Python

Anthony Briggs

Foreword by Steve Holden

MANNING

Hello! Python

by Anthony Briggs

Chapters 3, 7, and 10
ISBN 9781935182085

To pre-order or learn more about this book go to www.manning.com/briggs/

Brief contents

3

Interacting with the world

This chapter covers

- *What libraries are*
- *How to use libraries, including Python's standard library*
- *An example program that uses Python's* `os` *and* `sys` *libraries*
- *Python's dictionary data type*

One of the key strengths of Python is its standard library. Installed along with Python, the standard library is a large suite of program code that covers common tasks like finding and iterating over files, handling user input, downloading and parsing pages from the web, and accessing databases. If you make good use of the standard library, you can often write programs in a fraction of the time that it would take you otherwise, with less typing and far fewer bugs.

GUIDO'S TIME MACHINE

The standard library is so extensive that one of the running jokes in the Python community is that Guido (the inventor of Python) owns a time machine. When someone asks for a module that performs a particular task, Guido hops in his time machine, travels back to the beginning of Python, and—"poof!"—it's already there.

In chapter 2, you used the `choice` function in Python's random module to pick something from a list, so you've already used a library. In this chapter, we'll go in depth and find out more about how to use libraries, what other libraries exist, and how to use Python's documentation to learn about specific libraries. In the process, you'll also pick up a few other missing pieces of Python, such as how you can read files, and you'll discover another of Python's data types—the dictionary.

The program in this chapter solves a common problem that you've probably faced before: you have two similar folders (perhaps one's a backup of your holiday photos), and you'd like to know which files differ between the two of them. You'll be tackling this program from a different angle than in chapter 2, though. Rather than write most of your own code, you'll be using Python to glue together several standard libraries to get the job done.

Let's start by learning more about Python libraries.

"Batteries included": Python's libraries

What are libraries used for? Normally, they're geared toward a single purpose, such as sending data via a network, writing CSV or Excel files, displaying graphics, or handling user input. But libraries can grow to cover a large number of related functions; there's no hard or fast rule.

LIBRARY Program code that is written so that it can be used by other programs.

Python libraries can do anything that Python can, and more. In some (rare) cases, like intensive number crunching or graphics processing, Python can be too slow to do what you need; but it's possible to extend Python to use libraries written in C.

In this section, you'll learn about Python's standard library, see which other libraries you can add, try them out, and get a handle on exploring a single library.

Python's standard library

Python installs with a large number of libraries that cover most of the common tasks that you'll need to handle when programming.

If you find yourself facing a tricky problem, it's a good habit to read through the modules in Python's standard library to see if something covers what you need to do. The Python manuals are installed with the standard Windows installer, and there's normally a documentation package when installing under Linux. The latest versions are also available at http://docs.python.org if you're connected to the internet. Being able to use a good library can save you hours of programming, so 5 or 10 minutes up front can pay big dividends.

The Python standard library is large enough that it can be hard to find what you need. Another way to learn it is to take it one piece at a time. The Python Module of the Week blog (www.doughellmann.com/PyMOTW/) covers most of Python's standard library and is an excellent way to familiarize yourself with what's available, because it often contains far more explanation than the standard Python documentation.

Other libraries

You're not limited to the libraries that Python installs. It's easy to download and install extra libraries to add the additional functionality that you need. Most add-on libraries come with their own installers or installation script; those that don't can normally be copied into the library folder of your Python directory. You'll find out how to install libraries in later chapters, Once the extra libraries are installed, they behave like Python's built-in ones; there's no special syntax that you need to know.

Using libraries

Once installed, using a library is straightforward: just add an import line at the top of the script. There are several ways to do it, but here are the three most common.

INCLUDE EVERYTHING

You can include everything from a library into your script by using a line like

```
from os import *
```

This will read everything from the `os` module and drop it straight into your script. If you want to use the `access` function from `os`, you can use it directly, like `access("myfile.txt")`. This has the advantage of saving some typing, but with serious downsides:

- You now have a lot of strange functions in your script.
- Worse, if you include more than one module in this way, then you run the risk of functions in the later module overwriting the functions from the first module—ouch!
- Finally, it's much harder to remember which module a particular function came from, which makes your program difficult to maintain.

Fortunately, there are much better ways to import modules.

INCLUDE THE MODULE

A better way to handle things is with a line like `import os`. This will import everything in `os` but make it available only through an `os` object. Now, if you want to use the access function, you need to use it like this: `os.access("myfile.txt")`. It's a bit more typing, but you won't run the risk of overwriting any other functions.

INCLUDE ONLY THE BITS THAT YOU WANT

If you're using the functions from a module a lot, you might find that your code becomes hard to read, particularly if the module has a long name. There's a third option in this case: you can use a line like `from os import access`. This will import directly so that you can use `access ("myfile.txt")` without the module name, but only include the access function, not the entire `os` module. You still run the risk of overwriting

with a later module, but, because you have to specify the functions and there are fewer of them, it's much less likely.

What's in a library, anyway?

Libraries can include anything that comes with standard Python—variables, functions, and classes, as well as Python code that should be run when the library is loaded. You're not limited in any way; anything that's legal in Python is fine to put in a library. When using a library for the first time, it helps to know what's in it, and what it does. There are two main ways to find out.

TIP `dir` and `help` aren't only useful for libraries. You can try them on all of the Python objects, such as classes and functions. They even support strings and numbers.

READ THE FINE MANUAL

Python comes with a detailed manual on every aspect of its use, syntax, standard libraries—pretty much everything you might need to reference when writing programs. It doesn't cover every possible use, but the majority of the standard library is there. If you have internet access, you can view it at http://docs.python.org, and it's normally installed alongside Python, too.

EXPLORATION

One useful function for finding out what a library contains is `dir()`. You can call it on any object to find out what methods it supports, but it's particularly useful with libraries. You can combine it with the `__doc__` special variable, which is set to the docstring defined for a function or method, to get a quick overview of a library's or class's methods and what they do. This combination is so useful that there's a shortcut called `help()` that is defined as one of Python's built-in functions.

For the details, you're often better off looking at the documentation; but if you only need to jog your memory, or if the documentation is patchy

or confusing, `dir()`, `__doc__`, and `help()` are much faster. The following listing is an example of looking up some information about the `os` library.

Listing 3.1 Finding out more about the os.path library

```
>>> import os
>>> dir(os.path)
['__all__', '__builtins__', '__doc__', '__file__',
 '__name__', '__package__', '_getfullpathname',
 'abspath', 'altsep', 'basename', 'commonprefix',
 'curdir', 'defpath', 'devnull', 'dirname', 'exists',
 'expanduser', 'expandvars', 'extsep', 'genericpath',
 'getatime', 'getctime', 'getmtime', 'getsize',
 'isabs', 'isdir', 'isfile', 'islink', 'ismount',
 'join', 'lexists', 'normcase', 'normpath', 'os',
 'pardir', 'pathsep', 'realpath', 'relpath', 'sep',
 'split', 'splitdrive', 'splitext', 'splitunc',
 'stat', 'supports_unicode_filenames', 'sys',
 'walk', 'warnings']
>>> print os.path.__doc__
Common pathname manipulations, WindowsNT/95 version.

Instead of importing this module directly, import os
and refer to this module as os.path.

>>> print os.path.isdir.__doc__
Return true if the pathname refers to an existing
directory.
>>> print os.path.isdir('c:/')
True
>>> print os.path.isdir('c:/windows/system.ini')
False

>>> help (os)
Help on module os:

NAME
    os - OS routines for Mac, NT, or Posix depending
    on what system we're on.

FILE
    c:\python26\lib\os.py
```

1 Import os

2 Explore os.path

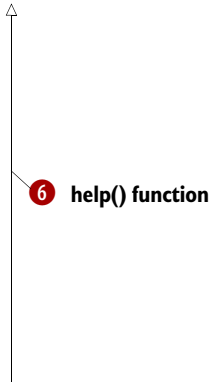
3 Docstring for os.path module

4 Docstring for the isdir function

5 Test functions

6 help() function

```
DESCRIPTION
    This exports:
      - all functions from posix, nt, os2, or ce,
        e.g. unlink, stat, etc.
      - os.path is one of the modules posixpath,
        or ntpath
      - os.name is 'posix', 'nt', 'os2', 'ce' or
        'riscos'
      - os.curdir is a string representing the
        current directory ('.' or ':')
      - os.pardir is a string representing the
        parent directory ('..' or '::')
```



First, you need to import the `os` module ❶. You can import `os.path` directly, but this is the way that it's normally done, so you'll have fewer surprises later. Next, you call the `dir()` function on `os.path`, to see what's in it ❷. The function will return a big list of function and variable names, including some built-in Python ones like `__doc__` and `__name__`.

Because you can see a `__doc__` variable in `os.path`, print it and see what it contains ❸. It's a general description of the `os.path` module and how it's supposed to be used.

If you look at the `__doc__` variable for a function in `os.path` ❹, it shows much the same thing—a short description of what the function is supposed to do.

Once you've found a function that you think does what you need, you can try it out to make sure ❺. Here, you're calling `os.path.isdir()` on a couple of different files and directories to see what it returns. For more complicated libraries, you might find it easier to write a short program rather than type it all in at the command line.

Finally, the output of the `help()` function ❻ contains all the same information that `__doc__` and `dir()` do, but printed nicely. It also looks through the whole object and returns all of its variables and methods without you having to look for them. You can press space or page up and down to read the output, and Q when you want to go back to the interpreter.

In practice, it can often take a combination of these methods before you understand enough about the library for it to be useful. A quick overview of the library documentation, followed by some experimenting at the command line and a further read of the documentation, will provide you with some of the finer points once you understand how it all fits together. Also, bear in mind that you don't necessarily have to understand the entire library at once, as long as you can pick and choose the pieces you need.

Now that you know the basics of Python libraries, let's see what you can do with them.

Another way to ask questions

There's one thing that you need to know before you can start putting your program together. Actually, there are a couple of other things, but you can pick those up on the way. What you'd like to be able to do in order to begin is tell the computer which directories you want to compare. If this were a normal program, you'd probably have a graphical interface where you could click the relevant directories. But that sounds hard, so you'll pick something simpler to write: a command-line interface.

Using command-line arguments

Command-line arguments are often used in system-level programs. When you run a program from the command line, you can specify additional parameters by typing them after the program's name. In this case, you'll be typing in the names of the two directories that you want to compare; something like this:

```
python difference.py directory1 directory2
```

If you have spaces in your directory name, you can surround the parameters with quotation marks; otherwise, your operating system will interpret it as two different parameters:

```
python difference.py "My Documents\directory1" "My Documents\directory2"
```

Now that you have your parameters, what are you going to do with them?

Using the sys module

In order to read the parameters you've fed in, you'll need to use the `sys` module that comes with Python's standard library. `sys` deals with all sorts of system-related functionality, such as finding out which version of Python a script is running on, information about the script, paths, and so on. You'll be using `sys.argv`, which is an array containing the script's name and any parameters that it was called with. Your initial program is listing 3.2, which will be the starting point for the comparison script.

Listing 3.2 Reading parameters using `sys`

```
import sys

if len(sys.argv) < 3:
        print "You need to specify two directories:"
        print sys.argv[0], "<directory 1> <directory 2>"
        sys.exit()

directory1 = sys.argv[1]
directory2 = sys.argv[2]

print "Comparing:"
print directory1
print directory2
print
```

❶ Check parameters

❷ Store parameter values

❸ Debug strings

First, you check to make sure that the script has been called with enough parameters ❶. If there are too few, then you return an error to the user. Note also that you're using `sys.argv[0]` to find out what the name of your script is and `sys.exit` to end the program early.

Because you know now that there are at least two other values, you can store them for later use ❷. You could use `sys.argv` directly, but this way, you've got a nice variable name, which makes the program easier to understand.

Once you have the variables set, you can print them out ❸ to make sure they're what you're expecting. You can test it out by trying the commands from the section "Using command-line arguments." The script should respond back with whatever you've specified.

NOTE File objects are an important part of Python. Quite a few libraries use file-like objects to access other things, like web pages, strings, and the output returned from other programs.

If you're happy with the results, it's time to start building the program in the next section.

Reading and writing files

The next thing you'll need to do in your duplicate checker is to find your files and directories and open them to see if they're the same. Python has built-in support for handling files as well as good cross platform file and directory support via the `os` module. You'll be using both of these in your program.

Paths and directories (a.k.a. dude, where's my file?)

Before you open your file, you need to know where to find it. You want to find all of the files in a directory and open them, as well as any files in directories within that directory, and so on. That's pretty tricky if you're writing it yourself; fortunately, the `os` module has a function called `os.walk()` that does exactly what you want. The `os.walk()` function returns a list of all of the directories and files for a path. If you append listing 3.3 to the end of listing 3.2, it will call `os.walk()` on the directories that you've specified.

Listing 3.3 Using `os.walk()`

```
import os

for directory in [directory1, directory2]:
    if not os.access(directory, os.F_OK):
        print directory, "isn't a valid directory!"
        sys.exit()

    print "Directory", directory
    for item in os.walk(directory):
        print item
    print
```

❶ Don't repeat yourself

❷ Input checking

❸ Walk over directory

You're going to be doing the same thing for both `directory1` and `directory2` ❶. You could repeat your code over again for `directory2`, but if you want to change it later, you'll have to change it in two places. Worse, you could accidentally change one but not the other, or change it slightly differently. A better way is to use the directory names in a `for` loop like this, so you can reuse the code within the loop.

It's good idea to check the input that your script's been given ❷. If there's something amiss, then exit with a reasonable error message to let the user know what's gone wrong.

❸ is the part where you walk over the directory. For now, you're printing the raw output that's returned from `os.walk()`, but in a minute you'll do something with it.

I've set up two test directories on my computer with a few directories that I found lying around. It's probably a good idea for you to do the same, so you can test your program and know you're making progress.

If you run the program so far, you should see something like the following output:

```
D:\code>python difference_engine_2_os.py . test1 test2
Comparing:
test1
test2

Directory test1
('C:\\test1', ['31123', 'My Music', 'My Pictures', 'test'], [])
('C:\\test1\\31123', [], [])
('C:\\test1\\My Music', [], ['Desktop.ini', 'Sample Music.lnk'])
('C:\\test1\\My Pictures', [], ['Sample Pictures.lnk'])
('C:\\test1\\test', [], ['foo1.py', 'foo1.pyc', 'foo2.py', 'foo2.pyc',
  'os.walk.py', 'test.py'])

Directory test2
('C:\\test2', ['31123', 'My Music', 'My Pictures', 'test'], [])
('C:\\test2\\31123', [], [])
('C:\\test2\\My Music', [], ['Desktop.ini', 'Sample Music.lnk'])
```

```
('C:\\test2\\My Pictures', [], ['Sample Pictures.lnk'])
('C:\\test2\\test', [], ['foo1.py', 'foo1.pyc', 'foo2.py', 'foo2.pyc',
'os.walk.py', 'test.py'])
```

In Python strings, some special characters can be created by using a backslash in front of another character. If you want a tab character, for example, you can put `\t` into your string. When Python prints it, it will be replaced with a literal tab character. If you do want a backslash, though—as you do here—then you'll need to use two backslashes, one after the other.

The output for each line gives you the name of a directory within your path, then a list of directories within that directory, then a list of the files … handy, and definitely beats writing your own version.

Paths

If you want to use a file or directory, you'll need what's called a *path*. A path is a string that gives the exact location of a file, including any directories that contain it. For example, the path to Python on my computer is C:\python26\python.exe, which looks like `"C:\\python26\\python.exe"` when expressed as a Python string.

If you wanted a path for foo2.py in the last line of the previous listing, you can use `os.path.join('C:\\test2\\test', 'foo2.py')`, to get a path that looks like `'C:\\test2\\test\\foo2.py'`. You'll see more of the details when you start putting your program together in a minute.

TIP One thing to keep in mind when using paths is that the separator will be different depending on which platform you're using. Windows uses a backslash (\) character, and Linux and Macintosh use a forward slash (/). To make sure your programs work on all three systems, it's a good idea to get in the habit of using the `os.path.join()` function, which takes a list of strings and joins them with whatever the path separator is on the current computer.

Once you have the location of your file, the next step is opening it.

File, open!

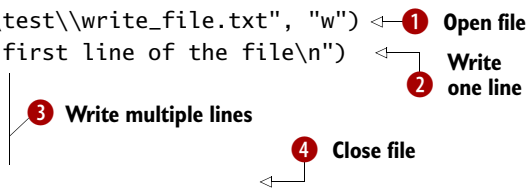
To open a file in Python, you can use the `file()` or `open()` built-in function. They're exactly the same behind the scenes, so it doesn't matter which one you use. If the file exists and you can open it, you'll get back a file object, which you can read using the `read()` or `readlines()` method. The only difference between `read()` and `readlines()` is that `readlines()` will split the file into strings, but `read()` will return the file as one big string. This code shows how you can open a file and read its contents:

```
read_file = file(os.path.join("c:\\test1\\test", "foo2.py"))
file_contents = list(read_file.readlines())
print "Read in", len(file_contents), "lines from foo2.py"
print "The first line reads:", file_contents[0]
```

First, create a path using `os.path.join()`, and then use it to open the file at that location. You'll want to put in the path to a text file that exists on your computer. `read_file` will now be a file object, so you can use the `readlines()` method to read the entire contents of the file. You're also turning the file contents into a list using the `list()` function. You don't normally treat files like this, but it helps to show you what's going on. `file_contents` is a list now, so you can use the `len()` function to see how many lines it has, and print the first line by using an index of 0.

