CHAPTER 14

Network Programming

In this chapter, I give you a sample of the various ways in which Python can help you write programs that use a network, such as the Internet, as an important component. Python is a very powerful tool for network programming. Many libraries for common network protocols and for various layers of abstractions on top of them are available, so you can concentrate on the logic of your program, rather than on shuffling bits across wires. Also, it's easy to write code for handling various protocol formats that may *not* have existing code, because Python's really good at tackling patterns in byte streams (you've already seen this in dealing with text files in various ways).

Because Python has such an abundance of network tools available for you to use, I can only give you a brief peek at its networking capabilities here. You can find some other examples elsewhere in this book. Chapter 15 includes a discussion of web-oriented network programming, and several of the projects in later chapters use networking modules to get the job done. If you want to know even *more* about network programming in Python, I can heartily recommend John Goerzen's *Foundations of Python Network Programming* (Apress, 2004), which deals with the subject very thoroughly.

In this chapter, I give you an overview of some of the networking modules available in the Python standard library. Then comes a discussion of the `SocketServer` class and its friends, followed by a brief look at the various ways in which you can handle several connections at once. Finally, I give you a look at the Twisted framework, a rich and mature framework for writing networked applications in Python.

Note If you have a strict firewall in place, it will probably warn you once you start running your own network programs and stop them from connecting to the network. You should either configure your firewall to let your Python do its work, or, if the firewall has an interactive interface (such as the Windows XP firewall), simply allow the connections when asked. Note, though, that any software connected to a network is a potential security risk, even if (or especially if) you wrote the software yourself.

A Handful of Networking Modules

You can find plenty of networking modules in the standard library, and many more elsewhere. In addition to those that clearly deal mainly with networking, several modules (such as those that deal with various forms of data encoding for network transport) may be seen as network related. I've been fairly restrictive in my selection of modules here.

The socket Module

A basic component in network programming is the *socket*. A socket is basically an "information channel" with a program on both ends. The programs may be on different computers (connected through a network) and may send information to each other through the socket. Most network programming in Python hides the basic workings of the `socket` module and doesn't interact with the sockets directly.

Sockets come in two varieties: server sockets and client sockets. After you create a server socket, you tell it to wait for connections. It will then listen at a certain network address (a combination of an IP address and a port number) until a client socket connects. The two can then communicate.

Dealing with client sockets is usually quite a bit easier than dealing with the server side, because the server must be ready to deal with clients whenever they connect, and it must deal with multiple connections, while the client simply connects, does its thing, and disconnects. Later in this chapter, I discuss server programming through the `SocketServer` class family and the Twisted framework.

A socket is an instance of the `socket` class from the `socket` module. It is instantiated with up to three parameters: an address family (defaulting to `socket.AF_INET`), whether it's a stream (`socket.SOCK_STREAM`, the default) or a datagram (`socket.SOCK_DGRAM`) socket, and a protocol (defaulting to 0, which should be okay). For a plain-vanilla socket, you don't really need to supply any arguments.

A server socket uses its `bind` method followed by a call to `listen` to listen to a given address. A client socket can then connect to the server by using its `connect` method with the same address as used in `bind`. (On the server side, you can, for example, get the name of the current machine using the function `socket.gethostname`.) In this case, an address is just a tuple of the form `(host, port)`, where `host` is a host name (such as `www.example.com`) and `port` is a port number (an integer). The `listen` method takes a single argument, which is the length of its backlog (the number of connections allowed to queue up, waiting for acceptance, before connections start being disallowed).

Once a server socket is listening, it can start accepting clients. This is done using the `accept` method. This method will block (wait) until a client connects, and then it will return a tuple of the form `(client, address)`, where `client` is a client socket and `address` is an address, as explained earlier. The server can deal with the client as it sees fit, and then start waiting for new connections, with another call to `accept`. This is usually done in an infinite loop.

