



**Erle Robotics:
Python networking
programming**

Table of Contents

1. [Introduction](#)
2. [Introduction to Client/Server Networking](#)
 - i. [Virtualenv](#)
 - ii. [Installing virtualenv in Erle](#)
 - iii. [Create a virtual environment to test packages](#)
3. [Introduction to socket](#)
 - i. [What is socket?](#)
 - ii. [Creating a Socket](#)
 - iii. [Using sockets](#)
 - iv. [Disconnecting](#)
 - v. [Non - blocking sockets](#)
4. [UDP and TCP](#)
 - i. [Addresses and port numbers](#)
 - ii. [UDP](#)
 - i. [How UDP works?](#)
 - ii. [When to use UPD](#)
 - iii. [Socket \(UDP\)](#)
 - iv. [Unreliability, Backoff, Blocking, Timeouts](#)
 - v. [Connecting UDP Sockets](#)
 - vi. [Binding to Interfaces \(UDP\)](#)
 - vii. [UDP Fragmentation](#)
 - viii. [Socket Options](#)
 - iii. [TCP](#)
 - i. [How TCP works?](#)
 - ii. [When to use TCP](#)
 - iii. [What TCP Sockets Mean](#)
 - iv. [A Simple TCP Client and Server](#)
 - v. [Binding to Interfaces\(TCP\)](#)
 - vi. [Deadlock](#)
 - vii. [Closed Connections, Half-Open Connections](#)
 - viii. [Using TCP Streams like Files](#)
5. [Socket names and DNS](#)
 - i. [Socket names](#)
 - ii. [Five socket cordinates](#)
 - iii. [IPv6](#)
 - iv. [The getaddrinfo\(\) function](#)
 - i. [Asking getaddrinfo\(\) Where to Bind](#)
 - ii. [Asking getaddrinfo\(\) About Services](#)
 - iii. [Asking getaddrinfo\(\) for Pretty Hostnames](#)
 - iv. [Other getaddrinfo\(\) Flags](#)
 - v. [getaddrinfo\(\) in your own code](#)
 - v. [A Sketch of How DNS Works](#)
 - vi. [Using DNS](#)
6. [Network Data and Network Errors](#)
 - i. [Text and Encodings](#)
 - ii. [Network Byte Order](#)
 - iii. [Framing and Quoting](#)
 - iv. [Pickles and Self-Delimiting Formats](#)
 - v. [XML, JSON, Etc.](#)
 - vi. [Compression](#)
 - vii. [Network Exceptions](#)
 - viii. [Handling Exceptions](#)

7. TLS and SSL
 - i. Cleartext on the Network
 - ii. TLS Encrypts Your Conversations
 - iii. Supporting TLS in Python
 - iv. The Standard SSL Module
8. Server Architecture
 - i. Daemons and Logging
 - ii. Introductory example
 - iii. Elementary client
 - iv. Event-Driven Servers
 - v. The Semantics of Non-blocking
 - vi. Twisted Python
 - vii. Threading and Multi-processing
 - viii. Threading and Multi-processing Frameworks
9. Caches, Message Queues, and Map-Reduce
 - i. Using Memcached
 - ii. Memcached and Sharding
 - iii. Message Queues
 - iv. Using Message Queues from Python
 - v. Map-Reduce
10. HTTP
 - i. URL Anatomy
 - ii. Relative URLs
 - iii. Instrumenting urllib2
 - iv. The GET Method and The Host Header
 - v. Payloads and Persistent Connections
 - vi. POST And Forms
 - vii. REST And More HTTP Methods
 - viii. Identifying User Agents and Web Servers
 - ix. Content Type Negotiation
 - x. Compression
 - xi. HTTP Caching
 - xii. The HEAD Method
 - xiii. HTTPS Encryption
 - xiv. HTTP Authentication
 - xv. Cookies
 - xvi. HTTP Session Hijacking
 - xvii. Cross-Site Scripting Attacks
11. Screen Scraping
 - i. Fetching Web Pages
 - ii. Downloading Pages Through Form Submission
 - iii. The Structure of Web Pages
 - iv. Three Axes
 - v. Diving into an HTML Document
 - vi. Selectors
12. Web Applications
 - i. Web Servers and Python
 - ii. Choosing a Web Server
 - iii. WSGI
 - iv. WSGI Middleware
 - v. Python Web Frameworks
 - vi. URL Dispatch Techniques
 - vii. Templates
 - viii. Pure-Python Web Servers
 - ix. Common Gateway Interface (CGI)
 - x. mod_python

13. E-mail Composition and Decoding
 - i. E-mail Messages
 - ii. Composing Traditional Messages
 - iii. Parsing Traditional Messages
 - iv. Parsing Dates
 - v. Understanding MIME
 - vi. Composing MIME Attachments
 - vii. MIME Alternative Parts
 - viii. Composing Non-English Headers
 - ix. Composing Nested Multiparts
 - x. Parsing MIME Messages
 - xi. Decoding Headers
14. Simple Mail Transport Protocol (SMTP)
 - i. E-mail Clients, Webmail Services
 - ii. How SMTP Is Used
 - iii. Sending E-Mail
 - iv. Introducing the SMTP Library
 - v. Error Handling and Conversation Debugging
 - vi. Getting Information from EHLO
 - vii. Using Secure Sockets Layer and Transport Layer Security
 - viii. Authenticated SMTP
15. Post Office Protocol (POP)
 - i. Connecting and Authenticating
 - ii. Obtaining Mailbox Information
 - iii. Downloading and Deleting Messages
16. Internet Message Access Protocol (IMAP)
 - i. Understanding IMAP in Python
 - ii. IMAPClient
 - iii. Message Numbers vs. UIDs
 - iv. Summary Information
 - v. Downloading an Entire Mailbox
 - vi. Downloading Messages Individually
 - vii. Flagging and Deleting Messages
 - viii. Searching and Manipulating Messages
17. Telnet and SSH
 - i. Command-Line Automation
 - ii. Command-Line Expansion and Quoting
 - iii. Unix Has No Special Characters
 - iv. Quoting Characters for Protection
 - v. Things Are Different in a Terminal
 - vi. Terminals Do Buffering
 - vii. Telnet
 - viii. SSH: The Secure Shell
 - ix. SSH Host Keys
 - x. SSH Authentication
 - xi. Shell Sessions and Individual Commands
 - xii. SFTP: File Transfer Over SSH
18. File Transfer Protocol (FTP)
 - i. What to Use Instead of FTP
 - ii. Communication Channels
 - iii. Using FTP in Python
 - iv. ASCII and Binary Files
 - v. Advanced Binary Downloading
 - vi. Uploading Data
 - vii. Advanced Binary Uploading
 - viii. Handling Errors

- ix. [Detecting Directories and Recursive Download](#)
- x. [Creating Directories, Deleting Things](#)
- 19. [Remote Procedure Call \(RPC\)](#)
 - i. [Features of RPC](#)
 - ii. [XML-RPC](#)
 - iii. [JSON-RPC](#)
 - iv. [Self-documenting Data](#)
 - v. [Talking About Objects: Pyro and RPyC](#)
 - vi. [An RPyC Example](#)
 - vii. [RPC, Web Frameworks, Message Queues](#)

Erle Robotics: Python Networking Programming

book passing

Book

This book teaches the reader about **Python networking** in Linux, using [Erle Robotics autopilots](#). **Erle Robotics** creates **small-size Linux computers for making drones**.

With Python networking we refer to how using this programming language to control the incoming/outcoming connections, to use different protocols such as IP.



About

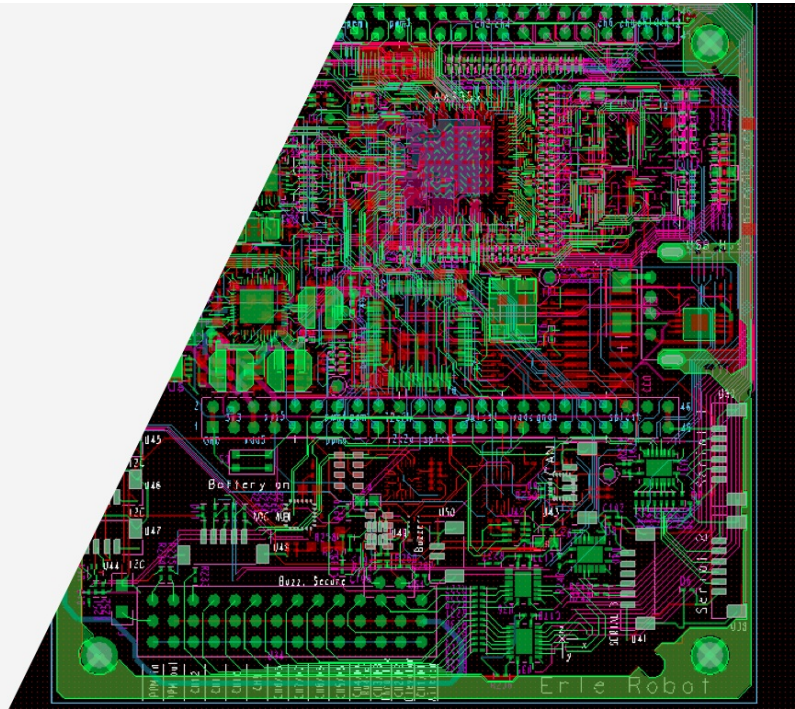
For years we've been working in the robotics field, particularly with drones. We have passed through different Universities and research centers and in all these places we actually found that most of the **drones are black boxes** (check out our [60s pitch](#)). Not meant to be used for learning, research. The software they use is in most of the cases unknown, closed source or **not documented**. Given these conditions, how are we going to educate the next generations on this technologies? How do you get started programming drones if you don't have \$1000+ budget? Which platform allows me to get started with drones without risking a hand?

We are coming up with an answer to all these questions, our technology: **Erle**.

Erle board

A palm size Linux-based computer for making drones

We have designed a small computer with about 40 sensors, plenty of I/O and processing power for real-time analysis. Erle is the enabling technology for the next generation of aerial and terrestrial robots.



Inspired by the BeagleBone development board, we have designed a small computer with about 36+ sensors, plenty of I/O and processing power for real-time analysis. Erle is the enabling technology for the next generation of aerial and terrestrial robots that will be used in cities solving tasks such as surveillance, enviromental monitoring or even providing aid at catastrophes.

Our small-size Linux computer is bringing robotics to the people and businesses.

License

This book has been based on diferent Linux documentation avialible on the internet. Refer to the sources for the corresponding licenses:

- [Python Documentation](#)
- [Python Standard Library](#)
- [Python Package Index](#)

All Python releases are Open Source (see [link](#) for the Open Source Definition).

- *Foundations of Python Network Programming* by Brandon Rhodes and John Goerzen

Unless specified, this content is licensed under the Creative Commons Attribution-NonComercial-Share Alike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.



All derivative works are to be attributed to *Silvia Núñez Rivero* of **Erle Robotics S.L.**

For any questions, concerns, or issues submit them to support [at] erlerobot.com.

Introduction to Client/Server Networking

This chapter is about network programming with the Python language: about accomplishing a specific set of tasks that all involve a particular technology—computer networks—using a general-purpose programming language that can do all sorts of things.

For now on, we will use frequently:

- [Python Standard Library documentation](#)
- [Python Package Index](#)

Virtualenv

A common situation is that you find a *Python package* that sounds like it might already do exactly what you want, and that you want to try it out on your system. For this you should be introduced to very best Python technology for quickly trying out a new library: **virtualenv**

In the old days, installing a Python package was a gruesome and irreversible act that required administrative privileges on your machine and left your system Python install permanently altered.

Careful Python programmers do not suffer from this situation any longer. Many of them install only one Python package system-wide: virtualenv. Once virtualenv is installed, you have the power to create any number of small, self-contained “virtual Python environments” where packages can be installed, un-installed, and experimented with without contaminating your system-wide Python. When a particular project or experiment is over, you simply remove its virtual environment directory, and your system is clean.

Installing virtualenv in Erle

This is the [official website of virtualenv](#), where you can find information about the installation and the usage.

If you are connected to the Internet from Erle (by using a wireless nado usb) then you only need to type:

```
root@erlerobot:~# pip install virtualenv
```

If not the process must be a bit more tedious:

First of all you need to download the virtualenv from [here](#). Download the file called `virtualenv-1.11.6.tar.gz` (md5, pgp) to your PC. Then copy it to Erleboard, you can find in [this tutorial](#) how to do it. Once you have copied it, type :

```
root@erlerobot:~# tar xvfz virtualenv-1.11.6.tar.gz
root@erlerobot:~# cd virtualenv-1.11.6
root@erlerobot:~# python setup.py install
```

Congratulations you are now ready to use it!

Create a virtual environment to test packages

We are now going to use virtualenv to create a new environment and install the `googlemaps` package on it. You can read more about this package [here](#).

Now you type the following:

```
root@erlerobot:~# virtualenv --no-site-packages gmapenv
New python executable in gmapenv/bin/python
Installing setuptools, pip...done.
root@erlerobot:~#
root@erlerobot:~# cd gmapenv
root@erlerobot:~/gmapenv# ls
bin  include  lib  local
root@erlerobot:~/gmapenv# . bin/activate
(gmapenv)root@erlerobot:~/gmapenv# python -c 'import googlemaps'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named googlemaps
(gmapenv)root@erlerobot:~/gmapenv#
```

As you can see, the `googlemaps` package is not yet available. To install it, use the `pip` command that is inside your virtualenv and that is now on your path thanks to the `activate` command that you ran:

```
(gmapenv)root@erlerobot:~/gmapenv# pip install googlemaps
Downloading/unpacking googlemaps
Downloading googlemaps-1.0.2.tar.gz (60Kb): 60Kb downloaded
Running setup.py egg_info for package googlemaps
Installing collected packages: googlemaps
Running setup.py install for googlemaps
Successfully installed googlemaps
Cleaning up...
```

The python binary inside the virtualenv will now have the `googlemaps` package available:

```
(gmapenv)root@erlerobot:~/gmapenv# python -c 'import googlemaps'
```

When you install a packet, you should be careful: it must be suitable for Erle architecture.

Introduction to socket

We will use sockets a lot in future chapters. Thus, this chapter's aim is to introduce you the basic concepts of socket.

What is socket?

Rather than trying to invent its own API for doing networking, Python made an interesting decision: it simply provides a slightly object-based interface to all of the normal, gritty, low-level operating system calls that are normally used to accomplish networking tasks on POSIX-compliant operating systems.

So, Python exposes the normal POSIX calls for raw UDP and TCP connections rather than trying to invent any of its own. And the normal POSIX networking calls operate around a central concept called a socket.

That means that communication between different entities on a network is based on the classic concept Python sockets. Sockets are an abstract concept that designates the end point of a connection. The programs use sockets to communicate with other programs, which may be located on different computers. A socket is defined by the IP address of the machine, the port on which it listens, and the protocol used.

Moreover, if you have ever worked with POSIX before, you will probably have run across the fact that instead of making you repeat a file name over and over again, the calls let you use the file name to create a “file descriptor” that represents a connection to the file, and through which you can access the file until you are done working with it. Sockets provide the same idea for the networking realm: when you ask for access to a line of communication—like a UDP port, as we are about to see—you create one of these abstract “socket” objects and then ask for it to be bound to the port you want to use. If the binding is successful, then the socket “holds on to” that port number for.

You should, as well, be aware of that part of the trouble with understanding these things is that “socket” can mean a number of subtly different things, depending on context. So first, let’s make a distinction between a “client” socket - an endpoint of a conversation, and a “server” socket, which is more like a switchboard operator. The client application (your browser, for example) uses “client” sockets exclusively; the web server it’s talking to uses both “server” sockets and “client” sockets.

From [Python documentation](#) we can extract more info about socket module.

Creating a Socket

Roughly speaking, when you clicked on the link that brought you to this page, your browser did something like the following:

```
#create an INET, STREAMing socket
s = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM)
#now connect to the web server on port 80
# - the normal http port
s.connect(("www.mcmillan-inc.com", 80))
```

When the connect completes, the socket `s` can be used to send in a request for the text of the page. The same socket will read the reply, and then be destroyed. That's right, destroyed. Client sockets are normally only used for one exchange (or a small set of sequential exchanges).

What happens in the web server is a bit more complex. First, the web server creates a "server socket":

```
#create an INET, STREAMing socket
serversocket = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM)
#bind the socket to a public host,
# and a well-known port
serversocket.bind((socket.gethostname(), 80))
#become a server socket
serversocket.listen(5)
```

A couple things to notice: we used `socket.gethostname()` so that the socket would be visible to the outside world. If we had used `s.bind('localhost', 80)` or `s.bind('127.0.0.1', 80)` we would still have a "server" socket, but one that was only visible within the same machine. `s.bind('', 80)` specifies that the socket is reachable by any address the machine happens to have.

A second thing to note: low number ports are usually reserved for "well known" services (HTTP, SNMP etc). If you're playing around, use a nice high number (4 digits).