Although you won't be using it in your program, it's also possible to write text into a file as well as read from it. To do this, you'll need to open the file with a write mode instead of the default read-only mode, and use the `write()` or `writelines()` function of the file object. Here's a quick example:

```
write_file = file("C:\\test2\\test\\write_file.txt", "w")
write_file.write("This is the first line of the file\n")
write_file.writelines(
    ["and the second\n",
    "and the third!\n"])
write_file.close()
```



You're using the same `file()` function you used before, but here you're feeding it an extra parameter, the string `"w"`, to tell Python that you want to open it for writing 1.

Once you have the file object back, you can write to it by using the `.write()` method, with the string you want to write as a parameter ❷. The `"\n"` at the end is a special character for a new line; without it, all of the output would be on one line. You can also write multiple lines at once, by putting them into a list and using the `.writelines()` method instead ❸.

Once you're done with a file, it's normally a good idea to close it ❹, particularly if you're writing to it. Files can sometimes be buffered, which means they're not written onto the disk straight away—if your computer crashes, it might not be saved.

That's not all you can do with files, but it's enough to get started. For your difference engine you won't need to write files, but it will help for future programs. For now, let's turn our attention to the last major feature you'll add to your program.

Comparing files

We're almost there, but there's one last hurdle. When you're running your program, you need to know whether you've seen a particular file in the other directory, and if so, whether it has the same content, too. You could read in all the files in and compare their content line by line, but what if you have a large directory with big images? That's a lot of storage, which means Python is likely to run slowly.

NOTE It's often important to consider how fast your program will run, or how much data it will need to store, particularly if the problem that you're working on is open ended—that is, if it might be run on a large amount of data.

Fingerprinting a file

Fortunately, there's another library to help you, called `hashlib`, which is used to generate a hash for a particular piece of data. A hash is like a fingerprint for a file: from the data it's given, it will generate a list of numbers and letters that's virtually guaranteed to be unique for that data. If even a small part of the file changes, the hash will be completely different, and you'll be able to detect the change. Best of all, the

hashes are relatively small, so they won't take up much space. The following listing features a small script that shows how you might generate a hash for one file.

Listing 3.4 Generating a hash for a file

```
import hashlib
import sys

file_name = sys.argv[1]
read_file = file(file_name)
the_hash = hashlib.md5()
for line in read_file.readlines():
    the_hash.update(line)
print the_hash.hexdigest()
```

❶ Open file
❷ Create hash object
❸ Update hash
❹ Print digest

After importing your libraries, you read a file name from the command line and open it ❶. Next, you create a hash object here ❷, which will handle all of the hash generation. I'm using md5, but there are many others in `hashlib`.

Once you have an open file and a hash object, you feed each line of the file into the hash with the `update()` method ❸.

After you've fed all the lines into the hash, you can get the final hash in `hexdigest` form ❹. It uses only numbers and the letters *a–f*, so it's easy to display on screen or paste into an email.

An easy way to test the script is to run it on itself. After you've run it once, try making a minor change to the script, such as adding an extra blank line at the end of the file. If you run the script again, the output should be completely different.

Here, I'm running the hash-generating script on itself. For the same content, it will always generate the same output:

```
D:\test>python hash.py hash.py
df16fd6453cedecdea3dddca83d070d4
D:\test>python hash.py hash.py
df16fd6453cedecdea3dddca83d070d4
```

These are the results of adding one blank line to the end of the hash.py file. It's a minor change (most people wouldn't notice it), but now the hash is completely different:

```
D:\test>hash.py hash.py
47eeac6e2f3e676933e88f096e457911
```

Now that your hashes are working, let's see how you can use them in your program.

Mugshots: storing your files' fingerprints in a dictionary

Now that you can generate a hash for any given file, you need somewhere to put it. One option is to put the hashes into a list, but searching over a list every time you want to find a particular file is slow, particularly if you have a large directory with lots of files. There's a better way to do it, by using Python's other main data type: the dictionary.

You can think of dictionaries as a bag of data. You put data in, give it a name, and then, later, when you want the data back, you give the dictionary its name, and the dictionary will return the data. In Python's terminology, the name is called a *key* and the data is called the *value* for that key. Let's see how you use a dictionary by taking a look at the following listing.

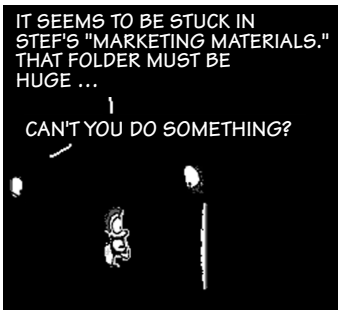
Listing 3.5 How to use a dictionary

```
test_dictionary = {}
test_dictionary = {'one' : 1, 'two' : 2}
test_dictionary = {
    'list' : [1,2,3],
    'dict' : {'one' : 1, 'two' : 2},
}
print test_dictionary['list']
del test_dictionary['list']
print test_dictionary.keys()
print test_dictionary.values()
print test_dictionary.items()
```

1. Put anything you like in dictionary
2. Access values
3. Remove values
4. Useful dictionary methods

Dictionaries are fairly similar to lists, except that you use curly braces instead of square brackets, and you separate keys and their values with a colon.

The other similarity to lists is that you can include anything that you like as a value ❶, including lists, dictionaries, and other objects. You're not limited to storing simple types like strings or numbers, or one type of thing. The only constraint is on the key: it can only be something that isn't modifiable, like a string or number.



To get your value back once you've put it in the dictionary, use the dictionary's name with the key after it in square brackets ❷. If you're finished with a value, it's easy to remove it by using `del` followed by the dictionary and the key that you want to delete ❸.

Dictionaries are objects, so they have some useful methods ❹ as well as direct access. `keys()` returns all of the keys in a dictionary, `values()` will return its values, and `items()` returns both the keys and values. Typically, you'll use it in a `for` loop, like this:

```
for key, value in test_dictionary.items(): ...
```

When deciding what keys and values to use for a dictionary, the best option is to use something unique for the key, and the data you'll need in your program as the value. You might need to convert the data somehow when building your dictionary, but it normally makes your code easier to write and easier to understand. For your dictionary, you'll use the path to the file as the key, and the checksum you've generated as the value.

Now that you know about hashes and dictionaries, let's put your program together.

Putting it all together

"Measure twice, cut once" is an old adage that often holds true. When programming, you always have your undo key, but you can't undo the time you spent writing the code you end up throwing away.

When developing a program, it often helps to have some sort of plan in place as to how you'll proceed. Your plan doesn't have to be terribly detailed; but it can help you to avoid potential roadblocks or trouble spots if you can foresee them. Now that you think you have all of the parts you'll need, let's plan out the overall design of your program at a high level. It should go something like

- Read in and sanity-check the directories you want to compare.
- Build a dictionary containing all the files in the first directory.
- For each file in the second directory, compare it to the same file in the first dictionary.

That seems pretty straightforward. In addition to having this overall structure, it can help to think about the four different possibilities for each file, as shown in the following figure.

Case 1	**The file doesn't exist in directory 2.**	Case 2	**The file exists, but is different in each directory.**
Case 3	**The files are identical in both.**	Case 4	**The file exists in directory 2, but not in your first directory.**

Figure 3.1 The four possibilities for differences between files

Given this rough approach, a couple of issues should stand out. First, your initial plan of building all the checksums right away may not be such a good idea after all. If the file isn't in the second directory, then you'll have gone to all the trouble of building a checksum that you'll never use. For small files and directories it might not make much difference, but for larger ones (for example, photos from a digital camera or MP3s), the extra time might be significant. The solution to this is to put a placeholder into the dictionary that you build and only generate the checksum once you know you have both files.

CAN'T YOU USE A LIST? If you're putting a placeholder into your dictionary instead of a checksum, you'd normally start by using a list. Looking up a value in a dictionary is typically much faster, though; for large lists, Python needs to check each value in turn, whereas a

dictionary needs a single lookup. Another good reason is that dictionaries are more flexible and easier to use than lists if you're comparing independent objects.

Second, what happens if a file is in the first directory but not the second? Given the rough plan we just discussed, you're only comparing the second directory to the first one, not vice versa. You won't notice a file if it's not in the second directory. One solution to this is to delete the files from the dictionary as you compare them. Once you've finished the comparisons, you know that anything left over is missing from the second directory.

Planning like this can take time, but it's often faster to spend a little time up front working out potential problems. What's better to throw away when you change your mind: five minutes of design or half an hour of writing code? Listings 3.6 and 3.7 show the last two parts of your program based on the updated plan. You can join these together with listings 3.2 and 3.3 to get a working program.

Listing 3.6 Utility functions for your difference program

```
import hashlib

def md5(file_path):
    """Return an md5 checksum for a file."""
    read_file = file(file_path)
    the_hash = hashlib.md5()
    for line in read_file.readlines():
        the_hash.update(line)
    return the_hash.hexdigest()

def directory_listing(dir_name):
    """Return all of the files in a directory."""
    dir_file_list = {}
    dir_root = None
    dir_trim = 0
    for path, dirs, files in os.walk(dir_name):
        if dir_root is None:
            dir_root = path
```

1 MD5 function

2 Directory listing function

3 Finding root of directory

```
        dir_trim = len(dir_root)
        print "dir", dir_name,
        print "root is", dir_root
    trimmed_path = path[dir_trim:]
    if trimmed_path.startswith(os.path.sep):
        trimmed_path = trimmed_path[1:]
    for each_file in files:
        file_path = os.path.join(
                        trimmed_path, each_file)
        dir_file_list[file_path] = True
return (dir_file_list, dir_root)
```

3 Finding root of directory

4 Building dictionary of files

5 Returning multiple values

This is the program from listing 3.5, rolled up into a function. Notice how a docstring has been added as the second line 1 so it's easy to remember what the function does.

Because you'll be building a list of files for two directories, it makes sense to have a function that returns all the information you need about a directory 2, so you can reuse it each time. The two things you need are the *root*, or lowest-level directory (the one typed in at the command line) and a list of all the files relative to that root so you can compare the two directories easily. For example, `C`:\test\test_dir\file.txt and C:\test2\test_dir\file.txt should both be entered into their respective dictionaries as \test_dir\file.txt.

Because `os.walk()` starts at the root of a directory by default, all you need to do is remember the first directory that it returns 3. You do that by setting `dir_root` to `None` before you enter the `for` loop. `None` is a special value in Python that means "not set" or "value unknown." It's what you use if you need to define a variable but don't know its value yet. Inside the loop, if `dir_root` is `None`, you know it's the first time through the loop and you have to set it. You're setting a `dir_trim` variable too, so that later you can easily trim the first part of each directory that's returned.

Once you have your directory root, you can chop off the common part of your directories

and path separators from the front of the path returned by `os.walk()` ❹. You do that by using string slices, which will return a subsection of a string. It works in exactly the same way as a list index, so it starts at 0 and can go up to the length of the string.

When you're done, you return both the directory listing and the root of the directory ❺ using a special Python data type called a *tuple*. Tuples are similar to lists, except that they're immutable—you can't change them after they've been created.

Now that you've checked your inputs and set up all of your program's data, you can start making use of it. As in chapter 2, when you simplified Hunt the Wumpus, the part of the program that does stuff is fairly short, clear, and easy to understand. All the tricky details have been hidden away inside functions, as you can see in the next listing.

Listing 3.7 Finding the differences between directories

```
dir1_file_list, dir1_root = directory_listing(directory1)
dir2_file_list, dir2_root = directory_listing(directory2)

for file_path in dir2_file_list.keys():
    if file_path not in dir1_file_list:
        print file_path, "not found in directory 1"
    else:
        print file_path, "found in directory 1 and 2"
        file1 = os.path.join(dir1_root, file_path)
        file2 = os.path.join(dir2_root, file_path)
        if md5(file1) != md5(file2):
             print file1, "and", file2, "differ!"
        del dir1_file_list[file_path]

for key, value in dir1_file_list.items():
    print key, "not found in directory 2"
```

❶ Use directory functions

❷ Files not in directory 1

❸ Compare checksums

❹ Files not in directory 2

To assign both of the variables you get back from your function, you separate them with a comma ❶. You've already seen this when using `dictionary.items()` in a `for` loop.

Here's the first comparison ❷: if the file isn't in directory 1, then you warn the user. You can use `in` with a dictionary in the same way that you would for a list, and Python will return `True` if the object is in the dictionaries' keys.

If the file exists in both directories, then you build a checksum for each file and compare them ❸. If they're different, then you know the files are different and you again warn the user. If the checksums are the same then you keep quiet, because you don't want to overwhelm people with screens and screens of output—they want to know the differences.

Once you've compared the files in section 3, you delete them from the dictionary. Any that are left over you know aren't in directory 2 and you tell the user about them ❹.

That seems to about do it for your program, but are you sure it's working? Time to test it.

Testing your program

If you haven't already, now's probably a good time to create some test directories so you can try your script and make sure it's working. It's especially important as you start working on problems that have real-world consequences. For example, if you're backing up some family photos and your program doesn't report that a file has changed (or doesn't exist), you won't know to back it up and might lose it if your hard drive crashes. Or it might report two files as the same when they're actually different.

You can test your script on directories that you already have, but specific test directories are a good idea, mainly because you can exercise all the features you're expecting. At a minimum, I'd suggest

- Adding at least two directory levels, to make sure paths are handled properly
- Creating a directory with at least one space in its name

- Using both text and binary files (for example, images)
- Setting up all the cases you're expecting (files missing, file differences, files that are the same)

By thinking about all the possible cases, you can catch bugs in your program before you run it over a real directory and miss something or, worse, lose important data. The following figure shows the initial test directory (called test) that I set up on my computer.

Figure 3.2 A test directory for the difference engine

This test directory doesn't get all the possible failures, but it does check for most of them. The next step was to copy that directory (I called it test2) and make some changes for the difference engine to work on, as shown in figure 3.3. I've used the numbers 1 to 4 within the files to represent each of the possible cases, with 1 and 4 being missing files, 2 for files that have some differences, and 3 for files that are identical in both directories.

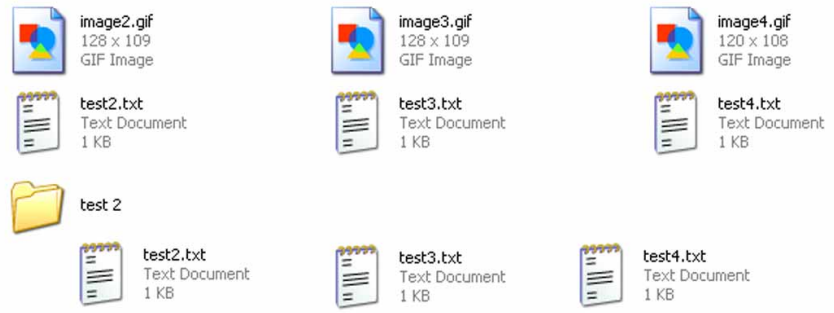


Figure 3.3 test2, an almost identical copy of the first test directory

You can see the output of running your script over these directories:

```
D:\>python code\difference_engine.py test test2
Comparing:
test
test2
dir test root is test
dir test2 root is test2
test\test 2\test2.txt and test2\test 2\test2.txt differ!
image4.gif not found in directory 1
test 2\test4.txt not found in directory 1
test\image2.gif and test2\image2.gif differ!
test4.txt not found in directory 1
test\test2.txt and test2\test2.txt differ!
test1.txt not found in directory 2
test 2\test1.txt not found in directory 2
image1.gif not found in directory 2
```

That seems to be pretty much what you were expecting. The script is descending into the test 2 directory in each case and is picking up the differences between the files—1 and 4 are missing, 2 is different, and 3 isn't reported because the files are identical.

Now that you've tested out your script, let's see what you can do to improve it.

Improving your script

Your script so far works, but it could do with a few improvements. For a start, the results it returns are out of order. The files that are missing from the second directory appear right at the end. Ideally, you'd have them appear next to the other entries for that directory, to make it easier to see what the differences are.

NOTE Does this strategy look familiar? It's exactly what you did when developing Hunt the Wumpus. You start by writing a program that's as simple as you can make it and then build on the extra features that you need.

Putting results in order

It initially might be difficult to see how you might go about ordering the results, but if you think back to chapter 2, one of the strategies that

you used with Hunt the Wumpus was to separate the program from its interface. In your difference engine, you haven't done so much of that so far—now might be a good time to start. You need two parts to your program: one part that does the work and stores the data it generates, and another to display that data. The following listing shows how you generate your results and store them.

Listing 3.8 Separating generated results from display

```
dir1_file_list, dir1_root = directory_listing(directory1)
dir2_file_list, dir2_root = directory_listing(directory2)
results = {}                                                    # 1 Results dictionary

for file_path in dir2_file_list.keys():
    if file_path not in dir1_file_list:
        results[file_path] = "not found in directory 1"
    else:
        file1 = os.path.join(dir1_root, file_path)              # 2 Store results
        file2 = os.path.join(dir2_root, file_path)
        if md5(file1) != md5(file2):
            results[file_path] = "is different in directory 2"
        else:
            results[file_path] = "is the same in both"

for file_path, value in dir1_file_list.items():
    if file_path not in results:
        results[file_path] = "not found in directory 2"
```

Here's the trick. Rather than try to display the results as soon as you get them, which means you're trying to shoehorn your program structure into your display structure, you store the results in a dictionary to display later ❶.

The result of each comparison is stored in `result` ❷, with the file path as the key and a description of the result of the comparison as the value.

That should take care of storing the results; let's take a look at how you display them:

```
print
for file_path, result in sorted(results.items()):               # 1 Sort results
```

```
    if os.path.sep not in file_path and "same" not in result:
        print path, result
```

❷ Check in strings

```
for path, result in sorted(results.items()):
    if os.path.sep in file_path and "same" not in result:
        print path, result
```

❸ Other directories

`sorted()` is a built-in Python function that sorts groups of items ❶. You can give it lists, dictionary keys, values or items, strings, and all sorts of other things. In this case, you're using it to sort `result.items()` by `file_path`, the first part of `result.items()`.

