Core Modules

"Since the functions in the C runtime library are not part of the Win32 API, we believe the number of applications that will be affected by this bug to be very limited"

Microsoft, January 1999

Overview

Python's standard library covers a wide range of modules. Everything from modules that are as much a part of the Python language as the types and statements defined by the language specification, to obscure modules that are probably useful only to a small number of programs.

This section describes a number of fundamental standard library modules. Any larger Python program is likely to use most of these modules, either directly or indirectly.

Built-in Functions and Exceptions

Two modules are even more basic than all other modules combined: the **__builtin__** module defines built-in functions (like **len**, **int**, and **range**), and the **exceptions** module defines all built-in exceptions.

Python imports both modules when it starts up, and makes their content available for all programs.

Operating System Interface Modules

There are a number of modules providing platform-independent interfaces to the underlying operating system. They are modeled after the POSIX standard API and the standard C library.

The modules in this group include **os**, which provides file and process operations, **os.path** which offers a platform-independent way to pull apart and put together file names, and **time** which provides functions to work with dates and times.

To some extent, networking and thread support modules could also belong in this group, but they are not supported by all Python implementations.

Type Support Modules

Several built-in types have support modules in the standard library. The **string** module implements commonly used string operations, the **math** module provides math operations and constants, and the **cmath** module does the same for complex numbers.

Regular Expressions

The **re** module provides regular expressions support for Python. Regular expressions are string patterns written in a special syntax, which can be used to match strings, and extract substrings.

Language Support Modules

sys gives you access to various interpreter variables, such as the module search path, and the interpreter version. **operator** provides functional equivalents to many built-in operators. **copy** allows you to copy objects. And finally, **gc** gives you more control over the garbage collector facilities in Python 2.0.

The __builtin__ module

This module contains built-in functions which are automatically available in all Python modules. You usually don't have to import this module; Python does that for you when necessary.

Calling a function with arguments from a tuple or dictionary

Python allows you to build function argument lists on the fly. Just put all the arguments in a tuple, and call the built-in **apply** function:

Example: Using the apply function

```
# File:builtin-apply-example-1.py

def function(a, b):
    print a, b

apply(function, ("whither", "canada?"))
apply(function, (1, 2 + 3))
```

```
whither canada?
1 5
```

To pass keyword arguments to a function, you can use a dictionary as the third argument to **apply**:

Example: Using the apply function to pass keyword arguments

```
# File:builtin-apply-example-2.py

def function(a, b):
    print a, b

apply(function, ("crunchy", "frog"))
apply(function, ("crunchy",), {"b": "frog"})
apply(function, (), {"a": "crunchy", "b": "frog"})
```

```
crunchy frog
crunchy frog
crunchy frog
```

One common use for **apply** is to pass constructor arguments from a subclass on to the base class, especially if the constructor takes a lot of arguments.

Example: Using the apply function to call base class constructors

```
# File:builtin-apply-example-3.py

class Rectangle:
    def __init__(self, color="white", width=10, height=10):
        print "create a", color, self, "sized", width, "x", height

class RoundedRectangle(Rectangle):
    def __init__(self, **kw):
        apply(Rectangle.__init__, (self,), kw)

rect = Rectangle(color="green", height=100, width=100)
rect = RoundedRectangle(color="blue", height=20)
```

```
create a green <Rectangle instance at 8c8260> sized 100 x 100
create a blue <RoundedRectangle instance at 8c84c0> sized 10 x 20
```

Python 2.0 provides an alternate syntax. Instead of **apply**, you can use an ordinary function call, and use * to mark the tuple, and ** to mark the dictionary.

The following two statements are equivalent:

```
result = function(*args, **kwargs)
result = apply(function, args, kwargs)
```

Loading and reloading modules

If you've written a Python program larger than just a few lines, you know that the **import** statement is used to import external modules (you can also use the **from-import** version). What you might not know already is that **import** delegates the actual work to a built-in function called **__import__**.

The trick is that you can actually call this function directly. This can be handy if you have the module name in a string variable, like in the following example, which imports all modules whose names end with "**-plugin**":

Example: Using the __import__ function to load named modules

```
# File:builtin-import-example-1.py

import glob, os

modules = []

for module_file in glob.glob("*-plugin.py"):
    try:
        module_name, ext = os.path.splitext(os.path.basename(module_file))
        module = __import__(module_name)
        modules.append(module)
    except ImportError:
        pass # ignore broken modules

# say hello to all modules
for module in modules:
    module.hello()
```

```
example-plugin says hello
```

Note that the plugin modules have hyphens in the name. This means that you cannot import such a module using the ordinary **import** command, since you cannot have hyphens in Python identifiers.

Here's the plugin used in this example:

Example: A sample plugin

```
# File:example-plugin.py

def hello():
    print "example-plugin says hello"
```

The following example shows how to get a function object, given that you have the module and function name as strings:

Example: Using the __import__ function to get a named function

```
# File:builtin-import-example-2.py

def getfunctionbyname(module_name, function_name):
    module = __import__(module_name)
    return getattr(module, function_name)

print repr(getfunctionbyname("dumbdbm", "open"))
```

```
<function open at 794fa0>
```

You can also use this function to implement lazy loading of modules. In the following example, the **string** module is imported when it is first used:

Example: Using the __import__ function to implement lazy import

```
# File:builtin-import-example-3.py

class LazyImport:
    def __init__(self, module_name):
        self.module_name = module_name
        self.module = None
    def __getattr__(self, name):
        if self.module is None:
            self.module = __import__(self.module_name)
        return getattr(self.module, name)

string = LazyImport("string")

print string.lowercase
```

```
abcdefghijklmnopqrstuvwxyz
```

Python provides some basic support for reloading modules that you've already imported. The following example loads the **hello.py** file three times:

Example: Using the reload function

```
# File:builtin-reload-example-1.py

import hello
reload(hello)
reload(hello)
```

```
hello again, and welcome to the show
hello again, and welcome to the show
hello again, and welcome to the show
```

reload uses the module name associated with the module object, not the variable name. This means that even if you've renamed the module, **reload** will still be able to find the original module.

Note that when you reload a module, it is recompiled, and the new module replaces the old one in the module dictionary. However, if you have created instances of classes defined in that module, those instances will still use the old implementation.

Likewise, if you've used **from-import** to create references to module members in other modules, those references will not be updated.

Looking in namespaces

The **dir** function returns a list of all members of a given module, class, instance, or other type. It's probably most useful when you're working with an interactive Python interpreter, but can also come in handy in other situations.

Example: Using the dir function

```
# File:builtin-dir-example-1.py

def dump(value):
    print value, "=>", dir(value)

import sys

dump(0)
dump(1.0)
dump(0.0j) # complex number
dump([]) # list
dump({}) # dictionary
dump("string")
dump(len) # function
dump(sys) # module
```

```
0 => []
1.0 => []
0j => ['conjugate', 'imag', 'real']
[] => ['append', 'count', 'extend', 'index', 'insert',
    'pop', 'remove', 'reverse', 'sort']
{} => ['clear', 'copy', 'get', 'has_key', 'items',
    'keys', 'update', 'values']
string => []
<built-in function len> => ['__doc__', '__name__', '__self__']
<module 'sys' (built-in)> => ['__doc__', '__name__',
    '__stderr__', '__stdin__', '__stdout__', 'argv',
    'builtin_module_names', 'copyright', 'dllhandle',
    'exc_info', 'exc_type', 'exec_prefix', 'executable',
...
```

In the following example, the **getmember** function returns all class-level attributes and methods defined by a given class:

Example: Using the dir function to find all members of a class

```
# File:builtin-dir-example-2.py

class A:
    def a(self):
        pass
    def b(self):
        pass
```

```
class B(A):
    def c(self):
        pass
    def d(self):
        pass

def getmembers(klass, members=None):
    # get a list of all class members, ordered by class
    if members is None:
        members = []
    for k in klass.__bases__:
        getmembers(k, members)
    for m in dir(klass):
        if m not in members:
            members.append(m)
    return members

print getmembers(A)
print getmembers(B)
print getmembers(IOError)
```

```
['__doc__', '__module__', 'a', 'b']
['__doc__', '__module__', 'a', 'b', 'c', 'd']
['__doc__', '__getitem__', '__init__', '__module__', '__str__']
```

Note that the **getmembers** function returns an ordered list. The earlier a name appears in the list, the higher up in the class hierarchy it's defined. If order doesn't matter, you can use a dictionary to collect the names instead of a list.

The **vars** function is similar, but it returns a dictionary containing the current value for each member. If you use it without an argument, it returns a dictionary containing what's visible in the current local namespace:

Example: Using the vars function

```
# File:builtin-vars-example-1.py

book = "library2"
pages = 250
scripts = 350

print "the %(book)s book contains more than %(scripts)s scripts" % vars()
```

```
the library book contains more than 350 scripts
```

Checking an object's type

Python is a dynamically typed language, which means that a given variable can be bound to values of different types at different occasions. In the following example, the same function is called with an integer, a floating point value, and a string:

```
def function(value):
    print value

function(1)
function(1.0)
function("one")
```

The **type** function allows you to check what type a variable has. This function returns a *type descriptor*, which is a unique object for each type provided by the Python interpreter.

Example: Using the type function

```
# File:builtin-type-example-1.py

def dump(value):
    print type(value), value

dump(1)
dump(1.0)
dump("one")
```

```
<type 'int'> 1
<type 'float'> 1.0
<type 'string'> one
```

Each type has a single corresponding type object, which means that you can use the **is** operator (object identity) to do type testing:

Example: Using the type function to distinguish between file names and file objects

```
# File:builtin-type-example-2.py

def load(file):
    if isinstance(file, type("")):
        file = open(file, "rb")
    return file.read()

print len(load("samples/sample.jpg")), "bytes"
print len(load(open("samples/sample.jpg", "rb"))), "bytes"
```

```
4672 bytes
4672 bytes
```

The **callable** function checks if an object can be called (either directly or via **apply**). It returns true for functions, methods, lambda expressions, classes, and class instances which define the **__call__** method.