Finally, the argument to listen tells the socket library that we want it to queue up as many as 5 connect requests (the normal max) before refusing outside connections. If the rest of the code is written properly, that should be plenty.

Now that we have a "server" socket, listening on port 80, we can enter the mainloop of the web server:

```
while 1:
    #accept connections from outside
    (clientsocket, address) = serversocket.accept()
    #now do something with the clientsocket
    #in this case, we'll pretend this is a threaded server
    ct = client_thread(clientsocket)
    ct.run()
```

There's actually 3 general ways in which this loop could work - dispatching a thread to handle `clientsocket`, create a new process to handle `clientsocket`, or restructure this app to use non-blocking sockets, and multiplex between our "server" socket and any active `clientsockets` using `select`. The important thing to understand now is this: this is all a "server" socket does. It doesn't send any data. It doesn't receive any data. It just produces "client" sockets. Each `clientsocket` is created in response to some other "client" socket doing a `connect()` to the host and port we're bound to. As soon as we've created that `clientsocket`, we go back to listening for more connections. The two "clients" are free to chat it up - they are using some dynamically allocated port which will be recycled when the conversation ends.

Using sockets

The first thing to note, is that the web browser's "client" socket and the web server's "client" socket are identical beasts. That is, this is a "peer to peer" conversation. Or to put it another way, as the designer, you will have to decide what the rules of etiquette are for a conversation. Normally, the connecting socket starts the conversation, by sending in a request, or perhaps a signon. But that's a design decision - it's not a rule of sockets.

Now there are two sets of verbs to use for communication. You can use `send()` and `recv()`, or you can transform your client socket into a file-like beast and use `read()` and `write()`. I'm not going to talk about it here, except to warn you that you need to use flush on sockets. These are buffered "files", and a common mistake is to write something, and then read for a reply. Without a flush in there, you may wait forever for the reply, because the request may still be in your output buffer.

Now we come to the major stumbling block of sockets - `send()` and `recv()` operate on the network buffers. They do not necessarily handle all the bytes you hand them (or expect from them), because their major focus is handling the network buffers. In general, they return when the associated network buffers have been filled (send) or emptied (recv). They then tell you how many bytes they handled. It is your responsibility to call them again until your message has been completely dealt with.

When a `recv()` returns 0 bytes, it means the other side has closed (or is in the process of closing) the connection. You will not receive any more data on this connection.

A protocol like HTTP uses a socket for only one transfer. The client sends a request, then reads a reply. That's it. The socket is discarded. This means that a client can detect the end of the reply by receiving 0 bytes.

But if you plan to reuse your socket for further transfers, you need to realize that there is no EOT on a socket. I repeat: if a socket `send()` or `recv()` returns after handling 0 bytes, the connection has been broken. If the connection has not been broken, you may wait on a `recv()` forever, because the socket will not tell you that there's nothing more to read (for now). Now if you think about that a bit, you'll come to realize a fundamental truth of sockets: messages must either be fixed length (yuck), or be delimited (shrug), or indicate how long they are (much better), or end by shutting down the connection. The choice is entirely yours, (but some ways are righter than others).

Assuming you don't want to end the connection, the simplest solution is a fixed length message:

```
class mysocket:
    '''demonstration class only
    - coded for clarity, not efficiency
    '''

    def __init__(self, sock=None):
        if sock is None:
            self.sock = socket.socket(
                socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock

    def connect(self, host, port):
        self.sock.connect((host, port))

    def mysend(self, msg):
        totalsent = 0
        while totalsent < MSGLEN:
            sent = self.sock.send(msg[totalsent:])
            if sent == 0:
                raise RuntimeError("socket connection broken")
            totalsent = totalsent + sent

    def myreceive(self):
        chunks = []
        bytes_recd = 0
        while bytes_recd < MSGLEN:
            chunk = self.sock.recv(min(MSGLEN - bytes_recd, 2048))
            if chunk == '':
                raise RuntimeError("socket connection broken")
```

```
chunks.append(chunk)
bytes_recd = bytes_recd + len(chunk)
return ''.join(chunks)
```

The sending code here is usable for almost any messaging scheme - in Python you send strings, and you can use `len()` to determine its length (even if it has embedded `\0` characters). It's mostly the receiving code that gets more complex.

The easiest enhancement is to make the first character of the message an indicator of message type, and have the type determine the length. Now you have two recvs - the first to get (at least) that first character so you can look up the length, and the second in a loop to get the rest. If you decide to go the delimited route, you'll be receiving in some arbitrary chunk size, (4096 or 8192 is frequently a good match for network buffer sizes), and scanning what you've received for a delimiter.

One complication to be aware of: if your conversational protocol allows multiple messages to be sent back to back (without some kind of reply), and you pass `recv()` an arbitrary chunk size, you may end up reading the start of a following message. You'll need to put that aside and hold onto it, until it's needed.

Prefixing the message with its length (say, as 5 numeric characters) gets more complex, because (believe it or not), you may not get all 5 characters in one recv. In playing around, you'll get away with it; but in high network loads, your code will very quickly break unless you use two recv loops - the first to determine the length, the second to get the data part of the message. Nasty. This is also when you'll discover that send does not always manage to get rid of everything in one pass. And despite having read this, you will eventually get bit by it!

We will discuss the issue of framing (delimiting messages) in later chapter: *Network data and Network errors*

Disconnecting

Strictly speaking, you're supposed to use `shutdown` on a socket before you close it. The `shutdown` is an advisory to the socket at the other end. Depending on the argument you pass it, it can mean "I'm not going to send anymore, but I'll still listen", or "I'm not listening, good riddance!". Most socket libraries, however, are so used to programmers neglecting to use this piece of etiquette that normally a close is the same as `shutdown() ; close()`. So in most situations, an explicit shutdown is not needed.

One way to use `shutdown` effectively is in an HTTP-like exchange. The client sends a request and then does a `shutdown(1)`. This tells the server "This client is done sending, but can still receive." The server can detect "EOF" by a receive of 0 bytes. It can assume it has the complete request. The server sends a reply. If the send completes successfully then, indeed, the client was still receiving.

Python takes the automatic shutdown a step further, and says that when a socket is garbage collected, it will automatically do a close if it's needed. But relying on this is a very bad habit. If your socket just disappears without doing a close, the socket at the other end may hang indefinitely, thinking you're just being slow. So, it is very recommendable close your sockets when you're done.

Non - blocking sockets

In Python, you use `socket.setblocking(0)` to make it non-blocking. You do this after creating the socket, but before using it. (Actually, if you're nuts, you can switch back and forth.)

The major mechanical difference is that `send()`, `recv()`, `connect` and `accept` can return without having done anything. You have (of course) a number of choices. You can check return code and error codes and generally drive yourself crazy. Your app will grow large, buggy and suck CPU. So let's skip the brain-dead solutions and do it right. Use `select`.

```
ready_to_read, ready_to_write, in_error = \
    select.select(
        potential_readers,
        potential_writers,
        potential_errs,
        timeout)
`
```

You pass `select` three lists: the first contains all sockets that you might want to try reading; the second all the sockets you might want to try writing to, and the last (normally left empty) those that you want to check for errors. You should note that a socket can go into more than one list. The `select` call is blocking, but you can give it a timeout. This is generally a sensible thing to do - give it a nice long timeout (say a minute) unless you have good reason to do otherwise.

In return, you will get three lists. They contain the sockets that are actually readable, writable and in error. Each of these lists is a subset (possibly empty) of the corresponding list you passed in.

If a socket is in the output readable list, you can be as-close-to-certain-as-we-ever-get-in-this-business that a `recv` on that socket will return something. Same idea for the writable list. You'll be able to send something. Maybe not all you want to, but something is better than nothing. (Actually, any reasonably healthy socket will return as writable - it just means outbound network buffer space is available.)

If you have a "server" socket, put it in the `potential_readers` list. If it comes out in the readable list, your `accept` will (almost certainly) work. If you have created a new socket to connect to someone else, put it in the `potential_writers` list. If it shows up in the writable list, you have a decent chance that it has connected.

One very nasty problem with `select`: if somewhere in those input lists of sockets is one which has died a nasty death, the `select` will fail. You then need to loop through every single damn socket in all those lists and do a `select([sock], [], [], 0)` until you find the bad one. That timeout of 0 means it won't take long, but it's ugly.

Actually, `select` can be handy even with blocking sockets. It's one way of determining whether you will block - the socket returns as readable when there's something in the buffers. However, this still doesn't help with the problem of determining whether the other end is done, or just busy with something else.

Portability alert: On Unix, `select` works both with the sockets and files. Don't try this on Windows. On Windows, `select` works with sockets only.

UDP and TCP

The two principal approaches when building a top IP are: UDP and TCP.

- The vast majority of applications today are built atop TCP, the Transmission Control Protocol, which offers ordered and reliable data streams between IP applications.
- A few protocols, usually with short, self-contained requests and responses, and simple clients that will not be annoyed if a request gets lost and they have to repeat it, choose UDP, the User Datagram Protocol.

This two methods are described in depth along this chapter, but for now have take a quick look to the differences between this two.

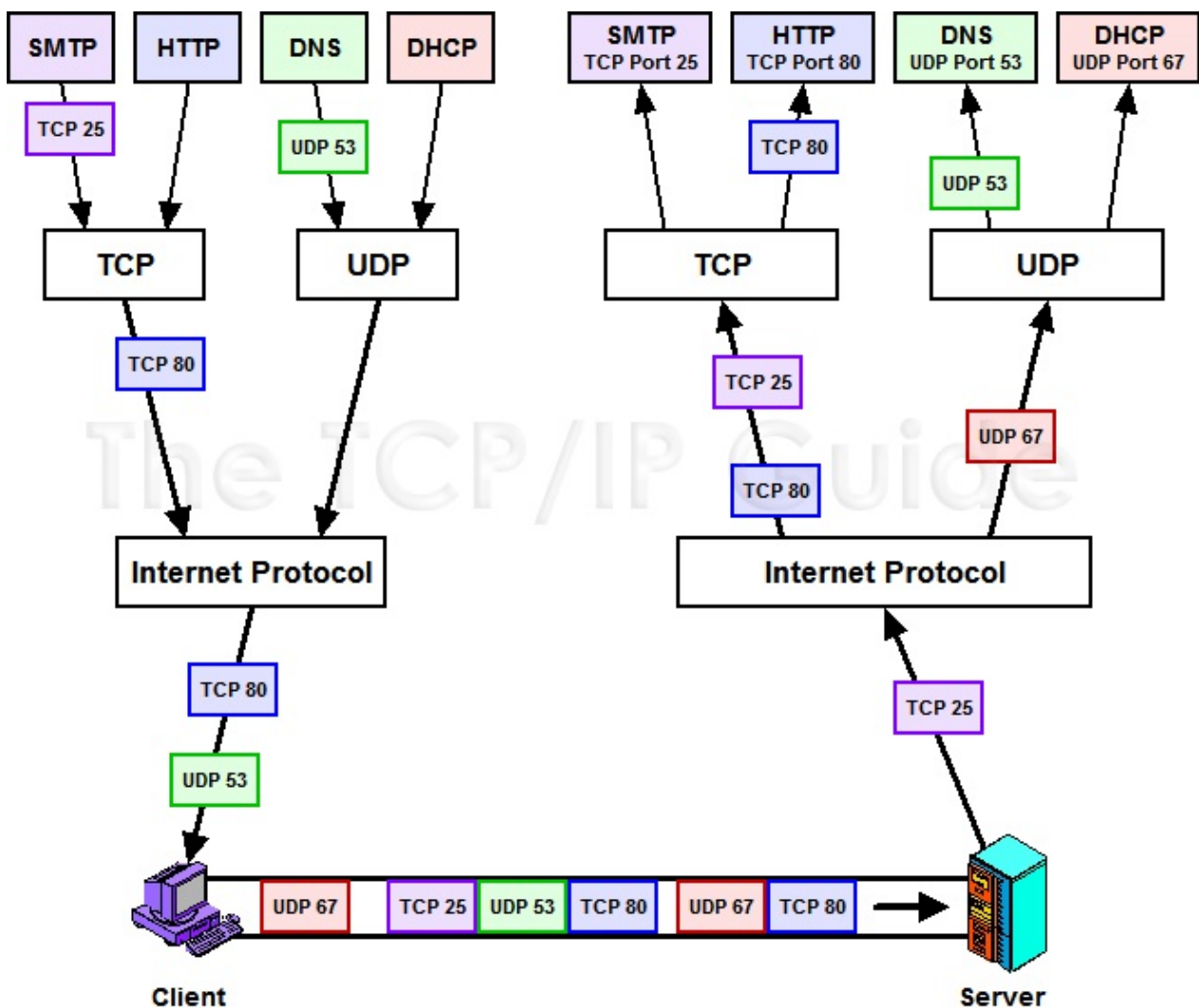
TCP	UDP
Reliable—monitors message transmission, tracks data transfer to ensure receipt of all packets	Unreliable—no concept of acknowledgment, retransmission, or timeout –
Ordered—buffering provisions to ensure correct order of data packets	Not ordered—data arrives in order of receipt
Heavyweight—dedicated connection, provisions for speed and congestion control	Lightweight—no dedicated end-to-end connection, no congestion control
Streaming	Datagram oriented
Heavy overhead	Light overhead
Lower speed	Higher speed

Addresses and port numbers

We are going to review a bit about this two topics:

The IP protocol assigns an IP address—which traditionally takes the form of a four-octet code, like 18.9.22.69—to every machine connected to an IP network. In fact, it does a bit more than this: a machine with several network cards connected to the network will typically have a different IP address for each card, so that other hosts can choose the network over which you want to contact the machine. But even if an IP-connected machine has only one network card, it also has at least one other network address: the address 127.0.0.1 is how machines can connect to themselves. It serves as a stable name that each machine has for itself, that stays the same as network cables are plugged and unplugged and as wireless signals come and go. And these IP addresses allow millions of different machines, using all sorts of different network hardware, to pass packets to each other over the fabric of an IP network.

But with UDP and TCP we now take a big step, and stop thinking about the routing needs of the network as a whole and start considering the needs of specific applications that are running on a particular machine. And the first thing we notice is that a single computer today can have many dozens of programs running on it at any given time—and many of these will want to use the network at the same moment. You might be checking e-mail with Thunderbird while a web page is downloading in Google Chrome, or installing a Python package with pip over the network while checking the status of a remote server with SSH. Somehow, all of those different and simultaneous conversations need to take place without interfering with each other. This problem is known as need for multiplexing: the need for a single channel to be shared unambiguously by several different conversations.



You also should remember that when a program on your computer sends or receives data over the Internet it sends that data to an ip address and a specific port on the remote computer, and receives the data on a usually random port on its

own computer. If it uses the TCP protocol to send and receive the data then it will connect and bind itself to a TCP port. If it uses the UDP protocol to send and receive data, it will use a UDP port.

UDP

Now, we are going to centre in UDP (User Datagram Protocol).

How UDP works?

The UDP scheme is really quite simple; an IP address and port are all that is necessary to direct a packet to its destination.

Imagine, for example, that you set up a DNS server (Chapter 4) on one of your machines, with the IP address 192.168.1.9. To allow other computers to find the service, the server will ask the operating system for permission to take control of the UDP port with the standard DNS port number 53. Assuming that no process is already running that has claimed that port number, the DNS server will be granted that port.

Next, imagine that a client machine with the IP address 192.168.1.30 on your network is given the IP address of this new DNS server and wants to issue a query. It will craft a DNS query in memory, and then ask the operating system to send that block of data as a UDP packet. Since there will need to be some way to identify the client when the packet returns, and since the client has not explicitly requested a port number, the operating system assigns it a random one—say, port 44137.

The packet will therefore wing its way toward port 53 with labels that identify its source as the IP address and UDP port numbers (here separated by a colon):

```
192.168.1.30:44137
```

And it will give its destination as the following:

```
192.168.1.9:53
```

This destination address, simple though it looks—just the number of a computer, and the number of a port—is everything that an IP network stack needs to guide this packet to its destination. The DNS server will receive the request from its operating system, along with the originating IP and port number. Once it has formulated a response, the DNS server will ask the operating system to send the response as a UDP packet to the IP address and UDP port number from which the request originally came. The reply packet will have the source and destination swapped from what they were in the original packet, and upon its arrival at the source machine, it will be delivered to the waiting client program.

When to use UDP

So, The User Data Protocol, UDP, lets user-level programs send individual packets across an IP network. Typically, a client program sends a packet to a server, which then replies back using the return address built into every UDP packet. You might think that UDP would be very efficient for sending small messages. Actually, UDP is efficient only if your host ever only sends one message at a time, then waits for a response.

There are two good reasons to use UDP:

- Because you are implementing a protocol that already exists, and it uses UDP.
- Because unreliable subnet broadcast is a great pattern for your application, and UDP supports it perfectly.

Socket (UDP)

As we have seen sockets makes talking to arbitrary machines around the world unbelievably easy (at least compared to other schemes).