Within the body of the loop, you're using `in` to check the contents of the strings ❷. You want to know whether this path is part of a directory, in which case it will have `os.path.sep` somewhere within it, and you also want to know whether the result shows that the files are the same.

Now that you've displayed everything within the root of the directory, you can go ahead and show everything within the subdirectories ❸. You're reversing the sense of the `if` statement to show everything that wasn't shown the first time around.

In hindsight, that was relatively easy. Following the pattern you established in Hunt the Wumpus, separating data from its display is a powerful tactic that can make complicated problems easy to understand and program.

Comparing directories

The other thing your program should probably handle is the case where you have empty directories. Currently it only looks for files, and any empty directories will be skipped. Although unnecessary for your initial use case (checking for missing images before you back up), it will almost certainly be useful somewhere down the track. Once you've added this feature, you'll be able to spot any change in the directories,

short of permission changes to the files—and it requires surprisingly little code. The next listing shows how I did it.

Listing 3.9 Comparing directories, too

```
def md5(file_path):
    if os.path.isdir(file_path):
        return '1'
    read_file = file(file_path)

...

    for path, dirs, files in os.walk(directory_name):
        ...
        for each_file in files + dirs:
            file_path = os.path.join(trimmed_path, each_file)
            dir_file_list[file_path] = True
```

❶ Don't try to checksum directories

❷ Include directory and file paths

The first thing to do is to include directory paths as well as files when generating a listing ❷. To do that, you join the `dirs` and `files` lists with the `+` operator.

If you try to open a directory to read its contents, you'll get an error ❶; this is because directories don't have contents the same way files do. To get around that, it's ok to cheat a little bit. You alter the md5 function and use `os.path.isdir()` to find out whether it's a directory. If it is, you return a dummy value of `'1'`. It doesn't matter what the contents of a directory are, because the files will be checked in turn, and you only care whether a directory exists (or not).

Once you've made those changes, you're done. Because the directories follow the same data structure as the files, you don't need to make any changes to the comparison or display parts of your program. You'll probably want to add some directories to both your test directories to make sure the program is working properly.

You've improved your script, but that doesn't mean there isn't more you can do.

Where to from here?

The program as it stands now is feature-complete based on your initial need, but you can use the code you've written so far for other purposes. Here are some ideas:

- If you're sure you won't have any different files, you can extend the program to create a merged directory from multiple sources. Given a number of directories, consolidate their contents into a third, separate location.
- A related task would be to find all the identical copies of a file in a directory—you might have several old backups and want to know whether there are any sneaky extra files you've put in one of them.
- You could create a change monitor—a script that notifies you of changes in one directory. One script would look at a directory and store the results in a file. The second script would look at that file and directory and tell you if any of the output has changed. Your storage file doesn't have to be complicated—a text file containing a path and checksum for each file should be all you need.
- You can also use your `os.walk` functions as a template to do something other than check file contents. A script to check directory sizes could be useful. Your operating system will probably give you information about how much space a particular directory takes up, but what if you want to graph usage over time, or break your results down by file type? A script is much more flexible, and you can make it do whatever you need.

You'll need to avoid the temptation of reinventing the wheel. If a tool has already been written that solves your problem, it's generally better to use that, or at least include it in your script if possible. For example, you might consider writing a program that shows you the changes between different versions of files as well as whether they're different—but that program's already been written; it's called `diff`. It's

widely available as a command-line program under Linux, but it's also available for Windows and comes in graphical versions, too.

One of the other programming tricks is knowing when to stop. Gold-plating your program can be fun, but you could always be working on your next project instead!

Summary

In this chapter, you learned about some of the standard library packages available with every installation of Python, as well as how to include and use them and how to learn about unfamiliar ones. You built what would normally be a fairly complex application, but, because you made good use of several Python libraries, the amount of code you had to write was minimal.

In the next chapter, we'll look at another way of organizing programs, as well as other uses for functions and some other Python techniques that can help you to write clearer, more concise code. The program in this chapter was fairly easy to test, but not all programs will be that straightforward, so we'll also look at another way of testing programs to make sure they work.

7

Sufficiently advanced technology...

This chapter covers

- *More advanced features of classes*
- *Generators*
- *Functional programming*

In this chapter, we're going to be looking at some of the more advanced tasks Python can do. In chapter 1, you learned that Python is known as a multi-paradigm language, which means it doesn't confine you to just one way of doing things. There are three main styles of programming: imperative, object-oriented, and functional. Python lets you work with all three, and even mix and match them where necessary.

We've already covered imperative and most of object-oriented programming in the chapters so far, so this chapter will focus mostly on functional programming and the more advanced parts of object-oriented programming in Python.

Object orientation

Let's start by taking a second look at how object-oriented classes should be organized, using two separate methods: mixin classes and the `super()` method.

Mixin classes

Sometimes you don't need an entire class to be able to do something. Perhaps you only need to add logging, or the ability to save the state of your class to disk. In these cases, you could add the functionality to a base class, or to each class that needs it, but that can get pretty repetitive. There's an easier way, called a *mixin class*.

The idea is that a mixin class contains only a small, self-contained piece of functionality, usually a few methods or variables, which are unlikely to conflict with anything in the child class. Take a look at this listing, which creates a `Loggable` class.

Listing 7.1 A logging mixin

```
class Loggable(object):
    """Mixin class to add logging."""
    log_file_name = 'log.txt'
    def log(self, log_line):
        file(self.log_file_name).write(log_line)

class MyClass(Loggable):
    """A class that you've written."""
    log_file_name = "myclass_log.txt"
    def do_something(self):
        self.log("I did something!')
```

The mixin class is defined in exactly the same way as a regular class. Here, you add a class variable for the file name and a method to write a line to that file. If you want to use the mixin class, all you need to do is inherit from it.

Once you're in the child class, all of the mixin's methods and variables become available, and you can override them if you need to.

Using simple file logging like this works well, but the following listing features a slightly more involved version that uses Python's built-in logging module. The advantage of this version is that, as your program grows, you can take advantage of some of the different logging methods—you can send it to your system's logs, or automatically roll over to a new file if the old one gets too big.

Listing 7.2 Using Python's logging module

```
import logging

class Loggable(object):
    """Mixin class to add logging."""

    def __init__(self,
                 log_file_name = 'log.txt',
                 log_level = logging.INFO,
                 log_name = 'MyApp'):
        self.log_file_name = log_file_name
        self.log_level = log_level
        self.log_name = log_name
        self.logger = self._get_logger()

    def _get_logger(self):
        logger = logging.getLogger(self.log_name)
        logger.setLevel(self.log_level)

        handler = logging.FileHandler(
                     self.log_file_name)
        logger.addHandler(handler)

        formatter = logging.Formatter(
            "%(asctime)s: %(name)s - "
            "%(levelname)s - %(message)s")
        handler.setFormatter(formatter)

        return logger

    def log(self, log_line, severity=None):
        self.logger.log(severity or self.log_level,
                        log_line)

    def warn(self, log_line):
        self.logger.warn(log_line)
    ...
```

1 Initialize Loggable

2 Create Logger object

3 Logging methods

```
class MyClass(Loggable):
    """A class that you've written."""

    def __init__(self):
        Loggable.__init__(self,
                          log_file_name = 'log2.txt')
        #super(MyClass, self).__init__(
        #                  log_file_name = 'log2.txt')

    def do_something(self):
        print "Doing something!"
        self.log("I did something!")
        self.log("Some debugging info", logging.DEBUG)
        self.warn("Something bad happened!")

test = MyClass()
test.do_something()
```

4 How do you call Loggable.__init__()?

5 Create class and logging methods

Rather than rely on class methods, it's better to instantiate them properly from an `__init__` method ❶. This way, you can take care of any extra initialization you need to do, or require that variables be specified on creation.

❷ is all the setup you need to do when creating a logger from Python's logging module. First, you create a logger instance. Then, you can add a handler to it, to specify what happens to log entries, and a formatter to that handler, to tell it how to write out log lines.

Your mixin class also needs methods so you can log ❸. One option is to use a generic log method that you give a severity when you call it, but a cleaner way is to use the logger's methods like `debug`, `info`, `warn`, `error`, and `critical`.

Now that you're using `__init__` in `Loggable`, you'll need to find a way to call it ❹. There are two ways. The first is to call each parent class explicitly by using its name and method directly, but passing in `self`. The second is to use Python's `super()` method, which finds the method

in the next parent class. In this case, they do much the same thing, but `super()` properly handles the case where you have a common grandparent class. See the next section for potential problems when using this method, and the sample code in super_test.py in the code tarball for this book.

Once all that's done, you can use the logging class exactly the same way you did in the previous version ❺. Note that you've also exposed the logger object itself, so if you need to, you can call its methods directly.

super() and friends

Using the `super()` method with *diamond inheritance* (see figure 7.1) can be fraught with peril—the main reason being that, when you use it with common methods such as `__init__`, you're not guaranteed which class's `__init__` method you'll be calling. Each will be called, but they could be in any order. To cover for these cases, it helps to remember the following things when using `super()`:

- Use `**kwargs`, avoid using plain arguments, and always pass all the arguments you receive to any parent methods. Other parent methods might not have the same number or type of arguments as the subclass, particularly when calling `__init__`.
- If one of your classes uses `super()`, then they all should. Being inconsistent means an `__init__` method might not be called or might be called twice.

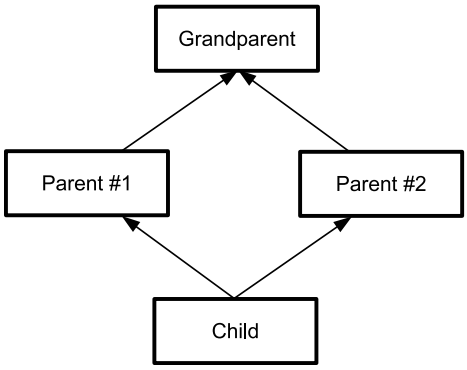


Figure 7.1
A diamond inheritance structure

- You don't necessarily *need* to use `super()` if you can design your programs to avoid diamond inheritance—that is, without parent classes sharing a grandparent.

Now that you have a better sense of how classes should be organized and what to watch out for when using multiple inheritance, let's take a look at some of the other things you can do with classes.

Customizing classes

Python gives you a great deal of power when it comes to defining how the methods in your class work and which methods are called. Not only do you have access to all of Python's introspection power, but you can also decide to use different methods at runtime—even methods that don't exist.

When Python looks up an attribute or method on a class (for example, `self.log_file_name` or `test.do_something()` in listing 7.2), it will look up that value in a dictionary called `__dict__`. `__dict__` stores all the user-defined values for a class and is used for most lookups, but it's possible to override it at several points.

Python provides a number of possible ways to customize attribute access by overriding some built-in methods. You do so in the same way you've been using `__init__` to initialize classes.

__getattr__

`__getattr__` is used to provide methods or attributes when they're not found in the class or a parent class. You can use this to catch missing methods or write wrappers around other classes or programs. The following listing shows how you can use the `__getattr__` method to override the way Python looks up missing attributes.

Listing 7.3 Using __getattr__

```
class TestGetAttr(object):

    def __getattr__(self, name):
        print "Attribute '%s' not found!" % name
        return 42
```

```
test_class = TestGetAttr()
print test_class.something

test_class.something = 43
print test_class.some-
thing
```

The `__getattr__` method takes one argument, the attribute name, and returns what the value should be. In this case, you print the name and then return a default value, but you could do anything—log to a file, call an API, or hand over the responsibility to another class or function.

Now when you try to access a value that doesn't exist in the class, `__getattr__` will step in and return your default value.

Because `__getattr__` is only called when the attribute isn't found, setting an attribute first means that `__getattr__` won't be run.

Now that you can get your attributes, let's also learn how to set them.

__setattr__

`__setattr__` is used to change the way that Python alters attributes or methods. You can intercept calls to your class, log them, or do whatever you need to. The following listing shows a simple way to catch attribute access and redirect it to a different dictionary instead of inserting it into the default `__dict__`.

Listing 7.4 Using __setattr__

```
class TestSetAttr(object):

    def __init__(self):
        self.__dict__['things'] = {}

    def __setattr__(self, name, value):
        print "Setting '%s' to '%s'" % (name, value)
        self.things[name] = value

    def __getattr__(self, name):
        try:
            return self.things[name]
```

1 **Set up replacement dictionary**

2 **__setattr__ inserts into things**

3 **__getattr__ reads from things**

```
        except KeyError:
            raise AttributeError(
                "'%s' object has no attribute '%s'" %
                    (self.__class__.__name__, name))
```

❸

```
test_class2 = TestSetAttr()
test_class2.something = 42
print test_class2.something
print test_class2.things
print test_class2.something_else
```

❹ **Use class**

❶ is where you set `things`, which will store all the attributes you'll set. One catch when using `__setattr__` is that you can't directly set something in the class, because that will result in `__setattr__` calling itself and looping until Python runs out of recursion room. You'll need to set the value in the class's `__dict__` attribute directly, as you do here.

Once `things` is set in `__dict__`, though, you can read from it normally, because `__getattr__` won't be called when you access `self.things`. `__setattr__` takes a name and a value, and in this case you're inserting the value into the `things` dictionary ❷ instead of into the class.

This version of `__getattr__` looks in the `self.things` dictionary for your value ❸. If it's not there, you raise an `AttributeError` to mimic Python's normal handling.

The class you've written behaves exactly like a normal class, except you have close to complete control over how its methods and attributes are read ❹. If you want to override everything, though, you'll need to use `__getattribute__`.

__getattribute__

Another approach is to override all method access entirely. If `__getattribute__` exists in your class, it will be called for all method and attribute access, right?

Well, that's sort of true. Strictly speaking, even `__getattribute__` doesn't override everything. There are still a number of methods, such as `__len__` and `__init__`, which are accessed directly by Python and won't be overridden. But everything else, even `__dict__`, goes through `__getattribute__`. This works, but in practice it means you'll have a hard time getting to any of your attributes. If you try something like `self.thing`, then you'll end up in an infinite `__getattribute__` loop.

How do you fix this? `__getattribute__` won't be much use if you can't access the real variables. The answer is to use a different version of `__getattribute__`: the one you would normally be using if you hadn't just overridden it. The easiest way to get to a fresh `__getattribute__` is via the base `object` class, and feed in `self` as the instance. The following listing shows you how.

Listing 7.5 Using `__getattribute__`

```
class TestGetAttribute(object):

    def __init__(self, things=None):
        my_dict = object.__getattribute__(
                                    self, '__dict__')
        if not things:
            my_dict['things'] = {}
        else:
            my_dict['things'] = things

    def __setattr__(self, name, value):
        print "Setting '%s' to '%s'" % (name, value)
        my_dict = get_real_attr(self, '__dict__')
        my_dict['things'][name] = value

    def __getattribute__(self, name):
        try:
            my_dict = get_real_attr(self, '__dict__')
            return my_dict['things'][name]
        except KeyError:
            my_class = get_real_attr(self, '__class__')
            raise AttributeError(
                "'%s' object has no attribute '%s'" %
                    (my_class.__name__, name))
```

1 Borrow object's getattribute

2 Easier way

3 _getattribute__

```
def get_real_attr(instance, name):
    return object.__getattribute__(instance, name)
```

❷ Easier way

```
test_class3 = TestGetAttribute({'foo': 'bar'})
print object.__getattribute__(test_class3, '__dict__')
test_class3.something = 43
print object.__getattribute__(test_class3, '__dict__')
print test_class3.foo
```

❹ Use class

Python methods are functions, so it's relatively easy to call back to `object`. The only thing you need to do is to pass it `self` as the instance, and the name of the attribute you want ❶. I've also updated `__init__` so you can pass in values to set up the internal `things` dictionary.

To tidy up the calls to object, you can define a helper function to make the call for you. ❷ is a version of `__setattr__` that uses it.

Other than the fact that you need to use `object` to get the dictionary you're editing, the call to `__getattribute__` ❸ is much like the one to `__getattr__`; it receives a name and returns a value, converting `KeyError` to `AttributeError` along the way.

After you've been through all that, your class is ready to be used ❹. It follows the same usage pattern, but you can now hide the `things` dictionary from casual inspection (it's still visible if you use the old `object.__getattribute__`, though).

If using `__getattribute__` seems like a lot of work, don't worry. Most of the time, you won't need to use it. But many third-party libraries make use of it in addition to `__getattr__` and `__setattr__`. If you need to use them, they can save a lot of work and make your class interfaces a lot more Pythonic and easy to use.

Properties

A more specific method for customizing your attributes is to use Python's `property` function. Whereas `__getattr__` and `__getattribute__` work across the entire class, `property` allows

you to specify functions, commonly known as *getters* and *setters*, that are responsible for controlling access to an attribute or method.

Properties solve a common programming problem: how to customize attribute access without altering your class's external interface. Without properties, it's standard practice to use getters and setters for every attribute, even if you don't need them, due to the difficulty in switching from attribute access to using a function. Python allows you to do this without having to change everything that uses your class. The next listing shows how you might create an integer attribute that can only be set to an integer from 0 to 31.