Example: Using the callable function

```
# File:builtin-callable-example-1.py

def dump(function):
    if callable(function):
        print function, "is callable"
    else:
        print function, "is *not* callable"

class A:
    def method(self, value):
        return value

class B(A):
    def __call__(self, value):
        return value

a = A()
b = B()

dump(0) # simple objects
dump("string")
dump(callable)
dump(dump) # function

dump(A) # classes
dump(B)
dump(B.method)

dump(a) # instances
dump(b)
dump(b.method)
```

```
0 is *not* callable
string is *not* callable
<built-in function callable> is callable
<function dump at 8ca320> is callable
A is callable
B is callable
<unbound method A.method> is callable
<A instance at 8caa10> is *not* callable
<B instance at 8cab00> is callable
<method A.method of B instance at 8cab00> is callable
```

Note that the class objects (**A** and **B**) are both callable; if you call them, they create new objects. However, instances of class **A** are not callable, since that class doesn't have a **__call__** method.

You'll find functions to check if an object is of any of the built-in number, sequence, or dictionary types in the **operator** module. However, since it's easy to create a class that implements e.g. the basic sequence methods, it's usually a bad idea to use explicit type testing on such objects.

Things get even more complicated when it comes to classes and instances. Python doesn't treat classes as types per se. Instead, all classes belong to a special class type, and all class instances belong to a special instance type.

This means that you cannot use **type** to test if an instance belongs to a given class; *all instances have the same type!* To solve this, you can use the **isinstance** function, which checks if an object is an instance of a given class (or of a subclass to it).

Example: Using the isinstance function

```
# File:builtin-isinstance-example-1.py

class A:
    pass

class B:
    pass

class C(A):
    pass

class D(A, B):
    pass

def dump(object):
    print object, "=>",
    if isinstance(object, A):
        print "A",
    if isinstance(object, B):
        print "B",
    if isinstance(object, C):
        print "C",
    if isinstance(object, D):
        print "D",
    print

a = A()
b = B()
c = C()
d = D()

dump(a)
dump(b)
dump(c)
dump(d)
dump(0)
dump("string")
```

```
<A instance at 8ca6d0> => A
<B instance at 8ca750> => B
<C instance at 8ca780> => A C
<D instance at 8ca7b0> => A B D
0 =>
string =>
```

The **issubclass** function is similar, but checks whether a class object is the same as a given class, or is a subclass of it.

Note that while **isinstance** accepts any kind of object, the **issubclass** function raises a **TypeError** exception if you use it on something that is not a class object.

Example: Using the issubclass function

```
# File:builtin-issubclass-example-1.py

class A:
    pass

class B:
    pass

class C(A):
    pass

class D(A, B):
    pass

def dump(object):
    print object, "=>",
    if issubclass(object, A):
        print "A",
    if issubclass(object, B):
        print "B",
    if issubclass(object, C):
        print "C",
    if issubclass(object, D):
        print "D",
    print

dump(A)
dump(B)
dump(C)
dump(D)
dump(0)
dump("string")
```

```
A => A
B => B
C => A C
D => A B D
0 =>
Traceback (innermost last):
  File "builtin-issubclass-example-1.py", line 29, in ?
  File "builtin-issubclass-example-1.py", line 15, in dump
TypeError: arguments must be classes
```

Evaluating Python expressions

Python provides several ways to interact with the interpreter from within a program. For example, the **eval** function evaluates a string as if it were a Python expression. You can pass it a literal, simple expressions, or even use built-in functions:

Example: Using the eval function

```
# File:builtin-eval-example-1.py

def dump(expression):
    result = eval(expression)
    print expression, "=>", result, type(result)

dump("1")
dump("1.0")
dump("'string'")
dump("1.0 + 2.0")
dump("'*' * 10")
dump("len('world')")
```

```
1 => 1 <type 'int'>
1.0 => 1.0 <type 'float'>
'string' => string <type 'string'>
1.0 + 2.0 => 3.0 <type 'float'>
'*' * 10 => ********** <type 'string'>
len('world') => 5 <type 'int'>
```

A problem with **eval** is that if you cannot trust the source from which you got the string, you may get into trouble. For example, someone might use the built-in **__import__** function to load the **os** module, and then remove files on your disk:

Example: Using the eval function to execute arbitrary commands

```
# File:builtin-eval-example-2.py

print eval("__import__('os').getcwd()")
print eval("__import__('os').remove('file')")
```

```
/home/fredrik/librarybook
Traceback (innermost last):
 File "builtin-eval-example-2", line 2, in ?
 File "<string>", line 0, in ?
os.error: (2, 'No such file or directory')
```

Note that you get an **os.error** exception, which means that *Python actually tried to remove the file!*

Luckily, there's a way around this problem. You can pass a second argument to **eval**, which should contain a dictionary defining the namespace in which the expression is evaluated. Let's pass in an empty namespace:

```
>>> print eval("__import__('os').remove('file')", {})
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<string>", line 0, in ?
os.error: (2, 'No such file or directory')
```

Hmm. We still end up with an **os.error** exception.

The reason for this is that Python looks in the dictionary before it evaluates the code, and if it doesn't find a variable named **__builtins__** in there (note the plural form), it adds one:

```
>>> namespace = {}
>>> print eval("__import__('os').remove('file')", namespace)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<string>", line 0, in ?
os.error: (2, 'No such file or directory')
>>> namespace.keys()
['__builtins__']
```

If you print the contents of the namespace variable, you'll find that it contains the full set of built-in functions.

The solution to this little dilemma isn't far away: since Python doesn't add this item if it is already there, you just have to add a dummy item called **__builtins__** to the namespace before calling **eval**:

Example: Safely using the eval function to evaluate arbitrary strings

```
# File:builtin-eval-example-3.py

print eval("__import__('os').getcwd()", {})
print eval("__import__('os').remove('file')", {"__builtins__": {}})
```

```
/home/fredrik/librarybook
Traceback (innermost last):
  File "builtin-eval-example-3.py", line 2, in ?
  File "<string>", line 0, in ?
NameError: __import__
```

Note that this doesn't product you from CPU or memory resource attacks (for example, something like **eval("'*'*1000000*2*2*2*2*2*2*2*2*2")** will most likely cause your program to run out of memory after a while)

Compiling and executing code

The **eval** function only works for simple expressions. To handle larger blocks of code, use the **compile** and **exec** functions:

Example: Using the compile function to check syntax

```
# File:builtin-compile-example-1.py

NAME = "script.py"

BODY = """
prnt 'owl-stretching time'
"""

try:
    compile(BODY, NAME, "exec")
except SyntaxError, v:
    print "syntax error:", v, "in", NAME
```

```
syntax error: invalid syntax in script.py
```

When successful, the **compile** function returns a code object, which you can execute with the **exec** statement:

Example: Compiling and executing compiled code

```
# File:builtin-compile-example-2.py

BODY = """
print 'the ant, an introduction'
"""

code = compile(BODY, "<script>", "exec")

print code

exec code
```

```
<code object ? at 8c6be0, file "<script>", line 0>
the ant, an introduction
```

To generate code on the fly, you can use the class shown in the following example. Use the **write** method to add statements, and **indent** and **dedent** to add structure, and this class takes care of the rest.

Example: A simple code generator tool

```
# File:builtin-compile-example-3.py

import sys, string

class CodeGeneratorBackend:
    "Simple code generator for Python"

    def begin(self, tab="\t"):
        self.code = []
        self.tab = tab
        self.level = 0

    def end(self):
        self.code.append("") # make sure there's a newline at the end
        return compile(string.join(self.code, "\n"), "<code>", "exec")

    def write(self, string):
        self.code.append(self.tab * self.level + string)

    def indent(self):
        self.level += 1
        # in Python 1.5.2 and earlier, use this instead:
        # self.level = self.level + 1

    def dedent(self):
        if self.level == 0:
            raise SyntaxError, "internal error in code generator"
        self.level -= 1
        # in Python 1.5.2 and earlier, use this instead:
        # self.level = self.level - 1

#
# try it out!

c = CodeGeneratorBackend()
c.begin()
c.write("for i in range(5):")
c.indent()
c.write("print 'code generation made easy!'")
c.dedent()
exec c.end()
```

```
code generation made easy!
code generation made easy!
code generation made easy!
code generation made easy!
code generation made easy!
```

Python also provides a function called **execfile**. It's simply a shortcut for loading code from a file, compiling it, and executing it. The following example shows how to use and emulate this function.

Example: Using the execfile function

```
# File:builtin-execfile-example-1.py

execfile("hello.py")

def EXECFILE(filename, locals=None, globals=None):
    exec compile(open(filename).read(), filename, "exec") in locals, globals

EXECFILE("hello.py")
```

```
hello again, and welcome to the show
hello again, and welcome to the show
```

The **hello.py** file used in this example has the following contents:

Example: The hello.py script

```
# File:hello.py

print "hello again, and welcome to the show"
```

Overloading functions from the __builtin__ module

Since Python looks among the built-in functions *after* it has checked the local and module namespace, there may be situations when you need to explicitly refer to the **__builtin__** module. For example, the following script overloads the **open** function with a version that opens an ordinary file and checks that it starts with a "magic" string. To be able to use the original **open** function, it explicitly refers to it using the module name.