■ **Note** The form of server programming discussed here is called *blocking* or *synchronous* network programming. In the section "Multiple Connections" later in this chapter, you'll see examples of nonblocking or asynchronous network programming, as well as using threads to be able to deal with several clients at once.

For transmitting data, sockets have two methods: `send` and `recv` (for "receive"). You can call `send` with a string argument to send data, and `recv` with a desired (maximum) number of bytes to receive data. If you're not sure which number to use, 1024 is as good a choice as any.

Listings 14-1 and 14-2 show an example client/server pair that is about as simple as it gets. If you run them on the same machine (starting the server first), the server should print out a message about getting a connection, and the client should then print out a message it has received from the server. You can run several clients while the server is still running. By replacing the call to `gethostname` in the client with the actual host name of the machine where the server is running, you can have the two programs connect across a network from one machine to another.

Note The port numbers you use are normally restricted. In a Linux or UNIX system, you need administrator privileges to use a port below 1024. These low-numbered ports are used for standard services, such as port 80 for your web server (if you have one). Also, if you stop a server with Ctrl+C, for example, you might need to wait for a bit before using the same port number again (you may get an "Address already in use" error).

Listing 14-1. *A Minimal Server*

```
import socket

s = socket.socket()

host = socket.gethostname()
port = 1234
s.bind((host, port))

s.listen(5)
while True:
    c, addr = s.accept()
    print 'Got connection from', addr
    c.send('Thank you for connecting')
    c.close()
```

Listing 14-2. *A Minimal Client*

```
import socket

s = socket.socket()

host = socket.gethostname()
port = 1234

s.connect((host, port))
print s.recv(1024)
```

You can find more information about the socket module in the Python Library Reference (http://python.org/doc/lib/module-socket.html) and in Gordon McMillan's Socket Programming HOWTO (http://docs.python.org/dev/howto/sockets.html).

The urllib and urllib2 Modules

Of the networking libraries available, the ones that probably give you the most bang for the buck are urllib and urllib2. They enable you to access files across a network, just as if they were located on your computer. Through a simple function call, virtually anything you can refer to with a Uniform Resource Locator (URL) can be used as input to your program. Just imagine the possibilities you get if you combine this with the re module: you can download web pages, extract information, and create automatic reports of your findings.

The two modules do more or less the same job, with urllib2 being a bit more "fancy." For simple downloads, urllib is quite all right. If you need HTTP authentication or cookies, or you want to write extensions to handle your own protocols, then urllib2 might be the right choice for you.

Opening Remote Files

You can open remote files almost exactly as you do local files; the difference is that you can use only read mode, and instead of open (or file), you use urlopen from the urllib module:

```
>>> from urllib import urlopen
>>> webpage = urlopen('http://www.python.org')
```

If you are online, the variable webpage should now contain a file-like object linked to the Python web page at http://www.python.org.

■Note If you want to experiment with urllib but aren't currently online, you can access local files with URLs that start with file:, such as file:c:\text\somefile.txt. (Remember to escape your backslashes.)

The file-like object that is returned from urlopen supports (among others) the close, read, readline, and readlines methods, as well as iteration.

Let's say you want to extract the (relative) URL of the "About" link on the Python page you just opened. You could do that with regular expressions (for more information about regular expressions, see the section about the re module in Chapter 10):

```
>>> import re
>>> text = webpage.read()
>>> m = re.search('<a href="([^"]+)" .*?>about</a>', text, re.IGNORECASE)
>>> m.group(1)
'/about/'
```

■ **Note** You may need to modify the regular expression if the web page has changed since the time of writing, of course.