Listing 7.6 Using properties

```
class TestProperty(object):

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

    def get_x(self):
        return self._x

    def set_x(self, value):
        if not (type(value) == int and 0 <= value < 32):
            raise ValueError("TestProperty.x "
                "must be an integer from 0 to 31")
        self._x = value

    x = property(get_x, set_x)

test = TestProperty(10)
print test.x
test.x = 11
test.x += 1
assert test.x == 12
print test.x

try:
    test2 = TestProperty(42)
except ValueError:
    # ValueError: TestProperty.x must be
    # an integer between 0 and 32
    print "test2 not set to 42"
```

1 Class setup

2 x is really _x

3 Set _x

4 Define property

5 Interface

6 Bounds checking

The initial setup ❶ looks similar to any class's `__init__` function. Some introductions set the hidden variable directly; but I prefer it this way, because it means you can't have `x` set to something out of bounds.

❷ is your getter, which returns the value of `_x`—although you could convert it to whatever you liked, or even return `None`.

❸ is your setter, which first checks to make sure the value is an integer from 0 to 31. If it isn't, then you raise a `ValueError`.

Finally, you set `x` on the class to be a property ❹ and pass it the getter and setter functions, `get_x` and `set_x`. Note that you can also define a read-only property if you only pass a getter. If you then try to set `x`, you'll get `AttributeError: can't set attribute`.

If you didn't know it was a property, you wouldn't be able to tell by using the class. The interface ❺ for your defined `x` is exactly the same as if it were a regular attribute.

The only exception to the interface is the one you've included. If you try to set the value of `test2.x` to something out of bounds, you'll get an exception ❻.

In practice, you'll want to use the methods that are most suited for your use case. Some situations, such as logging, wrapping a library, and security checking, call for `__getattribute__` or `__getattr__`, but if all you need to do is customize a few specific methods, then properties are normally the best way to do it.

Emulating other types

One other common practice is to write classes to emulate certain types, such as lists or dictionaries. If you have a class that is supposed to behave similarly to a list or a number, it helps when the class behaves in exactly the same way, supporting the same methods and raising the same exceptions when you misuse it.

There are a number of methods you can define that Python will use when you use your class in certain ways. For example, if you need two instances of your class to compare as equal, you can define an `__eq__` method that takes two objects and returns `True` if they should be treated as equal.

The next listing provides an example: here, two methods are added to the previous class so you can compare them to each other. I've renamed the class `LittleNumber`, to make its purpose clearer (you'll also want to rename the class name in the exception).

Listing 7.7 Extending properties

```
class LittleNumber(object):
    ...

    def __eq__(self, other):
        return self.x == other.x

    def __lt__(self, other):
        return self.x < other.x

    def __add__(self, other):
        try:
            if type(other) == int:
                return LittleNumber(self.x + other)
            elif type(other) == LittleNumber:
                return LittleNumber(self.x + other.x)
            else:
                return NotImplemented
        except ValueError:
            raise ValueError(
                "Sum of %d and %d is out of bounds "
                "for LittleNumber!" % (self.x, other.x))

    def __str__(self):
        return "<LittleNumber: %d>" % self.x

one = LittleNumber(1)
two = LittleNumber(2)
print one == one
print not one == two
```

1 `__eq__`

2 `__lt__`

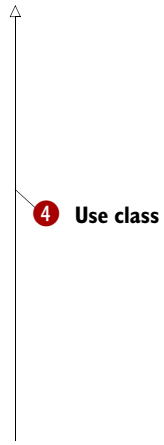
3 Add values

4 Use class

```
print one != two
print one < two
print two > one
print not one > two
print two >= one
print two >= two

onetoo = LittleNumber(1)
print onetoo == one
print not onetoo == two

print onetoo + one
print one
print onetoo + one == two
```



This method ❶ checks the value against the other one you're given. Whenever Python encounters `a == b`, it will call `a.__eq__(b)` to figure out what the real value should be.

In the same way as `__eq__`, `__lt__` ❷ will compare two values and return `True` if the current instance is less than the one passed in.

`__add__` is also useful and should return the result of adding something to the class ❸. This case is somewhat more complex—you return a new `LittleNumber` if you're passed an integer or another `LittleNumber`, but you need to catch two cases: where the value goes out of bounds and where someone passes you a different type, such as a string. If you can't (or won't) handle a particular case, you can return `NotImplemented`, and Python will raise a `TypeError`. Again, a more understandable error message here will save you a lot of debugging further down the track.

Believe it or not, that's all you need to get your class to behave something like an integer ❹. Note that you don't necessarily need to implement all of the mirror functions like `__gt__` and `__ne__`, because Python will try their opposites if they're not defined. All of the expressions here should return `True`.

Here's a table of some of the most common methods you'll want to override if you're providing a class similar to some of the built-in types.

Table 7.1 Common methods you may want to override

Type	Methods	Description

Most types	`__eq__(self, other)`
	`__ne__(self, other)`
	`__gt__(self, other)`
	`__lt__(self, other)`
	`__le__(self, other)`
	`__ge__(self, other)`
		Tests for equality and relative value, ==, !=, >, <, <=, and >=.

	`__str__(self)` `__repr__(self)`	Returns a printable version of the class and a printable representation of the class.

Dictionary, list, or other container	`__getitem__(self, key)`
	`__setitem__(self, key, value)`
	`__delitem__(self, key)`
		Gets, sets, and deletes an entry.

	`keys(self)`	Returns a list of keys (dictionaries only).
	`__len__(self)`	Returns the number of entries.
	`__iter__(self)` and `iterkeys(self)`	If your object is large, you might want to consider using these methods to return an iterator (see the next section for details).
	`__contains__(self, value)`	The value of an entry (of a list or set), or a key (of a dictionary).

Numbers	`__add__(self, other)`
	`__sub__(self, other)`
	`__mul__(self, other)`
	`__floordiv__(self, other)`
	`__pow__(self, other)`
		Returns the result of adding, multiplying, dividing, and raising to a power.

	`__int__(self)` `__float__(self)`	Converts an instance of a class into an integer or float.

These are by no means the only methods you can set, but they're by far the most common, unless you're doing something exotic. Let's look at a practical example of how these methods are used in practice, by examining Python's generators and iterators.

Generators and iterators

Generators are one of Python's best-kept secrets after list comprehensions, and they're worth looking into in more detail. They're intended to solve the problem of storing state in between function calls, but they're also useful for cases where you need to deal with large amounts of data, perhaps too large to fit easily in memory.

First we'll look at iterators, Python's method for dealing with looping objects. Then we'll look at how you can make use of generators to quickly deal with large amounts of data in log files.

Iterators

You've been using iterators throughout the book, right from chapter 2, because every time you use a `for` loop or a list comprehension, iterators have been acting behind the scenes. You don't need to know how iterators work in order to make use of them, but they're useful for understanding how generators operate.

NOTE Iterators are a common solution to a frequent programming task: you have a bunch of things—now how do you do something to each one of them? The trick is that most Python collections, be they lists, files, or sets, can be used as iterators.

The interface of an iterator is straightforward. It has a `.next()` method, which you call over and over again to get each value in the sequence. Once all the values are gone, the iterator raises a `StopIteration` exception, which tells Python to stop looping. In practice, using an iterator looks something like figure 7.2.

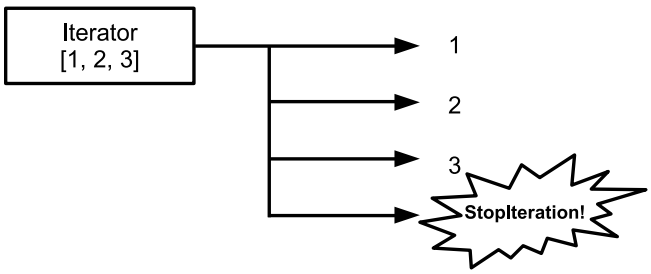


Figure 7.2 The iterator protocol: once you run through three iterations, it stops.

When you ask Python to iterate over an object such as a list, the first thing it does is call `iter(object)`, which calls that object's `__iter__` method and expects to get an iterator object. You don't need to use the `__iter__` call directly unless you're creating your own custom iterator. In that case, you'll need to implement both the `__iter__` and the `next()` methods yourself. This listing shows how to use the `iter` function to create an iterator from any iterable Python object.

Listing 7.8 Using an iterator the hard way

```
>>> my_list = [1, 2, 3]
>>> foo = iter(my_list)
>>> foo
<listiterator object at 0x8b3fbec>
>>> foo.next()
1
>>> foo.next()
2
>>> foo.next()
3
>>> foo.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

❶ Create iterator

❷ Iterate over it

❸ No more values

You use the `iter()` function to create an iterator object for `my_list` ❶. If you print it, you can see that's it's a new type of object—a `listiterator`—not a list. When you call the `next()` method of your iterator ❷, it returns the values from the list: 1, 2, and 3.

Once you've run out of values ❸, the iterator will raise a `StopIteration` exception to signal that the iterator has finished.

The iterator protocol is simple, but it's a fundamental underpinning of the looping and iteration mechanisms in Python. Let's have a look at how generators use this protocol to make your programming life easier.

Generators

Generators are similar to iterators. They use exactly the same `.next()` method, but they're easier to create and to understand. Generators are defined exactly like functions, except they use a `yield` statement instead of a `return`. Here's a simple example of a generator, which counts down to zero from a specified value:

```
def counter(value):
    print "Starting at", value
    while value > 0:
        yield value
        value -= 1
```

1 Definition
2 Looping
3 Yield

Let's look at this one step at a time.

Generators start out like functions, including the way you give them arguments 1. We're also including a debugging string here, so you can follow how the generator is called.

Generators need some way to return values repeatedly, so usually you'll see a loop 2 within the body.

The `yield` statement 3 will stop your function and return the value you give it.

Finally, after each call, Python will return into the generator via the `next()` method. You subtract 1 from the value each time, until it's no longer greater than zero, and then the generator will finish with a `StopIteration` exception.

That's only half of the puzzle, though—you still need to be able to call your generator. The following listing shows how you can do that, by creating a counter and then using it in a `for` loop. You can also call it directly with a line like `for x in counter(5)`.

Listing 7.9 Using your counter generator

```
>>> foo = counter(5)
>>> foo
<generator object at 0x896054c>
>>> for x in foo:
...     print x
```

1 Create counter generator
2 Use generator in loop

```
...
counting down from 5
5
4
3
2
1
>>>
```

❸ Output from generator

First, you create the counter ❶. Even though it looks like a function, it doesn't start right away; instead, it returns a generator object.

You can use the generator object in a loop like ❷. Python will call the generator repeatedly until it runs out of values or the generator exits normally.

❸ is what the loop prints out. The first line is the initial debug from the generator, and the other lines are the results it returns.

There's one last mechanism you should know, which can save you a lot of time setting up generator functions.

Generator expressions

Generator expressions are a lot like list comprehensions, except that, behind the scenes, they use generators rather than building a whole list. Try running the following expressions in a Python prompt:

```
foo = [x**2 for x in range(100000000)]
bar = (x**2 for x in range(100000000))
```

Depending on your computer, the list comprehension will either take a long time to return or else raise a `MemoryError`. That's because it's creating a hundred million results and inserting them into a list.

This generator, on the other hand, will return immediately. It hasn't created any results at all—it will only do that if you try to iterate over `bar`. If you break out of that loop after 10 results, then the other 99,999,990 values won't need to be calculated.

If generators and iterators are so great, why would you ever use lists or list comprehensions? They're still useful if you want to do anything else

with your data other than loop over it. If you want to access your values in a random order—say the fourth value, then the second, then the eighteenth—then your generators won't help because they access values linearly from the first through to the one millionth. Similarly, if you need to add extra values to your list or modify them in some way, then generators won't help—you'll need a list.

Now that you know how generators work, let's look at where they can be useful when you write your programs.

Using generators

As you learned at the start of the chapter, you can use generators in cases where reading in a large amount of data would slow your program down or make it run out of memory and crash.

NOTE In case you haven't realized it yet, Python is an intensely pragmatic language. Every feature has gone through a rigorous community-based vetting process known as a Python Enhancement Proposal (PEP), so there will be strong use cases for each feature. You can read more about PEPs at www.python.org/dev/peps/pep-0001/.

A common problem that involves a large amount of data is the processing of files. If you have a few hundred log files and need to find out which ones have a certain string in them, or collate data across several website directories, then it can be hard to make sense of what's happening within a reasonable amount of time. Let's take a look at a few simple generators that can make your life easier if you run into this sort of problem.

Reading files

One of the areas where Python makes use of generators is in the file-processing sections of the `os` module. `os.walk` is a good example—it allows you to iterate recursively through directories, building lists of the files and subdirectories within them, but because it builds the list as it goes, it's nice and fast. You've

already encountered `os.walk` in chapter 3, when you were building a program to compare files. A typical use is shown in the following listing, which is a program to read a directory and return the files that are of a certain type—in this case, .log files.

Listing 7.10 os.walk revisited

```
import os

dir_name = '/var/log'
file_type = '.log'

for path, dirs, files in os.walk(dir_name):
    print path
    print dirs
    print [f for f in files if f.endswith(file_type)]
    print '-' * 42
```

1 Directory and file type

2 Iterate using os.walk

3 Log files

First, you specify your directory and the file type you want to search for 1. `os.walk` returns a generator you can use to iterate over the directory 2. It will give you the path of the directory, as well as any subdirectories and files within it.

You assume that anything that ends in .log is a log file 3. Depending on your specific situation, an assumption like this may or may not be warranted, but because you will, in practice, have control of the web server, you can add the .log part if you need to.

When you run the program in listing 7.10, it will output something like the following. Each section contains the directory you're iterating over and then its subdirectories and the log files within the current directory.

Listing 7.11 The output from os.walk

```
/var/log
['landscape', 'lighttpd', 'dist-upgrade', 'apparmor', ... ]
['wpa_supplicant.log', 'lpr.log', 'user.log', ... ]
------------------------------------------
/var/log/landscape
[]
['sysinfo.log']
```

```
----------------------------------------
/var/log/dist-upgrade
[]
['main.log', 'apt-term.log', 'xorg_fix_intrepid.log', ... ]
----------------------------------------
...
```

You can use some generators to make your code a little easier to work with. As an example, let's say you're monitoring your web server for errors, so you want to find out which of your log files have the word *error* in them. You'd also like to print out the line itself so you can track down what's going on if there are any errors. Here are three generators that will help you do that.

Listing 7.12 Generators to work through a directory

```
import os

def log_files(dir_name, file_type):
    if not os.path.exists(dir_name):
        raise ValueError(dir_name + " not found!")
    if not os.path.isdir(dir_name):
        raise ValueError(dir_name + " is not a directory!")
    for path, dirs, files in os.walk(dir_name):
        log_files = [f for f in files
                     if f.endswith(file_type)]
        for each_file in log_files:
            yield os.path.join(path, each_file)

def log_lines(dir_name, file_type):
    for each_file in log_files(dir_name, file_type):
        for each_line in file(each_file).readlines():
            yield (each_file, each_line.strip())

def list_errors(dir_name, file_type):
    return (each_file + ': ' + each_line.strip()
            for each_file, each_line in
                log_lines(dir_name, file_type)
            if 'error' in each_line.lower())

if __name__ == '__main__':
```

1 **Wrap os.walk in generator**

2 **Generator for each line of files**

3 **Filter out non-error lines**

```
dir_name = '/var/log'
file_type = '.log'
for each_file in log_files(dir_name, file_type):
    print each_file
print
for each_error in list_errors(dir_name, file_type):
    print each_error
```

4 Create generators

This is the same code you saw in listing 7.10, but wrapped in a generator function 1. One issue with `os.walk` is that it won't raise an exception if you give it a nonexistent directory or something that's not a directory, so you catch both of those cases before you start.

Now that you have `log_files` as a generator, you can use it to build further generators. `log_lines` reads each file in turn and yields successive lines of each log file, along with the name of the file 2.

This generator builds on the `log_lines` generator to return only those lines that have the word *error* in them 3. Notice also that you're returning a generator comprehension instead of using `yield`. This is an alternative way of doing things, one that can make sense for small generators, or where the values you're returning fit the generator comprehension style well.

Once you've done all the hard work of creating the generators 4, calling them is easy—give them the directory and file type, and do what you need to with each result.

Now you can find all the error lines in all the log files in a certain directory. Returning the lines with *error* in them isn't particularly useful, though. What if you had an error that didn't contain the word *error?* Instead, it could contain something like *Process #3456 out of memory!* There are all sorts of conditions you'd like to check in your log files, so you'll need something a little more powerful.

Getting to grips with your log lines

You'd like to have a lot more control over the data in your log files, including being able to filter by any field or combination of fields. In practice, this means you'll need to break the data from each line in your log file up and interpret the bits. The following listing shows some examples from an old Apache access log I had lying around.

Listing 7.13 Apache log lines

```
124.150.110.226 - - [26/Jun/2008:06:48:29 +0000] "GET / HTTP/1.1" 200 99
"-" "Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.14) Gecko/20080419
     Ubuntu/8.04 (hardy) Firefox/2.0.0.14"

66.249.70.40 - - [26/Jun/2008:08:41:18 +0000] "GET /robots.txt HTTP/1.1"
404 148 "-" "Mozilla/5.0 (compatible; Googlebot/2.1;
             +http://www.google.com/bot.html)"