Example: Explicitly accessing functions in the __builtin__ module

```
# File:builtin-open-example-1.py

def open(filename, mode="rb"):
    import __builtin__
    file = __builtin__.open(filename, mode)
    if file.read(5) not in("GIF87", "GIF89"):
        raise IOError, "not a GIF file"
    file.seek(0)
    return file

fp = open("samples/sample.gif")
print len(fp.read()), "bytes"

fp = open("samples/sample.jpg")
print len(fp.read()), "bytes"
```

```
3565 bytes
Traceback (innermost last):
  File "builtin-open-example-1.py", line 12, in ?
  File "builtin-open-example-1.py", line 5, in open
IOError: not a GIF file
```

The exceptions module

This module provides the standard exception hierarchy. It's automatically imported when Python starts, and the exceptions are added to the **__builtin__** module. In other words, you usually don't need to import this module.

This is a Python module in 1.5.2, and a built-in module in 2.0 and later.

The following standard exceptions are defined by this module:

- **Exception** is used as a base class for all exceptions. It's strongly recommended (but not yet required) that user exceptions are derived from this class too.
- **SystemExit(Exception)** is raised by the **sys.exit** function. If it propagates to the top level without being caught by a **try-except** clause, the interpreter is terminated without a traceback message.
- **StandardError(Exception)** is used as a base class for all standard exceptions (except **SystemExit**, that is).
- **KeyboardInterrupt(StandardError)** is raised when the user presses Control-C (or any other interrupt key). Note that this may cause strange errors if you use "catch all" **try-except** statements.
- **ImportError(StandardError)** is raised when Python fails to import a module.
- **EnvironmentError** is used as a base class for exceptions that can be caused by the interpreter's environment (that is, they're usually not caused by bugs in the program).
- **IOError(EnvironmentError)** is used to flag I/O-related errors.
- **OSError(EnvironmentError)** is used to flag errors by the **os** module.
- **WindowsError(OSError)** is used to flag Windows-specific errors from the **os** module.
- **NameError(StandardError)** is raised when Python fails to find a global or local name.
- **UnboundLocalError(NameError)** is raised if your program attempts to access a local variable before it has been assigned a value. This exception is only used in 2.0 and later; earlier versions raise a plain **NameError** exception instead.
- **AttributeError(StandardError)** is raised when Python fails to find (or assign to) an instance attribute, a method, a module function, or any other qualified name.
- **SyntaxError(StandardError)** is raised when the compiler stumbles upon a syntax error.
- (2.0 and later) **IndentationError(SyntaxError)** is raised for syntax errors caused by bad indentation. This exception is only used in 2.0 and later; earlier versions raise a plain **SyntaxError** exception instead.
- (2.0 and later) **TabError(IndentationError)** is raised by the interpreter when the **-tt** option is used to check for inconsistent indentation. This exception is only used in 2.0 and later; earlier versions raise a plain **SyntaxError** exception instead.

- **TypeError(StandardError)** is raised when an operation cannot be applied to an object of the given type.
- **AssertionError(StandardError)** is raised when an **assert** statement fails (if the expression is false, that is).
- **LookupError(StandardError)** is used as a base class for exceptions raised when a sequence or dictionary type doesn't contain a given index or key.
- **IndexError(LookupError)** is raised by sequence objects when the given index doesn't exist.
- **KeyError(LookupError)** is raised by dictionary objects when the given key doesn't exist.
- **ArithmeticError(StandardError)** is used as a base class for math-related exceptions.
- **OverflowError(ArithmeticError)** is raised when an operations overflows (for example, when an integer is too large to fit in the given type).
- **ZeroDivisionError(ArithmeticError)** is raised when you try to divide a number by zero.
- **FloatingPointError(ArithmeticError)** is raised when a floating point operation fails.
- **ValueError(StandardError)** is raised if an argument has the right type, but an invalid value.
- (2.0 and later) **UnicodeError(ValueError)** is raised for type problems related to the Unicode string type. This is only used in 2.0 and later.
- **RuntimeError(StandardError)** is used for various run-time problems, including attempts to get outside the box when running in restricted mode, unexpected hardware problems, etc.
- **NotImplementedError(RuntimeError)** can be used to flag functions that hasn't been implemented yet, or methods that should be overridden.
- **SystemError(StandardError)** is raised if the interpreter messes up, and knows about it. The exception value contains a more detailed description (usually something cryptic, like "**eval_code2: NULL globals**" or so). I cannot recall ever seeing this exception in over five years of full-time Python programming, but maybe that's just me.
- **MemoryError(StandardError)** is raised when the interpreter runs out of memory. Note that this only happens when the underlying memory allocation routines complain; you can often send your poor computer into a mindless swapping frenzy before that happens.

You can create your own exception classes. Just inherit from the built-in **Exception** class (or a proper standard exception), and override the constructor and/or **__str__** method as necessary.

Example: Using the exceptions module

```
# File:exceptions-example-1.py

# python imports this module by itself, so the following
# line isn't really needed
# import exceptions

class HTTPError(Exception):
    # indicates an HTTP protocol error
    def __init__(self, url, errcode, errmsg):
        self.url = url
        self.errcode = errcode
        self.errmsg = errmsg
    def __str__(self):
        return (
            "<HTTPError for %s: %s %s>" %
            (self.url, self.errcode, self.errmsg)
            )

try:
    raise HTTPError("http://www.python.org/foo", 200, "Not Found")
except HTTPError, error:
    print "url", "=>", error.url
    print "errcode", "=>", error.errcode
    print "errmsg", "=>", error.errmsg
    raise # reraise exception
```

```
url => http://www.python.org/foo
errcode => 200
errmsg => Not Found
Traceback (innermost last):
  File "exceptions-example-1", line 16, in ?
HTTPError: <HTTPError for http://www.python.org/foo: 200 Not Found>
```

The os module

This module provides a unified interface to a number of operating system functions.

Most of the functions in this module are implemented by platform specific modules, such as **posix** and **nt**. The **os** module automatically loads the right implementation module when it is first imported.

Working with files

The built-in **open** function lets you create, open, and modify files. This module adds those extra functions you need to rename and remove files:

Example: Using the os module to rename and remove files

```
# File:os-example-3.py

import os
import string

def replace(file, search_for, replace_with):
    # replace strings in a text file

    back = os.path.splitext(file)[0] + ".bak"
    temp = os.path.splitext(file)[0] + ".tmp"

    try:
        # remove old temp file, if any
        os.remove(temp)
    except os.error:
        pass

    fi = open(file)
    fo = open(temp, "w")

    for s in fi.readlines():
        fo.write(string.replace(s, search_for, replace_with))

    fi.close()
    fo.close()

    try:
        # remove old backup file, if any
        os.remove(back)
    except os.error:
        pass

    # rename original to backup...
    os.rename(file, back)

    # ...and temporary to original
    os.rename(temp, file)
```

```
#
# try it out!

file = "samples/sample.txt"

replace(file, "hello", "tjena")
replace(file, "tjena", "hello")
```

Working with directories

The os module also contains a number of functions that work on entire directories.

The **listdir** function returns a list of all filenames in a given directory. The current and parent directory markers used on Unix and Windows (**.** and **..**) are not included in this list.

Example: Using the os module to list the files in a directory

```
# File:os-example-5.py

import os

for file in os.listdir("samples"):
    print file
```

```
sample.au
sample.jpg
sample.wav
...
```

The **getcwd** and **chdir** functions are used to get and set the current directory:

Example: Using the os module to change the working directory

```
# File:os-example-4.py

import os

# where are we?
cwd = os.getcwd()
print "1", cwd

# go down
os.chdir("samples")
print "2", os.getcwd()

# go back up
os.chdir(os.pardir)
print "3", os.getcwd()
```

```
1 /ematter/librarybook
2 /ematter/librarybook/samples
3 /ematter/librarybook
```

The **makedirs** and **removedirs** functions are used to create and remove directory hierarchies.

Example: Using the os module to create and remove multiple directory levels

```
# File:os-example-6.py

import os

os.makedirs("test/multiple/levels")

fp = open("test/multiple/levels/file", "w")
fp.write("inspector praline")
fp.close()

# remove the file
os.remove("test/multiple/levels/file")

# and all empty directories above it
os.removedirs("test/multiple/levels")
```

Note that **removedirs** removes all empty directories along the given path, starting with the last directory in the given path name. In contrast, the **mkdir** and **rmdir** functions can only handle a single directory level.

Example: Using the os module to create and remove directories

```
# File:os-example-7.py

import os

os.mkdir("test")
os.rmdir("test")

os.rmdir("samples") # this will fail
```

```
Traceback (innermost last):
  File "os-example-7", line 6, in ?
OSError: [Errno 41] Directory not empty: 'samples'
```

To remove non-empty directories, you can use the **rmtree** function in the **shutil** module.

Working with file attributes

The **stat** function fetches information about an existing file. It returns a 9-tuple which contains the size, inode change timestamp, modification timestamp, and access privileges.

Example: Using the os module to get information about a file

```
# File:os-example-1.py

import os
import time

file = "samples/sample.jpg"

def dump(st):
    mode, ino, dev, nlink, uid, gid, size, atime, mtime, ctime = st
    print "- size:", size, "bytes"
    print "- owner:", uid, gid
    print "- created:", time.ctime(ctime)
    print "- last accessed:", time.ctime(atime)
    print "- last modified:", time.ctime(mtime)
    print "- mode:", oct(mode)
    print "- inode/dev:", ino, dev

#
# get stats for a filename

st = os.stat(file)

print "stat", file
dump(st)
print

#
# get stats for an open file

fp = open(file)

st = os.fstat(fp.fileno())

print "fstat", file
dump(st)
```

```
stat samples/sample.jpg
- size: 4762 bytes
- owner: 0 0
- created: Tue Sep 07 22:45:58 1999
- last accessed: Sun Sep 19 00:00:00 1999
- last modified: Sun May 19 01:42:16 1996
- mode: 0100666
- inode/dev: 0 2

fstat samples/sample.jpg
- size: 4762 bytes
- owner: 0 0
- created: Tue Sep 07 22:45:58 1999
- last accessed: Sun Sep 19 00:00:00 1999
- last modified: Sun May 19 01:42:16 1996
- mode: 0100666
- inode/dev: 0 0
```

Some fields don't make sense on non-Unix platforms; for example, the (inode, dev) tuple provides a unique identity for each file on Unix, but can contain arbitrary data on other platforms.