Retrieving Remote Files

The `urlopen` function gives you a file-like object you can read from. If you would rather have `urllib` take care of downloading the file for you, storing a copy in a local file, you can use `urlretrieve` instead. Rather than returning a file-like object, it returns a tuple `(filename, headers)`, where `filename` is the name of the local file (this name is created automatically by `urllib`), and `headers` contains some information about the remote file. (I'll ignore `headers` here; look up `urlretrieve` in the standard library documentation of `urllib` if you want to know more about it.) If you want to specify a file name for the downloaded copy, you can supply that as a second parameter:

```
urlretrieve('http://www.python.org', 'C:\\python_webpage.html')
```

This retrieves the Python home page and stores it in the file `C:\python_webpage.html`. If you don't specify a file name, the file is put in some temporary location, available for you to open (with the `open` function), but when you're finished with it, you may want to have it removed so that it doesn't take up space on your hard drive. To clean up such temporary files, you can call the function `urlcleanup` without any arguments, and it takes care of things for you.

SOME UTILITIES

In addition to reading and downloading files through URLs, `urllib` also offers some functions for manipulating the URLs themselves. (The following assumes some knowledge of URLs and CGI.) The following functions are available:

- `quote(string[, safe])`: Returns a string in which all special characters (characters that have special significance in URLs) have been replaced by URL-friendly versions (such as `%7E` instead of ~). This can be useful if you have a string that might contain such special characters and you want to use it as a URL. The `safe` string includes characters that should not be coded like this. The default is `'/'`.
- `quote_plus(string[, safe])`: Works like `quote`, but also replaces spaces with plus signs.
- `unquote(string)`: The reverse of `quote`.
- `unquote_plus(string)`: The reverse of `quote_plus`.
- `urlencode(query[, doseq])`: Converts a mapping (such as a dictionary) or a sequence of two-element tuples—of the form `(key, value)`—into a "URL-encoded" string, which can be used in CGI queries. (Check the Python documentation for more information.)

Other Modules

As mentioned, beyond the modules explicitly discussed in this chapter, there are hordes of network-related modules in the Python library and elsewhere. Table 14-1 lists some network-related modules from the Python standard library. As noted in the table, some of these modules are discussed elsewhere in the book.

Table 14-1. *Some Network-Related Modules in the Standard Library*

Module	Description
`asynchat`	Additional functionality for `asyncore` (see Chapter 24)
`asyncore`	Asynchronous socket handler (see Chapter 24)
`cgi`	Basic CGI support (see Chapter 15)
`Cookie`	Cookie object manipulation, mainly for servers
`cookielib`	Client-side cookie support
`email`	Support for e-mail messages (including MIME)
`ftplib`	FTP client module
`gopherlib`	Gopher client module
`httplib`	HTTP client module
`imaplib`	IMAP4 client module
`mailbox`	Reads several mailbox formats
`mailcap`	Access to MIME configuration through mailcap files
`mhlib`	Access to MH mailboxes
`nntplib`	NNTP client module (see Chapter 23)
`poplib`	POP client module
`robotparser`	Support for parsing web server robot files
`SimpleXMLRPCServer`	A simple XML-RPC server (see Chapter 27)
`smtpd`	SMTP server module
`smtplib`	SMTP client module
`telnetlib`	Telnet client module
`urlparse`	Support for interpreting URLs
`xmlrpclib`	Client support for XML-RPC (see Chapter 27)

SocketServer and Friends

As you saw in the section about the `socket` module earlier, writing a simple socket server isn't really hard. If you want to go beyond the basics, however, getting some help can be nice. The `SocketServer` module is the basis for a framework of several servers in the standard library, including BaseHTTPServer, SimpleHTTPServer, CGIHTTPServer, SimpleXMLRPCServer, and DocXMLRPCServer, all of which add various specific functionality to the basic server.

SocketServer contains four basic classes: TCPServer, for TCP socket streams; UDPServer, for UDP datagram sockets; and the more obscure UnixStreamServer and UnixDatagramServer. You probably won't need the last three.

