CHAPTER 10

Batteries Included

You now know most of the basic Python language. While the core language is powerful in itself, Python gives you more tools to play with. A standard installation includes a set of modules called the *standard library*. You have already seen some of them (`math` and `cmath`, which contain mathematical functions for real and complex numbers, for example), but there are many more. This chapter shows you a bit about how modules work, and how to explore them and learn what they have to offer. Then the chapter offers an overview of the standard library, focusing on a few selected useful modules.

Modules

You already know about making your own programs (or *scripts*) and executing them. You have also seen how you can fetch functions into your programs from external modules using `import`:

```
>>> import math
>>> math.sin(0)
0.0
```

Let's take a look at how you can write your own modules.

Modules Are Programs

Any Python program can be imported as a module. Let's say you have written the program in Listing 10-1 and stored it in a file called `hello.py` (the name is important).

Listing 10-1. *A Simple Module*

```
# hello.py
print "Hello, world!"
```

Where you save it is also important; in the next section you learn more about that, but for now let's say you save it in the directory `C:\python` (Windows) or `~/python` (UNIX/Mac OS X).

Then you can tell your interpreter where to look for the module by executing the following (using the Windows directory):

```
>>> import sys
>>> sys.path.append('c:/python')
```

■ **Tip** In UNIX, you cannot simply append the string `'~/python'` to `sys.path`. You must use the full path (such as `'/home/yourusername/python'`) or, if you want to automate it, use `sys.path.expanduser('~/python')`.

This simply tells the interpreter that it should look for modules in the directory `c:\python` in addition to the places it would normally look. After having done this, you can import your module (which is stored in the file `c:\python\hello.py`, remember?):

```
>>> import hello
Hello, world!
```

■ **Note** When you import a module, you may notice that a new file appears—in this case `c:\python\hello.pyc`. The file with the `.pyc` extension is a (platform-independent) processed ("compiled") Python file that has been translated to a format that Python can handle more efficiently. If you import the same module later, Python will import the `.pyc` file rather than the `.py` file, unless the `.py` file has changed; in that case, a new `.pyc` file is generated. Deleting the `.pyc` file does no harm (as long as there is an equivalent `.py` file available)—a new one is created when needed.

As you can see, the code in the module is executed when you import it. However, if you try to import it again, nothing happens:

```
>>> import hello
>>>
```

Why doesn't it work this time? Because modules aren't really meant to *do* things (such as printing text) when they're imported. They are mostly meant to *define* things, such as variables, functions, classes, and so on. And because you need to define things only once, importing a module several times has the same effect as importing it once.

WHY ONLY ONCE?

The import-only-once behavior is a substantial optimization in most cases, and it can be very important in one special case: if two modules import each other.

In many cases, you may write two modules that need to access functions and classes from each other to function properly. For example, you may have created two modules—`clientdb` and `billing`—containing code for a client database and a billing system, respectively. Your client database may contain calls to your billing system (for example, automatically sending a bill to a client every month), while the billing system probably needs to access functionality from your client database to do the billing correctly.

If each module could be imported several times, you would end up with a problem here. The module `clientdb` would import `billing`, which again imports `clientdb`, which . . . you get the picture. You get an endless loop of imports (endless recursion, remember?). However, because nothing happens the second time you import the module, the loop is broken.

If you *insist* on reloading your module, you can use the built-in function `reload`. It takes a single argument (the module you want to reload) and returns the reloaded module. This may be useful if you have made changes to your module and want those changes reflected in your program while it is running. To reload the simple `hello` module (containing only a `print` statement), I would use the following:

```
>>> hello = reload(hello)
Hello, world!
```

Here, I assume that `hello` has already been imported (once). By assigning the result of `reload` to `hello`, I have replaced the previous version with the reloaded one. As you can see from the printed greeting, I am really importing the module here.

If you've created an object `x` by instantiating the class `Foo` from the module `bar`, and you then reload `bar`, the object `x` refers to will not be re-created in any way. `x` will still be an instance of the old version of `Foo` (from the old version of `bar`). If, instead, you want `x` to be based on the new `Foo` from the reloaded module, you will need to create it anew.

Note that the `reload` function has disappeared in Python 3.0. While you can achieve similar functionality using `exec`, the best thing in most cases is simply to stay away from module reloading.

Modules Are Used to Define Things

So modules are executed the first time they are imported into your program. That seems sort of useful, but not very. What makes them worthwhile is that they (just like classes) keep their scope around afterward. That means that any classes or functions you define, and any variables you assign a value to, become attributes of the module. This may seem complicated, but in practice it is very simple.

Defining a Function in a Module

Let’s say you have written a module like the one in Listing 10-2 and stored it in a file called `hello2.py`. Also assume that you’ve put it in a place where the Python interpreter can find it, either using the `sys.path` trick from the previous section or the more conventional methods from the section “Making Your Modules Available,” which follows.

■ **Tip** If you make a program (which is meant to be executed, and not really used as a module) available in the same manner as other modules, you can actually execute it using the `-m` switch to the Python interpreter. Running the command `python -m progname args` will run the program `progname` with the command-line arguments `args`, provided that the file `progname.py` (note the suffix) is installed along with your other modules (that is, provided you have imported `progname`).

Listing 10-2. *A Simple Module Containing a Function*

```
# hello2.py
def hello():
    print "Hello, world!"
```

You can then import it like this:

```
>>> import hello2
```

The module is then executed, which means that the function `hello` is defined in the scope of the module, so you can access the function like this:

```
>>> hello2.hello()
Hello, world!
```

Any name defined in the global scope of the module will be available in the same manner.

Why would you want to do this? Why not just define everything in your main program? The primary reason is *code reuse*. If you put your code in a module, you can use it in more than one of your programs, which means that if you write a good client database and put it in a module called `clientdb`, you can use it when billing, when sending out spam (though I hope you won’t), and in any program that needs access to your client data. If you hadn’t put this in a separate module, you would need to rewrite the code in each one of these programs. So, remember: to make your code reusable, make it modular! (And, yes, this is definitely related to abstraction.)

Adding Test Code in a Module

Modules are used to define things such as functions and classes, but every once in a while (quite often, actually), it is useful to add some test code that checks whether things work as they should. For example, if you wanted to make sure that the hello function worked, you might rewrite the module hello2 into a new one, hello3, defined in Listing 10-3.

Listing 10-3. *A Simple Module with Some Problematic Test Code*

```
# hello3.py
def hello():
    print "Hello, world!"

# A test:
hello()
```

This seems reasonable—if you run this as a normal program, you will see that it works. However, if you import it as a module, to use the hello function in another program, the test code is executed, as in the first hello module in this chapter:

```
>>> import hello3
Hello, world!
>>> hello3.hello()
Hello, world!
```

This is not what you want. The key to avoiding it is "telling" the module whether it's being run as a program on its own or being imported into another program. To do that, you need the variable __name__:

```
>>> __name__
'__main__'
>>> hello3.__name__
'hello3'
```

As you can see, in the "main program" (including the interactive prompt of the interpreter), the variable __name__ has the value '__main__'. In an imported module, it is set to the name of that module. Therefore, you can make your module's test code more well behaved by putting in an if statement, as shown in Listing 10-4.

Listing 10-4. *A Module with Conditional Test Code*

```
# hello4.py

def hello():
    print "Hello, world!"

def test():
    hello()

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

If you run this as a program, the hello function is executed; if you import it, it behaves like a normal module:

```
>>> import hello4
>>> hello4.hello()
Hello, world!
```

As you can see, I've wrapped up the test code in a function called test. I could have put the code directly into the if statement; however, by putting it in a separate test function, you can test the module even if you have imported it into another program:

```
>>> hello4.test()
Hello, world!
```

■ **Note** If you write more thorough test code, it might be a good idea to put it in a separate program. See Chapter 16 for more on writing tests.

Making Your Modules Available

In the previous examples, I have altered sys.path, which contains a list of directories (as strings) in which the interpreter should look for modules. However, you don't want to do this in general. The ideal case would be for sys.path to contain the correct directory (the one containing your module) to begin with. There are two ways of doing this: put your module in the right place or tell the interpreter where to look. The following sections discuss these two solutions.

Putting Your Module in the Right Place

Putting your module in the right place (or, rather *a* right place, because there may be several possibilities) is quite easy. It's just a matter of finding out where the Python interpreter looks for modules and then putting your file there.

■ **Note** If the Python interpreter on the machine you're working on has been installed by an administrator and you do not have administrator permissions, you may not be able to save your module in any of the directories used by Python. You will then need to use the alternative solution: tell the interpreter where to look.

As you may remember, the list of directories (the so-called search path) can be found in the path variable in the sys module:

```
>>> import sys, pprint
>>> pprint.pprint(sys.path)
```

```
['C:\\Python25\\Lib\\idlelib',
 'C:\\WINDOWS\\system32\\python25.zip',
 'C:\\Python25',
 'C:\\Python25\\DLLs',
 'C:\\Python25\\lib',
 'C:\\Python25\\lib\\plat-win',
 'C:\\Python25\\lib\\lib-tk',
 'C:\\Python25\\lib\\site-packages']
```

■ **Tip** If you have a data structure that is too big to fit on one line, you can use the `pprint` function from the `pprint` module instead of the normal `print` statement. `pprint` is a pretty-printing function, which makes a more intelligent printout.

This is a relatively standard `path` for a Python 2.5 installation on Windows. You may not get the exact same result. The point is that each of these strings provides a place to put modules if you want your interpreter to find them. Even though all these will work, the `site-packages` directory is the best choice because it's meant for this sort of thing. Look through your `sys.path` and find your `site-packages` directory, and save the module from Listing 10-4 in it, but give it another name, such as `another_hello.py`. Then try the following:

```
>>> import another_hello
>>> another_hello.hello()
Hello, world!
```

As long as your module is located in a place like `site-packages`, all your programs will be able to import it.

Telling the Interpreter Where to Look

Putting your module in the correct place might not be the right solution for you for a number of reasons:

- You don't want to clutter the Python interpreter's directories with your own modules.
- You don't have permission to save files in the Python interpreter's directories.
- You would like to keep your modules somewhere else.

The bottom line is that if you place your modules somewhere else, you must tell the interpreter where to look. As you saw earlier, one way of doing this is to edit `sys.path`, but that is *not* a common way to do it. The standard method is to include your module directory (or directories) in the environment variable `PYTHONPATH`.

Depending on which operating system you are using, the contents of `PYTHONPATH` varies (see the sidebar "Environment Variables"), but basically it's just like `sys.path`—a list of directories.

ENVIRONMENT VARIABLES

Environment variables are not part of the Python interpreter—they're part of your operating system. Basically, they are like Python variables, but they are set outside the Python interpreter. To find out how to set them, you should consult your system documentation, but here are a few pointers.

In UNIX and Mac OS X, you will probably set environment variables in some shell file that is executed every time you log in. If you use a shell such as `bash`, the file is `.bashrc`, found in your home directory. Add the following to that file to add the directory `~/python` to your `PYTHONPATH`:

```
export PYTHONPATH=$PYTHONPATH:~/python
```

Note that multiple directories are separated by colons. Other shells may have a different syntax for this, so you should consult the relevant documentation.

In Windows, you may be able to edit environment variables from your Control Panel (in reasonably advanced versions of Windows, such as Windows XP, 2000, NT, and Vista; on older versions such as Windows 98, this does not work, and you must edit your `autoexec.bat` file instead, as covered in the next paragraph). From the Start menu, select Start ➤ Settings ➤ Control Panel. In the Control Panel, double-click the System icon. In the dialog box that opens, select the Advanced tab and click the Environment Variables button. That brings up another dialog box with two tables: one with your user variables and one with system variables. You are interested in the user variables. If you see `PYTHONPATH` there already, select it, click Edit, and edit it. Otherwise, click New and use `PYTHONPATH` as the name; enter your directory as the value. Note that multiple directories are separated by semicolons.

If the previous tactic doesn't work, you can edit the file `autoexec.bat`, which you can find (assuming that you have a relatively standard setup) in the top directory of the C drive. Open the file in Notepad (or the IDLE text editor, for that matter) and add a line setting the `PYTHONPATH`. If you want to add the directory `C:\python`, type the following:

```
set PYTHONPATH=%PYTHONPATH%;C:\python
```

Note that the IDE you're using might have its own mechanisms for setting environment variables and the Python path.

■ **Tip** You don't need to change the `sys.path` by using `PYTHONPATH`. Path configuration files provide a useful shortcut to make Python do it for you. A path configuration file is a file with the file name extension `.pth` and contains directories that should be added to `sys.path`. Empty lines and lines beginning with # are ignored. Files beginning with `import` are executed. For a path configuration file to be executed, it must be placed in a directory where it can be found. For Windows, use the directory named by `sys.prefix` (probably something like `C:\Python22`); in UNIX and Mac OS X, use the `site-packages` directory. (For more information, look up the `site` module in the Python Library Reference. This module is automatically imported during initialization of the Python interpreter.)

Naming Your Module

As you may have noticed, the file that contains the code of a module must be given the same name as the module, with an additional .py file name extension. In Windows, you can use the file name extension .pyw instead. You learn more about what that file name extension means in Chapter 12.

Packages

To structure your modules, you can group them into *packages*. A package is basically just another type of module. The interesting thing about them is that they can contain other modules. While a module is stored in a file (with the file name extension .py), a package is a directory. To make Python treat it as a package, it must contain a file (module) named __init__.py. The contents of this file will be the contents of the package, if you import it as if it were a plain module. For example, if you had a package named constants, and the file constants/__init__.py contains the statement PI = 3.14, you would be able to do the following:

```
import constants
print constants.PI
```

To put modules inside a package, simply put the module files inside the package directory.

For example, if you wanted a package called drawing, which contained one module called shapes and one called colors, you would need the files and directories (UNIX pathnames) shown in Table 10-1.

Table 10-1. *A Simple Package Layout*

File/Directory	Description
~/python/	Directory in PYTHONPATH
~/python/drawing/	Package directory (drawing package)
~/python/drawing/__init__.py	Package code (drawing module)
~/python/drawing/colors.py	colors module
~/python/drawing/shapes.py	shapes module

In Table 10-1, it is assumed that you have placed the directory ~/python in your PYTHONPATH. In Windows, simply replace ~/python with c:\python and reverse the direction of the slashes (to backslashes).

With this setup, the following statements are all legal:

```
import drawing              # (1) Imports the drawing package
import drawing.colors       # (2) Imports the colors module
from drawing import shapes  # (3) Imports the shapes module
```

After the first statement, the contents of the `__init__` module in `drawing` would be available; the `shapes` and `colors` modules, however, would not be. After the second statement, the `colors` module would be available, but only under its full name, `drawing.colors`. After the third statement, the `shapes` module would be available, under its short name (that is, simply `shapes`). Note that these statements are just examples. There is no need, for example, to import the package itself before importing one of its modules as I have done here. The second statement could very well be executed on its own, as could the third. You may nest packages inside each other.

Exploring Modules

Before I describe some of the standard library modules, I'll show you how to explore modules on your own. This is a valuable skill because you will encounter many useful modules in your career as a Python programmer, and I couldn't possibly cover all of them here. The current standard library is large enough to warrant books all by itself (and such books have been written)—and it's growing. New modules are added with each release, and often some of the modules undergo slight changes and improvements. Also, you will most certainly find several useful modules on the Web, and being able to grok[1] them quickly and easily will make your programming much more enjoyable.

What's in a Module?

The most direct way of probing a module is to investigate it in the Python interpreter. The first thing you need to do is to import it, of course. Let's say you've heard rumors about a standard module called `copy`:

```
>>> import copy
```

No exceptions are raised—so it exists. But what does it do? And what does it contain?

Using dir

To find out what a module contains, you can use the `dir` function, which lists all the attributes of an object (and therefore all functions, classes, variables, and so on of a module). If you print out `dir(copy)`, you get a long list of names. (Go ahead, try it.) Several of these names begin with an underscore—a hint (by convention) that they aren't meant to be used outside the module. So let's filter them out with a little list comprehension (check the section on list comprehension in Chapter 5 if you don't remember how this works):

```
>>> [n for n in dir(copy) if not n.startswith('_')]
['Error', 'PyStringMap', 'copy', 'deepcopy', 'dispatch_table', 'error', 'name', 't']
```

The list comprehension is the list consisting of all the names from `dir(copy)` that don't have an underscore as their first letter. This list is much less confusing than the full listing.

1. The term *grok* is hackerspeak, meaning "to understand fully," taken from Robert A. Heinlein's novel *Stranger in a Strange Land* (Ace Books, reissue 1995).

The __all__ Variable

What I did with the little list comprehension in the previous section was to make a guess about what I was *supposed* to see in the copy module. However, you can get the correct answer directly from the module itself. In the full dir(copy) list, you may have noticed the name __all__. This is a variable containing a list similar to the one I created with list comprehension, except that this list has been set in the module itself. Let's see what it contains:

```
>>> copy.__all__
['Error', 'copy', 'deepcopy']
```

My guess wasn't all that bad. I got only a few extra names that weren't intended for my use. But where did this __all__ list come from, and why is it really there? The first question is easy to answer. It was set in the copy module, like this (copied directly from copy.py):

```
__all__ = ["Error", "copy", "deepcopy"]
```

So why is it there? It defines the public interface of the module. More specifically, it tells the interpreter what it means to import all the names from this module. So if you use this:

```
from copy import *
```

you get only the four functions listed in the __all__ variable. To import PyStringMap, for example, you would need to be explicit, by either importing copy and using copy.PyStringMap, or by using from copy import PyStringMap.

Setting __all__ like this is actually a useful technique when writing modules, too. Because you may have a lot of variables, functions, and classes in your module that other programs might not need or want, it is only polite to filter them out. If you don't set __all__, the names exported in a starred import defaults to all global names in the module that don't begin with an underscore.

Getting Help with help

Until now, you've been using your ingenuity and knowledge of various Python functions and special attributes to explore the copy module. The interactive interpreter is a very powerful tool for this sort of exploration because your mastery of the language is the only limit to how deeply you can probe a module. However, there is one standard function that gives you all the information you would normally need. That function is called help. Let's try it on the copy function:

```
>>> help(copy.copy)
Help on function copy in module copy:

copy(x)
    Shallow copy operation on arbitrary Python objects.

    See the module's __doc__ string for more info.

>>>
```

This tells you that copy takes a single argument x, and that it is a "shallow copy operation." But it also mentions the module's __doc__ string. What's that? You may remember that I mentioned docstrings in Chapter 6. A docstring is simply a string you write at the beginning of a function to document it. That string may then be referred to by the function attribute __doc__. As you may understand from the preceding help text, modules may also have docstrings (they are written at the beginning of the module), as may classes (they are written at the beginning of the class).

Actually, the preceding help text was extracted from the copy function's docstring:

```
>>> print copy.copy.__doc__
Shallow copy operation on arbitrary Python objects.

    See the module's __doc__ string for more info.
```

The advantage of using help over just examining the docstring directly like this is that you get more information, such as the function signature (that is, the arguments it takes). Try to call help(copy) (on the module itself) and see what you get. It prints out a lot of information, including a thorough discussion of the difference between copy and deepcopy (essentially that deepcopy(x) makes copies of the values found in x as attributes and so on, while copy(x) just copies x, binding the attributes of the copy to the same values as those of x).

Documentation

A natural source for information about a module is, of course, its documentation. I've postponed the discussion of documentation because it's often much quicker to just examine the module a bit yourself first. For example, you may wonder, "What were the arguments to range again?" Instead of searching through a Python book or the standard Python documentation for a description of range, you can just check it directly:

```
>>> print range.__doc__
range([start,] stop[, step]) -> list of integers

Return a list containing an arithmetic progression of integers.
range(i, j) returns [i, i+1, i+2,..., j-1]; start (!) defaults to 0.
When step is given, it specifies the increment (or decrement).
For example, range(4) returns [0, 1, 2, 3].  The end point is omitted!
These are exactly the valid indices for a list of 4 elements.
```

You now have a precise description of the range function, and because you probably had the Python interpreter running already (wondering about functions like this usually happens while you are programming), accessing this information took just a couple of seconds.

However, not every module and every function has a good docstring (although it should), and sometimes you may need a more thorough description of how things work. Most modules you download from the Web have some associated documentation. In my opinion, some of the most useful documentation for learning to program in Python is the Python Library Reference, which describes all of the modules in the standard library. If I want to look up some fact about Python, nine times out of ten, I find it there. The library reference is available for online browsing (at http://python.org/doc/lib) or for download, as are several other standard documents (such as the Python Tutorial and the Python Language Reference). All of the documentation is available from the Python web site at `http://python.org/doc`.

Use the Source

The exploration techniques I've discussed so far will probably be enough for most cases. But those of you who wish to truly understand the Python language may want to know things about a module that can't be answered without actually reading the source code. Reading source code is, in fact, one of the best ways to learn Python—besides coding yourself.

Doing the actual reading shouldn't be much of a problem, but where is the source? Let's say you wanted to read the source code for the standard module `copy`. Where would you find it? One solution would be to examine `sys.path` again and actually look for it yourself, just like the interpreter does. A faster way is to examine the module's `__file__` property:

```
>>> print copy.__file__
C:\Python24\lib\copy.py
```

■ **Note** If the file name ends with `.pyc`, just use the corresponding file whose name ends with `.py`.

There it is! You can open the `copy.py` file in your code editor (for example, IDLE) and start examining how it works.

■ **Caution** When opening a standard library file in a text editor, you run the risk of accidentally modifying it. Doing so might break it, so when you close the file, make sure that you don't save any changes you might have made.

Note that some modules don't have any Python source you can read. They may be built into the interpreter (such as the sys module) or they may be written in the C programming language.[2] (See Chapter 17 for more information on extending Python using C.)

The Standard Library: A Few Favorites

Chances are that you're beginning to wonder what the title of this chapter means. The phrase "batteries included" with reference to Python was originally coined by Frank Stajano and refers to Python's copious standard library. When you install Python, you get a lot of useful modules (the "batteries") for "free." Because there are so many ways of getting more information about these modules (as explained in the first part of this chapter), I won't include a full reference here (which would take up far too much space anyway); instead, I'll describe

2. If the module was written in C, the C source code should be available.

a few of my favorite standard modules to whet your appetite for exploration. You'll encounter more standard modules in the project chapters (Chapters 20 through 29). The module descriptions are not complete but highlight some of the interesting features of each module.

sys

The sys module gives you access to variables and functions that are closely linked to the Python interpreter. Some of these are shown in Table 10-2.

Table 10-2. *Some Important Functions and Variables in the sys Module*

Function/Variable	Description
`argv`	The command-line arguments, including the script name
`exit([arg])`	Exits the current program, optionally with a given return value or error message
`modules`	A dictionary mapping module names to loaded modules
`path`	A list of directory names where modules can be found
`platform`	A platform identifier such as `sunos5` or `win32`
`stdin`	Standard input stream—a file-like object
`stdout`	Standard output stream—a file-like object
`stderr`	Standard error stream—a file-like object

The variable `sys.argv` contains the arguments passed to the Python interpreter, including the script name.

The function `sys.exit` exits the current program. (If called within a `try/finally` block, discussed in Chapter 8, the `finally` clause is still executed.) You can supply an integer to indicate whether the program succeeded—a UNIX convention. You'll probably be fine in most cases if you rely on the default (which is zero, indicating success). Alternatively, you can supply a string, which is used as an error message and can be very useful for a user trying to figure out why the program halted; then, the program exits with that error message and a code indicating failure.

The mapping `sys.modules` maps module names to actual modules. It applies to only currently imported modules.

The module variable `sys.path` was discussed earlier in this chapter. It's a list of strings, in which each string is the name of a directory where the interpreter will look for modules when an `import` statement is executed.

The module variable `sys.platform` (a string) is simply the name of the "platform" on which the interpreter is running. This may be a name indicating an operating system (such as `sunos5` or `win32`), or it may indicate some other kind of platform, such as a Java Virtual Machine (for example, `java1.4.0`) if you're running Jython.

The module variables `sys.stdin`, `sys.stdout`, and `sys.stderr` are file-like stream objects. They represent the standard UNIX concepts of standard input, standard output, and standard error. To put it simply, `sys.stdin` is where Python gets its input (used in the functions `input` and raw_input, for example), and sys.stdout is where it prints. You learn more about files (and these three streams) in Chapter 11.

As an example, consider the problem of using printing arguments in reverse order. When you call a Python script from the command line, you may add some arguments after it—the so-called *command-line arguments*. These will then be placed in the list sys.argv, with the name of the Python script as sys.argv[0]. Printing these out in reverse order is pretty simple, as you can see in Listing 10-5.

Listing 10-5. *Reversing and Printing Command-Line Arguments*

```
# reverseargs.py
import sys
args = sys.argv[1:]
args.reverse()
print ' '.join(args)
```

As you can see, I make a copy of sys.argv. You can modify the original, but in general, it's safer not to because other parts of the program may also rely on sys.argv containing the original arguments. Notice also that I skip the first element of sys.argv—the name of the script. I reverse the list with args.reverse(), but I can't print the result of that operation. It is an in-place modification that returns None. An alternative approach would be the following:

```
print ' '.join(reversed(sys.argv[1:]))
```

Finally, to make the output prettier, I use the join string method. Let's try the result (assuming a UNIX shell here, but it will work equally well at an MS-DOS prompt, for example):

```
$ python reverseargs.py this is a test
test a is this
```

os

The os module gives you access to several operating system services. The os module is extensive; only a few of the most useful functions and variables are described in Table 10-3. In addition to these, os and its submodule os.path contain several functions to examine, construct, and remove directories and files, as well as functions for manipulating paths (for example, os.path.split and os.path.join let you ignore os.pathsep most of the time). For more information about this functionality, see the standard library documentation.

Table 10-3. *Some Important Functions and Variables in the os Module*

Function/Variable	Description
environ	Mapping with environment variables
system(command)	Executes an operating system command in a subshell
sep	Separator used in paths
pathsep	Separator to separate paths

Continued

Table 10-3. *Continued*

Function/Variable	Description
`linesep`	Line separator (`'\n'`, `'\r'`, or `'\r\n'`)
`urandom(n)`	Returns `n` bytes of cryptographically strong random data

The mapping `os.environ` contains environment variables described earlier in this chapter. For example, to access the environment variable `PYTHONPATH`, you would use the expression `os.environ['PYTHONPATH']`. This mapping can also be used to change environment variables, although not all platforms support this.

The function `os.system` is used to run external programs. There are other functions for executing external programs, including `execv`, which exits the Python interpreter, yielding control to the executed program, and `popen`, which creates a file-like connection to the program. For more information about these functions, consult the standard library documentation.

Tip In current versions of Python, the `subprocess` module is available. It collects the functionality of the `os.system`, `execv`, and `popen` functions.

The module variable `os.sep` is a separator used in pathnames. The standard separator in UNIX (and the Mac OS X command-line version of Python) is `/`. The standard in Windows is `\\` (the Python syntax for a single backslash), and in Mac OS, it is `:`. (On some platforms, `os.altsep` contains an alternate path separator, such as `/` in Windows.)

You use `os.pathsep` when grouping several paths, as in `PYTHONPATH`. The `pathsep` is used to separate the pathnames: `:` in UNIX (and the Mac OS X command-line version of Python), `;` in Windows, and `::` in Mac OS.

The module variable `os.linesep` is the line separator string used in text files. In UNIX (and, again, the command-line version in Mac OS X), this is a single newline character (`\n`), in Mac OS, it's a single carriage return character (`\r`); and in Windows, it's the combination of a carriage return and a newline (`\r\n`).

The `urandom` function uses a system-dependent source of "real" (or, at least, cryptographically strong) randomness. If your platform doesn't support it, you'll get a `NotImplementedError`.

As an example, consider the problem of starting a web browser. The `system` command can be used to execute any external program, which is very useful in environments such as UNIX where you can execute programs (or *commands*) from the command line to list the contents of a directory, send email, and so on. But it can be useful for starting programs with graphical user interfaces, too—such as a web browser. In UNIX, you can do the following (provided that you have a browser at `/usr/ bin/firefox`):

```
os.system('/usr/bin/firefox')
```

Here's a Windows version (again, use the path of a browser you have installed):

```
os.system(r'c:\"Program Files"\"Mozilla Firefox"\firefox.exe')
```

Note that I've been careful about enclosing `Program Files` and `Mozilla Firefox` in quotes; otherwise, DOS (which handles the command) balks at the whitespace. (This may be important for directories in your `PYTHONPATH` as well.) Note also that you must use backslashes here because DOS gets confused by forward slashes. If you run this, you will notice that the browser tries to open a web site named `Files"\Mozilla...`—the part of the command after the whitespace. Also, if you try to run this from IDLE, a DOS window appears, but the browser doesn't start until you close that DOS window. All in all, not exactly ideal behavior.

Another function that suits the job better is the Windows-specific function `os.startfile`:

```
os.startfile(r'c:\Program Files\Mozilla Firefox\firefox.exe')
```

As you can see, `os.startfile` accepts a plain path, even if it contains whitespace (that is, don't enclose `Program Files` in quotes as in the `os.system` example).

Note that in Windows, your Python program keeps on running after the external program has been started by `os.system` (or `os.startfile`); in UNIX, your Python program waits for the `os.system` command to finish.

A BETTER SOLUTION: WEBBROWSER

The `os.system` function is useful for a lot of things, but for the specific task of launching a web browser, there's an even better solution: the `webbrowser` module. It contains a function called `open`, which lets you automatically launch a web browser to open the given URL. For example, if you want your program to open the Python web site in a web browser (either starting a new browser or using one that is already running), you simply use this:

```
import webbrowser
webbrowser.open('http://www.python.org')
```

The page should pop up. Pretty nifty, huh?

fileinput

You learn a lot about reading from and writing to files in Chapter 11, but here is a sneak preview. The `fileinput` module enables you to easily iterate over all the lines in a series of text files. If you call your script like this (assuming a UNIX command line):

```
$ python some_script.py file1.txt file2.txt file3.txt
```

you will be able to iterate over the lines of `file1.txt` through `file3.txt` in turn. You can also iterate over lines supplied to standard input (`sys.stdin`, remember?), for example, in a UNIX pipe, using the standard UNIX command `cat`:

```
$ cat file.txt | python some_script.py
```

If you use `fileinput`, calling your script with `cat` in a UNIX pipe works just as well as supplying the file names as command-line arguments to your script. The most important functions of the `fileinput` module are described in Table 10-4.

Table 10-4. *Some Important Functions in the fileinput Module*

Function	Description
`input([files[, inplace[, backup]])`	Facilitates iteration over lines in multiple input streams
`filename()`	Returns the name of the current file
`lineno()`	Returns the current (cumulative) line number
`filelineno()`	Returns the line number within current file
`isfirstline()`	Checks whether the current line is first in file
`isstdin()`	Checks whether the last line was from `sys.stdin`
`nextfile()`	Closes the current file and moves to the next
`close()`	Closes the sequence

`fileinput.input` is the most important of the functions. It returns an object that you can iterate over in a `for` loop. If you don't want the default behavior (in which `fileinput` finds out which files to iterate over), you can supply one or more file names to this function (as a sequence). You can also set the `inplace` parameter to a true value (`inplace=True`) to enable in-place processing. For each line you access, you'll need to print out a replacement, which will be put back into the current input file. The optional `backup` argument gives a file name extension to a backup file created from the original file when you do in-place processing.

The function `fileinput.filename` returns the file name of the file you are currently in (that is, the file that contains the line you are currently processing).

The function `fileinput.lineno` returns the number of the current line. This count is cumulative so that when you are finished with one file and begin processing the next, the line number is not reset but starts at one more than the last line number in the previous file.

The function `fileinput.filelineno` returns the number of the current line within the current file. Each time you are finished with one file and begin processing the next, the file line number is reset, and restarts at 1.

The function `fileinput.isfirstline` returns a true value if the current line is the first line of the current file; otherwise, it returns a false value.

The function `fileinput.isstdin` returns a true value if the current file is `sys.stdin`; otherwise, it returns false.

The function `fileinput.nextfile` closes the current file and skips to the next one. The lines you skip do not count against the line count. This can be useful if you know that you are finished with the current file—for example, if each file contains words in sorted order, and you are looking for a specific word. If you have passed the word's position in the sorted order, you can safely skip to the next file.

The function `fileinput.close` closes the entire chain of files and finishes the iteration.

As an example of using `fileinput`, let's say you have written a Python script and you want to number the lines. Because you want the program to keep working after you've done this, you must add the line numbers in comments to the right of each line. To line them up, you can use string formatting. Let's allow each program line to get 40 characters maximum and add the comment after that. The program in Listing 10-6 shows a simple way of doing this with `fileinput` and the `inplace` parameter.

Listing 10-6. *Adding Line Numbers to a Python Script*

```
# numberlines.py

import fileinput

for line in fileinput.input(inplace=True):
    line = line.rstrip()
    num  = fileinput.lineno()
    print '%-40s # %2i' % (line, num)
```

If you run this program on itself, like this:

```
$ python numberlines.py numberlines.py
```

you end up with the program in Listing 10-7. Note that the program itself has been modified, and that if you run it like this several times, you will have multiple numbers on each line. Recall that `rstrip` is a string method that returns a copy of a string, where all the whitespace on the right has been removed (see the section "String Methods" in Chapter 3 and Table B-6 in Appendix B).

Listing 10-7. *The Line Numbering Program with Line Numbers Added*