The **stat** module contains a number of useful constants and helper functions for dealing with the members of the stat tuple. Some of these are shown in the examples below.

You can modify the mode and time fields using the **chmod** and **utime** functions:

Example: Using the os module to change a file's privileges and timestamps

```
# File:os-example-2.py

import os
import stat, time

infile = "samples/sample.jpg"
outfile = "out.jpg"

# copy contents
fi = open(infile, "rb")
fo = open(outfile, "wb")

while 1:
    s = fi.read(10000)
    if not s:
        break
    fo.write(s)

fi.close()
fo.close()

# copy mode and timestamp
st = os.stat(infile)
os.chmod(outfile, stat.S_IMODE(st[stat.ST_MODE]))
os.utime(outfile, (st[stat.ST_ATIME], st[stat.ST_MTIME]))

print "original", "=>"
print "mode", oct(stat.S_IMODE(st[stat.ST_MODE]))
print "atime", time.ctime(st[stat.ST_ATIME])
print "mtime", time.ctime(st[stat.ST_MTIME])

print "copy", "=>"
st = os.stat(outfile)
print "mode", oct(stat.S_IMODE(st[stat.ST_MODE]))
print "atime", time.ctime(st[stat.ST_ATIME])
print "mtime", time.ctime(st[stat.ST_MTIME])
```

```
original =>
mode 0666
atime Thu Oct 14 15:15:50 1999
mtime Mon Nov 13 15:42:36 1995
copy =>
mode 0666
atime Thu Oct 14 15:15:50 1999
mtime Mon Nov 13 15:42:36 1995
```

Working with processes

The **system** function runs a new command under the current process, and waits for it to finish.

Example: Using the os module to run an operating system command

```
# File:os-example-8.py

import os

if os.name == "nt":
   command = "dir"
else:
   command = "ls -l"

os.system(command)
```

```
-rwxrw-r--   1 effbot  effbot       76 Oct  9 14:17 README
-rwxrw-r--   1 effbot  effbot     1727 Oct  7 19:00 SimpleAsyncHTTP.py
-rwxrw-r--   1 effbot  effbot      314 Oct  7 20:29 aifc-example-1.py
-rwxrw-r--   1 effbot  effbot      259 Oct  7 20:38 anydbm-example-1.py
...
```

The command is run via the operating system's standard shell, and returns the shell's exit status. Under Windows 95/98, the shell is usually **command.com** whose exit status is always 0.

> ***Warning:***
>
> *Since **os.system** passes the command on to the shell as is, it can be dangerous to use if you don't check the arguments carefully (consider running **os.system("viewer %s" % file)** with the file variable set to **"sample.jpg; rm -rf $HOME"**). When unsure, it's usually better to use **exec** or **spawn** instead (see below).*

The **exec** function starts a new process, replacing the current one ("go to process", in other words). In the following example, note that the "goodbye" message is never printed:

Example: Using the os module to start a new process

```
# File:os-exec-example-1.py

import os
import sys

program = "python"
arguments = ["hello.py"]

print os.execvp(program, (program,) +  tuple(arguments))
print "goodbye"
```

```
hello again, and welcome to the show
```

Python provides a whole bunch of **exec** functions, with slightly varying behavior. The above example uses **execvp**, which searches for the program along the standard path, passes the contents of the

second argument tuple as individual arguments to that program, and runs it with the current set of environment variables. See the *Python Library Reference* for more information on the other seven ways to call this function.

Under Unix, you can call other programs from the current one by combining **exec** with two other functions, **fork** and **wait**. The former makes a copy of the current process, the latter waits for a child process to finish.

Example: Using the os module to run another program (Unix)

```
# File:os-exec-example-2.py

import os
import sys

def run(program, *args):
    pid = os.fork()
    if not pid:
        os.execvp(program, (program,) +  args)
    return os.wait()[0]

run("python", "hello.py")

print "goodbye"
```

```
hello again, and welcome to the show
goodbye
```

The **fork** returns zero in the new process (the return from **fork** is the first thing that happens in that process!), and a non-zero process identifier in the original process. Or in other words, "**not pid**" is true only if we're in the new process.

fork and **wait** are not available on Windows, but you can use the **spawn** function instead. Unfortunately, there's no standard version of **spawn** that searches for an executable along the path, so you have to do that yourself:

Example: Using the os module to run another program (Windows)

```
# File:os-spawn-example-1.py

import os
import string

def run(program, *args):
    # find executable
    for path in string.split(os.environ["PATH"], os.pathsep):
        file = os.path.join(path, program) + ".exe"
        try:
            return os.spawnv(os.P_WAIT, file, (file,) + args)
        except os.error:
            pass
    raise os.error, "cannot find executable"

run("python", "hello.py")
```

```
print "goodbye"
```

```
hello again, and welcome to the show
goodbye
```

You can also use **spawn** to run other programs in the background. The following example adds an optional **mode** argument to the **run** function; when set to **os.P_NOWAIT**, the script doesn't wait for the other program to finish.

The default flag value **os.P_WAIT** tells **spawn** to wait until the new process is finished. Other flags include **os.P_OVERLAY** which makes **spawn** behave like **exec**, and **os.P_DETACH** which runs the new process in the background, detached from both console and keyboard.

Example: Using the os module to run another program in the background (Windows)

```
# File:os-spawn-example-2.py

import os
import string

def run(program, *args, **kw):
    # find executable
    mode = kw.get("mode", os.P_WAIT)
    for path in string.split(os.environ["PATH"], os.pathsep):
        file = os.path.join(path, program) + ".exe"
        try:
            return os.spawnv(mode, file, (file,) + args)
        except os.error:
            pass
    raise os.error, "cannot find executable"

run("python", "hello.py", mode=os.P_NOWAIT)
print "goodbye"
```

```
goodbye
hello again, and welcome to the show
```

The following example provides a **spawn** method that works on either platform:

Example: Using either spawn or fork/exec to run another program

```
# File:os-spawn-example-3.py

import os
import string

if os.name in ("nt", "dos"):
    exefile = ".exe"
else:
    exefile = ""

def spawn(program, *args):
    try:
        # check if the os module provides a shortcut
        return os.spawnvp(program, (program,) + args)
    except AttributeError:
        pass
    try:
        spawnv = os.spawnv
    except AttributeError:
        # assume it's unix
        pid = os.fork()
        if not pid:
            os.execvp(program, (program,) + args)
        return os.wait()[0]
    else:
        # got spawnv but no spawnp: go look for an executable
        for path in string.split(os.environ["PATH"], os.pathsep):
            file = os.path.join(path, program) + exefile
            try:
                return spawnv(os.P_WAIT, file, (file,) + args)
            except os.error:
                pass
        raise IOError, "cannot find executable"

#
# try it out!

spawn("python", "hello.py")

print "goodbye"
```

```
hello again, and welcome to the show
goodbye
```

The above example first attempts to call a function named **spawnvp**. If that doesn't exist (it doesn't, in 2.0 and earlier), the function looks for a function named **spawnv** and searches the path all by itself. As a last resort, it falls back on **exec** and **fork**.

Working with daemon processes

On Unix, **fork** can also be used to turn the current process into a background process (a "daemon"). Basically, all you need to do is to fork off a copy of the current process, and terminate the original process:

Example: Using the os module to run as daemon (Unix)

```
# File:os-example-14.py

import os
import time

pid = os.fork()
if pid:
    os._exit(0) # kill original

print "daemon started"
time.sleep(10)
print "daemon terminated"
```

However, it takes a bit more work to create a real daemon. First, call **setpgrp** to make the new process a "process group leader". Otherwise, signals sent to a (by that time) unrelated process group might cause problems in your daemon:

```
os.setpgrp()
```

It's also a good idea to remove the user mode mask, to make sure files created by the daemon actually gets the mode flags specified by the program:

```
os.umask(0)
```

Then, you should redirect the stdout/stderr files, instead of just closing them. If you don't do this, you may get unexpected exceptions the day some of your code tries to write something to the console via stdout or stderr.

```
class NullDevice:
    def write(self, s):
        pass

sys.stdin.close()
sys.stdout = NullDevice()
sys.stderr = NullDevice()
```

In other words, while Python's **print** and C's **printf/fprintf** won't crash your program if the devices have been disconnected, **sys.stdout.write()** happily throws an **IOError** exception when the application runs as a daemon. But your program works just fine when running in the foreground...

By the way, the **_exit** function used in the examples above terminates the current process. In contrast to **sys.exit**, this works also if the caller happens to catch the **SystemExit** exception:

Example: Using the os module to exit the current process

```
# File:os-example-9.py

import os
import sys

try:
    sys.exit(1)
except SystemExit, value:
    print "caught exit(%s)" % value

try:
    os._exit(2)
except SystemExit, value:
    print "caught exit(%s)" % value

print "bye!"
```

```
caught exit(1)
```

The os.path module

This module contains functions that deal with long filenames (path names) in various ways. To use this module, import the **os** module, and access this module as **os.path**.

Working with file names

This module contains a number of functions that deal with long filenames in a platform independent way. In other words, without having to deal with forward and backward slashes, colons, and whatnot.