To write a server using the SocketServer framework, you put most of your code in a request handler. Each time the server gets a request (a connection from a client), a request handler is instantiated, and various handler methods are called on it to deal with the request. Exactly which methods are called depends on the specific server and handler class used, and you can subclass them to make the server call a custom set of handlers. The basic BaseRequestHandler class places all of the action in a single method on the handler, called handle, which is called by the server. This method then has access to the client socket in the attribute self.request. If you're working with a stream (which you probably are, if you use TCPServer), you can use the class StreamRequestHandler, which sets up two other attributes, self.rfile (for reading) and self.wfile (for writing). You can then use these file-like objects to communicate with the client.

The various other classes in the SocketServer framework implement basic support for HTTP servers, including running CGI scripts, as well as support for XML-RPC (discussed in Chapter 27).

Listing 14-3 gives you a SocketServer version of the minimal server from Listing 14-1. It can be used with the client in Listing 14-2. Note that the StreamRequestHandler takes care of closing the connection when it has been handled. Also note that giving '' as the host name means that you're referring to the machine where the server is running.

Listing 14-3. *A SocketServer-Based Minimal Server*

```
from SocketServer import TCPServer, StreamRequestHandler

class Handler(StreamRequestHandler):

    def handle(self):
        addr = self.request.getpeername()
        print 'Got connection from', addr
        self.wfile.write('Thank you for connecting')

server = TCPServer(('', 1234), Handler)
server.serve_forever()
```

You can find more information about the SocketServer framework in the Python Library Reference (http://python.org/doc/lib/module-SocketServer.html) and in John Goerzen's *Foundations of Python Network Programming* (Apress, 2004).

Multiple Connections

The server solutions discussed so far have been *synchronous*: only one client can connect and get its request handled at a time. If one request takes a bit of time, such as, for example, a complete chat session, it's important that more than one connection can be dealt with simultaneously.

You can deal with multiple connections in three main ways: forking, threading, and asynchronous I/O. Forking and threading can be dealt with very simply, by using mix-in classes with any of the `SocketServer` servers (see Listings 14-4 and 14-5). Even if you want to implement them yourself, these methods are quite easy to work with. They do have their drawbacks, however. Forking takes up resources, and may not scale well if you have many clients (although, for a reasonable number of clients, on modern UNIX or Linux systems, forking is quite efficient, and can be even more so if you have a multi-CPU system). Threading can lead to synchronization problems. I won't go into these problems in any detail here (nor will I discuss multithreading in depth), but I'll show you how to use the techniques in the following sections.

FORKS? THREADS? WHAT'S ALL THIS, THEN?

Just in case you don't know about forking or threads, here is a little clarification. *Forking* is a UNIX term. When you fork a process (a running program), you basically duplicate it, and both resulting processes keep running from the current point of execution, each with its own copy of the memory (variables and such). One process (the original one) will be the *parent* process, while the other (the copy) will be the *child*. If you're a science fiction fan, you might think of parallel universes; the forking operation creates a fork in the timeline, and you end up with two universes (the two processes) existing independently. Luckily, the processes are able to determine whether they are the original or the child (by looking at the return value of the `fork` function), so they can act differently. (If they couldn't, what would be the point, really? Both processes would do the same job, and you would just bog down your computer.)

In a forking server, a child is forked off for every client connection. The parent process keeps listening for new connections, while the child deals with the client. When the client is satisfied, the child process simply exits. Because the forked processes run in parallel, the clients don't need to wait for each other.

Because forking can be a bit resource intensive (each forked process needs its own memory), an alternative exists: threading. *Threads* are lightweight processes, or subprocesses, all of them existing within the same (real) process, sharing the same memory. This reduction in resource consumption comes with a downside, though. Because threads share memory, you must make sure they don't interfere with the variables for each other, or try to modify the same things at the same time, creating a mess. These issues fall under the heading of "synchronization." With modern operating systems (except Microsoft Windows, which doesn't support forking), forking is actually quite fast, and modern hardware can deal with the resource consumption much better than before. If you don't want to bother with synchronization issues, then forking may be a good alternative.

