in any form.
`REG_DWORD`	A 32-bit number.
`REG_DWORD_LITTLE_ENDIAN`	A 32-bit number in little-endian format.
`REG_DWORD_BIG_ENDIAN`	A 32-bit number in big-endian format.
`REG_EXPAND_SZ`	Null-terminated string containing references to environment variables (‘`%PATH%`’).
`REG_LINK`	A Unicode symbolic link.
`REG_MULTI_SZ`	A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)
`REG_NONE`	No defined value type.
`REG_RESOURCE_LIST`	A device-driver resource list.
`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.

22.2.1 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 `Close()` method on the object, or the `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 `__nonzero__()` - thus

```
if handle:
    print "Yes"
```

will print `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 (eg, using the builtin `int()` function, in which case the underlying Windows handle value is returned. You can also use the `Detach()` method to return the integer handle, and also disconnect the Windows handle from the handle object.

`Close``()`
Closes the underlying Windows handle.

If the handle is already closed, no error is raised.

`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.

22.3 `winsound` — Sound-playing interface for Windows

New in version 1.5.2.

The `winsound` module provides access to the basic sound-playing machinery provided by Windows platforms. It includes two 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 bitwise 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:

`PlaySound()` *name*	Corresponding Control Panel Sound name
`'SystemAsterisk'`	Asterisk
`'SystemExclamation'`	Exclamation
`'SystemExit'`	Exit Windows
`'SystemHand'`	Critical Stop
`'SystemQuestion'`	Question

For example:

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" 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.

APPENDIX

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.

A.1 Frameworks

Frameworks tend to be harder to document, but are well worth the effort spent.

`test` — Regression testing framework. This is used for the Python regression test, but is useful for other Python libraries as well. This is a package rather than a single module.

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).

`platform` — This module tries to retrieve as much platform identifying data as possible. It makes this information available via function APIs. If called from the command line, it prints the platform information concatenated as single string to `sys.stdout`. The output format is useable as part of a filename. New in version 2.3.

`smtpd` — An SMTP daemon implementation which meets the minimum requirements for RFC 821 conformance.

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

audiodev — Platform-independent API for playing audio data.

linuxaudiodev — Play audio data on the Linux audio device. Replaced in Python 2.3 by the `ossaudiodev` 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**.

ossaudiodev — Play audio data via the Open Sound System API. This is usable on Linux, some flavors of BSD, and some commercial UNIX platforms.

A.5 Obsolete

These modules are not normally available for import; additional work must be done to make them available.

Those which are written in Python will be installed into the directory 'lib-old/' installed as part of the standard library. To use these, the directory must be added to `sys.path`, possibly using PYTHONPATH.

Obsolete 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.

addpack — Alternate approach to packages. Use the built-in package support instead.

cmp — File comparison function. Use the newer `filecmp` instead.

cmpcache — Caching version of the obsolete `cmp` module. Use the newer `filecmp` instead.

codehack — Extract function name or line number from a function code object (these are now accessible as attributes: `co.co_name`, `func.func_name`, `co.co_firstlineno`).

dircmp — Class to build directory diff tools on (may become a demo or tool). **Deprecated since release 2.0.** The `filecmp` module replaces `dircmp`.

dump — Print python code that reconstructs a variable.

fmt — Text formatting abstractions (too slow).

lockfile — Wrapper around FCNTL file locking (use `fcntl.lockf()`/`flock()` instead; see `fcntl`).

newdir — New `dir()` function (the standard `dir()` is now just as good).

Para — Helper for `fmt`.

poly — Polynomials.

regex — Emacs-style regular expression support; may still be used in some old code (extension module). Refer to the *Python 1.6 Documentation* for documentation.

regsub — Regular expression based string replacement utilities, for use with `regex` (extension module). Refer to the *Python 1.6 Documentation* for documentation.

tb — Print tracebacks, with a dump of local variables (use `pdb.pm()` or `traceback` instead).

timing — Measure time intervals to high resolution (use `time.clock()` instead). (This is an extension module.)

tzparse — Parse a timezone specification (unfinished; may disappear in the future, and does not work when the TZ environment variable is not set).

util — Useful functions that don't fit elsewhere.

whatsound — Recognize sound files; use `sndhdr` instead.

zmod — Compute properties of mathematical "fields."

The following modules are obsolete, but are likely to re-surface as tools or scripts:

find — Find files matching pattern in directory tree.

grep — **grep** implementation in Python.

packmail — Create a self-unpacking UNIX shell archive.

The following modules were documented in previous versions of this manual, but are now considered obsolete. The source for the documentation is still available as part of the documentation source archive.

ni — Import modules in "packages." Basic package support is now built in. The built-in support is very similar to what is provided in this module.

rand — Old interface to the random number generator.

soundex — Algorithm for collapsing names which sound similar to a shared key. The specific algorithm doesn't seem to match any published algorithm. (This is an extension module.)

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).

APPENDIX

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). 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 near the bottom 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
(http://www-mice.cs.ucl.ac.uk/multimedia/software/documentation/ReportingBugs.html)
: 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
(http://www.mozilla.org/quality/bug-writing-guidelines.html)
: Information about writing a good bug report. Some of this is specific to the Mozilla project, but describes general good practices.

APPENDIX

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.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.1	2003	PSF	yes

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 2.3.2

1. This LICENSE AGREEMENT is between the Python Software Foundation (“PSF”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 2.3.2 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 2.3.2 alone or in any derivative version, provided, however, that PSF’s License Agreement and PSF’s notice of copyright, i.e., “Copyright © 2001-2003 Python Software Foundation; All Rights Reserved” are retained in Python 2.3.2 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 2.3.2 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 2.3.2.

4. PSF is making Python 2.3.2 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 2.3.2 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.3.2 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.3.2, 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 2.3.2, 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

MODULE INDEX

S

T

U

W

X

Z

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