Example: Using the os.path module to handle filename

```
# File:os-path-example-1.py

import os

filename = "my/little/pony"

print "using", os.name, "..."
print "split", "=>", os.path.split(filename)
print "splitext", "=>", os.path.splitext(filename)
print "dirname", "=>", os.path.dirname(filename)
print "basename", "=>", os.path.basename(filename)
print "join", "=>", os.path.join(os.path.dirname(filename),
                                 os.path.basename(filename))
```

```
using nt ...
split => ('my/little', 'pony')
splitext => ('my/little/pony', '')
dirname => my/little
basename => pony
join => my/little\pony
```

Note that **split** only splits off a single item.

This module also contains a number of functions that allow you to quickly figure out what a filename represents:

Example: Using the os.path module to check what a filename represents

```
# File:os-path-example-2.py

import os

FILES = (
    os.curdir,
    "/",
    "file",
    "/file",
    "samples",
    "samples/sample.jpg",
    "directory/file",
    "../directory/file",
    "/directory/file"
    )

for file in FILES:
    print file, "=>",
    if os.path.exists(file):
        print "EXISTS",
    if os.path.isabs(file):
        print "ISABS",
    if os.path.isdir(file):
        print "ISDIR",
    if os.path.isfile(file):
        print "ISFILE",
    if os.path.islink(file):
        print "ISLINK",
    if os.path.ismount(file):
        print "ISMOUNT",
    print
```

```
. => EXISTS ISDIR
/ => EXISTS ISABS ISDIR ISMOUNT
file =>
/file => ISABS
samples => EXISTS ISDIR
samples/sample.jpg => EXISTS ISFILE
directory/file =>
../directory/file =>
/directory/file => ISABS
```

The **expanduser** function treats a user name shortcut in the same way as most modern Unix shells (it doesn't work well on Windows).

Example: Using the os.path module to insert the user name into a filename

```
# File:os-path-expanduser-example-1.py

import os

print os.path.expanduser("~/.pythonrc")
```

```
/home/effbot/.pythonrc
```

The **expandvars** function inserts environment variables into a filename:

Example: Using the os.path module to insert variables into a filename

```
# File:os-path-expandvars-example-1.py

import os

os.environ["USER"] = "user"

print os.path.expandvars("/home/$USER/config")
print os.path.expandvars("$USER/folders")
```

```
/home/user/config
user/folders
```

Traversing a file system

The **walk** function helps you find all files in a directory tree. It takes a directory name, a callback function, and a data object that is passed on to the callback.

Example: Using the os.path module to traverse a file system

```
# File:os-path-walk-example-1.py

import os

def callback(arg, directory, files):
    for file in files:
        print os.path.join(directory, file), repr(arg)

os.path.walk(".", callback, "secret message")
```

```
./aifc-example-1.py 'secret message'
./anydbm-example-1.py 'secret message'
./array-example-1.py 'secret message'
...
./samples 'secret message'
./samples/sample.jpg 'secret message'
./samples/sample.txt 'secret message'
./samples/sample.zip 'secret message'
./samples/articles 'secret message'
./samples/articles/article-1.txt 'secret message'
./samples/articles/article-2.txt 'secret message'
...
```

The **walk** function has a somewhat obscure user interface (maybe it's just me, but I can never remember the order of the arguments). The **index** function in the next example returns a list of filenames instead, which lets you use a straightforward **for-in** loop to process the files:

Example: Using os.listdir to traverse a file system

```
# File:os-path-walk-example-2.py

import os

def index(directory):
    # like os.listdir, but traverses directory trees
    stack = [directory]
    files = []
    while stack:
        directory = stack.pop()
        for file in os.listdir(directory):
            fullname = os.path.join(directory, file)
            files.append(fullname)
            if os.path.isdir(fullname) and not os.path.islink(fullname):
                stack.append(fullname)
    return files

for file in index("."):
    print file
```

```
.\aifc-example-1.py
.\anydbm-example-1.py
.\array-example-1.py
...
```

If you don't want to list all files (for performance or memory reasons), the following example uses a different approach. Here, the **DirectoryWalker** class behaves like a sequence object, returning one file at a time:

Example: Using a directory walker to traverse a file system

```
# File:os-path-walk-example-3.py

import os

class DirectoryWalker:
    # a forward iterator that traverses a directory tree

    def __init__(self, directory):
        self.stack = [directory]
        self.files = []
        self.index = 0

    def __getitem__(self, index):
        while 1:
            try:
                file = self.files[self.index]
                self.index = self.index + 1
            except IndexError:
                # pop next directory from stack
                self.directory = self.stack.pop()
                self.files = os.listdir(self.directory)
                self.index = 0
            else:
                # got a filename
                fullname = os.path.join(self.directory, file)
                if os.path.isdir(fullname) and not os.path.islink(fullname):
                    self.stack.append(fullname)
                return fullname

for file in DirectoryWalker("."):
    print file
```

```
.\aifc-example-1.py
.\anydbm-example-1.py
.\array-example-1.py
...
```

Note that this class doesn't check the index passed to the **__getitem__** method. This means that it won't do the right thing if you access the sequence members out of order.

Finally, if you're interested in the file sizes or timestamps, here's a version of the class that returns both the filename and the tuple returned from **os.stat**. This version saves one or two **stat** calls for each file (both **os.path.isdir** and **os.path.islink** uses **stat**), and runs quite a bit faster on some platforms.

Example: Using a directory walker to traverse a file system, returning both the filename and additional file information

```
# File:os-path-walk-example-4.py

import os, stat

class DirectoryStatWalker:
    # a forward iterator that traverses a directory tree, and
    # returns the filename and additional file information

    def __init__(self, directory):
        self.stack = [directory]
        self.files = []
        self.index = 0

    def __getitem__(self, index):
        while 1:
            try:
                file = self.files[self.index]
                self.index = self.index + 1
            except IndexError:
                # pop next directory from stack
                self.directory = self.stack.pop()
                self.files = os.listdir(self.directory)
                self.index = 0
            else:
                # got a filename
                fullname = os.path.join(self.directory, file)
                st = os.stat(fullname)
                mode = st[stat.ST_MODE]
                if stat.S_ISDIR(mode) and not stat.S_ISLNK(mode):
                    self.stack.append(fullname)
                return fullname, st

for file, st in DirectoryStatWalker("."):
    print file, st[stat.ST_SIZE]
```

```
.\aifc-example-1.py 336
.\anydbm-example-1.py 244
.\array-example-1.py 526
```

The stat module

This module contains a number of constants and test functions that can be used with the **os.stat** function.

Example: Using the stat module

```
# File:stat-example-1.py

import stat
import os, time

st = os.stat("samples/sample.txt")

print "mode", "=>", oct(stat.S_IMODE(st[stat.ST_MODE]))

print "type", "=>",
if stat.S_ISDIR(st[stat.ST_MODE]):
    print "DIRECTORY",
if stat.S_ISREG(st[stat.ST_MODE]):
    print "REGULAR",
if stat.S_ISLNK(st[stat.ST_MODE]):
    print "LINK",
print

print "size", "=>", st[stat.ST_SIZE]

print "last accessed", "=>", time.ctime(st[stat.ST_ATIME])
print "last modified", "=>", time.ctime(st[stat.ST_MTIME])
print "inode changed", "=>", time.ctime(st[stat.ST_CTIME])
```

```
mode => 0664
type => REGULAR
size => 305
last accessed => Sun Oct 10 22:12:30 1999
last modified => Sun Oct 10 18:39:37 1999
inode changed => Sun Oct 10 15:26:38 1999
```

The string module

This module contains a number of functions to process standard Python strings.

Example: Using the string module

```
# File:string-example-1.py

import string

text = "Monty Python's Flying Circus"

print "upper", "=>", string.upper(text)
print "lower", "=>", string.lower(text)
print "split", "=>", string.split(text)
print "join", "=>", string.join(string.split(text), "+")
print "replace", "=>", string.replace(text, "Python", "Java")
print "find", "=>", string.find(text, "Python"), string.find(text, "Java")
print "count", "=>", string.count(text, "n")
```

```
upper => MONTY PYTHON'S FLYING CIRCUS
lower => monty python's flying circus
split => ['Monty', "Python's", 'Flying', 'Circus']
join => Monty+Python's+Flying+Circus
replace => Monty Java's Flying Circus
find => 6 -1
count => 3
```

In Python 1.5.2 and earlier, this module uses functions from the **strop** implementation module where possible.

In Python 1.6 and later, most string operations are made available as string methods as well, and many functions in the **string** module are simply wrapper functions that call the corresponding string method.

Example: Using string methods instead of string module functions (Python 1.6 and later)

```
# File:string-example-2.py

text = "Monty Python's Flying Circus"

print "upper", "=>", text.upper()
print "lower", "=>", text.lower()
print "split", "=>", text.split()
print "join", "=>", "+".join(text.split())
print "replace", "=>", text.replace("Python", "Perl")
print "find", "=>", text.find("Python"), text.find("Perl")
print "count", "=>", text.count("n")
```

```
upper => MONTY PYTHON'S FLYING CIRCUS
lower => monty python's flying circus
split => ['Monty', "Python's", 'Flying', 'Circus']
join => Monty+Python's+Flying+Circus
replace => Monty Perl's Flying Circus
find => 6 -1
count => 3
```

In addition to the string manipulation stuff, the **string** module also contains a number of functions which convert strings to other types:

Example: Using the string module to convert strings to numbers

```
# File:string-example-3.py

import string

print int("4711"),
print string.atoi("4711"),
print string.atoi("11147", 8), # octal
print string.atoi("1267", 16), # hexadecimal
print string.atoi("3mv", 36) # whatever...

print string.atoi("4711", 0),
print string.atoi("04711", 0),
print string.atoi("0x4711", 0)

print float("4711"),
print string.atof("1"),
print string.atof("1.23e5")
```

```
4711 4711 4711 4711 4711
4711 2505 18193
4711.0 1.0 123000.0
```

In most cases (especially if you're using 1.6 or later), you can use the **int** and **float** functions instead of their **string** module counterparts.