The best thing may, however, be to avoid this sort of parallelism altogether. In this chapter, you find other solutions, based on the `select` function. Another way to avoid threads and forks is to switch to Stackless Python (`http://stackless.com`), a version of Python designed to be able to switch between different contexts quickly and painlessly. It supports a form of thread-like parallelism called *microthreads*, which scale much better than real threads. For example, Stackless Python microthreads have been used in EVE Online (`http://www.eve-online.com`) to serve thousands of users.

Asynchronous I/O is a bit more difficult to implement at a low level. The basic mechanism is the `select` function of the `select` module (described in the section "Asynchronous I/O with select and poll"), which is quite hard to deal with. Luckily, frameworks exist that work with asynchronous I/O on a higher level, giving you a simple, abstract interface to a very powerful and scalable mechanism. A basic framework of this kind, which is included in the standard library, consists of the asyncore and asynchat modules, discussed in Chapter 24. Twisted (which is discussed last in this chapter) is a very powerful asynchronous network programming framework.

Forking and Threading with SocketServer

Creating a forking or threading server with the SocketServer framework is so simple it hardly needs any explanation. Listings 14-4 and 14-5 show you how to make the server from Listing 14-3 forking and threading, respectively. The forking or threading behavior is useful only if the handle method takes a long time to finish. Note that forking doesn't work in Windows.

Listing 14-4. *A Forking Server*

```
from SocketServer import TCPServer, ForkingMixIn, StreamRequestHandler

class Server(ForkingMixIn, TCPServer): pass

class Handler(StreamRequestHandler):

    def handle(self):
        addr = self.request.getpeername()
        print 'Got connection from', addr
        self.wfile.write('Thank you for connecting')

server = Server(('', 1234), Handler)
server.serve_forever()
```

Listing 14-5. *A Threading Server*

```
from SocketServer import TCPServer, ThreadingMixIn, StreamRequestHandler

class Server(ThreadingMixIn, TCPServer): pass

class Handler(StreamRequestHandler):

    def handle(self):
        addr = self.request.getpeername()
        print 'Got connection from', addr
        self.wfile.write('Thank you for connecting')

server = Server(('', 1234), Handler)
server.serve_forever()
```

Asynchronous I/O with select and poll

When a server communicates with a client, the data it receives from the client may come in fits and spurts. If you're using forking and threading, that's not a problem. While one parallel waits for data, other parallels may continue dealing with their own clients. Another way to go, however, is to deal only with the clients that actually have something to say at a given moment. You don't even have to hear them out—you just hear (or, rather, *read*) a little, and then put it back in line with the others.

This is the approach taken by the frameworks asyncore/asynchat (see Chapter 24) and Twisted (see the following section). The basis for this kind of functionality is the select function, or, where available, the poll function, both from the select module. Of the two, poll is more scalable, but it is available only in UNIX systems (that is, not in Windows).

The select function takes three sequences as its mandatory arguments, with an optional timeout in seconds as its fourth argument. The sequences are file descriptor integers (or objects with a fileno method that return such an integer). These are the connections that we're waiting for. The three sequences are for input, output, and exceptional conditions (errors and the like). If no timeout is given, select blocks (that is, waits) until one of the file descriptors is ready for action. If a timeout is given, select blocks for at most that many seconds, with zero giving a straight poll (that is, no blocking). select returns three sequences (a triple—that is, a tuple of length three), each representing an active subset of the corresponding parameter. For example, the first sequence returned will be a sequence of input file descriptors where there is something to read.

The sequences can, for example, contain file objects (not in Windows) or sockets. Listing 14-6 shows a server using select to serve several connections. (Note that the server socket itself is supplied to select, so that it can signal when there are new connections ready to be accepted.) The server is a simple logger that prints out (locally) all data received from its clients. You can test it by connecting to it using telnet (or by writing a simple socket-based client that feeds it some data). Try connecting with multiple telnet connections to see that it can serve more than one client at once (although its log will then be a mixture of the input from the two).