When you craft programs that accept port numbers from user input like the command line or configuration files, it is friendly to allow not just numeric port numbers but to let users type humanreadable names for well-known ports. These names are standard, and are available through the `getservbyname()` call supported by Python's standard socket module. If we want to ask where the Domain Name Service lives, we could have found out this way:

```
import socket
socket.getservbyname('domain')
53
```

Now examine the following code which shows a simple server and client. You can see already that all sorts of operations are taking place that are drawn from the socket module in the Python Standard Library.

```
#UDP client and server on localhost
import socket, sys
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
MAX = 65535
PORT = 1060
if sys.argv[1:] == ['server']:
    s.bind(('127.0.0.1', PORT))
    print 'Listening at', s.getsockname()
    while True:
        data, address = s.recvfrom(MAX)
        print 'The client at', address, 'says', repr(data)
        s.sendto('Your data was %d bytes' % len(data), address)
elif sys.argv[1:] == ['client']:
    print 'Address before sending:', s.getsockname()
    s.sendto('This is my message', ('127.0.0.1', PORT))
    print 'Address after sending', s.getsockname()
    data, address = s.recvfrom(MAX) # overly promiscuous - see text!
    print 'The server', address, 'says', repr(data)
else:
    print >>sys.stderr, 'usage: udp_local.py server|client'
```

When running it, you should get something similar to this:

```
root@erlerobot:~/Python_files# python udp_local.py
usage: udp_local.py server|client
```

Now try to run first the server:

```
root@erlerobot:~/Python_files# python ude_local.py server
Listening at ('127.0.0.1', 1060)
```

And then in a new Terminal window the client:

```
root@erlerobot:~/Python_files# python udp_local.py client
Address before sending: ('0.0.0.0', 0)
Address after sending ('0.0.0.0', 59726)
The server ('127.0.0.1', 1060) says 'Your data was 18 bytes'
```

In the server window will appear a new line:

```
The client at ('127.0.0.1', 59726) says 'This is my message'
```

Note that the Python program can always use a socket's `getsockname()` method to retrieve the current IP and port to which the socket is bound. Once the socket has been bound successfully, the server is ready to start receiving requests! It enters a loop and repeatedly runs `recvfrom()`, telling the routine that it will happily receive messages up to a maximum length of `MAX`, which is equal to 65535 bytes—a value that happens to be the greatest length that a UDP packet can possibly have, so that we will always be shown the full content of each packet. Until we send a message with a client, our `recvfrom()` call will wait forever.

Unreliability, Backoff, Blocking, Timeouts

Because the client and server in the previous section were both running on the same machine and talking through its loopback interface—which is not even a physical network card that could experience a signaling glitch and lose a packet, but merely a virtual connection back to the same machine deep in the network stack—there was no real way that packets could get lost, and so we did not actually see any of the inconvenience of UDP.

You can run this client and server example on two different machines on the Internet. And instead of always answering client requests, this server randomly chooses to answer only half of the requests coming in from clients—which will let us demonstrate how to build reliability into our client code, without waiting what might be hours for a real dropped packet to occur.

```
import random, socket, sys
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
MAX = 65535
PORT = 1060
if 2 <= len(sys.argv) <= 3 and sys.argv[1] == 'server':
    interface = sys.argv[2] if len(sys.argv) > 2 else ''
    s.bind((interface, PORT))
    print 'Listening at', s.getsockname()
    while True:
        data, address = s.recvfrom(MAX)
        if random.randint(0, 1):
            print 'The client at', address, 'says:', repr(data)
            s.sendto('Your data was %d bytes' % len(data), address)
        else:
            print 'Pretending to drop packet from', address
elif len(sys.argv) == 3 and sys.argv[1] == 'client':
    hostname = sys.argv[2]
    s.connect((hostname, PORT))
    print 'Client socket name is', s.getsockname()
    delay = 0.1
    while True:
        s.send('This is another message')
        print 'Waiting up to', delay, 'seconds for a reply'
        s.settimeout(delay)
        try:
            data = s.recv(MAX)
        except socket.timeout:
            delay *= 2 # wait even longer for the next request
            if delay > 2.0:
                raise RuntimeError('I think the server is down!')
        except:
            raise # a real error, so we let the user see it
        else:
            break # we are done, and can stop looping
    print 'The server says', repr(data)
else:
    print >>sys.stderr, 'usage: udp_remote.py server [ <interface> ]'
    print >>sys.stderr, ' or: udp_remote.py client <host>'
    sys.exit(2)
```

Running the file itself result on:

```
root@erlerobot:~/Python_files# python socket1.py
usage: udp_remote.py server [ <interface> ]
or: udp_remote.py client <host>
```

Then run the server:

```
root@erlerobot:~/Python_files# python udp_remote.py server
Listening at ('0.0.0.0', 1060)
```

And now the client, remember to pass the hostname where the server script is being run(in this case the same machine):

```
root@erlerobot:~/Python_files# python udep_remote.py client 127.0.0.1
Client socket name is ('127.0.0.1', 54770)
Waiting up to 0.1 seconds for a reply
Waiting up to 0.2 seconds for a reply
Waiting up to 0.4 seconds for a reply
Waiting up to 0.8 seconds for a reply
The server says 'Your data was 23 bytes'
```

As you can see, each time a request is received, the server uses `randint()` to flip a coin to decide whether this request will be answered, so that we do not have to keep running the client all day waiting for a real dropped packet. The client will find that one or more of its requests never result in replies.

Connecting UDP Sockets

The remote UDP client in `socket1.py` uses a new call that we have not discussed before: the `connect()` socket operation. You can see easily enough what it does. Instead of having to use `sendto()` and an explicit UDP address every time we want to send something to the server, the `connect()` call lets the operating system know ahead of time which remote address to which we want to send packets, so that we can simply supply data to the `send()` call and not have to repeat the server address again. But `connect()` does something else important, which will not be obvious at all from reading the script of `udp_remote.py`. To approach this topic, let us return to `udp_local.py` file for a moment. You will recall that both its client and server use the loopback IP address and assume reliable delivery—the client will wait forever for a response. Try running the client in one window:

```
root@erlerobot:~/Python_files# python udp_local.py
Address before sending: ('0.0.0.0', 0)
Address after sending ('0.0.0.0', 52970)
```

The client is now waiting—perhaps forever—for a response in reply to the packet it has just sent to the localhost IP address at UDP port 1060. But what if we nefariously try sending it back a packet from a different server, instead? From another command prompt on the same system, try running Python and entering these commands—and for the port number, copy the integer that was just printed to the screen when you ran the UDP client:

```
>>> import socket
>>> s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
>>> s.sendto('Fake reply', ('127.0.0.1', 52970))
10
>>>
```

In the client window appears:

```
The server ('127.0.0.1', 65320) says 'Fake reply'
```

It turns out that our first client accepts answers from anywhere. Even though the server is running on the localhost, and remote network connectivity is not even desirable, the client will even accept packets from another machine. If I bring up a Python prompt on another box and run the same two lines of code as just shown, then a waiting client can even see the remote IP address.

There are, then, two ways to write UDP clients that are careful about the return addresses of the packets arriving back:

- You can use `sendto()` and direct each outgoing packet to a specific destination, and then use `recvfrom()` to receive the replies and carefully check the return address it gives you against the list of servers to which you have made outstanding requests.
- You can `connect()` your socket right after creating it, and then simply use `send()` and `recv()`, and the operating system will filter out unwanted packets for you. This works only for speaking to one server at a time, because running `connect()` a second time on the same socket does not add a second destination address to your UDP socket. Instead, it wipes out the first address entirely, so that no further replies from the earlier address will be delivered to your program.

Binding to Interfaces(UDP)

When using sockets, it is important to distinguish the act of “binding”—by which you grab a particular UDP port for the use of a particular socket—from the act that the client performs by “connecting,” which limits all replies received so that they can come only from the particular server to which you want to talk.

So far we have seen two possibilities for the IP address used in the `bind()` call that the server makes: you can use `'127.0.0.1'` to indicate that you only want packets from other programs running on the same machine, or use an empty string `''` as a wildcard, indicating that you are willing to receive packets from any interface. It actually turns out that there is a third choice: you can provide the IP address of one of the machine's external IP interfaces, like its Ethernet connection or wireless card, and the server will listen only for packets destined for those IPs. First, what if we bind solely to an external interface? Run the server like this, using whatever your operating system tells you is the external IP address of your system:

```
root@erlerobot:~/Python_files# python udp_remote.py server 192.168.1.35
Listening at ('192.168.1.35', 1060)
```

Connecting to this IP address from another machine should still work just fine:

```
root@erlerobot:~/Python_files# python udp_remote.py client 192.168.1.35
Client socket name is ('192.168.1.35', 58824)
Waiting up to 0.1 seconds for a reply
The server says 'Your data was 23 bytes'
```

But if you try connecting to the service through the loopback interface by running the client script on the same machine, the packets will never be delivered:

```
root@erlerobot:~/Python_files# python udp_remote.py client 127.0.0.1
Client socket name is ('127.0.0.1', 60251)
Waiting up to 0.1 seconds for a reply
Traceback (most recent call last):
...
socket.error: [Errno 111] Connection refused
```

If you run client again on the same machine, but this time use the external IP address of the box, even though the client and server are both running there, this will not give any error. So binding to an IP interface might limit which external hosts can talk to you; but it will certainly not limit conversations with other clients on the same machine, so long as they know the IP address that they should use to connect.

Now, stop all of the scripts that are running, and we can try running two servers on the same box.

```
root@erlerobot:~/Python_files# python udp_remote.py server 127.0.0.1
Listening at ('127.0.0.1', 1060)
```

And then we try running a second one, connected to the wildcard IP address that allows requests from any address:

```
root@erlerobot:~/Python_files# python udp_remote.py server
Traceback (most recent call last):
...
socket.error: [Errno 98] Address already in use
```

We have learned something about operating system IP stacks and the rules that they follow: they do not allow two different sockets to listen at the same IP address and port number, because then the operating system would not know where to

deliver incoming packets. But what if instead of trying to run the second server against all IP interfaces, we just ran it against an external IP interface—one that the first copy of the server is not listening to? Let us try:

```
root@erlerobot:~/Python_files# python udp_remote.py server 192.168.1.35
Listening at ('192.168.1.35', 1060)
```

It worked, this means that there are now two servers running on this machine, one of which is bound to the inward-looking port 1060 on the loopback interface, and the other looking outward for packets arriving on port 1060 from the network to which my wireless card has connected.

IP network stack never thinks of a UDP port as a lone entity that is either entirely available, or else in use, at any given moment. Instead, it thinks in terms of UDP “socket names” that are always a pair linking an IP interface—even if it is the wildcard interface—with a UDP port number. It is these socket names that must not conflict among the listening servers at any given moment, rather than the bare UDP ports that are in use.

UDP Fragmentation

The foregoing program listings have suggested that a UDP packet can be up to 64kB in size, whereas you probably already know that your Ethernet or wireless card can only handle packets of around 1,500 bytes instead.

The actual truth is that IP sends small UDP packets as single packets on the wire, but splits up larger UDP packets into several small physical packets. This means that large packets are more likely to be dropped, since if any one of their pieces fails to make its way to the destination, then the whole packet can never be reassembled and delivered to the listening operating system. But aside from the higher chance of failure, this process of fragmenting large UDP packets so that they will fit on the wire should be invisible to your application. There are three ways, however, in which it might be relevant:

- If you are thinking about efficiency, you might want to limit your protocol to small packets, to make retransmission less likely and to limit how long it takes the remote IP stack to reassemble your UDP packet and give it to the waiting application.
- If the ICMP packets are wrongfully blocked by a firewall that would normally allow your host to auto-detect the MTU between you and the remote host, then your larger UDP packets might disappear into oblivion without your ever knowing. The MTU is the “maximum transmission unit” or “largest packet size” that all of the network devices between two hosts will support.
- If your protocol can make its own choices about how it splits up data between different packets, and you want to be able to auto-adjust this size based on the actual MTU between two hosts, then some operating systems let you turn off fragmentation and receive an error if a UDP packet is too big. This lets you regroup and split it into several packets if that is possible.

Linux is one operating system that supports this last option. Take a look at `big_sender.py`, which sends a very large message to one of the servers that we have just designed.

```
import IN, socket, sys
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
MAX = 65535
PORT = 1060
if len(sys.argv) != 2:
    print >>sys.stderr, 'usage: big_sender.py host'
    sys.exit(2)
hostname = sys.argv[1]
s.connect((hostname, PORT))
s.setsockopt(socket.IPPROTO_IP, IN.IP_MTU_DISCOVER, IN.IP_PMTUDISC_DO)
try:
    s.send('#' * 65000)
except socket.error:
    print 'The message did not make it'
    option = getattr(IN, 'IP_MTU', 14) # constant taken from <linux/in.h>
    print 'MTU:', s.getsockopt(socket.IPPROTO_IP, option)
else:
    print 'The big message was sent! Your network supports really big packets!'
```

If we run this program against a server elsewhere on my home network, then we discover that my wireless network allows physical packets that are no bigger than the 1,500 bytes typically supported by Ethernet-style networks:

```
root@erlerobot:~/Python_files# python big_sender.py 127.0.0.0
The message did not make it
MTU: 1500
```


Socket Options

The POSIX socket interface also supports all sorts of socket options that control specific behaviors of network sockets. These are accessed through the Python socket methods `getsockopt()` and `setsockopt()`, using the options you will find documented for your operating system. You can find these options described in the [Python documentation](#).

When setting socket options, the set call is similar to:

```
value = s.getsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST)
s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, value)
```

Here are some of the more common options:

- `SO_BROADCAST`: Allows broadcast UDP packets to be sent and received; see the next section for details.
- `SO_DONTROUTE`: Only be willing to send packets that are addressed to hosts on subnets to which this computer is connected directly.
- `SO_TYPE`: When passed to `getsockopt()`, this returns to you regardless of whether a socket is of type `SOCK_DGRAM` and can be used for UDP, or it is of type `SOCK_STREAM` and instead supports the semantics of TCP.

NOTE:

If UDP has a superpower, it is its ability to support **broadcast**: instead of sending a packet to some specific other host, you can point it at an entire subnet to which your machine is attached and have the physical network card broadcast the packet so that all attached hosts see it without its having to be copied separately to each one of them. [Here](#) and [here](#) you can find two examples of broadcasting.

TCP

The Transmission Control Protocol (TCP) is the workhorse of the Internet. Protocols that carry documents and files nearly always ride atop TCP, including HTTP and all the major ways of transmitting e-mail. It is also the foundation of choice for protocols that carry on long conversations between people or computers, like SSH and many popular chat protocols

How TCP works

First, every packet is given a sequence number, so that the system on the receiving end can put them back together in the right order, and so that it can notice missing packets in the sequence and ask that they be re-transmitted. Instead of using sequential integers (1,2,...) to mark packets, TCP uses a counter that counts the number of bytes transmitted. So a 1,024-byte packet with a sequence number of 7,200 would be followed by a packet with a sequence number of 8,224. This means that a busy network stack does not have to remember how it broke a data stream up into packets; if asked for a re-transmission, it can break the stream up into packets some other way (which might let it fit more data into a packet if more bytes are now waiting for transmission), and the receiver can still put the packets back together.

Rather than running very slowly in lock-step by needing every packet to be acknowledged before it sends the next one, TCP sends whole bursts of packets at a time before expecting a response. The amount of data that a sender is willing to have on the wire at any given moment is called the size of the TCP "window." The TCP implementation on the receiving end can regulate the window size of the transmitting end, and thus slow or pause the connection. This is called "flow control." This lets it forbid the transmission of additional packets in cases where its input buffer is full and it would have to discard any more data if it were to arrive right now.

Finally, if TCP sees that packets are being dropped, it assumes that the network is becoming congested and stops sending as much data every second.

When to use TCP

TCP has very nearly become a universal default when two programs need to communicate, we should look at a few instances in which its behavior is not optimal for certain kinds of data, in case an application you are writing ever falls into one of these categories. First, TCP is unwieldy for protocols where clients want to send single, small requests to a server, and then are done and will not talk to it further. It takes three packets for two hosts to set up a TCP connection—the famous sequence of SYN, SYN-ACK, and ACK (which mean “I want to talk, here is the packet sequence number I will be starting with”; “okay, here’s mine”; “okay!”)—and then another three or four to shut the connection back down (either a quick FIN, FIN-ACK, ACK, or a slightly longer pair of separate FIN and ACK packets). That is six packets just to send a single request: Protocol designers quickly turn to UDP in such cases.

In view of this we are going to detail two situations where the use of TCP is not appropriate:

- Where UDP really shines over TCP, then, is where such a long-term relationship does not pertain between client and server, and especially where there are so many clients that a typical TCP implementation would run out of port numbers if it had to keep up with a separate data stream for each active client.
- The second situation where TCP is inappropriate is when an application can do something much smarter than simply re-transmit data when a packet has been lost. Imagine an audio chat conversation, for example: if a second’s worth of data is lost because of a dropped packet, then it will do little good to simply re-send that same second of audio, over and over, until it finally arrives.