```
# numberlines.py                        # 1
                                        # 2
import fileinput                        # 3
                                        # 4
for line in fileinput.input(inplace=1): # 5
    line = line.rstrip()                # 6
    num  = fileinput.lineno()           # 7
    print '%-40s # %2i' % (line, num)   # 8
```

■ Caution Be careful about using the `inplace` parameter—it's an easy way to ruin a file. You should test your program carefully *without* setting `inplace` (this will simply print out the result), making sure the program works before you let it modify your files.

For another example using `fileinput`, see the section about the `random` module, later in this chapter.

Sets, Heaps, and Deques

There are many useful data structures around, and Python supports some of the more common ones. Some of these, such as dictionaries (or hash tables) and lists (or dynamic arrays), are integral to the language. Others, although somewhat more peripheral, can still come in handy sometimes.

Sets

Sets were introduced in Python 2.3, through the Set class in the sets module. Although you may come upon Set instances in existing code, there is really very little reason to use them yourself, unless you want to be backward-compatible. In Python 2.3, sets were made part of the language, through the set type. This means that you don't need to import the sets module—you can just create sets directly:

```
>>> set(range(10))
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Sets are constructed from a sequence (or some other iterable object). Their main use is to check membership, and thus duplicates are ignored:

```
>>> set([0, 1, 2, 3, 0, 1, 2, 3, 4, 5])
set([0, 1, 2, 3, 4, 5])
```

Just as with dictionaries, the ordering of set elements is quite arbitrary and shouldn't be relied on:

```
>>> set(['fee', 'fie', 'foe'])
set(['foe', 'fee', 'fie'])
```

In addition to checking for membership, you can perform various standard set operations (which you may know from mathematics), such as union and intersection, either by using methods or by using the same operations as you would for bit operations on integers (see Appendix B). For example, you can find the union of two sets using either the union method of one of them or the bitwise OR operator, |:

```
>>> a = set([1, 2, 3])
>>> b = set([2, 3, 4])
>>> a.union(b)
set([1, 2, 3, 4])
>>> a | b
set([1, 2, 3, 4])
```

Here are some other methods and their corresponding operators; the names should make it clear what they mean:

```
>>> c = a & b
>>> c.issubset(a)
True
>>> c <= a
True
>>> c.issuperset(a)
False
>>> c >= a
False
>>> a.intersection(b)
set([2, 3])
```

```
>>> a & b
set([2, 3])
>>> a.difference(b)
set([1])
>>> a - b
set([1])
>>> a.symmetric_difference(b)
set([1, 4])
>>> a ^ b
set([1, 4])
>>> a.copy()
set([1, 2, 3])
>>> a.copy() is a
False
```

There are also various in-place operations, with corresponding methods, as well as the basic methods `add` and `remove`. For more information, see the section about set types in the Python Library Reference (`http://python.org/doc/lib/types-set.html`).

■ **Tip** If you need a function for finding, say, the union of two sets, you can simply use the unbound version of the `union` method, from the `set` type. This could be useful, for example, in concert with `reduce`:

```
>>> mySets = []
>>> for i in range(10):
...     mySets.append(set(range(i,i+5)))
...
>>> reduce(set.union, mySets)
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13])
```

Sets are mutable, and may therefore not be used, for example, as keys in dictionaries. Another problem is that sets themselves may contain only immutable (hashable) values, and thus may not contain other sets. Because sets of sets often occur in practice, this could be a problem. Luckily, there is the `frozenset` type, which represents *immutable* (and, therefore, hashable) sets:

```
>>> a = set()
>>> b = set()
>>> a.add(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: set objects are unhashable
>>> a.add(frozenset(b))
```

The `frozenset` constructor creates a copy of the given set. It is useful whenever you want to use a set either as a member of another set or as the key to a dictionary.

Heaps

Another well-known data structure is the *heap*, a kind of priority queue. A priority queue lets you add objects in an arbitrary order, and at any time (possibly in between the adding), find (and possibly remove) the smallest element. It does so much more efficiently than, say, using `min` on a list.

In fact, there is no separate heap type in Python—only a module with some heap-manipulating functions. The module is called `heapq` (the `q` stands for queue), and it contains six functions (see Table 10-5), the first four of which are directly related to heap manipulation. You must use a list as the heap object itself.

Table 10-5. *Some Important Functions in the fileinput Module*

Function	Description
`heappush(heap, x)`	Pushes `x` onto the heap
`heappop(heap)`	Pops off the smallest element in the heap
`heapify(heap)`	Enforces the `heap` property on an arbitrary list
`heapreplace(heap, x)`	Pops off the smallest element and pushes `x`
`nlargest(n, iter)`	Returns the `n` largest elements of `iter`
`nsmallest(n, iter)`	Returns the `n` smallest elements of `iter`

The `heappush` function is used to add an item to a heap. Note that you shouldn't use it on any old list—only one that has been built through the use of the various heap functions. The reason for this is that the order of the elements is important (even though it may look a bit haphazard; the elements aren't exactly sorted).

```
>>> from heapq import *
>>> from random import shuffle
>>> data = range(10)
>>> shuffle(data)
>>> heap = []
>>> for n in data:
...     heappush(heap, n)
>>> heap
[0, 1, 3, 6, 2, 8, 4, 7, 9, 5]
>>> heappush(heap, 0.5)
>>> heap
[0, 0.5, 3, 6, 1, 8, 4, 7, 9, 5, 2]
```

The order of the elements isn't as arbitrary as it seems. They aren't in strictly sorted order, but there is one guarantee made: the element at position `i` is always greater than the one in position `i // 2` (or, conversely, it's smaller than the elements at positions `2*i` and `2*i + 1`). This is the basis for the underlying heap algorithm. This is called the *heap property*.

The heappop function pops off the smallest element, which is always found at index 0, and makes sure that the smallest of the remaining elements takes over this position (while preserving the heap property). Even though popping the first element of a list isn't terribly efficient in general, it's not a problem here, because heappop does some nifty shuffling behind the scenes:

```
>>> heappop(heap)
0
>>> heappop(heap)
0.5
>>> heappop(heap)
1
>>> heap
[2, 5, 3, 6, 9, 8, 4, 7]
```

The heapify function takes an arbitrary list and makes it a legal heap (that is, it imposes the heap property) through the least possible amount of shuffling. If you don't build your heap from scratch with heappush, this is the function to use before starting to use heappush and heappop:

```
>>> heap = [5, 8, 0, 3, 6, 7, 9, 1, 4, 2]
>>> heapify(heap)
>>> heap
[0, 1, 5, 3, 2, 7, 9, 8, 4, 6]
```

The heapreplace function is not quite as commonly used as the others. It pops the smallest element off the heap and then pushes a new element onto it. This is a bit more efficient than a heappop followed by a heappush:

```
>>> heapreplace(heap, 0.5)
0
>>> heap
[0.5, 1, 5, 3, 2, 7, 9, 8, 4, 6]
>>> heapreplace(heap, 10)
0.5
>>> heap
[1, 2, 5, 3, 6, 7, 9, 8, 4, 10]
```

The remaining two functions of the heapq module, nlargest(n, iter) and nsmallest(n, iter), are used to find the n largest or smallest elements, respectively, of any iterable object iter. You could do this by using sorting (for example, using the sorted function) and slicing, but the heap algorithm is faster and more memory-efficient (and, not to mention, easier to use).

Deques (and Other Collections)

Double-ended queues, or *deques*, can be useful when you need to remove elements in the order in which they were added. In Python 2.4, the collections module was added, which contains the deque type.

Note As of Python 2.5, the `collections` module contains the `deque` type and `defaultdict`—a dictionary with a default value for nonexisting keys. Possible future additions are B-trees and Fibonacci heaps.

A deque is created from an iterable object (just like sets) and has several useful methods:

```
>>> from collections import deque
>>> q = deque(range(5))
>>> q.append(5)
>>> q.appendleft(6)
>>> q
deque([6, 0, 1, 2, 3, 4, 5])
>>> q.pop()
5
>>> q.popleft()
6
>>> q.rotate(3)
>>> q
deque([2, 3, 4, 0, 1])
>>> q.rotate(-1)
>>> q
deque([3, 4, 0, 1, 2])
```

The deque is useful because it allows appending and popping efficiently at the beginning (to the left), which you cannot do with lists. As a nice side effect, you can also rotate the elements (that is, shift them to the right or left, wrapping around the ends) efficiently. Deque objects also have the `extend` and `extendleft` methods, with `extend` working like the corresponding list method, and `extendleft` working analogously to `appendleft`. Note that the elements in the iterable object used in `extendleft` will appear in the deque in reverse order.

time

The `time` module contains functions for, among other things, getting the current time, manipulating times and dates, reading dates from strings, and formatting dates as strings. Dates can be represented as either a real number (the seconds since 0 hours, January 1 in the "epoch," a platform-dependent year; for UNIX, it's 1970), or a tuple containing nine integers. These integers are explained in Table 10-6. For example, the tuple

```
(2008, 1, 21, 12, 2, 56, 0, 21, 0)
```

represents January 21, 2008, at 12:02:56, which is a Monday and the twenty-first day of the year (no daylight savings).

Table 10-6. *The Fields of Python Date Tuples*

Index	Field	Value
0	Year	For example, 2000, 2001, and so on
1	Month	In the range 1–12
2	Day	In the range 1–31
3	Hour	In the range 0–23
4	Minute	In the range 0–59
5	Second	In the range 0–61
6	Weekday	In the range 0–6, where Monday is 0
7	Julian day	In the range 1–366
8	Daylight savings	0, 1, or –1

The range for seconds is 0–61 to account for leap seconds and double-leap seconds. The daylight savings number is a Boolean value (true or false), but if you use –1, `mktime` (a function that converts such a tuple to a timestamp measured in seconds since the epoch) will probably get it right. Some of the most important functions in the `time` module are described in Table 10-7.

Table 10-7. *Some Important Functions in the time Module*

Function	Description
`asctime([tuple])`	Converts a time tuple to a string
`localtime([secs])`	Converts seconds to a date tuple, local time
`mktime(tuple)`	Converts a time tuple to local time
`sleep(secs)`	Sleeps (does nothing) for `secs` seconds
`strptime(string[, format])`	Parses a string into a time tuple
`time()`	Current time (seconds since the epoch, UTC)

The function `time.asctime` formats the current time as a string, like this:

```
>>> time.asctime()
'Fri Dec 21 05:41:27 2008'
```

You can also supply a date tuple (such as those created by `localtime`) if you don't want the current time. (For more elaborate formatting, you can use the `strftime` function, described in the standard documentation.)

The function `time.localtime` converts a real number (seconds since epoch) to a date tuple, local time. If you want universal time,[3] use `gmtime` instead.

The function `time.mktime` converts a date tuple to the time since epoch in seconds; it is the inverse of `localtime`.

The function `time.sleep` makes the interpreter wait for a given number of seconds.

The function `time.strptime` converts a string of the format returned by `asctime` to a date tuple. (The optional format argument follows the same rules as those for `strftime`; see the standard documentation.)

The function `time.time` returns the current (universal) time as seconds since the epoch. Even though the epoch may vary from platform to platform, you can reliably time something by keeping the result of `time` from before and after the event (such as a function call), and then computing the difference. For an example of these functions, see the next section, which covers the `random` module.

The functions shown in Table 10-7 are just a selection of those available from the `time` module. Most of the functions in this module perform tasks similar to or related to those described in this section. If you need something not covered by the functions described here, take a look at the section about the `time` module in the Python Library Reference (`http://python.org/doc/lib/module-time.html`); chances are you may find exactly what you are looking for.