The **atoi** function takes an optional second argument, which specifices the number base. If the base is zero, the function looks at the first few characters before attempting to interpret the value: if "0x", the base is set to 16 (hexadecimal), and if "0", the base is set to 8 (octal). The default is base 10 (decimal), just as if you hadn't provided an extra argument.

In 1.6 and later, the **int** also accepts a second argument, just like **atoi**. But unlike the string versions, **int** and **float** also accepts Unicode strings.

The re module

> *"Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems"*
>
> *Jamie Zawinski, on comp.lang.emacs*

This module provides a set of powerful regular expression facilities. A regular expression is a string pattern written in a compact (and quite cryptic) syntax, and this module allows you to quickly check whether a given string *matches* a given pattern (using the **match** function), or *contains* such a pattern (using the **search** function).

The **match** function attempts to match a pattern against the beginning of the given string. If the pattern matches anything at all (including an empty string, if the pattern allows that!), **match** returns a *match object*. The **group** method can be used to find out what matched.

Example: Using the re module to match strings

```
# File:re-example-1.py

import re

text = "The Attila the Hun Show"

# a single character
m = re.match(".", text)
if m: print repr("."), "=>", repr(m.group(0))

# any string of characters
m = re.match(".*", text)
if m: print repr(".*"), "=>", repr(m.group(0))

# a string of letters (at least one)
m = re.match("\w+", text)
if m: print repr("\w+"), "=>", repr(m.group(0))

# a string of digits
m = re.match("\d+", text)
if m: print repr("\d+"), "=>", repr(m.group(0))
```

```
'.' => 'T'
'.*' => 'The Attila the Hun Show'
'\\w+' => 'The'
```

You can use parentheses to mark regions in the pattern. If the pattern matched, the **group** method can be used to extract the contents of these regions. **group(1)** returns the contents of the first group, **group(2)** the contents of the second, etc. If you pass several group numbers to the **group** function, it returns a tuple.

Example: Using the re module to extract matching substrings

```
# File:re-example-2.py

import re

text ="10/15/99"

m = re.match("(\d{2})/(\d{2})/(\d{2,4})", text)
if m:
    print m.group(1, 2, 3)
```

```
('10', '15', '99')
```

The **search** function searches for the pattern inside the string. It basically tries the pattern at every possible characters position, starting from the left, and returns a match object as soon it has found a match. If the pattern doesn't match anywhere, it returns **None**.

Example: Using the re module to search for substrings

```
# File:re-example-3.py

import re

text = "Example 3: There is 1 date 10/25/95 in here!"

m = re.search("(\d{1,2})/(\d{1,2})/(\d{2,4})", text)

print m.group(1), m.group(2), m.group(3)

month, day, year = m.group(1, 2, 3)
print month, day, year

date = m.group(0)
print date
```

```
10 25 95
10 25 95
10/25/95
```

The **sub** function can be used to replace patterns with another string.

Example: Using the re module to replace substrings

```
# File:re-example-4.py

import re

text = "you're no fun anymore..."

# literal replace (string.replace is faster)
print re.sub("fun", "entertaining", text)

# collapse all non-letter sequences to a single dash
print re.sub("[^\w]+", "-", text)

# convert all words to beeps
print re.sub("\S+", "-BEEP-", text)
```

```
you're no entertaining anymore...
you-re-no-fun-anymore-
-BEEP- -BEEP- -BEEP- -BEEP-
```

You can also use **sub** to replace patterns via a callback function. The following example also shows how to pre-compile patterns.

Example: Using the re module to replace substrings via a callback

```
# File:re-example-5.py

import re
import string

text = "a line of text\\012another line of text\\012etc..."

def octal(match):
    # replace octal code with corresponding ASCII character
    return chr(string.atoi(match.group(1), 8))

octal_pattern = re.compile(r"\\(\d\d\d)")

print text
print octal_pattern.sub(octal, text)
```

```
a line of text\012another line of text\012etc...
a line of text
another line of text
etc...
```

If you don't compile, the **re** module caches compiled versions for you, so you usually don't have to compile regular expressions in small scripts. In Python 1.5.2, the cache holds 20 patterns. In 2.0, the cache size has been increased to 100 patterns.

Finally, here's an example that shows you how to match a string against a list of patterns. The list of patterns are combined into a single pattern, and pre-compiled to save time.

Example: Using the re module to match against one of many patterns

```
# File:re-example-6.py

import re, string

def combined_pattern(patterns):
    p = re.compile(
        string.join(map(lambda x: "("+x+")", patterns), "|")
        )
    def fixup(v, m=p.match, r=range(0,len(patterns))):
        try:
            regs = m(v).regs
        except AttributeError:
            return None # no match, so m.regs will fail
        else:
            for i in r:
                if regs[i+1] != (-1, -1):
                    return i
    return fixup

#
# try it out!

patterns = [
    r"\d+",
    r"abc\d{2,4}",
    r"p\w+"
]

p = combined_pattern(patterns)

print p("129391")
print p("abc800")
print p("abc1600")
print p("python")
print p("perl")
print p("tcl")
```

```
0
1
1
2
2
None
```

The math module

This module implements a number of mathematical operations for floating point numbers. The functions are generally thin wrappers around the platform C library functions of the same name, so results may vary slightly across platforms in normal cases, or vary a lot in exceptional cases.

Example: Using the math module

```
# File:math-example-1.py

import math

print "e", "=>", math.e
print "pi", "=>", math.pi
print "hypot", "=>", math.hypot(3.0, 4.0)

# and many others...
```

```
e => 2.71828182846
pi => 3.14159265359
hypot => 5.0
```

See the *Python Library Reference* for a full list of functions.

The cmath module

This module contains a number of mathematical operations for complex numbers.

Example: Using the cmath module

```
# File:cmath-example-1.py

import cmath

print "pi", "=>", cmath.pi
print "sqrt(-1)", "=>", cmath.sqrt(-1)
```

```
pi => 3.14159265359
sqrt(-1) => 1j
```

See the *Python Library Reference* for a full list of functions.

The operator module

This module provides a "functional" interface to the standard operators in Python. The functions in this module can be used instead of some **lambda** constructs, when processing data with functions like **map** and **filter**.

They are also quite popular among people who like to write obscure code, for obvious reasons.

Example: Using the operator module

```
# File:operator-example-1.py

import operator

sequence = 1, 2, 4

print "add", "=>", reduce(operator.add, sequence)
print "sub", "=>", reduce(operator.sub, sequence)
print "mul", "=>", reduce(operator.mul, sequence)
print "concat", "=>", operator.concat("spam", "egg")
print "repeat", "=>", operator.repeat("spam", 5)
print "getitem", "=>", operator.getitem(sequence, 2)
print "indexOf", "=>", operator.indexOf(sequence, 2)
print "sequenceIncludes", "=>", operator.sequenceIncludes(sequence, 3)
```

```
add => 7
sub => -5
mul => 8
concat => spamegg
repeat => spamspamspamspamspam
getitem => 4
indexOf => 1
sequenceIncludes => 0
```

The module also contains a few functions which can be used to check object types:

Example: Using the operator module for type checking

```
# File:operator-example-2.py

import operator
import UserList

def dump(data):
    print type(data), "=>",
    if operator.isCallable(data):
        print "CALLABLE",
    if operator.isMappingType(data):
        print "MAPPING",
    if operator.isNumberType(data):
        print "NUMBER",
    if operator.isSequenceType(data):
        print "SEQUENCE",
    print

dump(0)
dump("string")
dump("string"[0])
dump([1, 2, 3])
dump((1, 2, 3))
dump({"a": 1})
dump(len) # function
dump(UserList) # module
dump(UserList.UserList) # class
dump(UserList.UserList()) # instance
```

```
<type 'int'> => NUMBER
<type 'string'> => SEQUENCE
<type 'string'> => SEQUENCE
<type 'list'> => SEQUENCE
<type 'tuple'> => SEQUENCE
<type 'dictionary'> => MAPPING
<type 'builtin_function_or_method'> => CALLABLE
<type 'module'> =>
<type 'class'> => CALLABLE
<type 'instance'> => MAPPING NUMBER SEQUENCE
```

Note that the operator module doesn't handle object instances in a sane fashion. In other words, be careful when you use the **isNumberType**, **isMappingType**, and **isSequenceType** functions. It's easy to make your code less flexible than it has to be.

Also note that a string sequence member (a character) is also a sequence. If you're writing a recursive function that uses **isSequenceType** to traverse an object tree, you better not pass it an ordinary string (or anything containing one).

The copy module

This module contains two functions which are used to copy objects.

copy(object) -> object creates a "shallow" copy of the given object. In this context, shallow means that the object itself is copied, but if the object is a container, the members will still refer to the original member objects.

Example: Using the copy module to copy objects

```
# File:copy-example-1.py

import copy

a = [[1],[2],[3]]
b = copy.copy(a)

print "before", "=>"
print a
print b

# modify original
a[0][0] = 0
a[1] = None

print "after", "=>"
print a
print b
```

```
before =>
[[1], [2], [3]]
[[1], [2], [3]]
after =>
[[0], None, [3]]
[[0], [2], [3]]
```

Note that you can make shallow copies of lists using the [:] syntax (a full slice), and make copies of dictionaries using the **copy** method.

In contrast, **deepcopy(object) -> object** creates a "deep" copy of the given object. If the object is a container, all members are copied as well, recursively.

Example: Using the copy module to copy collections

```
# File:copy-example-2.py

import copy

a = [[1],[2],[3]]
b = copy.deepcopy(a)

print "before", "=>"
print a
print b

# modify original
a[0][0] = 0
a[1] = None

print "after", "=>"
print a
print b
```

```
before =>
[[1], [2], [3]]
[[1], [2], [3]]
after =>
[[0], None, [3]]
[[1], [2], [3]]
```

The sys module

This module provides a number of functions and variables that can be used to manipulate different parts of the Python runtime environment.