What TCP Sockets Mean

As we have mentioned before, TCP uses port numbers to distinguish different applications running at the same IP address, and follows exactly the same conventions regarding well-known and ephemeral port number. With a stateful stream protocol like TCP, the `connect()` call becomes the fundamental act upon which all other network communication hinges. TCP `connect()` can fail: The remote host might not answer; it might refuse the connection; or more obscure protocol errors might occur like the immediate receipt of a RST (“reset”) packet. Because a stream connection involves setting up a persistent connection between two hosts, the other host needs to be listening and ready to accept your connection.

On the “server side”—which, for the purpose of this chapter, is the conversation partner not doing the `connect()` call but receiving the SYN packet that it initiates—an incoming connection generates an even more momentous event, the creation of a new socket. This is because the standard POSIX interface to TCP actually involves two completely different kinds of sockets: “passive” listening sockets and active “connected” ones:

- A *passive socket* holds the “socket name”—the address and port number—at which the server is ready to receive connections. No data can ever be received or sent by this kind of port; it does not represent any actual network conversation. Instead, it is how the server alerts the operating system to its willingness to receive incoming connections in the first place.
- An *active socket* (connected socket), is bound to one particular remote conversation partner, who has their own IP address and port number. It can be used only for talking back and forth with that partner, and can be read and written to without worrying about how the resulting data will be split up into packets—in many cases, a connected socket can be passed to another POSIX program that expects to read from a normal file, and the program will never even know that it is talking to the network.

Note that while a passive socket is made unique by the interface address and port number at which it is listening (so that no one else is allowed to grab that same address and port), there can be many active sockets that all share the same local socket name.

What makes an active socket unique is, rather, the four-part coordinate: (*local_ip*, *local_port*, *remote_ip*, *remote_port*). It is this four-tuple by which the operating system names each active TCP connection, and incoming TCP packets are examined to see whether their source and destination address associate them with any of the currently active sockets on the system.

A Simple TCP Client and Server

Here you can find the code of a simple TCP client and server that send and receive 16 octets:

```
import socket, sys
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
HOST = sys.argv.pop() if len(sys.argv) == 3 else '127.0.0.1'
PORT = 1060
def recv_all(sock, length):
    data = ''
    while len(data) < length:
        more = sock.recv(length - len(data))
        if not more:
            raise EOFError('socket closed %d bytes into a %d-byte message'
                            % (len(data), length))
        data += more
    return data
if sys.argv[1:] == ['server']:
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((HOST, PORT))
    s.listen(1)
    while True:
        print 'Listening at', s.getsockname()
        sc, sockname = s.accept()
        print 'We have accepted a connection from', sockname
        print 'Socket connects', sc.getsockname(), 'and', sc.getpeername()
        message = recv_all(sc, 16)
        print 'The incoming sixteen-octet message says', repr(message)
        sc.sendall('Farewell, client')
        sc.close()
        print 'Reply sent, socket closed'
elif sys.argv[1:] == ['client']:
    s.connect((HOST, PORT))
    print 'Client has been assigned socket name', s.getsockname()
    s.sendall('Hi there, server')
    reply = recv_all(s, 16)
    print 'The server said', repr(reply)
    s.close()
else:
    print >>sys.stderr, 'usage: tcp_local.py server|client [host]'
```

First, the TCP `connect()` call is not the innocuous bit of local socket configuration that it is in the case of UDP, where it merely sets a default address used with any subsequent `send()` calls, and places a filter on packets arriving at our socket. Here, `connect()` is a real live network operation that kicks off the three-way handshake between the client and server machine so that they are ready to communicate. This means that `connect()` can fail, as you can verify quite easily by executing this script when the server is not running:

```
root@erlerobot:~/Python_files# python tcp_sixteen.py client
Traceback (most recent call last):
File "tcp_sixteen.py", line 29, in <module>
    s.connect((HOST, PORT))
File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/socket.py", line 224, in meth
    return getattr(self._sock,name)(*args)
socket.error: [Errno 61] Connection refused
```

You will see that this TCP client is in one way much simpler than our UDP client, because it does not need to make any provision for missing data. Because of the assurances that TCP provides, it can `send()` data without checking whether the remote end receives it, and run `recv()` without having to consider the possibility of re-transmitting its request.

When we perform a TCP `send()`, our operating system's networking stack will face one of three situations:

- The data can be immediately accepted by the system, either because the network card is immediately free to transmit, or because the system has room to copy the data to a temporary outgoing buffer so that your program can continue running. In these cases, `send()` returns immediately, and it will return the length of your data string because the whole string was transmitted.

- Another possibility is that the network card is busy and that the outgoing data buffer for this socket is full and the system cannot—or will not—allocate any more space. In this case, the default behavior of `send()` is simply to block, pausing your program until the data can be accepted.
- There is a final, hybrid possibility: that the outgoing buffers are almost full, but not quite, and so part of the data you are trying to send can be immediately queued, but the rest will have to wait. In this case, `send()` completes immediately and returns the number of bytes accepted from the beginning of your data string, but leaves the rest of the data unprocessed.

Fortunately, Python does not force us to do this dance ourselves every time we have a block of data to send: the Standard Library socket implementation provides a friendly `sendall()` method. Not only is `sendall()` faster than doing it ourselves, it releases the Global Interpreter Lock during its loop so that other Python threads can run without contention until all of the data has been transmitted. Unfortunately, no equivalent is provided for the `recv()` call, despite the fact that it might return only part of the data that is on the way from the client. Internally, the operating system implementation of `recv()` uses logic very close to that used when sending:

- If no data is available, then `recv()` blocks and your program pauses until data arrives.
- If plenty of data is available already in the incoming buffer, then you are given as many bytes as you asked `recv()` for.
- But if the buffer contains a bit of data, but not as much as you are asking for, then you are immediately returned what does happen to be there, even if it is not as much as you have asked for.

In the code stored in `tcp_sixteen.py`, you can see how the distinction between active and listening socket is carried through in actual server code. The link, which might strike you as odd at first, is that a listening socket actually produces new connected sockets as the return value that you get by listening. Follow the steps in the program listing to see the order in which the socket operations occur.

Run the server:

```
root@erlerobot:~/Python_files# python tcp_sixteen.py server
Listening at ('127.0.0.1', 1060)
```

And then the client(in another terminal window):

```
root@erlerobot:~/Python_files# python tcp_sixteen.py client
Client has been assigned socket name ('127.0.0.1', 49607)
The server said 'Farewell, client'
```

The server returns this:

```
We have accepted a connection from ('127.0.0.1', 49607)
Socket connects ('127.0.0.1', 1060) and ('127.0.0.1', 49607)
The incoming sixteen-octet message says 'Hi there, server'
Reply sent, socket closed
Listening at ('127.0.0.1', 1060)
```

Binding to Interfaces(TCP)

the IP address that you pair with a port number when you perform a bind() operation tells the operating system which network interfaces you are willing to receive connections from. The example invocations of `tcp_sixteen.py` used the localhost IP address 127.0.0.1, which protects your code from connections originating on other machines. You can verify this by running `tcp_sixteen.py` in server mode as shown previously, and trying to connect with a client from another machine:

```
root@erlerobot:~/Python_files# python tcp_sixteen.py client 192.168.1.35
Traceback (most recent call last):
  File "tcp_sixteen.py", line 29, in <module>
    s.connect((HOST, PORT))
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/socket.py", line 224, in meth
    return getattr(self._sock,name)(*args)
socket.error: [Errno 61] Connection refused
```

But if you run the server with an empty string for the hostname, which tells the Python bind() routine that you are willing to accept connections through any of your machine's active network interfaces, then the client can connect successfully from another host:

```
root@erlerobot:~/Python_files# python tcp_sixteen.py server ""
Listening at ('0.0.0.0', 1060)
```

Run the client:

```
root@erlerobot:~/Python_files# python tcp_sixteen.py client 192.168.1.35
Client has been assigned socket name ('192.168.1.35', 49696)
The server said 'Farewell, client'
```

This appear into the server terminal:

```
We have accepted a connection from ('192.168.1.35', 49696)
Socket connects ('192.168.1.35', 1060) and ('192.168.1.35', 49696)
The incoming sixteen-octet message says 'Hi there, server'
Reply sent, socket closed
Listening at ('0.0.0.0', 1060)
```


Deadlock

The term “deadlock” is used for all sorts of situations in computer science where two programs, sharing limited resources, can wind up waiting on each other forever because of poor planning. It turns out that it can happen fairly easily when using TCP.

Take a look at `tcp_deadlock.py` for an example of a server and client that try to be a bit too clever without thinking through the consequences. Here, the server author has done something that is actually quite intelligent. His job is to turn an arbitrary amount of text into uppercase. Recognizing that its client's requests can be arbitrarily large, and that one could run out of memory trying to read an entire stream of input before trying to process it, the server reads and processes small blocks of 1,024 bytes at a time.

```
import socket, sys
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

HOST = '127.0.0.1'
PORT = 1060

if sys.argv[1] == ['server']:
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((HOST, PORT))
    s.listen(1)
    while True:
        print 'Listening at', s.getsockname()
        sc, sockname = s.accept()
        print 'Processing up to 1024 bytes at a time from', sockname
        n = 0
        while True:
            message = sc.recv(1024)
            if not message:
                break
            sc.sendall(message.upper()) # send it back uppercase
            n += len(message)
            print '\r%d bytes processed so far' % (n,),
            sys.stdout.flush()
        print
        sc.close()
        print 'Completed processing'

elif len(sys.argv) == 3 and sys.argv[1] == 'client' and sys.argv[2].isdigit():

    bytes = (int(sys.argv[2]) + 15) // 16 * 16 # round up to // 16
    message = 'capitalize this!' # 16-byte message to repeat over and over

    print 'Sending', bytes, 'bytes of data, in chunks of 16 bytes'
    s.connect((HOST, PORT))

    sent = 0
    while sent < bytes:
        s.sendall(message)
        sent += len(message)
        print '\r%d bytes sent' % (sent,),
        sys.stdout.flush()

    print
    s.shutdown(socket.SHUT_WR)

    print 'Receiving all the data the server sends back'

    received = 0
    while True:
        data = s.recv(42)
        if not received:
            print 'The first data received says', repr(data)
            received += len(data)
        if not data:
            break
        print '\r%d bytes received' % (received,),

    s.close()

else:
```

```
print >>sys.stderr, 'usage: tcp_deadlock.py server | client <bytes>'
```

If you start the server and then run the client with a command-line argument specifying a modest number of bytes— say, asking it to send 32 bytes of data (for simplicity, it will round whatever value you supply up to a multiple of 16 bytes)—then it will get its text back in all uppercase:

```
root@erlerobot:~/Python_files# python tcp_deadlock.py server
Listening at ('127.0.0.1', 1060)
```

```
root@erlerobot:~/Python_files# python tcp_deadlock.py client 32
Sending 32 bytes of data, in chunks of 16 bytes
32 bytes sent
Receiving all the data the server sends back
The first data received says 'CAPITALIZE THIS!CAPITALIZE THIS!'
32 bytes received
```

On the server screen this is displayed:

```
Processing up to 1024 bytes at a time from ('127.0.0.1', 49702)
32 bytes processed so far
Completed processing
Listening at ('127.0.0.1', 1060)
```

Now, try using the client to send a very large stream of data, say, one totaling a gigabyte:

```
root@erlerobot:~/Python_files# python tcp_deadlock.py client 1073741824
Sending 1073741824 bytes of data, in chunks of 16 bytes
1399600 bytes sent
```

In the server window:

```
Processing up to 1024 bytes at a time from ('127.0.0.1', 49703)
688032 bytes processed so far
```

You will see both the client and the server furiously updating their terminal windows as they breathlessly update you with the amount of data they have transmitted and received. The numbers will climb and climb until, quite suddenly, both connections freeze. The server's output buffer and the client's input buffer have both finally filled, and TCP has used its window adjustment protocol to signal this fact and stop the socket from sending more data that would have to be discarded and later re-sent.

Closed Connections, Half-Open Connections

`tcp_deadlock.py` shows us how a Python socket object behaves when an end-of-file is reached. You will see that the client makes a `shutdown()` call on the socket after it finishes sending its transmission. This solves an important problem: if the server is going to read forever until it sees end-of-file, then how will the client avoid having to do a full `close()` on the socket and thus forbid itself from doing the many `recv()` calls that it still needs to make to receive the server's response? The solution is to "half-close" the socket—that is, to permanently shut down communication in one direction but without destroying the socket itself—so that the server can no longer read any data, but can still send any remaining reply back in the other direction, which will still be open. The `shutdown()` call can be used to end either direction of communication in a two-way socket like this; its argument can be one of three symbols:

- `SHUT_WR`: This is the most common value used, since in most cases a program knows when its own output is finished but not about when its conversation partner will be done. This value says that the caller will be writing no more data into the socket, and that reads from its other end should act like it is closed.
- `SHUT_RD`: This is used to turn off the incoming socket stream, so that an end-of-file error is encountered if your peer tries to send any more data to you on the socket.
- `SHUT_RDWR`: This closes communication in both directions on the socket. It might not, at first, seem useful, because you can also just perform a `close()` on the socket and communication is similarly ended in both directions. The difference is a rather advanced one: if several programs on your operating system are allowed to share a single socket, then `close()` just ends your process's relationship with the socket, but keeps it open as long as another process is still using it; but `shutdown()` will always immediately disable the socket for everyone using it.

Using TCP Streams like Files

Since TCP supports streams of data, they might have already reminded you of normal files, which also support reading and writing as fundamental operations. Python does a very good job of keeping these concepts separate: file objects can `read()` and `write()`, sockets can `send()` and `recv()`, and no kind of object can do both. But sometimes you will want to treat a socket like a normal Python file object—often because you want to pass it into code like that of the many Python modules such as `pickle`, `json`, and `zlib` that can read and write data directly from a file. For this purpose, Python provides a `makefile()` method on every socket that returns a Python file object that is really calling `recv()` and `send()` behind the scenes:

```
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> hasattr(s, 'read')
False
>>> f = s.makefile()
>>> hasattr(f, 'read')
True
```

Sockets, like normal Python files, also have a `fileno()` method that lets you discover their file descriptor number in case you need to supply it to lower-level calls.

Socket names and DNS

In this chapter, we will discuss the topic of network addresses and will describe the distributed service that allows names to be resolved to raw IP addresses.

Socket names

The last chapter has already introduced you to the fact that sockets cannot be named with a single primitive Python value like a number or string. Instead, both TCP and UDP use integer port numbers to share a single machine's IP address among the many different applications that might be running there, and so the address and port number have to be combined in order to produce a socket name, like this:

```
('18.9.22.69', 80)
```

You will recall that socket names are important at several points in the creation and use of sockets. For your reference, here are all of the major socket methods that demand of you some sort of socket name as an argument:

- `mysocket.accept()` : Each time this is called on a listening TCP stream socket that has incoming connections ready to hand off to the application, it returns a tuple(ordered set of values) whose second item is the remote address that has connected (the first item in the tuple is the net socket connected to that remote address).
- `mysocket.bind(address)` : Assigns the socket the local address so that outgoing packets have an address from which to originate, and so that any incoming connections from other machines have a name that they can use to connect.
- `mysocket.connect(address)` : Establishes that data sent through this socket will be directed to the given remote address. For UDP sockets, this simply sets the default address used if the caller uses `send()` rather than `sendto()` ; for TCP sockets, this actually negotiates a new stream with another machine using a three-way handshake, and raises an exception if the negotiation fails.
- `mysocket.getpeername()` : Returns the remote address to which this socket is connected.
- `mysocket.getsockname()` : Returns the address of this socket's own local endpoint.
- `mysocket.recvfrom(...)` : For UDP sockets, this returns a tuple that pairs a string of returned data with the address from which it was just sent.
- `mysocket.sendto(data, address)` : An unconnected UDP port uses this method to fire off a data packet at a particular remote address.

In general, any of the foregoing methods can receive or return any of the sorts of addresses that follow, meaning that they will work regardless of whether you are using IPv4, IPv6 or others.