Additionally, two more recent time-related modules are available: `datetime` (which supports date and time arithmetic) and `timeit` (which helps you time pieces of your code). You can find more information about both in the Python Library Reference, and `timeit` is also discussed briefly in Chapter 16.

random

The `random` module contains functions that return random numbers, which can be useful for simulations or any program that generates random output.

■ **Note** Actually, the numbers generated are pseudo-random. That means that while they appear completely random, there is a predictable system that underlies them. However, because the module is so good at pretending to be random, you probably won't ever have to worry about this (unless you want to use these numbers for strong-cryptography purposes, in which case they may not be "strong" enough to withstand a determined attack—but if you're into strong cryptography, you surely don't need me to explain such elementary issues). If you need *real* randomness, you should check out the `urandom` function of the `os` module. The class `SystemRandom` in the `random` module is based on the same kind of functionality, and gives you data that is close to real randomness.

Some important functions in this module are shown in Table 10-8.

3. For more information about universal time, see `http://en.wikipedia.org/wiki/Universal_time`.

Table 10-8. *Some Important Functions in the random Module*

Function	Description
`random()`	Returns a random real number n such that $0 \; n \le 1$
`getrandbits(n)`	Returns n random bits, in the form of a long integer
`uniform(a, b)`	Returns a random real number n such that $a \; n \le b$
`randrange([start], stop, [step])`	Returns a random number from `range(start, stop, step)`
`choice(seq)`	Returns a random element from the sequence `seq`
`shuffle(seq[, random])`	Shuffles the sequence `seq` in place
`sample(seq, n)`	Chooses `n` random, unique elements from the sequence `seq`

The function `random.random` is one of the most basic random functions; it simply returns a pseudo-random number n such that $0 \; n \le 1$. Unless this is exactly what you need, you should probably use one of the other functions, which offer extra functionality. The function `random.getrandbits` returns a given number of bits (binary digits), in the form of a long integer. This is probably mostly useful if you're really into random stuff (for example, working with cryptography).

The function `random.uniform`, when supplied with two numerical parameters `a` and `b`, returns a random (uniformly distributed) real number n such that `a` $\; n \le$ `b`. So, for example, if you want a random angle, you could use `uniform(0,360)`.

The function `random.randrange` is the standard function for generating a random integer in the range you would get by calling `range` with the same arguments. For example, to get a random number in the range from 1 to 10 (inclusive), you would use `randrange(1,11)` (or, alternatively, `randrange(10)+1`), and if you want a random odd positive integer lower than 20, you would use `randrange(1,20,2)`.

The function `random.choice` chooses (uniformly) a random element from a given sequence.

The function `random.shuffle` shuffles the elements of a (mutable) sequence randomly, such that every possible ordering is equally likely.

The function `random.sample` chooses (uniformly) a given number of elements from a given sequence, making sure that they're all different.

■Note For the statistically inclined, there are other functions similar to `uniform` that return random numbers sampled according to various other distributions, such as betavariate, exponential, Gaussian, and several others.

Let's look at some examples using the `random` module. In these examples, I use several of the functions from the `time` module described previously. First, let's get the real numbers representing the limits of the time interval (the year 2008). You do that by expressing the date as a time tuple (using `-1` for day of the week, day of the year, and daylight savings, making Python calculate that for itself) and calling `mktime` on these tuples:

```
from random import *
from time import *
date1 = (2008, 1, 1, 0, 0, 0, -1, -1, -1)
time1 = mktime(date1)
date2 = (2009, 1, 1, 0, 0, 0, -1, -1, -1)
time2 = mktime(date2)
```

Then you generate a random number uniformly in this range (the upper limit excluded):

```
>>> random_time = uniform(time1, time2)
```

Then you simply convert this number back to a legible date:

```
>>> print asctime(localtime(random_time))
Mon Jun 24 21:35:19 2008
```

For the next example, let's ask the user how many dice to throw, and how many sides each one should have. The die-throwing mechanism is implemented with `randrange` and a `for` loop:

```
from random import randrange
num   = input('How many dice? ')
sides = input('How many sides per die? ')
sum = 0
for i in range(num): sum += randrange(sides) + 1
print 'The result is', sum
```

If you put this in a script file and run it, you get an interaction something like the following:

```
How many dice? 3
How many sides per die? 6
The result is 10
```

Now assume that you have made a text file in which each line of text contains a fortune. Then you can use the `fileinput` module described earlier to put the fortunes in a list, and then select one randomly:

```
# fortune.py
import fileinput, random
fortunes = list(fileinput.input())
print random.choice(fortunes)
```

In UNIX, you could test this on the standard dictionary file `/usr/dict/words` to get a random word:

```
$ python fortune.py /usr/dict/words
dodge
```

As a last example, suppose that you want your program to deal you cards, one at a time, each time you press Enter on your keyboard. Also, you want to make sure that you don't get the same card more than once. First, you make a "deck of cards"—a list of strings:

```
>>> values = range(1, 11) + 'Jack Queen King'.split()
>>> suits = 'diamonds clubs hearts spades'.split()
>>> deck = ['%s of %s' % (v, s) for v in values for s in suits]
```

The deck you just created isn't very suitable for a game of cards. Let's just peek at some of the cards:

```
>>> from pprint import pprint
>>> pprint(deck[:12])
['1 of diamonds',
 '1 of clubs',
 '1 of hearts',
 '1 of spades',
 '2 of diamonds',
 '2 of clubs',
 '2 of hearts',
 '2 of spades',
 '3 of diamonds',
 '3 of clubs',
 '3 of hearts',
 '3 of spades']
```

A bit too ordered, isn't it? That's easy to fix:

```
>>> from random import shuffle
>>> shuffle(deck)
>>> pprint(deck[:12])
['3 of spades',
 '2 of diamonds',
 '5 of diamonds',
 '6 of spades',
 '8 of diamonds',
 '1 of clubs',
 '5 of hearts',
 'Queen of diamonds',
 'Queen of hearts',
 'King of hearts',
 'Jack of diamonds',
 'Queen of clubs']
```

Note that I've just printed the 12 first cards here, to save some space. Feel free to take a look at the whole deck yourself.

Finally, to get Python to deal you a card each time you press Enter on your keyboard, until there are no more cards, you simply create a little `while` loop. Assuming that you put the code needed to create the deck into a program file, you could simply add the following at the end:

```
while deck: raw_input(deck.pop())
```

■ **Note** If you try the `while` loop shown here in the interactive interpreter, you'll notice that an empty string is printed out every time you press Enter. This is because `raw_input` returns what you write (which is nothing) and that will get printed. In a normal program, this return value from `raw_input` is simply ignored. To have it "ignored" interactively, too, just assign the result of `raw_input` to some variable you won't look at again and name it something like `ignore`.

---

shelve

In the next chapter, you learn how to store data in files, but if you want a really simple storage solution, the `shelve` module can do most of the work for you. All you need to do is supply it with a file name. The only function of interest in `shelve` is `open`. When called (with a file name) it returns a `Shelf` object, which you can use to store things. Just treat it as a normal dictionary (except that the keys must be strings), and when you're finished (and want things saved to disk), call its `close` method.

A Potential Trap

It is important to realize that the object returned by `shelve.open` is not an ordinary mapping, as the following example demonstrates:

```
>>> import shelve
>>> s = shelve.open('test.dat')
>>> s['x'] = ['a', 'b', 'c']
>>> s['x'].append('d')
>>> s['x']
['a', 'b', 'c']
```

Where did the `'d'` go?

The explanation is simple: when you look up an element in a `shelf` object, the object is reconstructed from its stored version; and when you assign an element to a key, it is stored. What happened in the preceding example was the following:

- The list `['a', 'b', 'c']` was stored in `s` under the key `'x'`.
- The stored representation was retrieved, a new list was constructed from it, and `'d'` was appended to the copy. This modified version was *not* stored!
- Finally, the original is retrieved again—without the `'d'`.

To correctly modify an object that is stored using the `shelve` module, you must bind a temporary variable to the retrieved copy, and then store the copy again after it has been modified:[4]

```
>>> temp = s['x']
>>> temp.append('d')
>>> s['x'] = temp
>>> s['x']
['a', 'b', 'c', 'd']
```

From Python 2.4 onward, there is another way around this problem: set the `writeback` parameter of the `open` function to `true`. If you do, all of the data structures that you read from or assign to the shelf will be kept around in memory (cached) and written back to disk only when you close the shelf. If you're not working with a huge amount of data, and you don't want to worry about these things, setting `writeback` to `true` (and making sure you close your shelf at the end) may be a good idea.

A Simple Database Example

Listing 10-8 shows a simple database application that uses the `shelve` module.

Listing 10-8. *A Simple Database Application*

```
# database.py
import sys, shelve

def store_person(db):
    """
    Query user for data and store it in the shelf object
    """
    pid = raw_input('Enter unique ID number: ')
    person = {}
    person['name']  = raw_input('Enter name: ')
    person['age']  = raw_input('Enter age: ')
    person['phone'] = raw_input('Enter phone number: ')

    db[pid] = person

def lookup_person(db):
    """
    Query user for ID and desired field, and fetch the corresponding data from
    the shelf object
    """
    pid = raw_input('Enter ID number: ')
    field = raw_input('What would you like to know? (name, age, phone)  ')
    field = field.strip().lower()
```

4. Thanks to Luther Blissett for pointing this out.

```
    print field.capitalize() + ':', \
          db[pid][field]

def print_help():
    print 'The available commands are:'
    print 'store  : Stores information about a person'
    print 'lookup : Looks up a person from ID number'
    print 'quit   : Save changes and exit'
    print '?      : Prints this message'

def enter_command():
    cmd = raw_input('Enter command (? for help): ')
    cmd = cmd.strip().lower()
    return cmd

def main():
    database = shelve.open('C:\\database.dat') # You may want to change this name
    try:
        while True:
            cmd = enter_command()
            if   cmd == 'store':
                store_person(database)
            elif cmd == 'lookup':
                lookup_person(database)
            elif cmd == '?':
                print_help()
            elif cmd == 'quit':
                return
    finally:
        database.close()

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

The program shown in Listing 10-8 has several interesting features:

- Everything is wrapped in functions to make the program more structured. (A possible improvement is to group those functions as the methods of a class.)
- The main program is in the `main` function, which is called only if `__name__ == '__main__'`. That means you can import this as a module and then call the `main` function from another program.
- I open a database (*shelf*) in the `main` function, and then pass it as a parameter to the other functions that need it. I could have used a global variable, too, because this program is so small, but it's better to avoid global variables in most cases, unless you have a reason to use them.

- After reading in some values, I make a modified version by calling `strip` and `lower` on them because if a supplied key is to match one stored in the database, the two must be *exactly* alike. If you always use `strip` and `lower` on what the users enter, you can allow them to be sloppy about using uppercase or lowercase letters and additional whitespace. Also, note that I've used `capitalize` when printing the field name.

- I have used `try` and `finally` to ensure that the database is closed properly. You never know when something might go wrong (and you get an exception), and if the program terminates without closing the database properly, you may end up with a corrupt database file that is essentially useless. By using `try` and `finally`, you avoid that.

So, let's take this database out for a spin. Here is a sample interaction:

```
Enter command (? for help): ?
The available commands are:
store  : Stores information about a person
lookup : Looks up a person from ID number
quit   : Save changes and exit
?      : Prints this message
Enter command (? for help): store
Enter unique ID number: 001
Enter name: Mr. Gumby
Enter age: 42
Enter phone number: 555-1234
Enter command (? for help): lookup
Enter ID number: 001
What would you like to know? (name, age, phone) phone
Phone: 555-1234
Enter command (? for help): quit
```

This interaction isn't terribly interesting. I could have done exactly the same thing with an ordinary dictionary instead of the `shelf` object. But now that I've quit the program, let's see what happens when I restart it—perhaps the following day?