Listing 14-6. *A Simple Server Using select*

```
import socket, select

s = socket.socket()

host = socket.gethostname()
port = 1234
s.bind((host, port))

s.listen(5)
inputs = [s]
while True:
    rs, ws, es = select.select(inputs, [], [])
    for r in rs:
        if r is s:
            c, addr = s.accept()
            print 'Got connection from', addr
            inputs.append(c)
        else:
            try:
                data = r.recv(1024)
                disconnected = not data
            except socket.error:
                disconnected = True

            if disconnected:
                print r.getpeername(), 'disconnected'
                inputs.remove(r)
            else:
                print data
```

The `poll` method is easier to use than `select`. When you call `poll`, you get a poll object. You can then register file descriptors (or objects with a `fileno` method) with the poll object, using its `register` method. You can later remove such objects again, using the `unregister` method. Once you've registered some objects (for example, sockets), you can call the `poll` method (with an optional timeout argument) and get a list (possibly empty) of pairs of the form `(fd, event)`, where `fd` is the file descriptor and `event` tells you what happened. It's a bitmask, meaning that it's an integer where the individual bits correspond to various events. The various events are constants of the `select` module, and are explained in Table 14-2. To check whether a given bit is set (that is, if a given event occurred), you use the bitwise and operator (`&`), like this:

```
if event & select.POLLIN: ...
```

Table 14-2. *Polling Event Constants in the select Module*

Event Name	Description
`POLLIN`	There is data to read available from the file descriptor.
`POLLPRI`	There is urgent data to read from the file descriptor.
`POLLOUT`	The file descriptor is ready for data, and will not block if written to.
`POLLERR`	Some error condition is associated with the file descriptor.
`POLLHUP`	Hung up. The connection has been lost.
`POLLNVAL`	Invalid request. The connection is not open.

The program in Listing 14-7 is a rewrite of the server from Listing 14-6, now using `poll` instead of `select`. Note that I've added a map (`fdmap`) from file descriptors (`ints`) to socket objects.

Listing 14-7. *A Simple Server Using poll*

```
import socket, select

s = socket.socket()
```

```
host = socket.gethostname()
port = 1234
s.bind((host, port))

fdmap = {s.fileno(): s}


s.listen(5)
p = select.poll()
p.register(s)
while True:
    events = p.poll()
    for fd, event in events:
        if fd in fdmap:
            c, addr = s.accept()
            print 'Got connection from', addr
            p.register(c)
            fdmap[c.fileno()] = c
        elif event & select.POLLIN:
            data = fdmap[fd].recv(1024)
            if not data: # No data -- connection closed
                print fdmap[fd].getpeername(), 'disconnected'
                p.unregister(fd)
                del fdmap[fd]
            else:
                print data
```

You can find more information about `select` and `poll` in the Python Library Reference (`http://python.org/doc/lib/module-select.html`). Also, reading the source code of the standard library modules `asyncore` and `asynchat` (found in the `asyncore.py` and `asynchat.py` files in your Python installation) can be enlightening.

Twisted

Twisted, from Twisted Matrix Laboratories (`http://twistedmatrix.com`), is an *event-driven* networking framework for Python, originally developed for network games but now used by all kinds of network software. In Twisted, you implement event handlers, much like you would in a GUI toolkit (see Chapter 12). In fact, Twisted works quite nicely together with several common GUI toolkits (Tk, GTK, Qt, and wxWidgets). In this section, I'll cover some of the basic concepts and show you how to do some relatively simple network programming using Twisted. Once you grasp the basic concepts, you can check out the Twisted documentation (available on the Twisted web site, along with quite a bit of other information) to do some more serious network programming. Twisted is a *very* rich framework and supports, among other things, web servers and clients, SSH2, SMTP, POP3, IMAP4, AIM, ICQ, IRC, MSN, Jabber, NNTP, DNS, and more!