Five socket coordinates

If you review previous code, you will notice that we have use:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('localhost', 1060))
```

We paid particular attention to the hostnames and IP addresses that their sockets used. But if you read each program listing from the beginning, you will see that these are only the last two coordinates of five major decisions that were made during the construction and deployment of each socket object. In order, here is the full list of values that had to be chosen, and you will see that there are five in all:

- First, the address family makes the biggest decision: it names what kind of network you want to talk to, out of the many kinds that a particular machine might support. We will always use the value `AF_INET`.
- Next after the address family comes the socket type. It chooses the particular kind of communication technique that you want to use on the network you have chosen. The socket interface designers decided to create more generic names for the broad idea of a packet-based socket, which goes by the name `SOCK_DGRAM`, and the broad idea of a reliable flowcontrolled data stream, which as we have seen is known as a `SOCK_STREAM`.
- The third field in the `socket()` call, the protocol, is rarely used because once you have specified the address family and socket type, you have narrowed down the possible protocols to one major option.
- The fourth and fifth fields are, then, the IP address and UDP or TCP port number that were explained in detail in the last chapters.

IPv6

And having explained all of that, it turns out that this book actually does need to introduce one additional address family, beyond the AF_INET we have used so far: the address family for IPv6, named AF_INET6, which is the way forward into a future where the world does not, in fact, run out of IP addresses.

In Python you can test directly for whether the underlying platform supports IPv6 by checking the `has_ipv6` Boolean attribute inside the socket module:

```
>>> import socket
>>> socket.has_ipv6
True
```

But note that this does not tell you whether an actual IPv6 interface is up and configured and can currently be used to send packets anywhere; it is purely an assertion about whether IPv6 support has been compiled into the operating system, not about whether it is in use.

The differences that IPv6 will make for your Python code might sound quite daunting, if listed one right after the other:

- Your sockets have to be prepared to have the family AF_INET6 if you are called upon to operate on an IPv6 network.
- No longer do socket names consist of just two pieces, an address and a port number; instead, they can also involve additional coordinates that provide “flow” information and a “scope” identifier.
- The pretty IPv4 octets like 18.9.22.69 that you might already be reading from configuration files or from your command-line arguments will now sometimes be replaced by IPv6 host addresses instead, which you might not even have good regular expressions for yet. They have lots of colons, they can involve hexadecimal numbers, and in general they look quite ugly.

The `getaddrinfo()` function

To make your code simple, powerful, and immune from the complexities of the transition from IPv4 to IPv6, you should turn your attention to one of the most powerful tools in the Python socket user's arsenal: `getaddrinfo()`. The `getaddrinfo()` function sits in the `socket` module along with most other operations that involve addresses (rather than being a socket method). Unless you are doing something specialized, it is probably the only routine that you will ever need to transform the hostnames and port numbers that your users specify into addresses that can be used by socket methods. Its approach is simple: rather than making you attack the addressing problem piecemeal, which is necessary when using the older routines in the `socket` module, it lets you specify everything you know about the connection that you need to make in a single call. In response, it returns all of the coordinates we discussed earlier that are necessary for you to create and connect a socket to the named destination.

If we visit [Python Official Documentation](#) we find this some interesting eplanations. First the syntaxis is the following:

```
socket.getaddrinfo(host, port[, family[, socktype[, proto[, flags]]]])
```

So what `getaddrinfo()` does is; translate the `host/port` argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. `host` is a domain name, a string representation of an IPv4/v6 address or `None`. `port` is a string service name such as `'http'`, a numeric port number or `None`. By passing `None` as the value of `host` and `port`, you can pass `NULL` to the underlying C API.

The function returns a list of 5-tuples with the following structure:

```
(family, socktype, proto, canonname, sockaddr)
```

In these tuples, `family`, `socktype`, `proto` are all integers and are meant to be passed to the `socket()` function. "canonname" will be a string representing the canonical name of the host if `AI_CANONNAME` is part of the `flags` argument; else `canonname` will be empty. "sockaddr" is a tuple describing a socket address, whose format depends on the returned `family` (a (address, port) 2-tuple for `AF_INET`, a (address, port, flow info, scope id) 4-tuple for `AF_INET6`), and is meant to be passed to the `socket.connect()` method.

Here you find a exapmle of use:

```
>>> import socket
>>> from pprint import pprint
>>> infolist = socket.getaddrinfo('gatech.edu', 'www')
>>> pprint(infolist)
[(2, 2, 17, '', ('130.207.160.173', 80)),
 (2, 1, 6, '', ('130.207.160.173', 80))]
>>>
>>> ftpca = infolist[0]
>>> ftpca[0:3]
(2, 2, 17)
>>> s = socket.socket(*ftpca[0:3])
>>> ftpca[4]
('130.207.160.173', 80)
>>> s.connect(ftpca[4])
>>>
```

`ftpca` here is an acronym for the order of the variables that are returned: "family, type, protocol, canonical name, and address," which contain everything you need to make a connection. Here, we have asked about the possible methods for connecting to the HTTP port of the host `gatech.edu`, and have been told that there are two ways to do it: by creating a `SOCK_STREAM` socket (socket type 1) that uses `IPPROTO_TCP` (protocol number 6) or else by using a `SOCK_DGRAM` (socket type 2) socket with `IPPROTO_UDP` (which is the protocol represented by the integer 17).

As you can see from the foregoing code snippet, `getaddrinfo()` generally allows not only the hostname but also the port name to be a symbol rather than an integer.

Asking getaddrinfo() Where to Bind

Before tackling all of the options that `getaddrinfo()` supports, it will be more useful to see how it is used to support three basic network operations. We will tackle them in the order that you might perform operations on a socket: binding, connecting, and then identifying a remote host who has sent you information.

```
>>> import socket
>>> from socket import getaddrinfo
>>> getaddrinfo(None, 'smtp', 0, socket.SOCK_STREAM, 0, socket.AI_PASSIVE)
[(2, 1, 6, '', ('0.0.0.0', 25)), (30, 1, 6, '', (':::', 25, 0, 0))]
>>> getaddrinfo(None, 53, 0, socket.SOCK_DGRAM, 0, socket.AI_PASSIVE)
[(2, 2, 17, '', ('0.0.0.0', 53)), (30, 2, 17, '', (':::', 53, 0, 0))]
>>>
```

Here we asked about where we should `bind()` a socket if we want to serve SMTP traffic using TCP, and if we want to serve DNS traffic using DCP, respectively. The answers we got back in each case are the appropriate wildcard addresses that will let us bind to every IPv4 and every IPv6 interface on the local machine with all of the right values for the socket family, socket type, and protocol in each case. If you instead want to `bind()` to a particular IP address that you know that the local machine holds, then omit the `AI_PASSIVE` flag and just specify the hostname. For example, here are two ways that you might try binding to localhost:

```
>>> getaddrinfo('127.0.0.1', 'smtp', 0, socket.SOCK_STREAM, 0)
[(2, 1, 6, '', ('127.0.0.1', 25))]
>>> getaddrinfo('localhost', 'smtp', 0, socket.SOCK_STREAM, 0)
[(30, 1, 6, '', (':::', 25, 0, 0)), (2, 1, 6, '', ('127.0.0.1', 25)), (30, 1, 6, '', ('fe80::1%lo0', 25, 0, 1))]
>>>
```

You can see that supplying the IPv4 address for the localhost locks you down to receiving connections only over IPv4, while using the symbolic name localhost (at least on a Linux laptop, with a well-configured `/etc/hosts` file) makes available both the IPv4 and IPv6 local names for the machine.

Asking getaddrinfo() About Services

The majority uses of `getaddrinfo()` are outward-looking, and generate information suitable for connecting you to other applications. In all such cases, you can either use an empty string to indicate that you want to connect back to the localhost using the loopback interface, or provide a string giving an IPv4 address, IPv6 address, or hostname to name your destination. The usual use of `getaddrinfo()` in all other cases—which, basically, is when you are preparing to `connect()` or `sendto()`—is to specify the `AI_ADDRCONFIG` flag, which filters out any addresses that are impossible for your computer to reach. For example, an organization might have both an IPv4 and an IPv6 range of IP addresses; but if your particular host supports only IPv4, then you will want the results filtered to include only addresses in that family. In case the local machine has only an IPv6 network interface but the service you are connecting to is supporting only IPv4, the `AI_V4MAPPED` will return you those IPv4 addresses re-encoded as IPv6 addresses that you can actually use. So you will usually use `getaddrinfo()` this way when connecting:

```
>>> import socket
>>> from socket import getaddrinfo
>>> getaddrinfo('ftp.kernel.org', 'ftp', 0, socket.SOCK_STREAM, 0, socket.AI_ADDRCONFIG | socket.AI_V4MAPPED)
[(2, 1, 6, '', ('199.204.44.194', 21)), (2, 1, 6, '', ('198.145.20.140', 21)), (2, 1, 6, '', ('149.20.4.69', 21))]
>>>
```

And we have gotten exactly what we wanted: every way to connect to a host named `ftp.kernel.org` through a TCP connection to its FTP port.

Here is another query, which describes how I can connect from my laptop to the HTTP interface of the IANA that assigns port numbers in the first place:

```
>>> getaddrinfo('iana.org', 'www', 0, socket.SOCK_STREAM, 0, socket.AI_ADDRCONFIG | socket.AI_V4MAPPED)
[(2, 1, 6, '', ('192.0.43.8', 80))]
>>>
```

If we take away our carefully chosen flags in the sixth parameter, then we will also be able to see their IPv6 address:

```
>>> getaddrinfo('iana.org', 'www', 0, socket.SOCK_STREAM, 0)
[(2, 1, 6, '', ('192.0.43.8', 80)), (30, 1, 6, '', ('2001:500:88:200::8', 80, 0, 0))]
>>>
```

Asking getaddrinfo() for Pretty Hostnames

One last circumstance that you will commonly encounter is where you either are making a new connection, or maybe have just received a connection to one of your own sockets, and you want an attractive hostname to display to the user or record in a log file. This is slightly dangerous because a hostname lookup can take quite a bit of time, even on the modern Internet, and might return a hostname that no longer works by the time you go and check your logs—so for log files, try to record both the hostname and raw IP address. But if you have a good use for the “canonical name” of a host, then try running `getaddrinfo()` with the `AI_CANONNAME` flag turned on, and the fourth item of any of the tuples that it returns—that were always empty strings in the foregoing examples, you will note—will contain the canonical name:

```
>>> import socket
>>> from socket import getaddrinfo
>>> getaddrinfo('iana.org', 'www', 0, socket.SOCK_STREAM, 0, socket.AI_ADDRCONFIG | socket.AI_V4MAPPED | socket.AI_CANONNAME)
[(2, 1, 6, 'iana.org', ('192.0.43.8', 80))]
```

Other getaddrinfo() Flags

The flags available vary somewhat by operating system, and you should always consult your own computer's documentation (not to mention its configuration) if you are confused about a value that it chooses to return. But there are several flags that tend to be cross-platform; here are some of the more important ones:

- `AI_ALL`: With `AI_V4MAPPED` option will save you in the situation where you are on a purely IPv6-connected host, but the host to which you want to connect advertises only IPv4 addresses: it resolves this problem by “mapping” the IPv4 addresses to their IPv6 equivalent. But if some IPv6 addresses do happen to be available, then they will be the only ones shown. Thus the existence of this option: if you want to see all of the addresses from your IPv6-connected host, even though some perfectly good IPv6 addresses are available, then combine this `AI_ALL` flag with `AI_V4MAPPED` and the list returned to you will have every address known for the target host.
- `AI_NUMERICHOST`: This turns off any attempt to interpret the hostname parameter (the first parameter to `getaddrinfo()`) as a textual hostname like `cern.ch`, and only tries to interpret the hostname string as a literal IPv4 or IPv6 hostname like `74.207.234.78` or `fe80::fcfd:4aff:fecf:ea4e`. This is much faster, as no DNS round-trip is incurred (see the next section), and prevents possibly untrusted user input from forcing your system to issue a query to a nameserver under someone else's control.
- `AI_NUMERICSERV`: This turns off symbolic port names like `www` and insists that port numbers like `80` be used instead. This does not necessarily have the networkquery implications of the previous option, since port-number databases are typically stored locally on IP-connected machines; on POSIX systems, resolving a symbolic port name typically requires only a quick scan of the `/etc/services` file (but check your `/etc/nsswitch.conf` file's `services` option to be sure). But if you know your port string should always be an integer, then activating this flag can be a useful sanity check.

getaddrinfo() in your own code

Here you have a quick example of how `getaddrinfo()` looks in actual code in `www_ping.py` .

```
import socket, sys

if len(sys.argv) != 2:
    print >>sys.stderr, 'usage: www_ping.py <hostname_or_ip>'
    sys.exit(2)

hostname_or_ip = sys.argv[1]

try:
    infolist = socket.getaddrinfo(
        hostname_or_ip, 'www', 0, socket.SOCK_STREAM, 0,
        socket.AI_ADDRCONFIG | socket.AI_V4MAPPED | socket.AI_CANONNAME,
    )
except socket.gaierror, e:
    print 'Name service failure:', e.args[1]
    sys.exit(1)

info = infolist[0] # per standard recommendation, try the first one
socket_args = info[0:3]
address = info[4]
s = socket.socket(*socket_args)
try:
    s.connect(address)
except socket.error, e:
    print 'Network failure:', e.args[1]
else:
    print 'Success: host', info[3], 'is listening on port 80'
```

It performs a simple are-you-there test of whatever web server you name on the command line by

attempting a quick connection to port 80 with a streaming socket. Using the script would look something like this:

```
root@erlerobot:~/Python_files#
root@erlerobot:~/Python_files# python www_ping.py mit.edu
Success: host mit.edu is listening on port 80
root@erlerobot:~/Python_files# python www_ping.py smtp.google.com
Name service failure: nodename nor servname provided, or not known
root@erlerobot:~/Python_files# www_ping.py no-such-host.com
Name service failure: nodename nor servname provided, or not known
root@erlerobot:~/Python_files#
```

Note that the `socket()` constructor does not take a list of three items as its parameter. Instead, the parameter list is introduced by an asterisk, which means that the three elements of the `socket_args` list are passed as three separate parameters to the constructor.

A Sketch of How DNS Works

The DNS Protocol purpose is to turn hostnames into IP addresses.

For example, consider the domain name `www.python.org`. If your web browser needs to know this address, then the browser runs a call like `getaddrinfo()` to ask the operating system to resolve that name. Your system will know either that it is running a nameserver of its own, or that the network to which it is attached provides name service. So, the first act of your DNS server will be to check its own cache of recently queried domain names to see if `www.python.org` has already been checked by some other machine served by the DNS server in the last few minutes or hours. If an entry is present and has not yet expired—and the owner of each domain name gets to choose its expiration timeout, because some organizations like to change IP addresses quickly if they need to, while others are happy to have old IP addresses linger for hours or days in the world's DNS caches—then it can be returned immediately. But let us imagine that it is morning and that you are the first person in your office or in the coffee shop to try talking to `www.python.org` today, and so the DNS server has to go find the hostname from scratch. Your DNS server will now begin a recursive process of asking about `www.python.org` at the very top of the world's DNS server hierarchy: the “root-level” nameservers that know all of the top-level domains (TLDs) like `.com`, `.org`, `.net`, and all of the country domains, and know the groups of servers that are responsible for each. Nameserver software generally comes with the IP addresses of these top-level servers built in, to solve the bootstrapping problem of how you find any domain nameservers before you are actually connected to the domain name system. With this first UDP round-trip, your DNS server will learn (if it did not know already from another recent query) which servers keep the full index of `.org` domain.

Now a second DNS request will be made, this time to one of the `.org` servers, asking who on earth runs the `python.org` domain. You can find out what those top-level servers know about a domain by running the `whois` command-line program on a POSIX system, or use one of the many “whois” web pages online, typing:

```
whois python.org
```

Wherever you are in the world, your DNS request for any hostname within `python.org` must be passed on to one of the two DNS servers named in that entry.

There are some reasons to **not use DNS**, and use `getaddrinfo()` or some other system-supported mechanism for resolving hostnames.

The DNS is often not the only way that a system gets name information.

- If your application runs off and tries to use DNS on its own as its first choice for resolving a domain name, then users will notice that some computer names that work everywhere else on your system—in their browser, in file share names, and so forth—suddenly do not work when they use your application, because you are not deferring to mechanisms like WINS or `/etc/hosts` like the operating system itself does.
- The local machine probably has a cache of recently queried domain names that might already know about the host whose IP address you need. If you try speaking DNS yourself to answer your query, you will be duplicating work that has already been done.
- The system on which your Python script is running already knows about the local domain nameservers, thanks either to manual intervention by your system administrator or a network configuration protocol like DHCP in your office, home, or coffee shop. To crank up DNS right inside your Python program, you will have to learn how to query your particular operating system for this information—an operating-system-specific action that we will not be covering in this book.
- If you do not use the local DNS server, then you will not be able to benefit from its own cache that would prevent your application and other applications running on the same network from repeating requests about a hostname that is in frequent use at your location.

- From time to time, adjustments are made to the world DNS infrastructure, and operating system libraries and daemons are gradually updated to accommodate this. If your program makes raw DNS calls of its own, then you will have to follow these changes yourself and make sure that your code stays up-to-date with the latest changes in TLD server IP addresses, conventions involving internationalization, and tweaks to the DNS protocol itself.

There is, however, a solid and legitimate reason to make a DNS call from Python: because you are a mail server, or at the very least a client trying to send mail directly to your recipients without needing to run a local mail relay, and you want to look up the MX records associated with a domain so that you can find the correct mail server for your friends at @example.com.

Using DNS

PyDNS provides a module for performing DNS queries from python applications. You can install it by:

```
pip install pydns
```

Your Python interpreter will then gain the ability to run our first DNS program listing, shown in `dns_basic.py`.