```
Enter command (? for help): lookup
Enter ID number: 001
What would you like to know? (name, age, phone) name
Name: Mr. Gumby
Enter command (? for help): quit
```

As you can see, the program reads in the file I created the first time, and Mr. Gumby is still there!

Feel free to experiment with this program, and see if you can extend its functionality and improve its user-friendliness. Perhaps you can think of a version that you have use for yourself? How about a database of your record collection? Or a database to help you keep track of friends who have borrowed books from you. (I know I could use that last one.)

re

Some people, when confronted with a problem, think, "I know, I'll use regular expressions." Now they have two problems.

—Jamie Zawinski

The `re` module contains support for *regular expressions*. If you've heard about regular expressions, you probably know how powerful they are; if you haven't, prepare to be amazed.

You should note, however, that mastering regular expressions may be a bit tricky at first. (Okay, very tricky, actually.) The key is to learn about them a little bit at a time—just look up (in the documentation) the parts you need for a specific task. There is no point in memorizing it all up front. This section describes the main features of the `re` module and regular expressions, and enables you to get started.

■ **Tip** In addition to the standard documentation, Andrew Kuchling's "Regular Expression HOWTO" (`http://amk.ca/python/howto/regex/`) is a useful source of information on regular expressions in Python.

What Is a Regular Expression?

A regular expression (also called a *regex* or *regexp*) is a pattern that can match a piece of text. The simplest form of regular expression is just a plain string, which matches itself. In other words, the regular expression `'python'` matches the string `'python'`. You can use this matching behavior for such things as searching for patterns in text, replacing certain patterns with some computed values, or splitting text into pieces.

The Wildcard

A regular expression can match more than one string, and you create such a pattern by using some special characters. For example, the period character (dot) matches any character (except a newline), so the regular expression `'.ython'` would match both the string `'python'` and the string `'jython'`. It would also match strings such as `'qython'`, `'+ython'`, or `' ython'` (in which the first letter is a single space), but not strings such as `'cpython'` or `'ython'` because the period matches a single letter, and neither two nor zero.

Because it matches "anything" (any single character except a newline), the period is called a *wildcard*.

Escaping Special Characters

When you use special characters in regular expressions, it's important to know that you may run into problems if you try to use them as normal characters. For example, imagine you want to match the string `'python.org'`. Do you simply use the pattern `'python.org'`? You could, but that would also match `'pythonzorg'`, for example, which you probably wouldn't want. (The dot matches any character except a newline, remember?) To make a special character behave like a normal one, you *escape* it, just as I demonstrated how to escape quotes in strings in Chapter 1. You place a backslash in front of it. Thus, in this example, you would use `'python\\.org'`, which would match `'python.org'` and nothing else.

■Note To get a single backslash, which is required here by the `re` module, you need to write two backslashes in the string—to escape it from the interpreter. Thus you have *two levels* of escaping here: (1) from the interpreter, and (2) from the `re` module. (Actually, in some cases you can get away with using a single backslash and have the interpreter escape it for you automatically, but don't rely on it.) If you are tired of doubling up backslashes, use a raw string, such as `r'python\.org'`.

Character Sets

Matching any character can be useful, but sometimes you want more control. You can create a so-called *character set* by enclosing a substring in brackets. Such a character set will match any of the characters it contains. For example, `'[pj]ython'` would match both `'python'` and `'jython'`, but nothing else. You can also use ranges, such as `'[a-z]'` to match any character from *a* to *z* (alphabetically), and you can combine such ranges by putting one after another, such as `'[a-zA-Z0-9]'` to match uppercase and lowercase letters and digits. (Note that the character set will match only *one* such character, though.)

To invert the character set, put the character ^ first, as in `'[^abc]'` to match any character except *a*, *b*, or *c*.

SPECIAL CHARACTERS IN CHARACTER SETS

In general, special characters such as dots, asterisks, and question marks must be escaped with a backslash if you want them to appear as literal characters in the pattern, rather than function as regular expression operators. Inside character sets, escaping these characters is generally not necessary (although perfectly legal). You should, however, keep in mind the following rules:

- You do need to escape the caret (`^`) if it appears at the beginning of the character set, unless you want it to function as a negation operator. (In other words, don't place it at the beginning unless you mean it.)
- Similarly, the right bracket (`]`) and the dash (`-`) must be put either at the beginning of the character set or escaped with a backslash. (Actually, the dash may also be put at the end, if you wish.)

Alternatives and Subpatterns

Character sets are nice when you let each letter vary independently, but what if you want to match only the strings `'python'` and `'perl'`? You can't specify such a specific pattern with character sets or wildcards. Instead, you use the special character for alternatives: the pipe character (`|`). So, your pattern would be `'python|perl'`.

However, sometimes you don't want to use the choice operator on the entire pattern—just a part of it. To do that, you enclose the part, or subpattern, in parentheses. The previous example could be rewritten as `'p(ython|erl)'`. (Note that the term *subpattern* can also apply to a single character.)

Optional and Repeated Subpatterns

By adding a question mark after a subpattern, you make it optional. It may appear in the matched string, but it isn't strictly required. So, for example, this (slightly unreadable) pattern:

```
r'(http://)?(www\.)?python\.org'
```

would match all of the following strings (and nothing else):

```
'http://www.python.org'
'http://python.org'
'www.python.org'
'python.org'
```

A few things are worth noting here:

- I've escaped the dots, to prevent them from functioning as wildcards.
- I've used a raw string to reduce the number of backslashes needed.
- Each optional subpattern is enclosed in parentheses.
- The optional subpatterns may or may not appear , independently of each other.

The question mark means that the subpattern can appear once or not at all. A few other operators allow you to repeat a subpattern more than once:

- `(pattern)*`: `pattern` is repeated zero or more times.
- `(pattern)+`: `pattern` is repeated one or more times.
- `(pattern){m,n}`: `pattern` is repeated from `m` to `n` times.

So, for example, `r'w*\.python\.org'` matches `'www.python.org'`, but also `'.python.org'`, `'ww.python.org'`, and `'wwwwwww.python.org'`. Similarly, `r'w+\.python\.org'` matches `'w.python.org'` but not `'.python.org'`, and `r'w{3,4}\.python\.org'` matches only `'www.python.org'` and `'wwww.python.org'`.

■ **Note** The term *match* is used loosely here to mean that the pattern matches the entire string. The `match` function (see Table 10-9) requires only that the pattern matches the beginning of the string.

The Beginning and End of a String

Until now, you've only been looking at a pattern matching an entire string, but you can also try to find a substring that matches the pattern, such as the substring `'www'` of the string `'www.python.org'` matching the pattern `'w+'`. When you're searching for substrings like this, it can sometimes be useful to anchor this substring either at the beginning or the end of the full string. For example, you might want to match `'ht+p'` at the beginning of a string, but not anywhere else. Then you use a caret (`'^'`) to mark the beginning. For example, `'^ht+p'` would match `'http://python.org'` (and `'htttttp://python.org'`, for that matter) but not `'www.http.org'`. Similarly, the end of a string may be indicated by the dollar sign (`$`).

Note For a complete listing of regular expression operators, see the section "Regular Expression Syntax" in the Python Library Reference (`http://python.org/doc/lib/re-syntax.html`).

Contents of the re Module

Knowing how to write regular expressions isn't much good if you can't use them for anything. The `re` module contains several useful functions for working with regular expressions. Some of the most important ones are described in Table 10-9.

Table 10-9. *Some Important Functions in the re Module*

Function	Description
`compile(pattern[, flags])`	Creates a pattern object from a string with a regular expression
`search(pattern, string[, flags])`	Searches for `pattern` in `string`
`match(pattern, string[, flags])`	Matches `pattern` at the beginning of `string`
`split(pattern, string[, maxsplit=0])`	Splits a `string` by occurrences of `pattern`
`findall(pattern, string)`	Returns a list of all occurrences of `pattern` in `string`
`sub(pat, repl, string[, count=0])`	Substitutes occurrences of `pat` in `string` with `repl`
`escape(string)`	Escapes all special regular expression characters in `string`

The function `re.compile` transforms a regular expression (written as a string) to a pattern object, which can be used for more efficient matching. If you use regular expressions represented as strings when you call functions such as `search` or `match`, they must be transformed into regular expression objects internally anyway. By doing this once, with the `compile` function, this step is no longer necessary each time you use the pattern. The pattern objects have the searching/matching functions as methods, so `re.search(pat, string)` (where `pat` is a regular expression written as a string) is equivalent to `pat.search(string)` (where `pat` is a pattern object created with `compile`). Compiled regular expression objects can also be used in the normal `re` functions.

The function `re.search` searches a given string to find the first substring, if any, that matches the given regular expression. If one is found, a `MatchObject` (evaluating to true) is returned; otherwise, `None` (evaluating to false) is returned. Due to the nature of the return values, the function can be used in conditional statements, like this:

```
if re.search(pat, string):
    print 'Found it!'
```

However, if you need more information about the matched substring, you can examine the returned `MatchObject`. (More about `MatchObject` in the next section.)

The function `re.match` tries to match a regular expression at the beginning of a given string. So `re.match('p', 'python')` returns true (a match object), while `re.match('p', 'www.python.org')` returns false (`None`).

Note The `match` function will report a match if the pattern matches the `beginning` of a string; the pattern is *not* required to match the entire string. If you want to do that, you need to add a dollar sign to the end of your pattern. The dollar sign will match the end of the string and thereby "stretch out" the match.

The function `re.split` splits a string by the occurrences of a pattern. This is similar to the string method `split`, except that you allow full regular expressions instead of only a fixed separator string. For example, with the string method `split`, you could split a string by the occurrences of the string `', '` but with `re.split` you can split on any sequence of space characters and commas:

```
>>> some_text = 'alpha, beta,,,,gamma   delta'
>>> re.split('[, ]+', some_text)
['alpha', 'beta', 'gamma', 'delta']
```

Note If the pattern contains parentheses, the parenthesized groups are interspersed between the split substrings. For example, `re.split('o(o)', 'foobar')` would yield `['f', 'o', 'bar']`.

As you can see from this example, the return value is a list of substrings. The `maxsplit` argument indicates the maximum number of splits allowed:

```
>>> re.split('[, ]+', some_text, maxsplit=2)
['alpha', 'beta', 'gamma   delta']
>>> re.split('[, ]+', some_text, maxsplit=1)
['alpha', 'beta,,,,gamma   delta']
```

The function `re.findall` returns a list of all occurrences of the given pattern. For example, to find all words in a string, you could do the following:

```
>>> pat = '[a-zA-Z]+'
>>> text = '"Hm... Err -- are you sure?" he said, sounding insecure.'
>>> re.findall(pat, text)
['Hm', 'Err', 'are', 'you', 'sure', 'he', 'said', 'sounding', 'insecure']
```

Or you could find the punctuation:

```
>>> pat = r'[.?\-",]+'
>>> re.findall(pat, text)
['"', '...', '--', '?"', ',', '.']
```

Note that the dash (`-`) has been escaped so Python won't interpret it as part of a character range (such as `a-z`).

The function `re.sub` is used to substitute the leftmost, nonoverlapping occurrences of a pattern with a given replacement. Consider the following example:

```
>>> pat = '{name}'
>>> text = 'Dear {name}...'
>>> re.sub(pat, 'Mr. Gumby', text)
'Dear Mr. Gumby...'
```

See the section "Group Numbers and Functions in Substitutions" later in this chapter for information about how to use this function more effectively.

The function `re.escape` is a utility function used to escape all the characters in a string that might be interpreted as a regular expression operator. Use this if you have a long string with a lot of these special characters and you want to avoid typing a lot of backslashes, or if you get a string from a user (for example, through the `raw_input` function) and want to use it as a part of a regular expression. Here is an example of how it works:

```
>>> re.escape('www.python.org')
'www\\.python\\.org'
>>> re.escape('But where is the ambiguity?')
'But\\ where\\ is\\ the\\ ambiguity\\?'
```

■ **Note** In Table 10-9, you'll notice that some of the functions have an optional parameter called `flags`. This parameter can be used to change how the regular expressions are interpreted. For more information about this, see the section about the `re` module in the Python Library Reference (`http://python.org/doc/lib/module-re.html`). The flags are described in the subsection "Module Contents."

Match Objects and Groups

The `re` functions that try to match a pattern against a section of a string all return `MatchObject` objects when a match is found. These objects contain information about the substring that matched the pattern. They also contain information about which parts of the pattern matched which parts of the substring. These parts are called *groups*.

A group is simply a subpattern that has been enclosed in parentheses. The groups are numbered by their left parenthesis. Group zero is the entire pattern. So, in this pattern:

```
'There (was a (wee) (cooper)) who (lived in Fyfe)'
```

the groups are as follows:

```
0  There was a wee cooper who lived in Fyfe
1  was a wee cooper
2  wee
3  cooper
4  lived in Fyfe
```

Typically, the groups contain special characters such as wildcards or repetition operators, and thus you may be interested in knowing what a given group has matched. For example, in this pattern:

```
r'www\.(.+)\.com$'
```

group 0 would contain the entire string, and group 1 would contain everything between `'www.'` and `'.com'`. By creating patterns like this, you can extract the parts of a string that interest you.

Some of the more important methods of re match objects are described in Table 10-10.

Table 10-10. *Some Important Methods of re Match Objects*

Method	Description
`group([group1, ...])`	Retrieves the occurrences of the given subpatterns (*groups*)
`start([group])`	Returns the starting position of the occurrence of a given group
`end([group])`	Returns the ending position (an exclusive limit, as in slices) of the occurrence of a given group
`span([group])`	Returns both the beginning and ending positions of a group

The method `group` returns the (sub)string that was matched by a given group in the pattern. If no group number is given, group 0 is assumed. If only a single group number is given (or you just use the default, 0), a single string is returned. Otherwise, a tuple of strings corresponding to the given group numbers is returned.

■ **Note** In addition to the entire match (group 0), you can have only 99 groups, with numbers in the range 1–99.

The method `start` returns the starting index of the occurrence of the given group (which defaults to 0, the whole pattern).

The method `end` is similar to `start`, but returns the ending index plus one.

The method `span` returns the tuple `(start, end)` with the starting and ending indices of a given group (which defaults to 0, the whole pattern).

Consider the following example:

```
>>> m = re.match(r'www\.(.*)\..{3}', 'www.python.org')
>>> m.group(1)
'python'
>>> m.start(1)
4
>>> m.end(1)
10
>>> m.span(1)
(4, 10)
```

Group Numbers and Functions in Substitutions

In the first example using `re.sub`, I simply replaced one substring with another—something I could easily have done with the `replace` string method (described in the section "String Methods" in Chapter 3). Of course, regular expressions are useful because they allow you to search in a more flexible manner, but they also allow you to perform more powerful substitutions.

The easiest way to harness the power of `re.sub` is to use group numbers in the substitution string. Any escape sequences of the form `'\\n'` in the replacement string are replaced by the string matched by group `n` in the pattern. For example, let's say you want to replace words of the form `'*something*'` with `'<em>something</em>'`, where the former is a normal way of expressing emphasis in plain-text documents (such as email), and the latter is the corresponding HTML code (as used in web pages). Let's first construct the regular expression:

```
>>> emphasis_pattern = r'\*([^\*]+)\*'
```

Note that regular expressions can easily become hard to read, so using meaningful variable names (and possibly a comment or two) is important if anyone (including you!) is going to view the code at some point.

■ **Tip** One way to make your regular expressions more readable is to use the `VERBOSE` flag in the `re` functions. This allows you to add whitespace (space characters, tabs, newlines, and so on) to your pattern, which will be ignored by `re`—except when you put it in a character class or escape it with a backslash. You can also put comments in such verbose regular expressions. The following is a pattern object that is equivalent to the emphasis pattern, but which uses the `VERBOSE` flag:

```
>>> emphasis_pattern = re.compile(r'''
...         \*      # Beginning emphasis tag -- an asterisk
...         (       # Begin group for capturing phrase
...         [^\*]+  # Capture anything except asterisks
...         )       # End group
...         \*      # Ending emphasis tag
...         ''', re.VERBOSE)
...
```

Now that I have my pattern, I can use `re.sub` to make my substitution:

```
>>> re.sub(emphasis_pattern, r'<em>\1</em>', 'Hello, *world*!')
'Hello, <em>world</em>!'
```

As you can see, I have successfully translated the text from plain text to HTML.

But you can make your substitutions even more powerful by using a function as the replacement. This function will be supplied with the `MatchObject` as its only parameter, and the string it returns will be used as the replacement. In other words, you can do whatever you want to the matched substring, and do elaborate processing to generate its replacement. What possible use could you have for such power, you ask? Once you start experimenting with regular expressions, you will surely find countless uses for this mechanism. For one application, see the section "A Sample Template System" a little later in the chapter.

GREEDY AND NONGREEDY PATTERNS

The repetition operators are by default *greedy*, which means that they will match as much as possible. For example, let's say I rewrote the emphasis program to use the following pattern:

```
>>> emphasis_pattern = r'\*(.+)\*'
```

This matches an asterisk, followed by one or more characters, and then another asterisk. Sounds perfect, doesn't it? But it isn't:

```
>>> re.sub(emphasis_pattern, r'<em>\1</em>', '*This* is *it*!')
'<em>This* is *it</em>!'
```

As you can see, the pattern matched everything from the first asterisk to the last—including the two asterisks between! This is what it means to be greedy: take everything you can.

In this case, you clearly don't want this overly greedy behavior. The solution presented in the preceding text (using a character set matching anything *except* an asterisk) is fine when you know that one specific letter is illegal. But let's consider another scenario. What if you used the form `'**something**'` to signify emphasis? Now it shouldn't be a problem to include single asterisks inside the emphasized phrase. But how do you avoid being too greedy?

Actually, it's quite easy—you just use a nongreedy version of the repetition operator. All the repetition operators can be made nongreedy by putting a question mark after them:

```
>>> emphasis_pattern = r'\*\*(.+?)\*\*'
>>> re.sub(emphasis_pattern, r'<em>\1</em>', '**This** is **it**!')
'<em>This</em> is <em>it</em>!'
```

Here I've used the operator `+?` instead of `+`, which means that the pattern will match one or more occurrences of the wildcard, as before. However, it will match as few as it can, because it is now nongreedy. So, it will match only the minimum needed to reach the next occurrence of `'\*\*'`, which is the end of the pattern. As you can see, it works nicely.

Finding the Sender of an Email

Have you ever saved an email as a text file? If you have, you may have seen that it contains a lot of essentially unreadable text at the top, similar to that shown in Listing 10-9.

Listing 10-9. *A Set of (Fictitious) Email Headers*

```
From foo@bar.baz  Thu Dec 20 01:22:50 2008
Return-Path: <foo@bar.baz>
Received: from xyzzy42.bar.com (xyzzy.bar.baz [123.456.789.42])
        by frozz.bozz.floop (8.9.3/8.9.3) with ESMTP id BAA25436
        for <magnus@bozz.floop>; Thu, 20 Dec 2004 01:22:50 +0100 (MET)
Received: from [43.253.124.23] by bar.baz
          (InterMail vM.4.01.03.27 201-229-121-127-20010626) with ESMTP
          id <20041220002242.ADASD123.bar.baz@[43.253.124.23]>;
          Thu, 20 Dec 2004 00:22:42 +0000
User-Agent: Microsoft-Outlook-Express-Macintosh-Edition/5.02.2022
Date: Wed, 19 Dec 2008 17:22:42 -0700
Subject: Re: Spam
From: Foo Fie <foo@bar.baz>
To: Magnus Lie Hetland <magnus@bozz.floop>
CC: <Mr.Gumby@bar.baz>
Message-ID: <B8467D62.84F%foo@baz.com>
In-Reply-To: <20041219013308.A2655@bozz.floop>
Mime-version: 1.0
Content-type: text/plain; charset="US-ASCII"
Content-transfer-encoding: 7bit
Status: RO
Content-Length: 55
Lines: 6

So long, and thanks for all the spam!


Yours,

Foo Fie
```

Let's try to find out who this email is from. If you examine the text, I'm sure you can figure it out in this case (especially if you look at the signature at the bottom of the message itself, of course). But can you see a general pattern? How do you extract the name of the sender, without the email address? Or how can you list all the email addresses mentioned in the headers? Let's handle the former task first.

The line containing the sender begins with the string `'From: '` and ends with an email address enclosed in angle brackets (`<` and `>`). You want the text found between those brackets. If you use the `fileinput` module, this should be an easy task. A program solving the problem is shown in Listing 10-10.

■ **Note** You could solve this problem without using regular expressions if you wanted. You could also use the `email` module.

Listing 10-10. *A Program for Finding the Sender of an Email*

```
# find_sender.py
import fileinput, re
pat = re.compile('From: (.*) <.*?>$')
for line in fileinput.input():
    m = pat.match(line)
    if m: print m.group(1)
```

You can then run the program like this (assuming that the email message is in the text file `message.eml`):

```
$ python find_sender.py message.eml
Foo Fie
```

You should note the following about this program:

- I compile the regular expression to make the processing more efficient.
- I enclose the subpattern I want to extract in parentheses, making it a group.
- I use a nongreedy pattern to so the email address matches only the last pair of angle brackets (just in case the name contains some brackets).
- I use a dollar sign to indicate that I want the pattern to match the entire line, all the way to the end.
- I use an `if` statement to make sure that I did in fact match something before I try to extract the match of a specific group.

To list all the email addresses mentioned in the headers, you need to construct a regular expression that matches an email address but nothing else. You can then use the method `findall` to find all the occurrences in each line. To avoid duplicates, you keep the addresses in a set (described earlier in this chapter). Finally, you extract the keys, sort them, and print them out:

```
import fileinput, re
pat = re.compile(r'[a-z\-\.]+@[a-z\-\.]+', re.IGNORECASE)
addresses = set()
```

```
for line in fileinput.input():
    for address in pat.findall(line):
        addresses.add(address)
for address in sorted(addresses):
    print address
```

The resulting output when running this program (with the email message in Listing 10-9 as input) is as follows:

```
Mr.Gumby@bar.baz
foo@bar.baz
foo@baz.com
magnus@bozz.floop
```

Note that when sorting, uppercase letters come before lowercase letters.

Note I haven't adhered strictly to the problem specification here. The problem was to find the addresses in the header, but in this case the program finds all the addresses in the entire file. To avoid that, you can call `fileinput.close()` if you find an empty line, because the header can't contain empty lines. Alternatively, you can use `fileinput.nextfile()` to start processing the next file, if there is more than one.

A Sample Template System

A *template* is a file you can put specific values into to get a finished text of some kind. For example, you may have a mail template requiring only the insertion of a recipient name. Python already has an advanced template mechanism: string formatting. However, with regular expressions, you can make the system even more advanced. Let's say you want to replace all occurrences of `'[something]'` (the "fields") with the result of evaluating `something` as an expression in Python. Thus, this string:

```
'The sum of 7 and 9 is [7 + 9].'
```

should be translated to this:

```
'The sum of 7 and 9 is 16.'
```

Also, you want to be able to perform assignments in these fields, so that this string:

```
'[name="Mr. Gumby"]Hello, [name]'
```

should be translated to this:

```
'Hello, Mr. Gumby'
```

This may sound like a complex task, but let's review the available tools:

- You can use a regular expression to match the fields and extract their contents.
- You can evaluate the expression strings with `eval`, supplying the dictionary containing the scope. You do this in a `try`/`except` statement. If a `SyntaxError` is raised, you probably have a statement (such as an assignment) on your hands and should use exec instead.
- You can execute the assignment strings (and other statements) with exec, storing the template's scope in a dictionary.
- You can use `re.sub` to substitute the result of the evaluation into the string being processed.

Suddenly, it doesn't look so intimidating, does it?

Tip If a task seems daunting, it almost always helps to break it down into smaller pieces. Also, take stock of the tools at your disposal for ideas on how to solve your problem.

See Listing 10-11 for a sample implementation.

Listing 10-11. *A Template System*

```
# templates.py

import fileinput, re

# Matches fields enclosed in square brackets:
field_pat = re.compile(r'\[(.+?)\]')


# We'll collect variables in this:
scope = {}

# This is used in re.sub:
def replacement(match):
    code = match.group(1)
    try:
        # If the field can be evaluated, return it:
        return str(eval(code, scope))
    except SyntaxError:
        # Otherwise, execute the assignment in the same scope...
        exec code in scope
        # ...and return an empty string:
        return ''

# Get all the text as a single string:
```

```
# (There are other ways of doing this; see Chapter 11)
lines = []
for line in fileinput.input():
    lines.append(line)
text = ''.join(lines)

# Substitute all the occurrences of the field pattern:
print field_pat.sub(replacement, text)
```

Simply put, this program does the following:

- Define a pattern for matching fields.
- Create a dictionary to act as a scope for the template.
- Define a replacement function that does the following:
  - Grabs group 1 from the match and puts it in code.
  - Tries to evaluate code with the scope dictionary as namespace, converts the result to a string, and returns it. If this succeeds, the field was an expression and everything is fine. Otherwise (that is, a `SyntaxError` is raised), go to the next step.
  - Execute the field in the same namespace (the scope dictionary) used for evaluating expressions, and then returns an empty string (because the assignment doesn't evaluate to anything).
- Use `fileinput` to read in all available lines, put them in a list, and join them into one big string.
- Replace all occurrences of `field_pat` using the replacement function in `re.sub`, and print the result.

■ **Note** In previous versions of Python, it was much more efficient to put the lines into a list and then join them at the end than to do something like this:

```
text = ''
for line in fileinput.input():
    text += line
```

Although this looks elegant, each assignment must create a new string, which is the old string with the new one appended, which can lead to a waste of resources and make your program slow. In older versions of Python, the difference between this and using `join` could be huge. In more recent versions, using the `+=` operator may, in fact, be *faster*. If performance is important to you, you could try out both solutions. And if you want a more elegant way to read in all the text of a file, take a peek at Chapter 11.

So, I have just created a really powerful template system in only 15 lines of code (not counting whitespace and comments). I hope you're starting to see how powerful Python becomes when you use the standard libraries. Let's finish this example by testing the template system. Try running it on the simple file shown in Listing 10-12.

Listing 10-12. *A Simple Template Example*

```
[x = 2]
[y = 3]
The sum of [x] and [y] is [x + y].
```

You should see this:

```


The sum of 2 and 3 is 5.

```

■ **Note** It may not be obvious, but there are three empty lines in the preceding output—two above and one below the text. Although the first two fields have been replaced by empty strings, the newlines following them are still there. Also, the `print` statement adds a newline, which accounts for the empty line at the end.

But wait, it gets better! Because I have used `fileinput`, I can process several files in turn. That means that I can use one file to define values for some variables, and then another file as a template where these values are inserted. For example, I might have one file with definitions as in Listing 10-13, named `magnus.txt`, and a template file as in Listing 10-14, named `template.txt`.

Listing 10-13. *Some Template Definitions*

```
[name     = 'Magnus Lie Hetland' ]
[email    = 'magnus@foo.bar'     ]
[language = 'python'             ]
```

Listing 10-14. *A Template*

```
[import time]
Dear [name],

I would like to learn how to program. I hear you use
the [language] language a lot -- is it something I
should consider?

And, by the way, is [email] your correct email address?
```

```
Fooville, [time.asctime()]

Oscar Frozzbozz
```

The `import time` isn't an assignment (which is the statement type I set out to handle), but because I'm not being picky and just use a simple `try`/`except` statement, my program supports any statement or expression that works with `eval` or exec. You can run the program like this (assuming a UNIX command line):

```
$ python templates.py magnus.txt template.txt
```

You should get some output similar to the following:

```




Dear Magnus Lie Hetland,

I would like to learn how to program. I hear you use
the python language a lot -- is it something I
should consider?

And, by the way, is magnus@foo.bar your correct email address?


Fooville, Wed Apr 24 20:34:29 2008

Oscar Frozzbozz
```

Even though this template system is capable of some quite powerful substitutions, it still has some flaws. For example, it would be nice if you could write the definition file in a more flexible manner. If it were executed with `execfile`, you could simply use normal Python syntax. That would also fix the problem of getting all those blank lines at the top of the output.

Can you think of other ways of improving the program? Can you think of other uses for the concepts used in this program? The best way to become really proficient in any programming language is to play with it—test its limitations and discover its strengths. See if you can rewrite this program so it works better and suits your needs.

■ **Note** There is, in fact, a perfectly good template system available in the standard libraries, in the `string` module. Just take a look at the `Template` class, for example.

Other Interesting Standard Modules

Even though this chapter has covered a lot of material, I have barely scratched the surface of the standard libraries. To tempt you to dive in, I'll quickly mention a few more cool libraries:

`functools`: Here, you can find functionality that lets you use a function with only *some* of its parameters (partial evaluation), filling in the remaining ones at a later time. In Python 3.0, this is where you will find `filter` and `reduce`.

`difflib`: This library enables you to compute how similar two sequences are. It also enables you to find the sequences (from a list of possibilities) that are "most similar" to an original sequence you provide. `difflib` could be used to create a simple searching program, for example.

`hashlib`: With this module, you can compute small "signatures" (numbers) from strings. And if you compute the signatures for two different strings, you can be almost certain that the two signatures will be different. You can use this on large text files. These modules have several uses in cryptography and security.[5]

`csv`: CSV is short for comma-separated values, a simple format used by many applications (for example, many spreadsheets and database programs) to store tabular data. It is mainly used when exchanging data between different programs. The `csv` module lets you read and write CSV files easily, and it handles some of the trickier parts of the format quite transparently.

`timeit`, `profile`, and `trace`: The `timeit` module (with its accompanying command-line script) is a tool for measuring the time a piece of code takes to run. It has some tricks up its sleeve, and you probably ought to use it rather than the `time` module for performance measurements. The `profile` module (along with its companion module, `pstats`) can be used for a more comprehensive analysis of the efficiency of a piece of code. The `trace` module (and program) can give you a coverage analysis (that is, which parts of your code are executed and which are not). This can be useful when writing test code, for example.

`datetime`: If the `time` module isn't enough for your time-tracking needs, it's quite possible that `datetime` will be. It has support for special date and time objects, and allows you to construct and combine these in various ways. The interface is in many ways a bit more intuitive than that of the `time` module.

`itertools`: Here, you have a lot of tools for creating and combining iterators (or other iterable objects). There are functions for chaining iterables, for creating iterators that return consecutive integers forever (similar to `range`, but without an upper limit), to cycle through an iterable repeatedly, and other useful stuff.

`logging`: Simply using `print` statements to figure out what's going on in your program can be useful. If you want to keep track of things even without having a lot of debugging output, you might write this information to a log file. This module gives you a standard set of tools for managing one or more central logs, with several levels of priority for your log messages, among other things.

5. See also the `md5` and `sha` modules.

getopt and optparse: In UNIX, command-line programs are often run with various *options* or *switches*. (The Python interpreter is a typical example.) These will all be found in sys.argv, but handling these correctly yourself is far from easy. The getopt library is a tried-and-true solution to this problem, while optparse is newer, more powerful, and much easier to use.

cmd: This module enables you to write a command-line interpreter, somewhat like the Python interactive interpreter. You can define your own commands that the user can execute at the prompt. Perhaps you could use this as the user interface to one of your programs?

A Quick Summary

In this chapter, you've learned about modules: how to create them, how to explore them, and how to use some of those included in the standard Python libraries.

Modules: A module is basically a subprogram whose main function is to *define things*, such as functions, classes, and variables. If a module contains any test code, it should be placed in an if statement that checks whether `__name__=='__main__'`. Modules can be imported if they are in the PYTHONPATH. You import a module stored in the file foo.py with the statement `import foo`.

Packages: A package is just a module that contains other modules. Packages are implemented as directories that contain a file named `__init__.py`.

Exploring modules: After you have imported a module into the interactive interpreter, you can explore it in many ways. Among them are using dir, examining the `__all__` variable, and using the help function. The documentation and the source code can also be excellent sources of information and insight.

The standard library: Python comes with several modules included, collectively called the standard library. Some of these were reviewed in this chapter:

- sys: A module that gives you access to several variables and functions that are tightly linked with the Python interpreter.
- os: A module that gives you access to several variables and functions that are tightly linked with the operating system.
- fileinput: A module that makes it easy to iterate over the lines of several files or streams.
- sets, heapq, and deque: Three modules that provide three useful data structures. Sets are also available in the form of the built-in type set.
- time: A module for getting the current time, and for manipulating and formatting times and dates.

- `random`: A module with functions for generating random numbers, choosing random elements from a sequence, and shuffling the elements of a list.
- `shelve`: A module for creating a persistent mapping, which stores its contents in a database with a given file name.
- `re`: A module with support for regular expressions.

If you are curious to find out more about modules, I again urge you to browse the Python Library Reference (`http://python.org/doc/lib`). It's really interesting reading.

New Functions in This Chapter

Function	Description
`dir(obj)`	Returns an alphabetized list of attribute names
`help([obj])`	Provides interactive help or help about a specific object
`reload(module)`	Returns a reloaded version of a module that has already been imported. To be abolished in Python 3.0.

What Now?

If you have grasped at least a few of the concepts in this chapter, your Python prowess has probably taken a great leap forward. With the standard libraries at your fingertips, Python changes from powerful to extremely powerful. With what you have learned so far, you can write programs to tackle a wide range of problems. In the next chapter, you learn more about using Python to interact with the outside world of files and networks, and thereby tackle problems of greater scope.