Downloading and Installing Twisted

Installing Twisted is quite easy. First, go to the Twisted Matrix web site (`http://twistedmatrix.com`) and, from there, follow one of the download links. If you're using Windows, download the Windows installer for your version of Python. If you're using some other system, download a source archive. (If you're using a package manager such as Portage, RPM, APT, Fink, or MacPorts, you can probably get it to download and install Twisted directly.) The Windows installer is a self-explanatory step-by-step wizard. It may take some time compiling and unpacking things, but all you have to do is wait. To install the source archive, you first unpack it (using `tar` and then either `gunzip` or `bunzip2`, depending on which type of archive you downloaded), and then run the Distutils script:

```
python setup.py install
```

You should then be able to use Twisted.

Writing a Twisted Server

The basic socket servers written earlier in this chapter are very explicit. Some of them have an explicit event loop, looking for new connections and new data. `SocketServer`-based servers have an implicit loop where the server looks for connections and creates a handler for each connection, but the handlers still must be explicit about trying to read data. Twisted (like the `asyncore`/`asynchat` framework, discussed in Chapter 24) uses an even more event-based approach. To write a basic server, you implement event handlers that deal with situations such as a new client connecting, new data arriving, and a client disconnecting (as well as many other events). Specialized classes can build more refined events from the basic ones, such as wrapping "data arrived" events, collecting the data until a newline is found, and then dispatching a "line of data arrived" event.

■ **Note** One thing I have not dealt with in this section, but which is somewhat characteristic of Twisted, is the concept of *deferreds* and deferred execution. See the Twisted documentation for more information (see, for example, the tutorial called "Deferreds are beautiful," available from the HOWTO page of the Twisted documentation).

Your event handlers are defined in a protocol. You also need a factory that can construct such protocol objects when a new connection arrives. If you just want to create instances of a custom protocol class, you can use the factory that comes with Twisted, the `Factory` class in the module `twisted.internet.protocol`. When you write your protocol, use the `Protocol` from the same module as your superclass. When you get a connection, the event handler `connectionMade` is called. When you lose a connection, `connectionLost` is called. Data is received from the client through the handler `dataReceived`. Of course, you can't use the event-handling strategy to send data back to the client—for that you use the object `self.transport`, which has a `write` method. It also has a `client` attribute, which contains the client address (host name and port).

Listing 14-8 contains a Twisted version of the server from Listings 14-6 and 14-7. I hope you agree that the Twisted version is quite a bit simpler and more readable. There is a little bit of setup involved; you need to instantiate `Factory` and set its `protocol` attribute so it knows which protocol to use when communicating with clients (that is, your custom protocol). Then you start listening at a given port with that factory standing by to handle connections by instantiating protocol objects. You do this using the `listenTCP` function from the `reactor` module. Finally, you start the server by calling the `run` function from the same module.

Listing 14-8. *A Simple Server Using Twisted*

```
from twisted.internet import reactor
from twisted.internet.protocol import Protocol, Factory

class SimpleLogger(Protocol):

    def connectionMade(self):
        print 'Got connection from', self.transport.client

    def connectionLost(self, reason):
        print self.transport.client, 'disconnected'

    def dataReceived(self, data):
        print data

factory = Factory()
factory.protocol = SimpleLogger

reactor.listenTCP(1234, factory)
reactor.run()
```

If you connected to this server using telnet to test it, you may have gotten a single character on each line of output, depending on buffering and the like. You could simply use `sys.sout.write` instead of `print`, but in many cases, you might like to get a single line at a time, rather than just arbitrary data. Writing a custom protocol that handles this for you would be quite easy, but there is, in fact, such a class available already. The module `twisted.protocols.basic` contains a couple of useful predefined protocols, among them `LineReceiver`. It implements `dataReceived` and calls the event handler `lineReceived` whenever a full line is received.