Working with command-line arguments

The **argv** list contains the arguments passed to the script, when the interpreter was started. The first item contains the name of the script itself.

Example: Using the sys module to get script arguments

```
# File:sys-argv-example-1.py

import sys

print "script name is", sys.argv[0]

if len(sys.argv) > 1:
    print "there are", len(sys.argv)-1, "arguments:"
    for arg in sys.argv[1:]:
        print arg
else:
    print "there are no arguments!"
```

```
script name is sys-argv-example-1.py
there are no arguments!
```

If you read the script from standard input (like "**python < sys-argv-example-1.py**"), the script name is set to an empty string. If you pass in the program as a string (using the **-c** option), the script name is set to "**-c**"

Working with modules

The **path** list contains a list of directory names where Python looks for extension modules (Python source modules, compiled modules, or binary extensions). When you start Python, this list is initialized from a mixture of built-in rules, the contents of the **PYTHONPATH** environment variable, and the registry contents (on Windows). But since it's an ordinary list, you can also manipulate it from within the program:

Example: Using the sys module to manipulate the module search path

```
# File:sys-path-example-1.py

import sys

print "path has", len(sys.path), "members"

# add the sample directory to the path
sys.path.insert(0, "samples")
import sample

# nuke the path
sys.path = []
import random # oops!
```

```
path has 7 members
this is the sample module!
Traceback (innermost last):
  File "sys-path-example-1.py", line 11, in ?
    import random # oops!
ImportError: No module named random
```

The **builtin_module_names** list contains the names of all modules built into the Python interpreter.

Example: Using the sys module to find built-in modules

```
# File:sys-builtin-module-names-example-1.py

import sys

def dump(module):
    print module, "=>",
    if module in sys.builtin_module_names:
        print "<BUILTIN>"
    else:
        module = __import__(module)
        print module.__file__

dump("os")
dump("sys")
dump("string")
dump("strop")
dump("zlib")
```

```
os => C:\python\lib\os.pyc
sys => <BUILTIN>
string => C:\python\lib\string.pyc
strop => <BUILTIN>
zlib => C:\python\zlib.pyd
```

The **modules** dictionary contains all loaded modules. The **import** statement checks this dictionary before it actually loads something from disk.

As you can see from the following example, Python loads quite a bunch of modules before it hands control over to your script.

Example: Using the sys module to find imported modules

```
# File:sys-modules-example-1.py

import sys

print sys.modules.keys()
```

```
['os.path', 'os', 'exceptions', '__main__', 'ntpath', 'strop', 'nt',
'sys', '__builtin__', 'site', 'signal', 'UserDict', 'string', 'stat']
```

Working with reference counts

The **getrefcount** function returns the reference count for a given object — that is, the number of places where this variable is used. Python keeps track of this value, and when it drops to zero, the object is destroyed.

Example: Using the sys module to find the reference count

```
# File:sys-getrefcount-example-1.py

import sys

variable = 1234

print sys.getrefcount(0)
print sys.getrefcount(variable)
print sys.getrefcount(None)
```

```
50
3
192
```

Note that this value is always larger than the actual count, since the function itself hangs on to the object while determining the value.

Checking the host platform

The **platform** variable contains the name of the host platform:

Example: Using the sys module to find the current platform

```
# File:sys-platform-example-1.py

import sys

#
# emulate "import os.path" (sort of)...

if sys.platform == "win32":
    import ntpath
    pathmodule = ntpath
elif sys.platform == "mac":
    import macpath
    pathmodule = macpath
else:
    # assume it's a posix platform
    import posixpath
    pathmodule = posixpath

print pathmodule
```

Typical platform names are **win32** for Windows 9X/NT and **mac** for Macintosh. For Unix systems,

the platform name is usually derived from the output of the "**uname -r**" command, such as **irix6**, **linux2**, or **sunos5** (Solaris).

Tracing the program

The **setprofiler** function allows you to install a profiling function. This is called every time a function or method is called, at every return (explicit or implied), and for each exception:

Example: Using the sys module to install a profiler function

```
# File:sys-setprofiler-example-1.py

import sys

def test(n):
    j = 0
    for i in range(n):
        j = j + i
    return n

def profiler(frame, event, arg):
    print event, frame.f_code.co_name, frame.f_lineno, "->", arg

# profiler is activated on the next call, return, or exception
sys.setprofile(profiler)

# profile this function call
test(1)

# disable profiler
sys.setprofile(None)

# don't profile this call
test(2)
```

```
call test 3 -> None
return test 7 -> 1
```

The **profile** module provides a complete profiler framework, based on this function.

The **settrace** function is similar, but the trace function is called for each new line:

Example: Using the sys module to install a trace function

```
# File:sys-settrace-example-1.py

import sys

def test(n):
    j = 0
    for i in range(n):
        j = j + i
    return n

def tracer(frame, event, arg):
    print event, frame.f_code.co_name, frame.f_lineno, "->", arg
    return tracer

# tracer is activated on the next call, return, or exception
sys.settrace(tracer)

# trace this function call
test(1)

# disable tracing
sys.settrace(None)

# don't trace this call
test(2)
```

```
call test 3 -> None
line test 3 -> None
line test 4 -> None
line test 5 -> None
line test 5 -> None
line test 6 -> None
line test 5 -> None
line test 7 -> None
return test 7 -> 1
```

The **pdb** module provides a complete debugger framework, based on the tracing facilities offered by this function.

Working with standard input and output

The **stdin**, **stdout** and **stderr** variables contain stream objects corresponding to the standard I/O streams. You can access them directly if you need better control over the output than **print** can give you. You can also *replace* them, if you want to redirect output and input to some other device, or process them in some non-standard way:

Example: Using the sys module to redirect output

```
# File:sys-stdout-example-1.py

import sys
import string

class Redirect:

    def __init__(self, stdout):
        self.stdout = stdout

    def write(self, s):
        self.stdout.write(string.lower(s))

# redirect standard output (including the print statement)
old_stdout = sys.stdout
sys.stdout = Redirect(sys.stdout)

print "HEJA SVERIGE",
print "FRISKT HUMÖR"

# restore standard output
sys.stdout = old_stdout

print "MÅÅÅÅL!"
```

```
heja sverige friskt humör
MÅÅÅÅL!
```

All it takes to redirect output is an object that implements the **write** method.

(Unless it's a C type instance, that is: Python uses an integer attribute called **softspace** to control spacing, and adds it to the object if it isn't there. You don't have to bother if you're using Python objects, but if you need to redirect to a C type, you should make sure that type supports the **softspace** attribute.)

Exiting the program

When you reach the end of the main program, the interpreter is automatically terminated. If you need to exit in midflight, you can call the **sys.exit** function instead. This function takes an optional integer value, which is returned to the calling program.

Example: Using the sys module to exit the program

```
# File:sys-exit-example-1.py

import sys

print "hello"

sys.exit(1)

print "there"
```

```
hello
```

It may not be obvious, but **sys.exit** doesn't exit at once. Instead, it raises a **SystemExit** exception. This means that you can trap calls to **sys.exit** in your main program:

Example: Catching the sys.exit call

```
# File:sys-exit-example-2.py

import sys

print "hello"

try:
    sys.exit(1)
except SystemExit:
    pass

print "there"
```

```
hello
there
```

If you want to clean things up after you, you can install an "exit handler", which is a function that is automatically called on the way out.

Example: Catching the sys.exit call

```
# File:sys-exitfunc-example-1.py

import sys

def exitfunc():
    print "world"

sys.exitfunc = exitfunc

print "hello"
sys.exit(1)
print "there" # never printed
```

```
hello
world
```

In Python 2.0, you can use the **atexit** module to register more than one exit handler.

The atexit module

(2.0 and later) This module allows you to register one or more functions that are called when the interpreter is terminated.

To register a function, simply call the **register** function. You can also add one or more extra arguments, which are passed as arguments to the exit function.

Example: Using the atexit module

```
# File:atexit-example-1.py

import atexit

def exit(*args):
    print "exit", args

# register three exit handlers
atexit.register(exit)
atexit.register(exit, 1)
atexit.register(exit, "hello", "world")
```

```
exit ('hello', 'world')
exit (1,)
exit ()
```

This module is a straightforward wrapper for the **sys.exitfunc** hook.

The time module

This module provides a number of functions to deal with dates and the time within a day. It's a thin layer on top of the C runtime library.

A given date and time can either be represented as a floating point value (the number of seconds since a reference date, usually January 1st, 1970), or as a time tuple.

Getting the current time

Example: Using the time module to get the current time

```
# File:time-example-1.py

import time

now = time.time()

print now, "seconds since", time.gmtime(0)[:6]
print
print "or in other words:"
print "- local time:", time.localtime(now)
print "- utc:", time.gmtime(now)
```

```
937758359.77 seconds since (1970, 1, 1, 0, 0, 0)

or in other words:
- local time: (1999, 9, 19, 18, 25, 59, 6, 262, 1)
- utc: (1999, 9, 19, 16, 25, 59, 6, 262, 0)
```

The tuple returned by **localtime** and **gmtime** contains (year, month, day, hour, minute, second, day of week, day of year, daylight savings flag), where the year number is four digits, the day of week begins with 0 for Monday, and January 1st is day number 1.