```
import sys, DNS

if len(sys.argv) != 2:
    print >>sys.stderr, 'usage: dns_basic.py <hostname>'
    sys.exit(2)

DNS.DiscoverNameServers()
request = DNS.Request()
for qt in DNS.Type.A, DNS.Type.AAAA, DNS.Type.CNAME, DNS.Type.MX, DNS.Type.NS:
    reply = request.req(name=sys.argv[1], qtype=qt)
    for answer in reply.answers:
        print answer['name'], answer['classstr'], answer['typename'], \
            repr(answer['data'])
```

Running this program will result on:

```
root@erlerobot:~/Python_files# dns_basic.py python.org
python.org IN A '82.94.164.162'
python.org IN AAAA '\x01\x08\x88 \x00\x00\r\x00\x00\x00\x00\x00\x00\xa2'
python.org IN MX (50, 'mail.python.org')
python.org IN NS 'ns2.xs4all.nl'
python.org IN NS 'ns.xs4all.nl'
```

The keys that get printed on each line are as follows:

- The name that we looked up.
- The “class,” which in all queries you are likely to see is IN, meaning it is a question about Internet addresses.
- The “type” of record; some common ones are A for an IPv4 address, AAAA for an IPv6 address, NS for a record that lists a nameserver, and MX for a statement about what mail server should be used for a domain.
- Finally, the “data” provides the information for which the record type was essentially a promise: the address, or data, or hostname associated with the name that we asked about.

Network Data and Network Errors

What data should we send? How should it be encoded and formatted? For what kinds of errors will our Python programs need to be prepared? We will look at the basic answers in this chapter, and learn how to use sockets responsibly so that our data arrives intact.

Text and Encodings

The use of ASCII for the basic English letters and numbers is nearly universal among network protocols these days. But when you begin to use more interesting characters, you have to be careful. In Python you should always represent a meaningful string of text with a “Unicode string” that is denoted with a leading u, like this:

```
>>> elvish = u'Namárië!'
```

But you cannot put such strings directly on a network connection without specifying which rival system of encoding you want to use to mix your characters down to bytes. A very popular system is UTF-8, because normal characters are represented by the same codes as in ASCII, and longer sequences of bytes are necessary only for international characters. Other encodings are available in Python; [the Standard Library documentation for the codecs package](#) lists them all. They each represent a full system for reducing symbols to bytes. Here are a few examples:

```
>>> elvish.encode('idna')
'xn--namri!-rta6f'
>>> elvish.encode('cp500')
'\xd5\x81\x94E\x99\x89S0'
>>> elvish.encode('utf_8_sig')
'\xef\xbb\xbfNam\xc3\xa1ri\xc3\xab!'
```

On the receiving end of such a string, simply take the byte string and call its decode() method with the name of the codec that was used to encode it:

```
>>> 'xn--namri!-rta6f'.decode('idna')
u'Nam\xe1ri\xeb!'
>>> '\xd5\x81\x94E\x99\x89S0'.decode('cp500')
u'Nam\xe1ri\xeb!'
>>> '\xef\xbb\xbfNam\xc3\xa1ri\xc3\xab!'.decode('utf_8_sig')
u'Nam\xe1ri\xeb!'
```

Network Byte Order

To understand the issue of byte order, consider the process of sending an integer over the network. To be specific, think about the integer 4253.

Many protocols, of course, will simply transmit this integer as the string '4253'—that is, as four distinct characters. The four digits will require at least four bytes to transmit, at least in any common text encoding. And using decimal digits will also involve some computational expense: since numbers are not stored inside computers in base 10, it will take repeated division—with inspection of the remainder—to determine that this number is in fact made of 4 thousands, plus 2 hundreds, plus 5 tens, plus 3 left over. And when the four-digit string '4253' is received, repeated addition and multiplication by powers of ten will be necessary to put the text back together into a number.

In any case, the string '4253' is not how your computer represents this number as an integer variable in Python. Instead it will store it as a binary number, using the bits of several successive bytes to represent the one's place, two's place, four's place, and so forth of a single large number. We can glimpse the way that the integer is stored by using the `hex()` built-in function at the Python prompt:

```
>>> hex(4253)
'0x109d'
```

Each hex digit corresponds to four bits, so each pair of hex digits represents a byte of data. Instead of being stored as four decimal digits 4, 4, 2, and 3 with the first 4 being the “most significant” digit (since tweaking its value would throw the number off by a thousand) and 3 being its least significant digit, the number is stored as a most significant byte 0x10 and a least significant byte 0x9d, adjacent to one another in memory.

Here we reach a great difference between computers. While they will all agree that the bytes in memory have an order, and they will all store a string like `Content-Length: 4253` in exactly that order starting with C and ending with 3, they do not share a single idea about the order in which the bytes of a binary number should be stored. Some computers are “big-endian” and put the most significant byte first; others are “little-endian” and put the least significant byte first.

Python makes it very easy to see the difference between the two endiannesses. Simply use the `struct` module, which provides a variety of operations for converting data to and from popular binary formats. Here is the number 4253 represented first in a little-endian format and then in a big-endian order:

```
>>> import struct
>>> struct.pack('<i', 4253)
'\x9d\x10\x00\x00'
>>> struct.pack('>i', 4253)
'\x00\x00\x10\x9d'
```

`struct` module performs conversions between Python values and C structs represented as Python strings. You can read more [here](#). We here used the code `i`, which uses four bytes to store an integer, so the two upper bytes are zero for a small number like 4253. It also supports an `unpack()` operation, which converts the binary data back to Python numbers:

```
>>> struct.unpack('>i', '\x00\x00\x10\x9d')
(4253,)
```

Therefore the `struct` module provides another symbol, `'!'`, which means the same thing as `'>'` when used in `pack()` and `unpack()` but says to other programmers (and, of course, to yourself as you read the code later), “I am packing this data so that I can send it over the network.”

Framing and Quoting

If you have made the far more common option of using a TCP stream for communication, then you will face the issue of framing, that is, the issue of how to delimit your messages so that the receiver can tell where one message ends and the next begins.

There is a **first pattern (streaming)** that can be used by extremely simple network protocols that involve only the delivery of data—no response is expected, so there never has to come a time when the receiver decides “Enough!” and turns around to send a response. In this case, the sender can loop until all of the outgoing data has been passed to `sendall()` and then `close()` the socket. The receiver need only call `recv()` repeatedly until the call finally returns an empty string, indicating that the sender has finally closed the socket. You can see this pattern in `streamer.py` :

```
import socket, sys
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

HOST = sys.argv.pop() if len(sys.argv) == 3 else '127.0.0.1'
PORT = 1060

if sys.argv[1:] == ['server']:
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((HOST, PORT))
    s.listen(1)
    print 'Listening at', s.getsockname()
    sc, sockname = s.accept()
    print 'Accepted connection from', sockname
    sc.shutdown(socket.SHUT_WR)
    message = ''
    while True:
        more = sc.recv(8192) # arbitrary value of 8k
        if not more: # socket has closed when recv() returns ''
            break
        message += more
    print 'Done receiving the message; it says:'
    print message
    sc.close()
    s.close()

elif sys.argv[1:] == ['client']:
    s.connect((HOST, PORT))
    s.shutdown(socket.SHUT_RD)
    s.sendall('Beautiful is better than ugly.\n')
    s.sendall('Explicit is better than implicit.\n')
    s.sendall('Simple is better than complex.\n')
    s.close()

else:
    print >>sys.stderr, 'usage: streamer.py server|client [host]'
```

If you run this script as a server and then, at another command prompt, run the client version, you

will see that all of the client's data makes it intact to the server, with the end-of-file event generated by the client closing the socket serving as the only framing that is necessary:

```
root@erlerobot:~/Python_files# python streamer.py server
Listening at ('127.0.0.1', 1060)
Accepted connection from ('127.0.0.1', 49592)
Done receiving the message; it says:
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
```

There is a **second pattern** is a variant on the first: **streaming in both directions**. The socket is initially left open in both directions. First, data is streamed in one direction—exactly and then that direction alone is shut down. Second, data is then

streamed in the other direction, and the socket is finally closed.

A **third pattern**, which we have already seen, is to use fixed-length messages, as illustrated in `tcp_sixteen.py`. You can use the Python `sendall()` method to keep sending parts of a string until the whole thing has been transmitted, and then use a `recv()` loop of our own devising to make sure that you receive the whole message.

A **fourth pattern** is to somehow delimit your messages with special characters. The receiver would wait in a `recv()` loop like the one just cited, but wait until the reply string it was accumulating finally contained the delimiter indicating the end-of-message.

A **fifth pattern** is to prefix each message with its length. This is a very popular choice for highperformance protocols since blocks of binary data can be sent verbatim without having to be analyzed, quoted, or interpolated. Of course, the length itself has to be framed using one of the techniques given previously—often it is simply a fixed-width binary integer, or else a variable-length decimal string followed by a delimiter. But either way, once the length has been read and decoded, the receiver can enter a loop and call `recv()` repeatedly until the whole message has arrived.

There is sixth pattern for which the unknown lengths are no problem. Instead of sending just one, try sending several blocks of data that are each prefixed with their length. This means that as each chunk of new information becomes available to the sender, it can be labeled with its length and placed on the outgoing stream. When the end finally arrives, the sender can emit an agreed-upon signal—perhaps a length field giving the number zero—that tells the receiver that the series of blocks is complete.

Following (`blocks.py`) you can find an example of this sixth pattern. Like the previous one, this sends data in only one direction—from the client to the server—but the data structure is much more interesting. Each message is prefixed with a 4-byte length; in a struct, 'I' means a 32-bit unsigned integer, meaning that these messages can be up to 4GB in length. A series of three such messages is sent to the server, followed by a zero-length message—which is essentially just a length field with zeros inside and then no message data after it—to signal that the series of blocks is over.

```
import socket, struct, sys
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

HOST = sys.argv.pop() if len(sys.argv) == 3 else '127.0.0.1'
PORT = 1060
format = struct.Struct('!I') # for messages up to 2**32 - 1 in length

def recvall(sock, length):
    data = ''
    while len(data) < length:
        more = sock.recv(length - len(data))
        if not more:
            raise EOFError('socket closed %d bytes into a %d-byte message'
                            % (len(data), length))
        data += more
    return data

def get(sock):
    lendata = recvall(sock, format.size)
    (length,) = format.unpack(lendata)
    return recvall(sock, length)

def put(sock, message):
    sock.send(format.pack(len(message)) + message)

if sys.argv[1:] == ['server']:
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((HOST, PORT))
    s.listen(1)
    print 'Listening at', s.getsockname()
    sc, sockname = s.accept()
    print 'Accepted connection from', sockname
    sc.shutdown(socket.SHUT_WR)
    while True:
        message = get(sc)
        if not message:
            break
        print 'Message says:', repr(message)
    sc.close()
    s.close()
```

```
elif sys.argv[1:] == ['client']:
    s.connect((HOST, PORT))
    s.shutdown(socket.SHUT_RD)
    put(s, 'Beautiful is better than ugly.')
    put(s, 'Explicit is better than implicit.')
    put(s, 'Simple is better than complex.')
    put(s, '')
    s.close()

else:
    print >>sys.stderr, 'usage: streamer.py server|client [host]'
```

Running first the server and then the client in different terminals, resulto on:

```
root@erlerobot:~/Python_files# python blocks.py server
Listening at ('127.0.0.1', 1060)
Accepted connection from ('127.0.0.1', 49692)
Message says: 'Beautiful is better than ugly.'
Message says: 'Explicit is better than implicit.'
Message says: 'Simple is better than complex.'
root@erlerobot:~/Python_files#
```

Pickles and Self-Delimiting Formats

Note that some kinds of data that you might send across the network already include some form of delimiting built-in. If you are transmitting such data, then you might not have to impose your own framing atop what the data is already doing. Consider Python “pickles” for example, the native form of serialization that comes with the Standard Library. The [pickle module](#) implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream is converted back into an object hierarchy. Moreover, using a quirky mix of text commands and data, a pickle stores the contents of a Python data structure so that you can reconstruct it later or on a different machine:

```
>>> import pickle
>>> pickle.dumps([5, 6, 7])
'(lp0\nI5\naI6\naI7\na.'
```

The interesting thing about the format is the '.' character that you see at the end of the foregoing string—it is the format's way of marking the end of a pickle. Upon encountering it, the loader can stop and return the value without reading any further. Thus we can take the foregoing pickle, stick some ugly data on the end, and see that `loads()` will completely ignore it and give us our original list back:

```
>>> pickle.loads('(lp0\nI5\naI6\naI7\na.UjJGdVpHRnNaZz09')
[5, 6, 7]
```

Of course, using `loads()` this way is not useful for network data, since it does not tell us how many bytes it processed in order to reload the pickle; we still do not know how much of our string is pickle data. But if we switch to reading from a file and using the pickle `load()` function, then the file pointer will be left right at the end of the pickle data, and we can start reading from there if we want to read what came after the pickle:

```
>>> from StringIO import StringIO
>>> f = StringIO('(lp0\nI5\naI6\naI7\na.UjJGdVpHRnNaZz09')
>>> pickle.load(f)
[5, 6, 7]
>>> f.pos
18
>>> f.read()
'UjJGdVpHRnNaZz09'
```


XML, JSON, Etc.

If your protocol needs to be usable from other programming languages—or if you simply prefer universal standards to formats specific to Python—then the JSON and XML data formats are each a popular choice. Note that neither of these formats supports framing, so you will have to first figure out how to extract a complete string of text from over the network before you can then process it.

JSON is among the best choices available today for sending data between different computer languages. Since Python 2.6, it has been included in the Standard Library as a [module named `json`](#). JSON, short for JavaScript Object Notation, is a lightweight format for data exchange. JSON is a subset of the object literal notation JavaScript that does not require the use of XML. For encoding basic Python object hierarchies:

```
>>> #The syntax is:
...
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> #Example:
...
>>> json.dumps([ 51, u'Namárië!' ])
'[51, "Nam\\u00e1rië\\u00eb!"]'
```

For decoding it you should use:

```
>>> #The syntax is:
...
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> #An example:
...
>>> json.loads('{"name": "Lancelot", "quest": "Grail"}')
{'quest': 'Grail', 'name': 'Lancelot'}
```

Note that the protocol fully supports Unicode strings. It does, however, have a weakness: a vast omission in the JSON standard is that it provides absolutely no provision for cleanly passing binary data like images or arbitrary documents. The XML format is better for documents, since its basic structure is to take strings and mark them up by wrapping them in angle-bracketed elements.

Compression

Since the time necessary to transmit data over the network is often more significant than the time your CPU spends preparing the data for transmission, it is often worthwhile to compress data before sending it. The popular HTTP protocol lets a client and server figure out whether they can both support compression.

An interesting fact about the most ubiquitous form of compression, the GNU [zlib facility](#) (For applications that require data compression, the functions in this module allow compression and decompression, using the zlib library) that is available through the Python Standard Library, is that it is self-framing. If you start feeding it a compressed stream of data, then it can tell you when the compressed data has ended and further, uncompressed data has arrived past its end.

Most protocols choose to do their own framing and then, if desired, pass the resulting block to zlib for decompression. But you could conceivably promise yourself that you would always tack a bit of uncompressed data onto the end of each zlib compressed string—here, we will use a single '.' byte— and watch for your compression object to split out that “extra data” as the signal that you are done. Consider this combination of two compressed data streams:

```
>>> import zlib
>>> data = zlib.compress('sparse') + '.' + zlib.compress('flat') + '.'
>>> data
'\x9c+.H,*N\x05\x00\t\r\x02\x8f.x\x9cK\xcbI,\x01\x00\x04\x16\x01\xa8.'
>>> len(data)
28
```

Imagine that these 28 bytes arrive at their destination in 8-byte packets. After processing the first packet, we will find the decompression object's `unused_data` slot still empty, which tells us that there is still more data coming, so we would `recv()` on our socket again:

```
>>> dobj = zlib.decompressobj()
>>> dobj.decompress(data[0:8]), dobj.unused_data
('spars', '')
```

But the second block of eight characters, when fed to our decompress object, both finishes out the compressed data we were waiting for (since the final 'e' completes the string 'sparse') and also finally has a non-empty `unused_data` value that shows us that we finally received our '.' byte:

```
>>> dobj.decompress(data[8:16]), dobj.unused_data
('e', '.x')
```

If another stream of compressed data is coming, then we have to provide everything past the '.'— in this case, the character 'x'—to our new decompress object, then start feeding it the remaining “packets”:

```
>>> dobj2 = zlib.decompressobj()
>>> dobj2.decompress('x'), dobj2.unused_data
('', '')
>>> dobj2.decompress(data[16:24]), dobj2.unused_data
('flat', '')
>>> dobj2.decompress(data[24:]), dobj2.unused_data
('', '.')
```

At this point, `unused_data` is again non-empty, meaning that we have read past the end of this second bout of compressed data and can examine its content.

Network Exceptions

Depending on the protocol implementation that you are using, you might have to deal only with exceptions specific to that protocol, or you might have to deal with both protocol-specific exceptions and with raw socket errors as well.

The exceptions that are specific to socket operations are:

- `socket.gaierror`: This exception is raised when `getaddrinfo()` cannot find a name or service that you ask about—hence the letters G, A, and I in its name.