■ **Tip** If you need to do something when you receive data in *addition* to using `lineReceived`, which depends on the `LineReceiver` implementation of `dataReceived`, you can use the new event handler defined by `LineReceiver` called `rawDataReceived`.

Switching the protocol requires only a minimum of work. Listing 14-9 shows the result. If you look at the resulting output when running this server, you'll see that the newlines are stripped; in other words, using `print` won't give you double newlines anymore.

Listing 14-9. *An Improved Logging Server, Using the LineReceiver Protocol*

```
from twisted.internet import reactor
from twisted.internet.protocol import Factory
from twisted.protocols.basic import LineReceiver


class SimpleLogger(LineReceiver):

    def connectionMade(self):
        print 'Got connection from', self.transport.client

    def connectionLost(self, reason):
        print self.transport.client, 'disconnected'

    def lineReceived(self, line):
        print line

factory = Factory()
factory.protocol = SimpleLogger

reactor.listenTCP(1234, factory)
reactor.run()
```

As noted earlier, there is a lot more to the Twisted framework than what I've shown you here. If you're interested in learning more, you should check out the online documentation, available at the Twisted web site (`http://twistedmatrix.com`).

A Quick Summary

This chapter has given you a taste of several approaches to network programming in Python. Which approach you choose will depend on your specific needs and preferences. Once you've chosen, you will, most likely, need to learn more about the specific method. Here are some of the topics this chapter touched upon:

Sockets and the `socket` module: Sockets are information channels that let programs (processes) communicate, possibly across a network. The `socket` module gives you low-level access to both client and server sockets. Server sockets listen at a given address for client connections, while clients simply connect directly.

`urllib` and `urllib2`: These modules let you read and download data from various servers, given a URL to the data source. The `urllib` module is a simpler implementation, while `urllib2` is very extensible and quite powerful. Both work through straightforward functions such as `urlopen`.

The `SocketServer` framework: This is a network of synchronous server base classes, found in the standard library, which lets you write servers quite easily. There is even support for simple web (HTTP) servers with CGI. If you want to handle several connections simultaneously, you need to use a *forking* or *threading* mix-in class.

`select` **and** `poll`: These two functions let you consider a set of connections and find out which ones are ready for reading and writing. This means that you can serve several connections piecemeal, in a round-robin fashion. This gives the illusion of handling several connections at the same time, and, although superficially a bit more complicated to code, is a much more scalable and efficient solution than threading or forking.

Twisted: This framework, from Twisted Matrix Laboratories, is very rich and complex, with support for most major network protocols. Even though it is large, and some of the idioms used may seem a bit foreign, basic usage is very simple and intuitive. The Twisted framework is also asynchronous, so it's very efficient and scalable. If you have Twisted available, it may very well be the best choice for many custom network applications.

New Functions in This Chapter

Function	Description
`urllib.urlopen(url[, data[, proxies]])`	Opens a file-like object from a URL
`urllib.urlretrieve(url[, fname[, hook[, data]]])`	Downloads a file from a URL
`urllib.quote(string[, safe])`	Quotes special URL characters
`urllib.quote_plus(string[, safe])`	The same as `quote`, but quotes spaces as +
`urllib.unquote(string)`	The reverse of `quote`
`urllib.unquote_plus(string)`	The reverse of `quote_plus`
`urllib.urlencode(query[, doseq])`	Encodes mapping for use in CGI queries
`select.select(iseq, oseq, eseq[, timeout])`	Finds sockets ready for reading/writing
`select.poll()`	Creates a poll object, for polling sockets
`reactor.listenTCP(port, factory)`	Twisted function; listens for connections
`reactor.run()`	Twisted function; main server loop

What Now?

You thought we were finished with network stuff now, huh? Not a chance. The next chapter deals with a quite specialized and much publicized entity in the world of networking: the Web.