65.55.211.90 - - [27/Jun/2008:23:33:52 +0000] "GET /robots.txt HTTP/1.1"
404 148 "-" "msnbot/1.1 (+http://search.msn.com/msnbot.htm)"
```

These lines are in Apache's Combined Log Format. Most of the fields are self explanatory—IP address is the computer making the request, referer is the URL of the page (if any) that was used to reach the page, HTTP request contains the path the user was requesting, size is the number of bytes transferred as a result of the request, and so on.

In listing 7.13, you should be able to see three separate requests: one from Firefox running under Linux, one from Google's search spider,

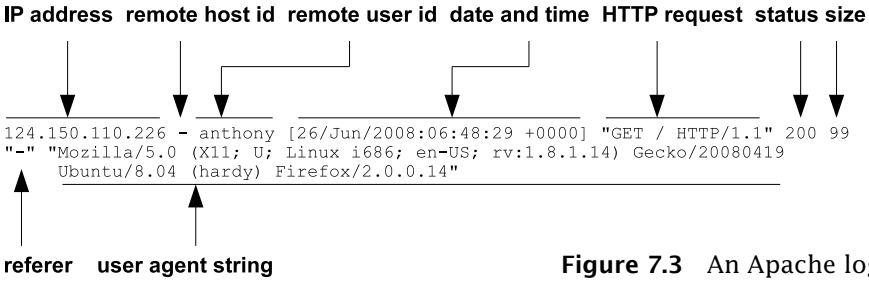


Figure 7.3 An Apache log line

and one from Microsoft's MSN. As you learned in chapter 5, the user agent string is supplied by the client and so can't be trusted, but in most cases it's accurate. The HTTP request part is the full command sent to the web server, so it includes the type of request (usually GET or POST) and the HTTP version as well as the path requested.

Pulling out the bits

That explains what the separate fields mean, but how are you going to get them? You could split on quotes or spaces, but it's possible they'll appear in odd places and throw the split function off track. For example, there are spaces in both the user-agent and the date-time string. It's also possible to have quotes within the user agent string and the URL, although they're supposed to be URL encoded.

TIP My rule of thumb is to use Python's string methods, like `endswith()` and `.split()`, when looking for simple things—but I find they can get unwieldy when you're trying to match against more complicated patterns, like the Apache log line.

In cases like this, it's usually easier to break out the big guns right at the start, rather than experiment with splitting the fields on various characters and trying to make sure it will work for everything. In this case, the fastest solution is probably to use a parsing tool called a *regular expression*, which is useful for reading single lines like this and breaking them down into chunks.

Regular expressions work by using special matching characters to designate particular types of character, such as spaces, digits, the letters *a* to *z*, *A* to *Z*, and so on. A full description of the gory details of regular expressions is out of the scope of this book, but the handy quick reference in table 7.2 will get you started.

Table 7.2 A regular expression cheat sheet

Expression	Definition
\	Regular expressions use a backslash for special characters. If you need to match an actual backslash, then you can use two backslashes together, \\.
\w	A "word" character: *a*–*z*, *A*–*Z*, 0–9, and a few others, such as underscore.

Table 7.2 A regular expression cheat sheet *(continued)*

Expression	Definition
`\W`	A non-word character—the opposite of `\w`.
`\s`	A whitespace character, such as space or tab.
`\S`	A non-whitespace character.
`\d`	A digit character, 0–9.
`.`	Any character at all.
`+`	Extends a special character to match one or more times. `\w+` will match at least one word character, but could match 20.
`*`	Like `+`, but matches zero or more instead of one or more.
`?`	You can use this after a `*` or `+` wildcard search to make them less "greedy." `.*?` will match the minimum it can, rather than as much as possible.
`()`	You can put brackets around a set of characters and pull them out later using the `.groups()` method of the match object.
`[]`	You can put characters between square brackets to match just those. `[aeiou]`, for example, matches vowels.
`r''`	A string preceded with `r` is a *raw* string, and Python won't escape any backslashes within it. For example, `"line 1\nline 2"` will normally be split over multiple lines, because Python will interpret `\n` as a return, but `r"line 1\nline 2"` won't.
`match` vs. `search`	Two main methods are used on the regular expression object. `match` will try to match from the start of a line, but `search` will look at the whole string. Normally, you'll want to use `search`, unless you know you want to match at the start.

The regular-expression string you're going to use to match the Apache log lines looks like this:

```
log_format = (r'(\S+) (\S+) (\S+) \[(.*?)\] '
              r'"(\S+) (\S+) (\S+)" (\S+) (\S+) '
              r'"(.+)" "(.+)"')
```

It looks complicated, but it's not so hard if you break it down and look at the individual parts:

- Most of the fields are groups of alphanumeric characters separated by spaces, so you can use `(\S+)` to match them. They're surrounded by brackets so you can access the fields after they've been matched. Each part corresponds to one field in the Apache log line.
- The date-and-time field is the only one with square brackets around it, so you can also match that easily and pull out everything, including spaces, with a wildcard match. Notice that you escape the `[` and `]` by putting a backslash in front of them so the regular expression treats them as normal characters.
- The referer and the user agent are also matched using wildcards, because they might have quotes or spaces in them.
- The whole string is wrapped in brackets so you can break it over multiple strings but still have Python consider them as a single string.

Now that you have a rough idea of how you can use regular expressions to match the fields in a log line, let's look at how you write the Python functions and generators to make sense of the overall scope of your log files. The following listing extends listing 7.7 to add new Apache-related functions and generators.

Listing 7.14 Parsing Apache log lines

```
import re
...

apache_log_headers = ['host', 'client_id', 'user_id',
    'datetime', 'method', 'request', 'http_proto',
    'status', 'size', 'referrer', 'user_agent']
log_format = (r'(\S+) (\S+) (\S+) \[(.*?)\] '
           r'"(\S+) (\S+) (\S+)" (\S+) (\S+) '
           r'"(.+)" "(.+)"')
log_regexp = re.compile(log_format)
```

1 Set up

```
def parse_apache(line):
    log_split = log_regexp.match(line)
    if not log_split:
        print "Line didn't match!", line
        return {}
    log_split = log_split.groups()

    result = dict(zip(apache_log_headers, log_split))
    result['status'] = int(result['status'])
    if result['size'].isdigit():
        result['size'] = int(result['size'])
    else:
        result['size'] = 0
    return result

def apache_lines(dir_name, file_type):
    return (parse_apache(line)
            for line in log_lines(dir_name, file_type))

...
if __name__ == '__main__':
    for each_line in log_lines('/var/log/apache2', '.log'):
        print each_line
        print parse_apache(each_line)

    print sum((each_line['size']
        for each_line in
            apache_lines('/var/log/apache2', '.log')
        if line.get('status', 0) == 200))
```

2 Parse apache line

3 Convert parsed line into dictionary

4 Generator for line parser

5 Use new generator

Before we get into the functions proper, it's a good idea to set up some of the variables that you'll need for your regular expressions 1. `apache_log_headers` is a list of names for all of the fields you'll see in your log file, and `log_format` is the regular expression string we looked at earlier. You also compile `log_format` into `log_regexp` so your matching is faster when you're parsing the log line.

First, you set up a function that is responsible for parsing a single line 2. Here's where you use the compiled regular expression object against the line you've been passed, using the `match` method. If it matches, `log_split` will be a match object, and you can call the

`.groups()` method to extract the parts you matched with brackets. If there's no match, `log_split` will be `None`, which means you have a line that is probably illegal. There's not much you can do in this case, so you'll return an empty dictionary.

If your function is going to be widely useful, you'll need to easily access different parts of the log line. The easiest way to do that is to put all of the fields into a dictionary ❸, so you can type `line['user_agent']` to access the user-agent string. A fast way to do that is by using Python's built-in `zip` function, which joins the fields together with the list of headers. It creates a sequence of tuples (identical to the results of an `.items()` call on a dictionary), and then you can cast that to a dictionary with the `dict()` function. Finally, you turn some of the results into integers to make them easier to deal with later.

Now that you have your line-parsing function, you can add a generator to call it for each line of the log file ❹.

If you have more information about what's in the line, you can search for more detail in your logs ❺. Here you're adding up the total size of the requests, but only where the request is successful (a status code of 200). You could also do things like exclude the Google, MSN, and Yahoo spiders to get "real" web traffic, see what volume of traffic is referred to you by Google, or add up individual IP addresses to get an idea of how many unique visitors you have.

When you run the program in listing 7.14, you should see a list of lines and their parsed representation, with a number at the end. That's the number of bytes that were transferred in successful transactions. Your program is complete, and you can start adding to it if there are particular features you'd like to add.

Functional programming

As you become more familiar with programming, you'll start to find that certain features are more or less error-prone than others. For example, if your program makes use of a lot of shared state or global variables, then you might find that a lot of your errors tend to be

around managing that state and tracking down which *!$%@%* function is replacing all your values with `None`.

It makes sense to try to find ways to design your program that don't involve error-prone features, and which are clearer and easier to understand. In turn, you'll be able to write larger programs with more features, and write them faster than you could before.

One of those strategies is called *functional programming*. Its main criterion is that it uses functions that have no side effects—their output is entirely determined by their input, and they don't modify anything outside the function. If a function is written like this, it makes it much easier to reason about and test, because you only need to consider the function by itself, not anything else.

Another feature of functional programming is that functions are objects in their own right — you can pass them as arguments to other functions and store them in variables. This might not seem particularly important, but it enables a lot of functionality that would otherwise be quite difficult to implement.

Side effects

Side effects refer to anything a function does that is outside its sphere of control or that doesn't relate to the value it returns. Modifying a global variable or one that's been passed in, writing to a file, and posting values to a URL are all examples of side effects. Functions should also only rely on values that are passed in, not on anything outside the function.

Map and filter

Once you know that functions are safe to run and aren't going to do anything weird, you can use them much more frequently—and for situations where you might not normally use functions.

Two common examples are `map` and `filter`. `map` takes a function and an iterable object, like a list or generator, and returns a list with the result

of that function applied to each item in the iterable. `filter`, on the other hand, takes an iterable function and returns only those items for which the function returns `True`.

In the case of your log files, you might have code like this:

```
errors = map(extract_error, filter(is_error, log_file.readlines()))
```

Note that `extract_error` pulls the error text from a log line, and `is_error` tells you whether the line is an error line. The result will be a new list of the error messages from your log file, and the original list will be untouched.

But, in practice, `map` and `filter` tend to make your programs less readable than using something like a list comprehension:

```
errors = [extract_error(line) for line in log_file.readlines()
          if is_error(line)]
```

A better use of functional programming is to use functions to change the behavior of other functions and classes. A good example is the use of decorators to change how functions and methods behave.

Passing and returning functions

Decorators are essentially wrappers around other functions. They take a function as an argument, potentially with other arguments, and return another function to call in its place. To use a decorator, you place its name above the function you're decorating, preceded by an `@` symbol and any arguments you need afterward, like a function.

A real-world example is Django's `user_passes_test` function, shown next, which is used to create decorators like `login_required`. `login_required` checks to see whether the user is logged in, and then either returns the regular web page if they are (Django calls them *views*) or redirects them to the site's login page if they aren't. It's fairly complex, but it uses most of the functional programming techniques described so far, plus a few others. I think you're ready to handle it, and we'll take it step by step.

Listing 7.15 Django's `user_passes_test` decorator

```
from functools import wraps                                    ← 3 Functools

def user_passes_test(test_func, login_url=None,                ← 1 User_passes_test
        redirect_field_name=REDIRECT_FIELD_NAME):                  returns decorator

    def decorator(view_func):                                  ← 2 Decorator function
        @wraps(view_func,                                      ← 3 Functools
               assigned=available_attrs(view_func))
        def _wrapped_view(request, *args, **kwargs):
            if test_func(request.user):                        ← 4 If user is
                return view_func(request,                          logged in
                                 *args, **kwargs)
            ...
            from django.contrib.auth.views \                   ← 5 If user isn't
                        import redirect_to_login                   logged in
            return redirect_to_login(
                path, login_url, redirect_field_name)
        return _wrapped_view                                   ← 2 Decorator function
    return decorator                                           ← 1 User_passes_test
                                                                   returns decorator
def login_required(function=None,                              ← 6 login_required
            redirect_field_name=REDIRECT_FIELD_NAME,               decorator
            login_url=None):
    actual_decorator = user_passes_test(
        lambda u: u.is_authenticated(),
        login_url=login_url,
        redirect_field_name=redirect_field_name
    )
    if function:
        return actual_decorator(function)
    return actual_decorator

@login_required                                                ← 7 Use
def top_secret_view(request, bunker_id, document_id):              decorator
        ...

@login_required(login_url="/super_secret/login")
def super_top_secret_view(request, bunker_id, document_id):
        ...
```

The first thing to notice is that `user_passes_test` isn't a decorator itself: it's a function that returns a function for you to use as a decorator ❶. This is a common trick if you need a few similar functions—pass in the bits that are different, and have the function return something you can use.

❷ is the decorator itself. Remember, all it has to do is return another function to use in place of `view_func`.

If you're planning on writing a few decorators, it's worth looking into `functools` ❸, a Python module that provides functional programming–related classes and functions. `wrap` makes sure the original meta information, such as the docstring and function name, are preserved in the final decorator. Notice also that you're using `*args` and `**kwargs` in your function, so the request's arguments can be passed through to the real view.

❹ is the first part of the test. If `test_func` returns `True`, then the user is logged in, and the decorator returns the results of calling the real view with the same arguments and keyword arguments.

If they're not logged in ❺, then you return a redirect to the login page instead. Note that I've snipped out some extra code that figures out `path` based on some Django internals—but that's not necessary to understand how the decorator works.

Next, you define the decorator ❻. You call `user_passes_test` with the relevant arguments and get back a function you can use in place of the real view. You also use `lambda`, which is a Python keyword you can use to define small, one-line functions. If your function is much more complex than this, though, it's usually better to define a separate function so you can name it and make it clearer.

Python will use the function returned by `login_required` in place of the real view ❼, so your `top_secret_view` function will first check to make sure the user is logged in before it returns any secret documents from one of your bunkers. You can also include arguments if you want the decorator to behave differently: in this case, by redirecting to a separate login system at /super_secret/login.

The emphasis in most programming is on objects and how they interact, but there's still a place for well-written, functional programs. Anywhere you need some extra configuration, have common functionality that can be extracted, or need to wrap something (without the overhead of a whole class), you can use functional programming.

Where to from here?

From here, you can extend your log-parsing script to capture different sorts of traffic. You could categorize log entries by type (visitor, logged-in user, search engine, bot), which section of your site they use, or what time of day they arrive. It's also possible to track individuals by IP address as they use your site, to work out how people make use of your site or to determine what they're looking for.

You can use Python's generators in other types of programs, too. If you were reading information from web pages rather than log files, you could still use the same strategies to help you reduce the amount of complexity in your code or the number of downloads you needed. Any program that needs to reduce the amount of data it has to read in, or that needs to call something repeatedly but still maintain state, can benefit from using generators.

You should also keep an eye out for areas in your programs might benefit from using some of the advanced functionality we looked at in this chapter. The secret is that, when you use it, it should make your programs simpler to understand by hiding the difficult or repetitive parts in a module or function. When you're writing an application in Django, you only need to include `@login_required` above each view you want protected—you don't need to explicitly check the request's user or redirect to a login page yourself.

Summary

In this chapter, you learned about the more advanced Python features, like generators and decorators, and you saw how you can modify classes' behavior and bend them to your will.

You saw how to alter the way a class's methods are looked up, catch a missing method, and even swap out the normal method lookups and

use your own criteria. You saw how to transparently swap out attributes for functions by using properties, and make your class behave like an integer, list, or dictionary by defining special methods.

We also looked at how to use generators to help you organize the data in your programs, and how they can reduce the memory required in your program by only loading data as it's needed, rather than ahead of time in one big chunk. We covered how to link generators together to help write more complicated programs, using an example where you parsed information from an Apache log file. We also explored using the regular expression module when you need a good way to match or extract information from some text.

Finally, we discussed functional programming, and you saw how Python supports it with `map` and `filter`, in addition to having functions that can be assigned to a variable. Then we looked at decorators and how they work in practice by defining and returning different functions.

We've covered most of Python's features, so from here on, we're going to take a slightly different tack and look at some common libraries that are used with Python. In the next chapter, we'll examine Django, the main web framework used with Python.

10

Twisted networking

This chapter covers

- *Writing networked programs in Python*
- *Designing multiplayer games (including testing them on your friends)*
- *Issues you'll encounter when writing asynchronous programs*

In this chapter, we'll be revisiting the adventure game you wrote in chapter 6 and extending it so you can log in and play it with other people via the internet. Normally these games are referred to as MUDs, which stands for Multi-User Dungeon. Depending on the person creating them, MUDs can range from fantasy hack-and-slash to science fiction, and players can compete or cooperate to earn treasure, points, or fame.

To get you started quickly, we'll use a framework called Twisted, which contains libraries for working with many different networking protocols and servers.

Installing Twisted

The first step is to install Twisted and get a test application running. Twisted comes with installers for Windows and Macintosh, which are available from the Twisted homepage at http://twistedmatrix.com/. Some

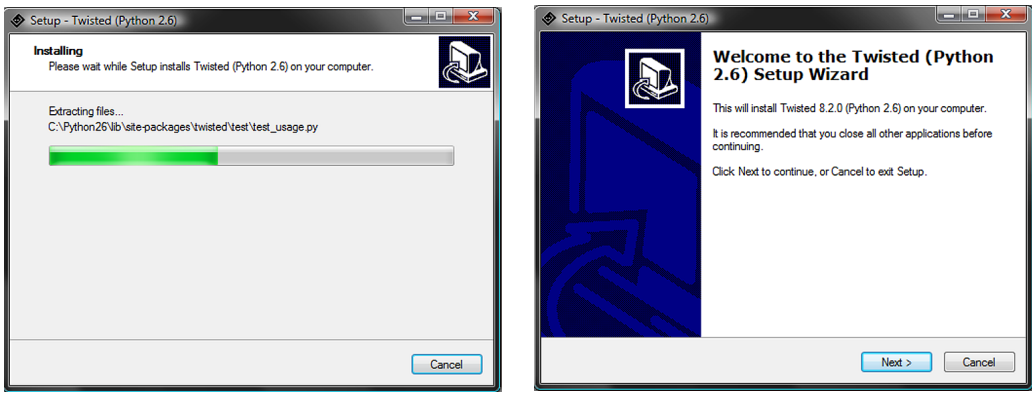


Figure 10.1 Installing Twisted on Windows

versions of MacOS ship with Twisted already installed, in which case it's easier to use that version. If you're using Linux, there should be packages available through your package manager.

The installer will pop up a window as it compiles things, but once you see the window on the right in figure 10.1, Twisted is installed!

The only other thing you need is a Telnet application. Most operating systems come with one built-in, and there are many free ones you can download. I normally use an SSH terminal program called PuTTY, which is available for Windows.

Your first application

You'll start by writing a simple chat server. The idea is that people will be able to log into it via a program called Telnet and send each other messages. It's a little more complex than "Hello World!" but you can extend this program and use it in your game later on in this chapter. Open a new file, and save it as something like chat_server.py.

Let's start with the first part of your application: the protocol for the chat server. In Twisted terminology, a *protocol* refers to the part of your application that handles the low-level details: opening connections, receiving data, and closing connections when you're finished. You can

do this in Twisted by subclassing its existing networking classes. The next listing shows a simple chat client, which you'll build on when you write your game in later sections.

Listing 10.1 A simple chat-server protocol