```
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> s.connect(('nonexistent.hostname.foo.bar', 80))
Traceback (most recent call last):
...
gaierror: [Errno -5] No address associated with hostname
```

- `socket.error`: This is the workhorse of the socket module, and will be raised for nearly every failure that can happen at any stage in a network transmission.
- `socket.timeout`: This exception is raised only if you, or a library that you are using, decides to set a timeout on a socket rather than wait forever for a `send()` or `recv()` to complete. It indicates that the timeout was reached before the operation could complete normally.

Handling Exceptions

There are four basic approaches of handling the errors that can occur.

- The first is not to handle exceptions at all. If only you or only other Python programmers will be using your script, then they will probably not be fazed by seeing an exception. If you are writing a library of calls to be used by other programmers, then this first approach is usually preferable, since by letting the exception through you give the programmer using your API the chance to decide how to present errors to his or her users.
- If you are indeed writing a library, then there is a second approach to consider: wrapping the network errors in an exception of your own.
- A third approach to exceptions is to wrap a `try...except` clause around every single network call that you ever make, and print out a pithy error message in its place. While suitable for short programs, this can become very repetitive when long programs are involved, without necessarily providing that much more information for the user.
- There is one final reason that might dictate where you add an exception handler to your network program: you might want to intelligently re-try an operation that failed.

TLS and SSL

Before you send sensitive data across a network, you need proof of the identity of the machine that you think is on the other end of the socket, and while sending the data, you need it protected against the prying eyes of anyone controlling the gateways and network switches that see all of your packets. The solution to this problem is to use Transport Layer Security (TLS). Because earlier versions of TLS were called the Secure Sockets Layer (SSL), nearly all of the libraries that you will use to speak TLS actually still have SSL somewhere in the name.

Cleartext on the Network

There are several security problems that TLS is designed to solve. They are best understood by considering the dangers of sending your network data as “cleartext” over a plain old socket, which copies your data byte-for-byte into the packets that get sent over the network.

What are the consequences of someone who can now observe, capture, and analyze your data at his leisure?

- He can see all of the data that passes over that segment of the network. The fraction of your data that he can capture depends on how much of it passes over that particular link.
- He will see any usernames and passwords that your clients use to connect to the servers behind them.
- Log messages can also be intercepted, if they are being sent to a central location and happen to travel over a compromised IP segment or device. This could be very useful if the observer wants to probe for vulnerabilities in your software.
- If your database server is not picky about who connects, aside from caring that the web front end sends a password, then the attacker can now launch a “replay attack,” in which he makes his own connection to your database and downloads all of the data that a front-end server is normally allowed to access.

Imagine an attacker who cannot yet alter traffic on your network itself, but who can compromise one of the services around the edges that help your servers find each other. Specifically, what if she can compromise the DNS service that lets your web front ends find your db.example.com server. Then some interesting tricks might become possible:

- When your front ends ask for the hostname db.example.com, she could answer with the IP address of her own server, located anywhere in the world, instead.
- The fake database server will be at a loss to answer requests with any real data that the intruder has not already copied down off the network.
- If your database is not carefully locked down and so is not picky about which servers connect, then the attacker can do something more interesting: as requests start arriving at her fake database server, he can have it turn around and forward those requests to the real database server. This is called a “man-in-the-middle” attack: he will be in fairly complete control of your application.
- While proxying the client requests through to the database, the attacker will probably also have the option of inserting queries of her own into the request stream. This could let her download entire tables of data and delete or change whatever data the front-end services are typically allowed to modify.

TLS Encrypts Your Conversations

The secret to TLS is public-key cryptography. There are several mathematical schemes that have been proved able to support public-key schemes, but they all have these three features:

- Anyone can generate a key pair, consisting of a private key that they keep to themselves and a public key that they can broadcast however they want.
- If the public key is used to encrypt information, then the resulting block of binary data cannot be read by anyone, anywhere in the world, except by someone who holds the private key.
- If the system that holds the private key uses it to encrypt information, then any copy of the public key can be used to decrypt the data.

We will focus on how public keys are used in the TLS system: Public keys are used at two different levels within TLS: first, to establish a certificate authority (CA) system that lets servers prove “who they really are” to the clients that want to connect; and, second, to help a particular client and server communicate securely.

Supporting TLS in Python

From the point of view of your network program, you start a TLS connection by turning control of a socket over to an SSL library. By doing so, you indicate that you want to stop using the socket for cleartext communication, and start using it for encrypted data under the control of the library.

From that point on, you no longer use the raw socket; doing so will cause an error and break the connection. Instead, you will use routines provided by the library to perform all communication. Both client and server should turn their sockets over to SSL at the same time, after reading all pending data off of the socket in both directions. There are two general approaches to using SSL:

- The most straightforward option is probably to use the [ssl package](#) that recent versions of Python ship with the Standard Library.
- The other alternative is to use a third-party Python library. There are several of these that support TLS, but many of them are decrepit and seem to have been abandoned. For example M2Crypto package.

The Standard SSL Module

Here you can find an example of the use of TLS. The first and last few lines of this file `sslclient.py` look completely normal: opening a socket to a remote server, and then sending and receiving data per the protocol that the server supports. The cryptographic protection is invoked by the few lines of code in the middle—two lines that load a certificate database and make the TLS connection itself, and then the call to `match_hostname()` that performs the crucial test of whether we are really talking to the intended server or perhaps to an impersonator.

```
import os, socket, ssl, sys
from backports.ssl_match_hostname import match_hostname, CertificateError

try:
    script_name, hostname = sys.argv
except ValueError:
    print >>sys.stderr, 'usage: sslclient.py <hostname>'
    sys.exit(2)

# First we connect, as usual, with a socket.

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((hostname, 443))

# Next, we turn the socket over to the SSL library!

ca_certs_path = os.path.join(os.path.dirname(script_name), 'certfiles.crt')
sslsock = ssl.wrap_socket(sock, ssl_version=ssl.PROTOCOL_SSLv3,
                          cert_reqs=ssl.CERT_REQUIRED, ca_certs=ca_certs_path)

# Does the certificate that the server proffered *really* match the
# hostname to which we are trying to connect? We need to check.

try:
    match_hostname(sslsock.getpeercert(), hostname)
except CertificateError, ce:
    print 'Certificate error:', str(ce)
    sys.exit(1)

# From here on, our `sslsock` works like a normal socket. We can, for
# example, make an impromptu HTTP call.

sslsock.sendall('GET / HTTP/1.0\r\n\r\n')
result = sslsock.makefile().read() # quick way to read until EOF
sslsock.close()
print 'The document https://%s/ is %d bytes long' % (hostname, len(result))
```

Note that the certificate database needs to be provided as a file named `certfiles.crt` in the same directory as the script.

```
root@erlerobot:~/Python_files# cat /etc/ssl/certs/* > certfiles.crt
```

```
root@erlerobot:~/Python_files# sslclient.py www.openssl.org
The document https://www.openssl.org/ is 15941 bytes long
```

Server Architecture

This chapter explores how network programming intersects with the general tools and techniques that Python developers use to write long-running daemons that can perform significant amounts of work by keeping a computer and its processors busy.

Daemons and Logging

A daemon is a computer program that runs as a background process, rather than being under the direct control of an interactive user. You can also install [python-daemon](#) from the Package, and its code will let your server program become a daemon entirely on its own power.

Another useful thing is the modern logging module, which can write to syslog, files, network sockets, or anything in between. The simplest pattern is to place something like this at the top of each of your daemon's source files:

```
import logging
log = logging.getLogger(__name__)
```

Then your code can generate messages very simply:

```
log.error('the system is down')
```

Introductory example

In this minimalist protocol `lancelot.py`, the client opens a socket, sends across one of the three questions asked of Sir Launcelot at the Bridge of Death in Monty Python's Holy Grail movie, and then terminates the message with a question mark: `What is your name?` The server replies by sending back the appropriate answer, which always ends with a period: `My name is Sir Launcelot of Camelot.` Both question and answer are encoded as ASCII.

```
import socket, sys

PORT = 1060
qa = (('What is your name?', 'My name is Sir Lancelot of Camelot.'),
      ('What is your quest?', 'To seek the Holy Grail.'),
      ('What is your favorite color?', 'Blue.))
qadict = dict(qa)

def recv_until(sock, suffix):
    message = ''
    while not message.endswith(suffix):
        data = sock.recv(4096)
        if not data:
            raise EOFError('socket closed before we saw %r' % suffix)
        message += data
    return message

def setup():
    if len(sys.argv) != 2:
        print >>sys.stderr, 'usage: %s interface' % sys.argv[0]
        exit(2)
    interface = sys.argv[1]
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind((interface, PORT))
    sock.listen(128)
    print 'Ready and listening at %r port %d' % (interface, PORT)
    return sock
```

The server code is `server_simple.py`:

```
import lancelot

def handle_client(client_sock):
    try:
        while True:
            question = lancelot.recv_until(client_sock, '?')
            answer = lancelot.qadict[question]
            client_sock.sendall(answer)
    except EOFError:
        client_sock.close()

def server_loop(listen_sock):
    while True:
        client_sock, sockname = listen_sock.accept()
        handle_client(client_sock)

if __name__ == '__main__':
    listen_sock = lancelot.setup()
    server_loop(listen_sock)
```

Anyway, this simple server has terrible performance characteristics. The difficulty comes when many clients all want to connect at the same time. The first client's socket will be returned by `accept()`, and the server will enter the `handle_client()` loop to start answering that first client's questions. But while the questions and answers are trundling back and forth across the network, all of the other clients are forced to queue up.

Elementary client

We will tackle the deficiencies of the simple server shown in `server_simple.py` in two discussions. First, in this section, we will discuss how much time it spends waiting even on one client that needs to ask several questions; and in the next section, we will look at how it behaves when confronted with many clients at once. A simple client for the Launcelot protocol connects, asks each of the three questions once, and then disconnects. The code of `client.py` is the following:

```
import socket, sys, lancebot

def client(hostname, port):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((hostname, port))
    s.sendall(lancebot.qa[0][0])
    answer1 = lancebot.recv_until(s, '.') # answers end with '.'
    s.sendall(lancebot.qa[1][0])
    answer2 = lancebot.recv_until(s, '.')
    s.sendall(lancebot.qa[2][0])
    answer3 = lancebot.recv_until(s, '.')
    s.close()
    print answer1
    print answer2
    print answer3

if __name__ == '__main__':
    if not 2 <= len(sys.argv) <= 3:
        print >>sys.stderr, 'usage: client.py hostname [port]'
        sys.exit(2)
    port = int(sys.argv[2]) if len(sys.argv) > 2 else lancebot.PORT
    client(sys.argv[1], port)
```

With these two scripts in place, we can start running our server in one console window:

```
...
root@erlerobot:~/Python_files# python server_simple.py localhost
Ready and listening at 'localhost' port 1060
...
```

We can then run our client in another window, and see the three answers returned by the server:

```
root@erlerobot:~/Python_files# python client.py localhost
My name is Sir Lancelot of Camelot.
To seek the Holy Grail.
Blue.
```

The client and server run very quickly here on my laptop. But appearances are deceiving, so we had better approach this client-server interaction more scientifically by bringing real measurements to bear upon its activity.

The solution for measuring the real waiting time when running the client and server on a single machine, but to send the connection through a round-trip to another machine by way of an SSH tunnel.

When doing this you will notice how the cost of communication dominates the performance. It will always seem to take less than 10 μ s for the server to run the answer = line and retrieve the response that corresponds to a particular question. If actually generating the answer were the client's only job, then we could expect it to serve more than 100,000 client requests per second way of an SSH tunnel. But look at all of the time that the client and server spend waiting for the network: every time one of them finishes a `sendall()` call, it takes between 500 μ s and 800 μ s before the other conversation partner is released from its `recv()` call and can proceed.

Now on, we may need a system for comparing the subsequent server designs that we explore. We are therefore going to turn now to a public tool: the [FunkLoad tool](#), written in Python and available from the Python Package Index.

```
root@erlerobot:~/Python_files# pip install funkload
```

Event-Driven Servers

The simple server we have been examining has the problem that the `recv()` call often finds that no data is yet available from the client, so the call “blocks” until data arrives. The time spent waiting, as we have seen, is time lost; it cannot be spent usefully by the server to answer requests from other clients.

But what if we avoided ever calling `recv()` until we knew that data had arrived from a particular client. The result would be an event-driven server that sits in a tight loop watching many clients; I have written an example, shown in `server_poll`.

```
import lancelot
import select

listen_sock = lancelot.setup()
sockets = { listen_sock.fileno(): listen_sock }
requests = {}
responses = {}

poll = select.poll()
poll.register(listen_sock, select.POLLIN)

while True:
    for fd, event in poll.poll():
        sock = sockets[fd]

        # Removed closed sockets from our list.
        if event & (select.POLLHUP | select.POLLERR | select.POLLNVAL):
            poll.unregister(fd)
            del sockets[fd]
            requests.pop(sock, None)
            responses.pop(sock, None)

        # Accept connections from new sockets.
        elif sock is listen_sock:
            newsock, sockname = sock.accept()
            newsock.setblocking(False)
            fd = newsock.fileno()
            sockets[fd] = newsock
            poll.register(fd, select.POLLIN)
            requests[newsock] = ''

        # Collect incoming data until it forms a question.
        elif event & select.POLLIN:
            data = sock.recv(4096)
            if not data: # end-of-file
                sock.close() # makes POLLNVAL happen next time
                continue
            requests[sock] += data
            if '?' in requests[sock]:
                question = requests.pop(sock)
                answer = dict(lancelot.qa)[question]
                poll.modify(sock, select.POLLOUT)
                responses[sock] = answer

        # Send out pieces of each reply until they are all sent.
        elif event & select.POLLOUT:
            response = responses.pop(sock)
            n = sock.send(response)
            if n < len(response):
                responses[sock] = response[n:]
            else:
                poll.modify(sock, select.POLLIN)
                requests[sock] = ''
```

The main loop in this program is controlled by the `poll` object, which is queried at the top of every iteration. The `poll()` call is a blocking call, the difference is that `recv()` has to wait on one single client, while `poll()` can wait on dozens or hundreds of clients, and return when any of them shows activity.

The way `poll()` works is that we tell it which sockets we need to monitor, and whether each socket interests us because we want to read from it or write to it. When one or more of the sockets are ready, `poll()` returns and provides a list of the

sockets that we can now use.

To keep things straight when reading the code, think about the lifespan of one particular client and trace what happens to its socket and data.

- The client will first do a `connect()`, and the server's `poll()` call will return and declare that there is data ready on the main listening socket. That can mean only one thing, a new client has connected. So we `accept()` the connection and tell our poll object that we want to be notified when data becomes available for reading from the new socket. To make sure that the `recv()` and `send()` methods on the socket never block and freeze our event loop, we call the `setblocking()` socket method with the value `False` (which means "blocking is not allowed").
- When data becomes available, the incoming string is appended to whatever is already in the requests dictionary under the entry for that socket. (sockets can safely be used as dictionary keys in Python)
- We keep accepting more data until we see a question mark, at which point the Launcelot question is complete. The questions are so short that, in practice, they probably all arrive in the very first `recv()` from each socket; but just to be safe, we have to be prepared to make several `recv()` calls until the whole question has arrived. We then look up the appropriate answer, store it in the responses dictionary under the entry for this client socket, and tell the poll object that we no longer want to listen for more data from this client but instead want to be told when its socket can start accepting outgoing data.
- Once a socket is ready for writing, we send as much of the answer as will fit into one `send()` call on the client socket. This, by the way, is a big reason `send()` returns a length: because if you use it in non-blocking mode, then it might be able to send only some of your bytes without making you wait for a buffer to drain back down.
- Once this server has finished transmitting the answer, we tell the poll object to swap the client socket back over to being listened to for new incoming data.
- After many question-answer exchanges, the client will finally close the connection. Oddly enough, the `POLLHUP`, `POLLERR`, and `POLLNVAL` circumstances that `poll()` can tell us about—all of which indicate that the connection has closed one way or another—are returned only if we are trying to write to the socket, not read from it. So when an attempt to read returns zero bytes, we have to tell the poll object that we now want to write to the socket so that we receive the official notification that the connection is closed.

Two things you should know

- A slightly older mechanism for writing event-driven servers that listen to sockets is to use the `select()` call, which like `poll()` is available from the Python `select` module in the Standard Library. I recommend to use `poll()` because it produces much cleaner code, but many people choose `select()` because it is supported on Windows.
- When talking about event-driven servers, you should take into account the following: **Event-Driven Servers are Blocking and Synchronous**. Referring to the event - driven servers, like the one in `server_poll.py`, some people call them "non-blocking," despite the fact that the `poll()` call blocks (they mean that it does not block waiting for any particular client), and others call them "asynchronous" despite the fact that the program executes its statements in their usual linear order.

The Semantics of Non-blocking

I should add a quick note about how `recv()` and `send()` behave in non-blocking mode, when you have called `setblocking(False)` on their socket. A `poll()` loop like the one just shown means that we never finish calling either of these functions when they cannot accept or provide data. But what if we find ourselves in a situation where we want to call either function in non-blocking mode and do not yet know whether the socket is ready?

For the `recv()` call, these are the rules:

- If data is ready, it is returned.
- If no data has arrived, `socket.error` is raised.
- If the connection has closed, `"` is returned.

Note that closed connection returns a value, but a still-open connection raises an exception. The logic behind this behavior is that the first and last possibilities are both possible in blocking mode as well: either you get data back, or finally the connection closes and you get back an empty string. So to communicate the extra, third possibility that can happen in non-blocking mode—that the connection is still open but no data is ready yet—an exception is used.

The behavior of non-blocking `send()` is similar:

- Some data is sent, and its length is returned.
- The socket buffers are full, so `socket.error` is raised.
- If the connection is closed, `socket.error` is also raised.

This evidence that `poll()` could say that a socket is ready for sending, but a FIN packet from the client could arrive right after the server is released from its `poll()` but before it can start up its `send()` call.

Twisted Python

There are a couple of Python facts to take into account when you are computing your own event-driven server.

It happens that Python comes with an event-driven framework built into the Standard Library. I am going to recommend that you ignore it entirely. It is a pair of ancient modules, `asyncore` and `asynchat`, that date from the early days of Python—you will note that all of the classes they define are lowercase, in defiance of both good taste and all subsequent practice—and that they are difficult to use correctly.

Instead, we will talk about [Twisted Python](#). Twisted Python is not simply a framework; it is an event-driven networking engine for Python. The Twisted community has developed a way of writing Python that is all their own.

Take a look at `server_twisted.py` for how simple our event-driven server can become if we leave the trouble of dealing with the low-level operating system calls to someone else.