Converting time values to strings

You can of course use standard string formatting operators to convert a time tuple to a string, but the **time** module also provides a number of standard conversion functions:

Example: Using the time module to format dates and times

```
# File:time-example-2.py

import time

now = time.localtime(time.time())

print time.asctime(now)
print time.strftime("%y/%m/%d %H:%M", now)
print time.strftime("%a %b %d", now)
print time.strftime("%c", now)
print time.strftime("%I %p", now)
print time.strftime("%Y-%m-%d %H:%M:%S %Z", now)

# do it by hand...
year, month, day, hour, minute, second, weekday, yearday, daylight = now
print "%04d-%02d-%02d" % (year, month, day)
print "%02d:%02d:%02d" % (hour, minute, second)
print ("MON", "TUE", "WED", "THU", "FRI", "SAT", "SUN")[weekday], yearday
```

```
Sun Oct 10 21:39:24 1999
99/10/10 21:39
Sun Oct 10
Sun Oct 10 21:39:24 1999
09 PM
1999-10-10 21:39:24 CEST
1999-10-10
21:39:24
SUN 283
```

Converting strings to time values

On some platforms, the **time** module contains a **strptime** function, which is pretty much the opposite of **strftime**. Given a string and a pattern, it returns the corresponding time tuple:

Example: Using the time.strptime function to parse dates and times

```
# File:time-example-6.py

import time

# make sure we have a strptime function!
try:
    strptime = time.strptime
except AttributeError:
    from strptime import strptime

print strptime("31 Nov 00", "%d %b %y")
print strptime("1 Jan 70 1:30pm", "%d %b %y %I:%M%p")
```

The **time.strptime** function is currently only made available by Python if it's provided by the platform's C libraries. For platforms that don't have a standard implementation (this includes Windows), here's a partial replacement:

Example: A strptime implementation

```
# File:strptime.py

import re
import string

MONTHS = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
          "Sep", "Oct", "Nov", "Dec"]

SPEC = {
    # map formatting code to a regular expression fragment
    "%a": "(?P<weekday>[a-z]+)",
    "%A": "(?P<weekday>[a-z]+)",
    "%b": "(?P<month>[a-z]+)",
    "%B": "(?P<month>[a-z]+)",
    "%C": "(?P<century>\d\d?)",
    "%d": "(?P<day>\d\d?)",
    "%D": "(?P<month>\d\d?)/(?P<day>\d\d?)/(?P<year>\d\d)",
    "%e": "(?P<day>\d\d?)",
    "%h": "(?P<month>[a-z]+)",
    "%H": "(?P<hour>\d\d?)",
    "%I": "(?P<hour12>\d\d?)",
    "%j": "(?P<yearday>\d\d?\d?)",
    "%m": "(?P<month>\d\d?)",
    "%M": "(?P<minute>\d\d?)",
    "%p": "(?P<ampm12>am|pm)",
    "%R": "(?P<hour>\d\d?):(?P<minute>\d\d?)",
    "%S": "(?P<second>\d\d?)",
    "%T": "(?P<hour>\d\d?):(?P<minute>\d\d?):(?P<second>\d\d?)",
    "%U": "(?P<week>\d\d)",
    "%w": "(?P<weekday>\d)",
    "%W": "(?P<weekday>\d\d)",
    "%y": "(?P<year>\d\d)",
    "%Y": "(?P<year>\d\d\d\d)",
    "%%": "%"
}

class TimeParser:

    def __init__(self, format):
        # convert strptime format string to regular expression
        format = string.join(re.split("(?:\s|%t|%n)+", format))
        pattern = []
        try:
            for spec in re.findall("%\w|%%|.", format):
                if spec[0] == "%":
                    spec = SPEC[spec]
                pattern.append(spec)
        except KeyError:
            raise ValueError, "unknown specificer: %s" % spec
        self.pattern = re.compile("(?i)" + string.join(pattern, ""))
```

```
    def match(self, daytime):
        # match time string
        match = self.pattern.match(daytime)
        if not match:
            raise ValueError, "format mismatch"
        get = match.groupdict().get
        tm = [0] * 9
        # extract date elements
        y = get("year")
        if y:
            y = int(y)
            if y < 68:
                y = 2000 + y
            elif y < 100:
                y = 1900 + y
            tm[0] = y
        m = get("month")
        if m:
            if m in MONTHS:
                m = MONTHS.index(m) + 1
            tm[1] = int(m)
        d = get("day")
        if d: tm[2] = int(d)
        # extract time elements
        h = get("hour")
        if h:
            tm[3] = int(h)
        else:
            h = get("hour12")
            if h:
                h = int(h)
                if string.lower(get("ampm12", "")) == "pm":
                    h = h + 12
                tm[3] = h
        m = get("minute")
        if m: tm[4] = int(m)
        s = get("second")
        if s: tm[5] = int(s)
        # ignore weekday/yearday for now
        return tuple(tm)

def strptime(string, format="%a %b %d %H:%M:%S %Y"):
    return TimeParser(format).match(string)

if __name__ == "__main__":
    # try it out
    import time
    print strptime("2000-12-20 01:02:03", "%Y-%m-%d %H:%M:%S")
    print strptime(time.ctime(time.time()))
```

```
(2000, 12, 20, 1, 2, 3, 0, 0, 0)
(2000, 11, 15, 12, 30, 45, 0, 0, 0)
```

Converting time values

Converting a time tuple back to a time value is pretty easy, at least as long as we're talking about local time. Just pass the time tuple to the **mktime** function:

Example: Using the time module to convert a local time tuple to a time integer

```
# File:time-example-3.py

import time

t0 = time.time()
tm = time.localtime(t0)

print tm

print t0
print time.mktime(tm)
```

```
(1999, 9, 9, 0, 11, 8, 3, 252, 1)
936828668.16
936828668.0
```

Unfortunately, there's no function in the 1.5.2 standard library that converts UTC time tuples *back* to time values (neither in Python nor in the underlying C libraries). The following example provides a Python implementation of such a function, called **timegm**:

Example: Converting a UTC time tuple to a time integer

```
# File:time-example-4.py

import time

def _d(y, m, d, days=(0,31,59,90,120,151,181,212,243,273,304,334,365)):
    # map a date to the number of days from a reference point
    return (((y - 1901)*1461)/4 + days[m-1] + d +
        ((m > 2 and not y % 4 and (y % 100 or not y % 400)) and 1))

def timegm(tm, epoch=_d(1970,1,1)):
    year, month, day, h, m, s = tm[:6]
    assert year >= 1970
    assert 1 <= month <= 12
    return (_d(year, month, day) - epoch)*86400 + h*3600 + m*60 + s

t0 = time.time()
tm = time.gmtime(t0)

print tm

print t0
print timegm(tm)
```

```
(1999, 9, 8, 22, 12, 12, 2, 251, 0)
936828732.48
936828732
```

In 1.6 and later, a similar function is available in the **calendar** module, as **calendar.timegm**.

Timing things

The **time** module can be used to time the execution of a Python program. You can measure either "wall time" (real world time), or "process time" (the amount of CPU time the process has consumed, this far).

Example: Using the time module to benchmark an algorithm

```
# File:time-example-5.py

import time

def procedure():
    time.sleep(2.5)

# measure process time
t0 = time.clock()
procedure()
print time.clock() - t0, "seconds process time"

# measure wall time
t0 = time.time()
procedure()
print time.time() - t0, "seconds wall time"
```

```
0.0 seconds process time
2.50903499126 seconds wall time
```

Not all systems can measure the true process time. On such systems (including Windows), **clock** usually measures the wall time since the program was started.

Also see the **timing** module, which measures the wall time between two events.

The process time has limited precision. On many systems, it wraps around after just over 30 minutes.

The types module

This module contains type objects for all object types defined by the standard interpreter. All objects of the same type share a single type object, so you can use **is** to test if an object has a given type.

Example: Using the types module

```
# File:types-example-1.py

import types

def check(object):
    print object,
    if type(object) is types.IntType:
        print "INTEGER",
    if type(object) is types.FloatType:
        print "FLOAT",
    if type(object) is types.StringType:
        print "STRING",
    if type(object) is types.ClassType:
        print "CLASS",
    if type(object) is types.InstanceType:
        print "INSTANCE",
    print

check(0)
check(0.0)
check("0")

class A:
    pass

class B:
    pass

check(A)
check(B)

a = A()
b = B()

check(a)
check(b)
```

```
0 INTEGER
0.0 FLOAT
0 STRING
A CLASS
B CLASS
<A instance at 796960> INSTANCE
<B instance at 796990> INSTANCE
```

Note that all classes have the same type, and so do all instances. To test what class hierarchy a class or an instance belongs to, use the built-in **issubclass** and **isinstance** functions.

The **types** module destroys the current exception state when it is first imported. In other words, don't import it (or *any* module that imports it!) from within an exception handler.

The gc module

(Optional, 2.0 and later) This module provides an interface to the built-in cyclic garbage collector.

Python uses reference counting to keep track of when to get rid of objects; as soon as the last reference to an object goes away, the object is destroyed.

Starting with version 2.0, Python also provides a cyclic garbage collector, which runs at regular intervals. This collector looks for data structures that point to themselves, and does what it can to break the cycles.

You can use the **gc.collect** function to force full collection. This function returns the number of objects destroyed by the collector.

Example: Using the gc module to collect cyclic garbage

```
# File:gc-example-1.py

import gc

# create a simple object that links to itself
class Node:

    def __init__(self, name):
        self.name = name
        self.parent = None
        self.children = []

    def addchild(self, node):
        node.parent = self
        self.children.append(node)

    def __repr__(self):
        return "<Node %s at %x>" % (repr(self.name), id(self))

# set up a self-referencing structure
root = Node("monty")

root.addchild(Node("eric"))
root.addchild(Node("john"))
root.addchild(Node("michael"))

# remove our only reference
del root

print gc.collect(), "unreachable objects"
print gc.collect(), "unreachable objects"
```

```
12 unreachable objects
0 unreachable objects
```

If you're sure that your program doesn't create any self-referencing data structures, you can use the **gc.disable** function to disable collection. After calling this function, Python 2.0 works exactly like 1.5.2 and earlier.