```
from twisted.conch.telnet import StatefulTelnetProtocol

class ChatProtocol(StatefulTelnetProtocol):
    def connectionMade(self):
        self.ip = self.transport.getPeer().host
        print "New connection from", self.ip
        self.msg_all(
            "New connection from %s" % self.ip,
            sender=None)
        self.factory.clients.append(self)

    def lineReceived(self, line):
        line = line.replace('\r', '')
        print ("Received line: %s from %s" %
                (line, self.ip))
        self.msg_all(line, sender=self)

    def connectionLost(self, reason):
        print "Lost connection to", self.ip
        self.factory.clients.remove(self)

    def msg_all(self, message, sender):
        self.factory.sendToAll(
            message, sender=sender)

    def msg_me(self, message):
        message = message.rstrip() + '\r'
        self.sendLine(message)
```

1 ChatProtocol is like Telnet

2 Override connectionMade

3 New connection

4 Handle data

5 Close tconnection

6 Convenience methods

For your chat server, you use Twisted's `StatefulTelnetProtocol` 1. It takes care of the low-level line-parsing code, which means you can write your code at the level of individual lines and not have to worry about whether you have a complete line or not.

You're customizing the protocol by overriding the built-in `connectionMade` method ❷. This will called by for each connection the first time it's made.

You're taking care of a bit of housekeeping here—storing the client's IP address and informing everyone who's already connected of the new connection ❸. You also store the new connection so you can send it broadcast messages in the future.

The Telnet protocol class provides the `lineReceived` method ❹, which gets called whenever a complete line is ready for you to use (that is, whenever the person at the other end presses the return key). In your chat server, all you need to do is send whatever's been typed to everyone else who's connected to the server. The only tricky thing you need to do is to remove any line feeds; otherwise, your lines will overwrite each other when you print them.

If the connection is lost for some reason—either the client disconnects, or is disconnected by you—`connectionLost` will be called so you can tidy things up ❺. In this case, you don't really need to do much, just remove the client from the list of connections so you don't send them any more messages.

To make the code easier to follow, I've created the `msg_all` and `msg_me` methods, which will send out a message to everyone and just you, respectively ❻. `msg_all` takes a `sender` attribute, which you can use to let people know who the message is coming from.

NOTE *Factory* is a programming term for something that creates a class for you to use. It's another way to hide some of the complexity of a library from the programmers who make use of it.

That takes care of how you want your program to behave. Now, how do you link it to Twisted? You use what Twisted refers to as a Factory, which is responsible for handling connections and creating new instances of `ChatProtocol` for each one. You can think of the Factory

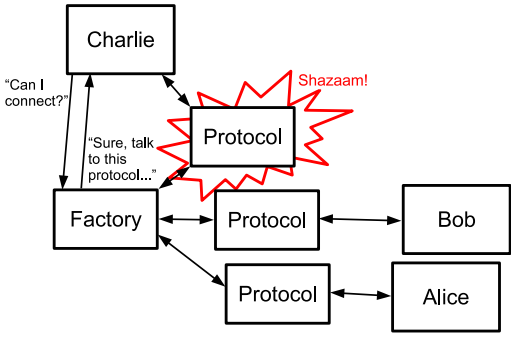


Figure 10.2 A Factory creating protocols

as a switchboard operator: as people connect to your server, the Factory creates new protocols and links them together, similar to figure 10.2.

So how do you do this in Twisted? Easy! Add a factory class, as shown in the next listing.

Listing 10.2 Connecting your protocol

```
from twisted.internet.protocol import ServerFactory
from twisted.internet import reactor
   ...
class ChatFactory(ServerFactory):
    protocol = ChatProtocol

    def __init__(self):
        self.clients = []

    def sendToAll(self, message, sender):
        message = message.rstrip() + '\r'
        for client in self.clients:
            if sender:
                client.sendLine(
                    sender.ip + ": " + message)
            else:
                client.sendLine(message)

print "Chat server running!"
factory = ChatFactory()
reactor.listenTCP(4242, factory)
reactor.run()
```

1 ChatFactory?

2 Wiring everything together

3 Talking to everyone

2 Wiring everything together

A Factory is object-oriented terminology for something that creates instances of another class 1. In this case, it will create instances of `ChatProtocol`.

The `ChatFactory` is the natural place to store data that is shared among all of the `ChatProtocol` instances. The `sendToAll` method is responsible for sending a message to each of the clients specified in the clients list 2. As you saw in listing 10.1, the client protocols are responsible for updating this list whenever they connect or disconnect.

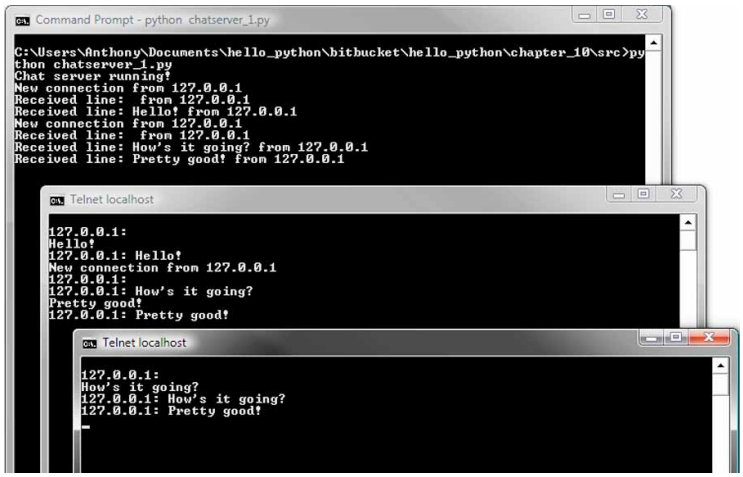
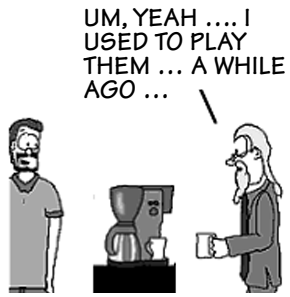


Figure10.3 Your chat server is running.

The final step is to let Twisted know about your new protocol and Factory. You do this by creating an instance of `ChatFactory`, binding it to a particular port with the `listenTCP` method, and then starting Twisted with a call to its main loop, `reactor.run()` ❸. Here you use 4242 as the port to listen to—it doesn't matter too much which one you use, as long as it's above 1024 so it doesn't interfere with existing network applications.

If you save the program and run it, you should see the message "Chat server running!" If you connect to your computer via Telnet on port 4242 (usually by typing `telnet localhost 4242`), then you should see something like figure 10.3.

It may not seem like much, but you've already got the basic functionality of the MUD server going. If you'd like to explore the chat server further, there's a more fully featured version included with the source code, available from http://manning.com/HelloPython/. That version adds commands to change your name and see who else is connected, as well as limit some common sorts of misbehavior and allow you to remove anyone who's behaving badly.



First steps with your MUD

Now you're ready to start connecting your adventure game to the network. You'll base it on the chat server, but instead of only broadcasting what's typed to everyone who's connected, you'll feed it directly into the adventure game. This is a common way to get things done when programming—find two programs (or functions or libraries) that do separate parts of what you need, and then "glue" them together.

The basic gist is that you'll have multiple players logged in at once, all trying to execute commands (such as "get sword") at the same time. This could potentially cause problems for the server because you're mixing real-time Twisted code with the one-step-at-a-time of your adventure game. You'll head off most of the issues by queuing up player commands and updating your game's state once per second.

Let's get started. Copy your adventure code from chapter 6 into a new folder, along with the chat-server code you just created. You'll probably also want to rename chat_server.py to something like mud_server.py, to help keep things straight, and rename your classes and variables as in the next listing.

Listing 10.3 Updating your chat protocol

```
from game import Game
from player import Player
import string

class MudProtocol(StatefulTelnetProtocol):

    def connectionMade(self):
        self.ip = self.transport.getPeer().host
        print "New connection from", self.ip

        self.msg_me("Welcome to the MUD server!")
        self.msg_me("")
        ...
        self.player = Player(game.start_loc)
        self.player.connection = self
        game.players.append(self.player)
```

1 Update imports and classes

2 Redirect input to Player class

3 Welcome message

4 Create player when connecting

```
def connectionLost(self, reason):
    ...
    game.players.remove(self.player)
    del self.player

def lineReceived(self, line):
    line = line.replace('\r', '')
    line = ''.join([ch for ch in line
                if ch in string.printable])
    self.player.input_list.insert(0, line)
```

5 Remove players when disconnecting

2 Redirect input to Player class

The first step is to import the `Game` and `Player` classes into your code **1**. I've also changed the name of the protocol so it's obvious what you're trying to write.

Next, you give a nice, friendly start when someone first connects to your MUD **3**.

Now you'll start to do the real work. But it turns out to not be that hard. I've assumed the game will keep track of its players somehow, and added a new player object to the game's list of players **4**. To make it possible for you to talk to the player from within the game, I've also added the protocol to the player. You'll see how that works in a minute.

You'll still need to handle the case where a player disconnects from the server **5**. But, again, it's straightforward: remove them from the game's list of players, and delete them.

Once players are connected, they'll want to type commands, like "go north" and "attack orc" **2**. First, you sanitize the input you've received (in testing, I found that different Telnet programs can send different weird characters). When it's trimmed down to only printable characters, you assume the player has a list of commands waiting to be executed, and push this one to the end.

Your protocol is done, but what about the Factory and the rest of it? It turns out that you don't need to do too much to your Factory—just change a few lines.

Listing 10.4 Updating your chat Factory

```
from twisted.internet import reactor, task
...
class MudFactory(ServerFactory):
    protocol = MudProtocol
    ...

game = Game()
# game.run()

def run_one_tick():
    game.run_one_tick()

print "Prototype MUD server running!"
factory = MudFactory()
game_runner = task.LoopingCall(run_one_tick)
game_runner.start(1.0)
reactor.listenTCP(4242, factory)
reactor.run()
```

❷ Update Factory

❸ Create Game instance

❶ Run game updates

You don't need to do too much to update your Factory ❷—change its protocol and rename it.

You'll need a `Game` object, too, so you create it here ❸. You don't want to use the old `run` method, though, because it still handles things the old way.

The design calls for you to run a game update once per second. Because you're using Twisted's event loop (that's the `reactor.run()` part), you'll need to use Twisted's `task.LoopingCall` to call the game's update method ❶, `run_one_tick`, which you'll also create shortly.

That should be all you need to do to the network code for now. You've made a few assumptions about how the game code will work, but often this is easier than jumping back and forth between the `Game` and `MudProtocol` classes and trying to fit it all together. Now that your protocol is written, you have to make `Game` and `Player` play along, too.

Listing 10.5 Changing your game code to work with the new interface

```
class Game(object):

    def __init__(self):
        ...
        #self.player = player.Player(cave1)
        self.players = []
        self.start_loc = cave1
    ...
    def run_one_tick(self):
        self.do_input()
        self.do_update()
        self.send_results()
       ...
    def send_results(self):
        """Send results of actions to players"""
        things_to_update = [thing for cave in self.caves
            for thing in cave.here
            if 'send_results' in dir(thing)]
        for thing in things_to_update:
            thing.send_results()
```

1 Don't create only one player

2 Game loop

3 Send results

The single-player version of the adventure game had one player, but you'll potentially have many, so you'll make it a list instead 1. You're also giving the starting cave a sensible name.

2 is the main loop you called from the networking part of your code. You should be able to follow what it's doing from the names—get input for each `Player` object (including monsters), run the update, and then send the results back.

You already have methods for getting input and processing orders, but you'll need something to send back the results of each player's actions 3. To do that, you'll make another assumption: that each `Player` object knows how to send results back to the player.

Now you have only two assumptions left to fill in, and they're both in the `Player` class. The first is that `Player` will have a list of pending commands, and the second is that it will have a way to send the results of any commands or events back to the player. The other thing you need

to do is make sure the `Player` class reads from the list of pending commands, rather than using `raw_input`.

Listing 10.6 Changing the `Player` code

```
class Player():
    def __init__(self, location):
        ...
        self.input_list = []
        self.result = []

    def get_input(self):
        #return raw_input(self.name+">")
        if self.input_list:
            return self.input_list.pop()
        else:
            return ''

    def send_results(self):
        for line in self.result:
            self.connection.msg_me(line)
        for line in self.events:
            self.connection.msg_me(line)

    def die(self):
        self.playing = False
        self.input = ""

        self.name = "A dead " + self.name


        for item in self.inventory:
            self.location.here.append(item)
            item.location = self.location
        self.inventory = []

        try:
            self.connection.msg_me("You have died! "
                                   "Better luck next time!")
            self.connection.transport.loseConnection()
        except AttributeError:
            pass # not everyone has a connection

class Monster(player.Player):
```

1 Add input and output buffers to Player

2 Update get_input

3 Add send_results methods

4 What happens when you die?

```
...
def send_results(self):
    pass

def get_input(self):
    if not self.playing:
        return ""
    player_present = [x for x in self.location.here
        if x.__class__ == player.Player
           and x.playing]
```

❺ **Stub out send_results in Monster**

❻ **May be more than one player**

You start by creating your list of pending commands and the result that needs to be sent back to the player ❶. They're just lists, and when they're in use they'll have a list of strings.

You can't use `raw_input` any more, so you need to read your next command from `self.input_list` ❷. `pop` removes the command for you so you don't have to worry about removing it from the list later. `pop` called on an empty list raises an exception, so you check for that case and assume the command is blank if there's nothing there.

To send the results of a player's actions ❸, you use the `self.connection` object that you set up in mudserver.py. Note that even if the player isn't doing anything, other players and monsters are, so you have two separate sections: one for the results of your actions and another for events.

In the old version of the game, when the player died, the game ended. That's no longer the case, so you'll need to gracefully handle the situation where a player dies ❹. To do that, you make the player drop whatever they're carrying, send them a message, and drop the connection. If you extend your game, you might want to make the player keep their items. Alternatively, you can allow other players to "get sword from Anthony" if you're feeling mean.

Monsters don't connect over the network and don't have the `self.connection` object, so the default `send_results` from the `Player` class won't

work. They don't need to know the results of their actions, so you'll stub out their version of `send_results` and return immediately ❺.

The previous adventure game looked at the player's name to figure out whether to attack them. Now that you have multiple players, who probably all have different names, you'll need to be a bit more discerning ❻. A better way is to examine the class of the object the monster is looking at, using the `__class__` method. That will return the class, which you can compare to `player.Player`.

NOTE This works so well because your game has only one point of communication with the player: the commands the player types and the responses the game returns.

That should be all you need to do. Now, when you run your server and connect via Telnet, you'll see your familiar adventure-game prompt, and you can run around the server collecting loot and slaying monsters. Go ahead and bask in the adventure and glory.

Well, sort of. Although the game works, and you can explore and do everything you need to, there are a few more things to take care of before your game is playable.

Making the game more fun

I made the previous code available to some of my friends online and got feedback from them. They raised two major issues: the monster was too hard to beat, and there wasn't enough interaction between the players. Normally, in an adventure game like this, you'll be able to change your name and description, talk to other players, look at their description, and so on.

Bad monster!

The problem with combat is pretty obvious once you run into the orc for the first time. You're limited to the actions you type in—but the monsters react at computer speed. The next figure shows what I mean.

Figure 10.4 Bad monster! No beating on the player!

The solution that most MUDs use is what's known as an *angry list*. Rather than attacking things directly, the game maintains a list of monsters and other players you're angry at. If you're not explicitly doing anything else, and there's something present that's on your angry list, then you'll attack it. If something attacks you, then it will go on your angry list, too, so you'll at least put up a token defense. Let's look at how you can implement the angry list in your game.

Listing 10.7 Angry lists

```
class Player(object):
    def __init__(self, game, location):
        ...
        self.angry_list = []                    # 1 Add angry list

    def update(self):
        self.result = self.process_input(self.input)

        if (self.playing and
            self.input == "" and
            self.angry_list):
            bad_guys = [x for x in self.location.here   # 2 Attack bad guys
                        if 'attack' in dir(x) and
                           x.name in self.angry_list]
```

```
            if bad_guys:
                bad_guy = random.choice(bad_guys)
                self.events += bad_guy.do_attack(self)

    def die(self):
        ...
        self.angry_list = []
    ...
    def stop(self, player, noun):
        self.angry_list = [x for x in self.angry_list
                           if x.name != noun]
        return ["Stopped attacking " + noun]

    def attack(self, player, noun):
        player.angry_list.append(self.name)
        self.angry_list.append(player.name)
        result = ["You attack the " + self.name]
        result += self.do_attack(player)
        return result

    def do_attack(self, player):
        """Called when <player> is attacking us (self)"""
        hit_chance = 2
        ...
    actions = [..., 'stop']
```

2 Attack bad guys

3 Dead players tell no tales

4 Stop attacking

5 Modify attack method

6 Add stop to list of actions

Both players and monsters will need a way to remember who they're angry at. You'll make it a list 1, because you're not expecting it to grow too large.

Next, you'll modify your `update` method. If your `input` attribute is blank, you know that the player (or monster) hasn't entered any commands, and you can go ahead and attack if necessary. You build a list of all the things you're angry at that are present, and then attack one of them 2.

If a player or monster is dead, they shouldn't keep attacking, so you clear their angry list 3.

The players will also need a way to stop attacking things (maybe they're friends again). The `stop` command will remove an attacker from the list of things that the player is angry at ❹.

The final major thing you'll do is make the `attack` command modify the angry lists of both the attacker and attacked ❺. Now, when something gets attacked, it will automatically fight back. Note how you build your result before you do the attack. That way, if the target dies, you won't see "You attack the dead orc." `do_attack` is the mechanism from your old `attack` attribute with a different name.

The final, final thing is to add `stop` to your list of commands ❻—otherwise you won't be able to use it!

Now the player should have half a chance against the orc. If the orc beats the player now, the player will at least feel that they haven't been completely robbed by the game. If you pick up the sword, you'll find it helps a lot, which is what you want. There are plenty of other opportunities for improving the combat system, but you need to deal with a more pressing problem, instead.

Back to the chat server

The second problem is that players can't interact with each other. This is often a big draw when it comes to a multiplayer game—players will come for the game but stay for the company. Fortunately, making your game more social is easy to do. You'll add a few extra commands to the `Player` class.

Listing 10.8 Social gaming

```
help_text = """
Welcome to the MUD

This text is intended to help you play the game.
Most of the usual MUD-type commands should work, including:
...
"""


class Player(object):
    ...
```

❶ Help!

```
def help(self, player, noun):
    return [help_text]

def name_(self, player, noun):
    self.name = noun
    return ["You changed your name to '%s'" % self.name]

def describe(self, player, noun):
    self.description = noun
    return ["You changed your description to '%s'" %
            self.description]

def look(self, player, noun):
    return ["You see %s." % self.name, self.description]

def say(self, player, noun):
    for object in self.location.here:
        if ('events' in dir(object) and
            object != self):
            object.events.append(
                self.name + " says: " + noun)
    return ["You say: " + noun]
```

1 Help!

Change name and description 2

3 Look

4 Talk to people

If a player is completely new to the game, you need to give them at least half an idea of what they can do. You'll make "help" output some helpful instructions 1. The full help text I added is in the `source` code.

Another easy win is to let players customize their appearance by changing their name and description 2. Rather than being "player #4," the player can now be "Grognir, Slayer of Orcs."

Of course, the description's not much good if other players can't see it 3.

You'll also need to add to most important case of all: a `say` command, so your players can talk to each other 4. All this command needs to do is send what you've typed to every other object in the current room. This simple change will allow players to interact on a human level, which will in turn help keep them coming back.

One of the issues you'll run into is that with the new commands, the old `find_handler` method will sometimes call the wrong thing. For example, both the `player` and the `location` have a `look` method, and which one is correct will depend on the context. Additionally, some of the commands you've just added only apply to the players themselves, and you shouldn't look for an object to apply them to. The following listing has an updated version that is a lot more explicit about which objects it should look at.

Listing 10.9 Updating `find_handler`

```
no_noun_verbs = ['quit', 'inv', 'name_', 'describe',
                 'help', 'say', 'shout', 'go']
...
def find_handler(self, verb, noun):
    if verb == 'name':
        verb = 'name_'
    if verb == "'":
        verb = 'say'

    if noun in ['me', 'self']:
        return getattr(self, verb, None)

    elif noun and verb not in self.no_noun_verbs:
        # Try and find the object
        object = [x for x in self.location.here + self.inventory
                  if x is not self and
                     x.name == noun and
                     verb in x.actions]
        if len(object) > 0:
            return getattr(object[0], verb)
        else:
            return False

    # if that fails, look in location and self
    if verb.lower() in self.location.actions:
        return getattr(self.location, verb)
    elif verb.lower() in self.actions:
        return getattr(self, verb)

def process_input(self, input):
    ...
```

1 Non-noun verbs

2 Special cases

3 Talking to yourself

4 Can't find that

```
    handler = self.find_handler(verb, noun)
    if handler is None:
        return [input+"? I don't know how to do that!"]
    elif handler is False:
        return ["I can't see the "+noun+"!"]
actions = […, 'name_', 'describe', 'look', 'help', 'say',...]
```

❹ Can't find that

Let's pay close attention to word choice. Some verbs don't apply to nouns, or else they implicitly apply to the player ❶.

There are a few special-case commands ❷ that you can't handle with your current system. You could rewrite the entire handler, but it's easier to catch those commands and explicitly convert them to something you *can* handle. Of course, if it becomes more than a handful of conversions, then you'll have to rethink things; but it will do for now.

So that you can see how you look, you'll add a `self` object, too ❸. "Look self" should return your description as it appears to other people.

❹ is another improvement to make things easier for the new player. Rather than have one error message when things go wrong, you'll have one for a command you don't understand, and another when you can't find what the player's looking for.

Now your players can chat to each other and compliment each other on their fine threads.

Finally, what would social gaming be without the opportunity to be antisocial, too? Most MUDs have the option to shout, which works much like speaking, except that everyone connected can hear you.

Listing 10.10 Antisocial gaming

```
def shout(self, player, noun):
    noun = noun.upper()
    for location in self.game.caves:
        for object in location.here:
            if ('events' in dir(object) and
                object != self):
```

❶ Shouting looks like shouting

❷ Send to all caves

❸ Find objects that can hear shout

```
                    object.events.append(
                        self.name + " shouts: " + noun)
        return ["You shout: " + noun]

class Player(object):
    def __init__(self, game, location):
        self.game = game
        ...

class Monster(player.Player):
    def __init__(self, game, location, name, description):
        player.Player.__init__(self, game, location)
        ...

class Game(object):
    def __init__(self):
        ...
        orc = monster.Monster(self, self.caves[1],
                'orc', 'A generic dungeon monster')

class MudProtocol(StatefulTelnetProtocol):
    def connectionMade(self):
        ...
        self.player = Player(game, game.start_loc)
```

3 Find objects that can hear shout

4 Update Player class

5 Update all Player and Monster instances

First, you'll convert the text to uppercase 1, SO THAT IT LOOKS A LOT MORE LIKE SHOUTING!

Now you need to visit each cave in turn—but there doesn't seem to be any way to find out what the caves are. For now, you'll assume you have access to the game's list of caves 2.

This is pretty much the same as when players talk to each other 3. Merging the two together—for example, by pushing the code into the location class—is left as an exercise for the reader.

Now you need to give your `Player` class access to the caves list from the game 4 by making it a variable you pass in from the game when you

create a player or monster. Then, update each place where you create an instance of a player or monster ❺, so it now knows about the `game` object and can tell where all the caves are.

There! That's a few more rough edges smoothed off your game. There's plenty left to do, but you won't be writing any new features for the game now. Instead, you'll focus on making the infrastructure around the game a bit more robust, so players won't be put off by having all their hard work disappear.

Making your life easier

If you only want to write the game for your friends, you can probably stop here; they can connect and play your game, after all. Currently, though, there are still a few issues that will make your life harder than it needs to be. Anyone can log on as anyone else, so the game isn't particularly secure; and the game doesn't save any progress, so every time you restart the game server, the player will have to start over from scratch.

Let's fix that. You'll add usernames and passwords to the game, as well as a mechanism to allow new players to register. Once you know who's logged on, you can save the players' progress every so often, and also when they quit the game. You'll need to learn a bit more about Twisted, though, because you'll be digging into the guts of one of its Telnet classes. But don't worry; it's straightforward once you get the hang of it.

Exploring unfamiliar code

Twisted is a large codebase and has a huge number of modules to help you network your application. That's great, because it means you don't have to write your own code to handle the networking in your application, but it raises a related problem: you must have at least a basic understanding of how everything fits together before you can make use of all that great code.

Ideally, the documentation for libraries like Twisted would be 100% up-to-date and cover everything you need to do with a nice, gentle introduction—but this isn't always the case. Often, you'll be able to

find something close, but then you'll need to piece together how the code works with some guessing, experimentation, and detective work.

It sounds hard, but in practice it's usually pretty easy. The trick is not to get too overwhelmed, and to use *all* the resources at your disposal. Here are some ideas on how you can get a grip on a large codebase and make it work in your application.

FIND AN EXAMPLE

Searching for "twisted tutorial" online gives you a number of starting points, and you can also add "telnet" or "telnet protocol" into the mix. As you learn more about Twisted, you'll find other keywords or method names that will help you narrow down what you're looking for. You can also start with a working example that sort of does what you need, and then tweak it until it covers exactly what you need it to do.

THE TWISTED DOCUMENTATION

There's reasonably comprehensive documentation available in the Conch section of the main Twisted site, http://twistedmatrix.com/documents/, but it doesn't cover all of what you need to do. There are some simple examples of SSH and Telnet servers, which you can skim through to get an idea of how everything fits together.

THE TWISTED API DOCS

Detailed, automatically generated documentation is available for the entire Twisted codebase, which you can see at http://twistedmatrix.com/documents/current/api/. Don't let the sheer number of packages put you off—we'll focus on the Telnet one: http://twistedmatrix.com/documents/current/api/twisted.conch.telnet.html.

THE TWISTED CODE

You can also read most of the Twisted code directly. The Windows version of Python stores its libraries at C:\Python26\Lib\site-packages\twisted; under Linux, it will be somewhere like /usr/lib/python2.6/dist-packages/twisted; and under Mac, it's usually at /Developer/SDKs/

MacOSX10.6.sdk/System/Library/Frameworks/Python.framework/Versions/2.5/Extras/lib/python/twisted. All the Twisted code is stored there, and you can open the files and read the code to find out exactly what a method does.

INTROSPECTION

If a library doesn't have API documentation, all is not lost. You can still create instances of classes and use `dir()`, `help()`, and `method.__doc__` to find out what they do. If you have a one-off method you need to know about, this can often be easier than reading the code or documentation.

In practice, none of these sources will cover all the details you need to know when writing your program, so you'll end up using a combination of them going back and forth as you learn new parts or run into problems.

Putting it all together

Let's get started putting together your login system. From a quick scan of Twisted's `Telnet` module, it looks like the best starting point is the `AuthenticatingTelnetProtocol` class. You'll get that working with your code, then make it register new players, and finally make the game able to save player data.

To start with, I looked at the Twisted documentation and the API reference for `AuthenticatingTelnetProtocol`. It sort of made sense, but from the methods and classes it's hard to see how to tie everything together. The protocol needs a `Portal`, which in turn depends on a `Realm`, an `Avatar`, and a `PasswordChecker`. Hmm, confusing. It looks like it's time to try to find an example of how the classes fit together.

There are a few different searches you could try: "twisted telnet," "twisted telnet example," and so on, but I didn't find much until I put in some terms from the code. The search "twisted telnet TelnetProtocol example" led me to www.mail-archive.com/twisted-python@twistedmatrix.com/msg01490.html, which, if you follow it through to the end, gives you some example code that is enough to see how the classes work together.

The basic gist is something like this: set up a `Realm` class, along with a `Portal` to get into it. The docs don't say whether it's a *magic* portal, but it should do. A `Portal` controls access to your `Realm`, using one or more password `Checker`s, via a `TelnetTransport`. Of course, the `AuthenticatingTelnetProtocol` only handles authentication, so you'll need to hand off to another protocol like your `MudProtocol` once you're logged in.

Got all that? No, me neither. I had to draw a picture to see how it all worked, and without the example I probably would've been lost. Figure 10.5 shows what I came up with.

Using the diagram and the example code, you can get a simple login going. The following listing shows how I changed the mudserver.py file.

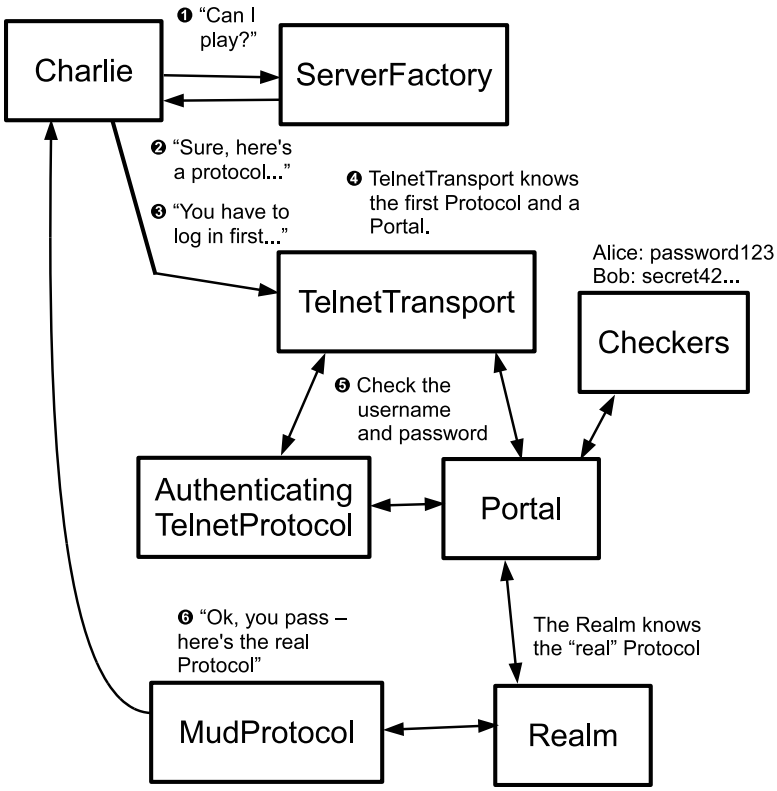


Figure 10.5 The Twisted class structure

Listing 10.11 Mudserver.py

```
import sys

from zope.interface import implements
from twisted.internet import protocol, reactor, task
from twisted.python import log

from twisted.cred import portal
from twisted.cred import checkers
from twisted.cred import credentials

from twisted.conch.telnet import AuthenticatingTelnetProtocol
from twisted.conch.telnet import StatefulTelnetProtocol
from twisted.conch.telnet import ITelnetProtocol
from twisted.conch.telnet import TelnetTransport

...
class Realm:
  implements(portal.IRealm)

  def requestAvatar(self, avatarId, mind, *interfaces):
    print "Requesting avatar..."
    if ITelnetProtocol in interfaces:
      av = MudProtocol()
      print "**", avatarId, dir(avatarId)
      print "**", mind, dir(mind)
      av.name = avatarId
      av.state = "Command"
      return ITelnetProtocol, av, lambda:None
    raise NotImplementedError("Not supported by this realm")

...
class MudProtocol(StatefulTelnetProtocol):

    def connectionMade(self):
        …
      # self.factory.clients.append(self)
        self.player.name = self.name
        checker = portal_.checkers.values()[0]
        self.player.password = checker.users[self.player.name]
        game.players.append(self.player)

    def connectionLost(self, reason):
        print "Lost connection to", self.ip
```

1 Lots of imports!

2 Create Realm

3 Use debugging strings

2 Create Realm

6 Set up ServerFactory

4 Find player's username and password

```
        if 'player' in dir(self):
            if self.player in game.players:
                game.players.remove(self.player)
            del self.player

if __name__ == '__main__':
    print "Prototype MUD server running!"

    realm = Realm()
    portal_ = portal.Portal(realm)
    checker = checkers.InMemoryUsernamePasswordDatabaseDontUse()
    checker.addUser("AA", "aa")
    portal_.registerChecker(checker)

    game = Game()
    ...
    factory = protocol.ServerFactory()
    factory.protocol = lambda: TelnetTransport(
        AuthenticatingTelnetProtocol, portal_)

    log.startLogging(sys.stdout)
    reactor.listenTCP(4242, factory)
    reactor.run()
```

4 Find player's username and password

5 Set up Realm and PasswordCheckers

6 Set up ServerFactory

7 Send logging to screen

To start, you'll import all the bits of Twisted you need 1. There are a lot, but think of it as code you don't have to write.

The `Realm` is the core class that represents your game's login 2. You only need to override one method: the one to get an `Avatar`. `Avatar`s are instances of the `MudProtocol` and represent the player's login. Notice that you set the player's name so you have access to it in `MudProtocol`, and set `state` to `"Command"`; otherwise, you'll get logged out right away.

NOTE The "code that you don't have to write" part is important. It's easy to overestimate how hard it is to learn how existing code works, and underestimate how hard it is to write new code that's as well tested.

While you're figuring out how everything works, it's perfectly fine to print out things to the screen to try and work out what each object does 3. You can use what you learn to search online, or through the code to find out what else uses these classes.

Most of `MudProtocol` is unchanged, but you'll need to know your player's username and password for later ❹, when you start saving to a file. The `Realm` has already given you the username, so you can use that to get the password from the checker. The other thing you change is the `connectionLost` method—if you lose the connection to the player, you want to clean up properly.

Now we're into the section where you set the code in motion. The first thing to do is create a `Realm` and then attach a `Portal` and `Checker`s to it. Once you've done that, you can insert usernames and passwords into your checker ❺. `InMemory..DontUse` is fine for your purposes, even though, in theory, it's insecure and you're not supposed to use it. There's also a file-based checker available, but it doesn't support saving new users back to the file.

Now that you're using `TelnetTransport` and your `Realm` to control things, you don't need a custom Factory, and you won't need to manually track the clients in the factory any more ❻. The `TelnetTransport` will use `AuthenticatingTelnetProtocol` to handle usernames and passwords, but once that's done it will hand off to the `Realm` to get the final protocol.

One last thing is that Twisted uses Python's log facility. To see what it's up to, you can add this line ❼, which will redirect the logging to `sys.stdout`—that is, print it on the screen.

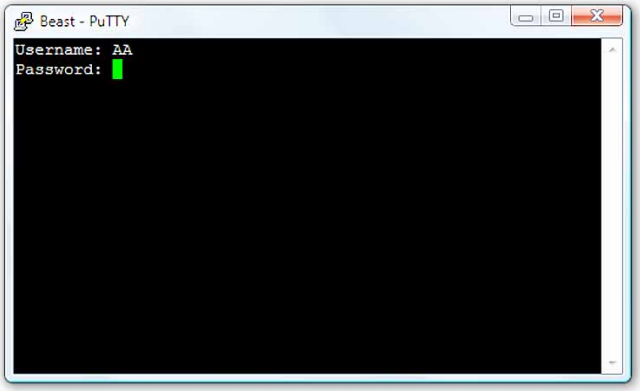


Figure 10.6 Logging in to your game

What does all this give you? Well, if you run your server now and try to connect to it, you should be presented with a login request instead of a password, similar to what's shown in figure 10.6. If you enter the username and password that are in the script, you should connect to the game.

That's not all you need to do, though. Remember that you want to allow players to register their own username and password. For that you'll have to learn a bit more about Twisted.

Write your own state machine

What you're going to do in this section is create a subclass of the class you've been using so far, which is `AuthenticatingTelnetProtocol`. It's what generates the `Username:` and `Password:` prompts in the login. What you'd like instead is a prompt that asks the player whether they want to log in or register a new account. If it's a registration, then it still asks you for a username and password, but creates the account instead of checking whether it exists.

Let's first take a look at `AuthenticatingTelnetProtocol`, to see how it's done. You can find the Telnet module on your computer at C:\Python26\Lib\site-packages\twisted\conch\telnet.py, or somewhere like /usr/lib/python2.6/ site-packages/twisted/conch/telnet.py if you're using Linux or MacOS X. If you open that file and scroll to the bottom, you'll find the class you're looking for; it's also shown in listing 10.12.

Listing 10.12 Twisted's `AuthenticatingTelnetProtocol` class

```
class AuthenticatingTelnetProtocol(StatefulTelnetProtocol):
    ...
    def telnet_User(self, line):
        self.username = line
        self.transport.will(ECHO)
        self.transport.write("Password: ")
        return 'Password'

    def telnet_Password(self, line):
        username, password = self.username, line
        del self.username
        def login(ignored):
            creds = credentials.UsernamePassword(
                        username, password)
            d = self.portal.login(creds, None, ITelnetProtocol)
            d.addCallback(self._cbLogin)
            d.addErrback(self._ebLogin)
```

1 AuthenticatingTelnetProtocol uses StatefulTelnetProtocol
2 Skip some bits
3 Username
4 Password
5 Fancy Twisted bits

```
        self.transport.wont(ECHO).addCallback(login)
        return 'Discard'

    def _cbLogin(self, ial):                          # 6 If everything goes great: callback
        interface, protocol, logout = ial
        assert interface is ITelnetProtocol
        self.protocol = protocol
        self.logout = logout
        self.state = 'Command'

        protocol.makeConnection(self.transport)
        self.transport.protocol = protocol

    def _ebLogin(self, failure):                      # 7 If everything goes bad: errorback
        self.transport.write("\nAuthentication failed\n")
        self.transport.write("Username: ")
        self.state = "User"
```

All the `Telnet` classes we've looked at so far are state machines—there are multiple steps involved in logging in, and the next one depends on the input you get. You're initially in the `"User"` state, which means input is fed to the `telnet_User` method ❶. Each method returns a string, which determines the next state.

There are a few other methods: `connectionMade` and `connectionLost`, but you don't need to deal with them in this case ❷.

The first line (after the initial greeting) goes to `telnet_User` and sets the username within the instance ❸. The `transport.will()` call tells the local client that the server (that is, you) will be responsible for echoing anything the user types—but in this case, it's the password, so you don't. Then `"Password"` is returned, so the next line goes to `telnet_Password`.

Now that you have the password, you can compare it with what you have for that username in the portal's password checker ❹.

Twisted has a mechanism called a *Deferred*, that helps to speed up the server ❺. A password checker might look at a file on disk, or connect to a different server to see whether the password is correct. If it waits

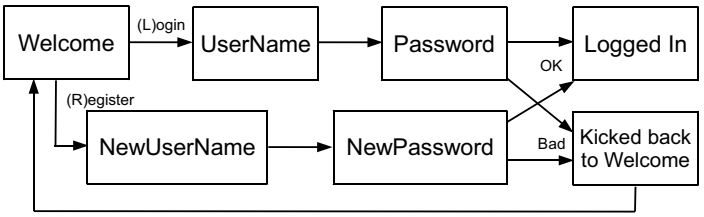


Figure 10.7 The states in `RegisteringTelnetProtocol`

for the result (normally known a *blocking*), nobody else will be able to do anything until the disk or remote server responds. Deferred objects are a way to say "When we get a response, handle it with this function" and then continue with other tasks. There are two possibilities: a callback and an error back.

If the checker responds that the password is right ❻, you can go ahead and do the rest of the login, which means storing some values, setting your state to `"Command"`, and switching out your protocol for the final one.

If the checker tells you the password or the username is wrong ❼, then you can tell the user off and switch back to the `"User"` state. The user will need to type in the username and password again—and you the user will get it right this time.

How can you subclass `AuthenticatingTelnetProtocol`? The answer is to add some new states so there's a registration branch as well as the normal login one, similar to the flowchart in figure 10.7.

The next listing adds a new protocol with three extra states—`"Welcome"`, `"NewUserName"`, and `"New Password"`—along with methods to handle each of them.

Listing 10.13 `RegisteringTelnetProtocol`

```
from twisted.conch.telnet import ECHO
class RegisteringTelnetProtocol(
          AuthenticatingTelnetProtocol):
    state = "Welcome"

    def connectionMade(self):
        self.transport.write("Welcome to the server!")
        self.transport.write("(L)ogin or (R)egister "
                             "a new account? ")
```

❶ **Welcome to server**

```
def telnet_Welcome(self, line):
    if line.strip().lower() == 'r':
        self.transport.write(
            "Enter your new username: ")
        return "NewUserName"
    elif line.strip().lower() == 'l':
        self.transport.write('Username: ')
        return "User"
    self.transport.write(
        "I don't understand that option.")
    return 'Welcome'

def telnet_NewUserName(self, line):
    for checker in self.portal.checkers.values():
        if line.strip() in checker.users:
            self.transport.write(
                "That account already exists! ")
            return "Welcome"
    self.username = line
    self.transport.will(ECHO)
    self.transport.write(
        "Enter your new password: ")
    return "NewPassword"

def telnet_NewPassword(self, line):
    self.transport.write(
        '\r\nWelcome to the server!\r\n')
    self.addNewUser(self.username, line)
    return self.telnet_Password(line)

def addNewUser(self, username, password):
    for checker in self.portal.checkers.values():
        checker.addUser(username, password)

def _ebLogin(self, failure):
    self.transport.write("\nAuthentication failed:"
        " %s (%s)\n" % (failure, dir(failure)))
    self.connectionMade()
    self.state = "Welcome"

factory = protocol.ServerFactory()
factory.protocol = lambda:
TelnetTransport(RegisteringTelnetProtocol, portal_)
```

2 Pick path

3 Register new name

3 Register new name

4 As long as it's not taken

5 Add user

6 Add user

7 Handle errors properly

8 Update factory protocol

Welcoming the user to the server ❶ is pretty much the same as the previous example, only with different values. You're prompting the user to enter `R` to register or `L` to login.

Because your previous state was `"Welcome"`, the first method is `telnet_Welcome`. The code is straightforward: `R` sets the state to `"NewUserName"`, `L` to `"User"`, and anything else will kick them back to `"Welcome"` ❷.

`telnet_NewUserName` is the same as `telnet_User`, too ❸. It prompts slightly differently and passes to a different state: `"NewPassword"` instead of `"Password"`.

Of course, you can't have two Gandalfs or Conans running around your server, so you need to check that the username doesn't already exist on the server ❹. If it does, you kick the user back to `"Welcome"`. Pick something more original!

Now that the player has passed all the hurdles you've set, you should probably add the player to the server ❺. To make life easier for the player, you also automatically log the player in.

The last bit didn't add the user, it only pretended to. ❻ will do the trick. You're calling each of your checkers in turn and calling their `addUser` method. Note that this won't work if you use the file-based checker, `twisted.cred.FilePasswordDB`—or at least not permanently, because it won't write the players back to the file.

If the login raises an error, you should return to the initial `"Welcome"` state ❼, rather than to `"User"`, so the user can register instead if the user can't remember their username (or if you've deleted it for some reason).

Finally, you need to update your factory's protocol so it uses `RegisteringTelnetProtocol` instead of the old `AuthenticatingTelnetProtocol` ❽.

Awesome! Now you won't have to enter usernames and passwords for everyone who wants to check out your cool new game. In practice, this will mean you'll get more players, because it sets the bar to entry much

lower, and the players won't have to wait around for you to check your email. The next step, if you're interested, is to include a password-reset or -retrieval mechanism, so the players (if they've set their email address in-game) can be sent their password if they forget it.

Making your world permanent

You have a few more pressing concerns now: players can register and log in, but if you restart the server for some reason (say, to add a new feature), then they lose all their progress and have to reregister! You don't have to save *everything*, though—what you'll do is save only the players and their items and restart all the monsters from scratch. This is common practice in most MUDs, so the monsters, puzzles, and stories reset each night.

NOTE One of the other reasons to implement saving is that it breaks the player's suspension of disbelief if everything suddenly vanishes. You want the player to believe on some level that the world you're creating is real, and real worlds don't disappear in a puff of virtual smoke.

Listing 10.14 Loading players

```
import os
import pickle

class Game(object):
    ...
    def __init__(self):
        ...
        self.start_loc = cave1
        self.player_store = {}
        self.load_players()

    def load_players(self):
        if os.access('players.pickle', os.F_OK) != 1:
            return
        load_file = open('players.pickle', 'rb')
        self.player_store = pickle.load(load_file)
```

1 Create player store

2 Load players

3 Load player store

You don't need to store every player, because you're only interested in players' data—what they've called themselves, how they look and which

items they're carrying. You'll put that information into the store ❶ so you can call it out at will.

The next thing you'll do is figure out how you're going to call the code you'll use to load the player store ❷. I think you'll be alright if you create a method.

The method to load the player store ❸ turns out to be pretty simple. Check to see if the file exists—if it does, then open it and load the `player_store` from it using Pickle.

Easy! Of course, you're not done yet—that only loads the player store. Now you need to work out what goes in the store, and save it to a file.

Listing 10.15 Saving players

```
class Game(object):
    ...
    def save(self):
        for player in self.players:
            self.player_store[player.name] = \
                                    player.save()
        print "Saving:", self.player_store
        save_file = open('players.pickle', 'wb')
        pickle.dump(self.player_store, save_file)

class Player(object):
     def __init__(self, game, location):
        ...
        self.password = ""

    def save(self):
        return {
            'name': self.name,
            'description': self.description,
            'password': self.password,
            'items': [(item.name, item.description)
                      for item in self.inventory], }
```

❶ Save each player

❷ Save file

Add password to player

❸ Create player store

You add each player to the player store in typical object-oriented fashion—by calling `player.save` to find out what should be stored for each player ❶.

Once you've refreshed the store, you can go ahead and save it to disk ❷, ready for the next time you start the game.

All the `player.save` method needs to do is make a dictionary of all of the player's data and return it ❸.

Now your `game.save` method should be working, and you can load from it. The last step is to trigger `game.save` at appropriate points and make sure the players are loaded with all their data when they log in.

Listing 10.16 Updating the server

```
from item import Item

class Player(object):
    ...
    def load(self, config):
        self.name = config['name']
        self.password = config['password']
        self.description = config['description']
        for item in config['items']:
            self.inventory.append(
                Item(item[0], item[1],
                     self.location))
    ...
    def quit(self, player, noun):
        self.playing = False
        self.game.player_store[self.name] = self.save()

        # drop all our stuff(?)
        for item in self.inventory:
            self.location.here.append(item)
            item.location = self.location
        self.inventory = []

        return ["Thanks for playing!"]
...
class Game(object):
    def connectionMade(self):
        ...
        self.player.password = \
            checker.users[self.player.name]
```

❶ **Load player**

❷ **Update quit method**

```
        if self.player.name in game.player_store:
            self.player.load(
                game.player_store[self.player.name])
        game.players.append(self.player)
if __name__ == '__main__':
    ...
    def do_save():
        print "Saving game..."
        game.save()
        print "Updating portal passwords..."
        for player in game.player_store.values():
            for checker in portal_.checkers.values():
                checker.users[player['name']] = \
                                player['password']

    do_save()
    game_saver = task.LoopingCall(do_save)
    game_saver.start(60.0)
```

Load player on creation

3 Save game

4 Refresh portal's password list

5 Save every minute

Loading the player is much the same as saving it ❶, only the other way around. Rather than dump your state into a dictionary, you update the state from one.

Rather than have the players die whenever they quit, they'll now save themselves and exit nicely ❷. For this game you only have one sword and one coin to share among all the players, so you'll drop all your items; but that's not normal practice for an adventure game.

To save everything ❸, you'll set up another periodic function using Twisted.

The players can change their passwords in game, so it makes sense to refresh the server's password list along with saving the game ❹. You do this right after the call to `game.save()`, so you know `game.player_store` is as fresh as possible.

Note that there's a bug in this code: when a player changes their name, the old name isn't removed. You'll want to either update the

name-changing code in `Player` to delete the old name from both the portals and `player_store`, or else disable the name-changing code. Disallowing name changes is probably the best option, because it also discourages bad behavior.

Once your function is complete, you only call it when you start up, and every minute or so after that ❺. I've picked 60 seconds as a reasonable timeframe, but you might find that a longer or shorter span works better for you. In practice, it will be a tradeoff between the load on the server when the game is saved, and the risk of losing your players' stuff.

That should be it. Now you have a stable base for your future development, and you don't have to worry about players not being able to log in, or having to respond to everyone who wants to log in.

Where to from here?

Your MUD is working and feature complete, but you've only scratched the surface of what you could do. One way to find out what needs to be done is to invite some of your friends to play—make sure they know it's a work in progress—and ask them for suggestions and bug fixes. If you don't have any friends who are into MUDs, the following is a list of some ideas you could try:

- Make the orc respawn once you've killed it (in a different location), or add different monsters. They might have different attacks, take more or fewer hits to kill, and drop different sorts of treasure.
- Saving the cave layout as well as the players' info will help players identify it more strongly as an actual place. Also, most MUDs will let you log in as a "wizard" and extend the game while you're playing it, adding rooms or monsters.

- Different items, armor, and weapons can add an extra level of interest, as players explore or save up their gold for new ones.
- Let the players gain experience and levels, with higher-level characters being tougher and more powerful. Different character

classes and statistics (strength, intelligence, dexterity, and so on) can help players identify with the game and make it more enjoyable.

- A number of open source MUDs are available, in several languages; download them and see how they work. Most of the core components will be similar, so you'll know what to look for when you're trying to make sense of them.

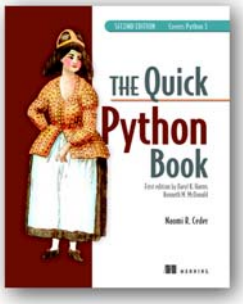
Summary

In this chapter, you learned how to add networking to a game and about the issues you need to deal with in networked environments. You started with a simple chat server and learned about Twisted's `Protocol` and `Server` classes, before creating a similar setup so you could play your game over Telnet. Because Twisted is asynchronous (does lots of things simultaneously), you also needed to learn how to use Twisted's `task.LoopingCall` for your game loop.

Once you'd done that, you opened your game for testing and discovered a few issues with the game play in the new environment. To fix these, you added some new features, such as angry lists, talking to other players, and commands to change player names and descriptions.

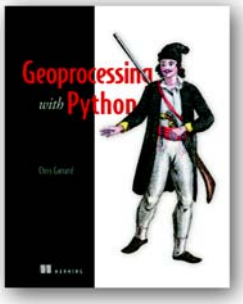
Finally, you set up a system so new players could log into your system without you having to add them to a list of users. You learned a bit more about the details of Twisted, particularly its Telnet implementation, but also about how it interfaces with `Protocol`s, `Server`s, and also `Deferred`s—one of Twisted's lower-level features.

RELATED MANNING TITLES



The Quick Python Book, Second Edition
Revised edition of The Quick Python Book by Daryl K. Harms and Kenneth M. McDonald
by Naomi R. Ceder

ISBN: 9781935182207
360 pages, $39.99
January 2010

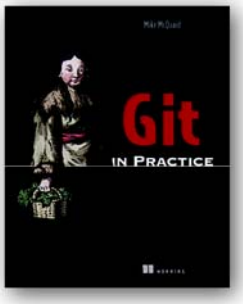


Geoprocessing with Python
by Chris Garrard

ISBN: 9781617292149
400 pages, $49.99
March 2016

OpenStack in Action
by V. K. Cody Bumgardner

ISBN: 9781617292163
375 pages, $54.99
October 2015



Git in Practice
by Mike McQuaid

ISBN: 9781617291975
272 pages, $39.99
September 2014

For ordering information go to www.manning.com