```
from twisted.internet.protocol import Protocol, ServerFactory
from twisted.internet import reactor
import lancelot

class Lancelot(Protocol):
    def connectionMade(self):
        self.question = ''

    def dataReceived(self, data):
        self.question += data
        if self.question.endswith('?'):
            self.transport.write(dict(lancelot.qa)[self.question])
            self.question = ''

factory = ServerFactory()
factory.protocol = Lancelot
reactor.listenTCP(1060, factory)
reactor.run()
```

From then on, every event on that socket is translated into a method call to our object, letting us write code that appears to be thinking about just one client at a time. But thanks to the fact that Twisted will create dozens or hundreds of our Lancelot protocol objects, one corresponding to each connected client, the result is an event loop that can respond to whichever client sockets are ready.

[Here](#) you can find more information about Twisted Python

Threading and Multi-processing

The essential idea of a threaded or multi-process server is that we take the simple and straightforward server that we started out with (the `server_simple.py`) and run several copies of it at once so that we can serve several clients at once, without making them wait on each other.

Using multiple threads or processes is very common, especially in high-capacity web and database servers. In the Standard Library you can find the [multiprocessing module](#).

(Note: The main program logic does not even know which solution is being used; the two classes have a similar enough interface that either `Thread` or `Process` can here be used interchangeably.)

Look the example at `server_multi.py`:

```
import sys, time, lancelot
from multiprocessing import Process
from server_simple import server_loop
from threading import Thread

WORKER_CLASSES = {'thread': Thread, 'process': Process}
WORKER_MAX = 10

def start_worker(Worker, listen_sock):
    worker = Worker(target=server_loop, args=(listen_sock,))
    worker.daemon = True # exit when the main process does
    worker.start()
    return worker

if __name__ == '__main__':
    if len(sys.argv) != 3 or sys.argv[2] not in WORKER_CLASSES:
        print >>sys.stderr, 'usage: server_multi.py interface thread|process'
        sys.exit(2)
    Worker = WORKER_CLASSES[sys.argv.pop()] # setup() wants len(argv)==2

    # Every worker will accept() forever on the same listening socket.

    listen_sock = lancelot.setup()
    workers = []
    for i in range(WORKER_MAX):
        workers.append(start_worker(Worker, listen_sock))

    # Check every two seconds for dead workers, and replace them.

    while True:
        time.sleep(2)
        for worker in workers:
            if not worker.is_alive():
                print worker.name, "died; starting replacement worker"
                workers.remove(worker)
                workers.append(start_worker(Worker, listen_sock))
```

As you can see it is letting multiple threads or processes all call `accept()` on the very same server socket, and instead of raising an error and insisting that only one thread at a time be able to wait for an incoming connection, the operating system patiently queues up all of our waiting workers and then wakes up one worker for each new connection that arrives. The fact that a listening socket can be shared at all between threads and processes, and that the operating system does round-robin balancing among the workers that are waiting on an `accept()` call, is one of the great glories of the POSIX network stack and execution model; it makes programs like this very simple to write.

Threading and Multi-processing Frameworks

The `SocketServer` module simplifies the task of writing network servers.

There are four basic server classes: `TCPServer`, `UDPServer`, `UnixDatagramServer` and `UnixStreamServer`.

These four classes process requests synchronously; each request must be completed before the next request can be started. This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process.

In `server_socketserver.py`, you can see how small our multi-threaded server becomes when it takes advantage of this framework. (There is also a `ForkingMixIn` that you can use if you want it to spawn several processes—at least on a POSIX system.)

```
from SocketServer import ThreadingMixIn, TCPServer, BaseRequestHandler
import lancelot, server_simple, socket

class MyHandler(BaseRequestHandler):
    def handle(self):
        server_simple.handle_client(self.request)

class MyServer(ThreadingMixIn, TCPServer):
    allow_reuse_address = 1
    # address_family = socket.AF_INET6 # if you need IPv6

server = MyServer(('', lancelot.PORT), MyHandler)
server.serve_forever()
```

Whereas our earlier example created the workers up front so that they were all sharing the same listening socket, the `SocketServer` does all of its listening in the main thread and creates one worker each time `accept()` returns a new client socket.

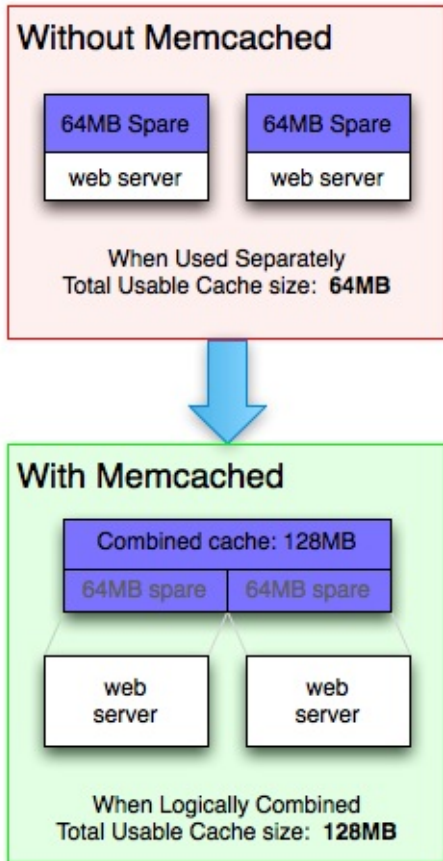
Caches, Message Queues, and Map-Reduce

This chapter surveys the handful of technologies that have together become fundamental building blocks for expanding applications to Internet scale.

This chapter's purpose is to introduce you to the problem that each tool solves; explain how to use the service to address that issue; and give a few hints about using the tool from Python.

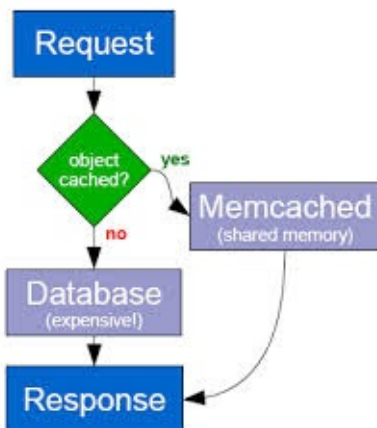
Using Memcached

Memcached is the “memory cache daemon.” Its impact on many large Internet services has been, by all accounts, revolutionary. After glancing at how to use it from Python, we will discuss its implementation, which will teach us about a very important modern network concept called sharding.



The actual procedures for using Memcached are designed to be very simple:

- You run a Memcached daemon on every server with some spare memory.
- You make a list of the IP address and port numbers of your new Memcached daemons, and distribute this list to all of the clients that will be using the cache.
- Your client programs now have access to an organization-wide blazing-fast keyvalue cache that acts something like a big Python dictionary that all of your servers can share. The cache operates on an LRU (least-recently-used) basis, dropping old items that have not been accessed for a while so that it has room to both accept new entries and keep records that are being frequently accessed.



Enough Python clients are currently listed for Memcached that I had better just send you to the page that lists them, rather than try to review them here: <http://code.google.com/p/memcached/wiki/Clients>. The client that they list first is written in pure Python, and therefore will not need to compile against any libraries. Memcached can be installed thanks to being available on the Python Package Index:

```
root@erlerobot:~/Python_files# pip install python-memcached
..
The interface is straightforward. Though you might have expected an interface that more strongly resembles a Python dictionary with native methods like __getitem__, the author of python-memcached chose instead to use the same method names as are used in other languages supported by Memcached—which I think was a good decision, since it makes it easier to translate Memcached examples into Python :
```python
>>> import memcache
>>> mc = memcache.Client(['127.0.0.1:11211'])
>>> mc.set('user:19', {'name: "Lancelot", quest: "Grail"})
True
>>> mc.get('user:19')
{'name: "Lancelot", quest: "Grail"}
```

The basic pattern by which Memcached is used from Python is shown in `squares.py`. Before embarking on an (artificially) expensive operation, it checks Memcached to see whether the answer is already present. If so, then the answer can be returned immediately; if not, then it is computed and stored in the cache before being returned.

```
import memcache, random, time, timeit
mc = memcache.Client(['127.0.0.1:11211'])

def compute_square(n):
 value = mc.get('sq:%d' % n)
 if value is None:
 time.sleep(0.001) # pretend that computing a square is expensive
 value = n * n
 mc.set('sq:%d' % n, value)
 return value

def make_request():
 compute_square(random.randint(0, 5000))

print 'Ten successive runs:',
for i in range(1, 11):
 print '%.2fs' % timeit.timeit(make_request, number=2000),
print
```

The Memcached daemon needs to be running on your machine at port 11211 for this example to succeed. For the first few hundred requests, of course, the program will run at its usual speed. But as the cache begins to accumulate more requests, it is able to accelerate an increasingly large fraction of them.

```
root@erlerobot:~/Python_files# python squares.py

Ten successive runs: 2.75s 1.98s 1.51s 1.14s 0.90s 0.82s 0.71s 0.65s 0.58s 0.55s
```

This pattern is generally characteristic of caching: a gradual improvement as the cache begins to cover the problem domain, and then stability as either the cache fills or the input domain has been fully covered.

You must always remember that Memcached is a cache; it is ephemeral, it uses RAM for storage, and, if re-started, it remembers nothing that you have ever stored! Your application should always be able to recover if the cache should disappear.

# Memcached and Sharding

The design of Memcached illustrates an important principle that is used in several other kinds of databases, and which you might want to employ in architectures of your own: the clients shard the database by hashing the keys' string values and letting the hash determine which member of the cluster is consulted for each key.

To understand why this is effective, consider a particular key/value pair—like the key `sq:42` and the value `1764` that might be stored by `squares.py`. To make the best use of the RAM it has available, the Memcached cluster wants to store this key and value exactly once. But to make the service fast, it wants to avoid duplication without requiring any coordination between the different servers or communication between all of the clients.

This means that all of the clients, without any other information to go on than (a) the key and (b) the list of Memcached servers with which they are configured, need some scheme for working out where that piece of information belongs. If they fail to make the same decision, then not only might the key and value be copied on to several servers and reduce the overall memory available, but also a client's attempt to remove an invalid entry could leave other invalid copies elsewhere.

The solution is that the clients all implement a single, stable algorithm that can turn a key into an integer  $n$  that selects one of the servers from their list. They do this by using a "hash" algorithm, which mixes the bits of a string when forming a number so that any pattern in the string is, hopefully, obliterated. You can find [hashlib module](#) in the Python Standard Library.

To see why patterns in key values must be obliterated, consider `hashing.py`. It loads a dictionary of English words (you might have to download a dictionary of your own or adjust the path to make the script run on your own machine), and explores how those words would be distributed across four servers if they were used as keys. The first algorithm tries to divide the alphabet into four roughly equal sections and distributes the keys using their first letter; the other two algorithms use hash functions.

```
import hashlib

def alpha_shard(word):
 """Do a poor job of assigning data to servers by using first letters."""
 if word[0] in 'abcdef':
 return 'server0'
 elif word[0] in 'ghijklm':
 return 'server1'
 elif word[0] in 'nopqrs':
 return 'server2'
 else:
 return 'server3'

def hash_shard(word):
 """Do a great job of assigning data to servers using a hash value."""
 return 'server%d' % (hash(word) % 4)

def md5_shard(word):
 """Do a great job of assigning data to servers using a hash value."""
 # digest() is a byte string, so we ord() its last character
 return 'server%d' % (ord(hashlib.md5(word).digest()[-1]) % 4)

words = open('/usr/share/dict/words').read().split()

for function in alpha_shard, hash_shard, md5_shard:
 d = {'server0': 0, 'server1': 0, 'server2': 0, 'server3': 0}
 for word in words:
 d[function(word.lower())] += 1
 print function.__name__[:-6], d
```

The `hash()` function is Python's own built-in hash routine, which is designed to be blazingly fast because it is used internally to implement Python dictionary lookup.

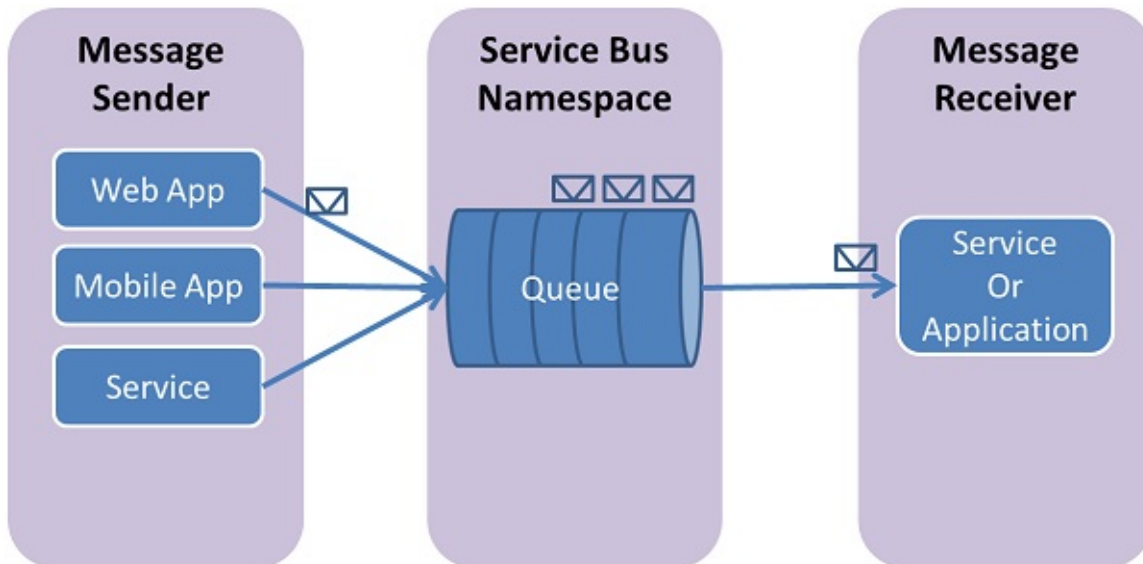


# Message Queues

Message queue protocols let you send reliable chunks of data called messages. Typically, a queue promises to transmit messages reliably, and to deliver them atomically: a message either arrives whole and intact, or it does not arrive at all. Clients never have to loop and keep calling something like `recv()` until a whole message has arrived. The other innovation that message queues offer is that, instead of supporting only the point-to-point connections that are possible with an IP transport like TCP, you can set up all kinds of topologies between messaging clients. Each brand of message queue typically supports several topologies.

A pipeline topology is the pattern that perhaps best resembles the picture you have in your head when you think of a queue: a producer creates messages and submits them to the queue, from which the messages can then be received by a consumer. For example, the front-end web machines of a photosharing web site might accept image uploads from end users and list the incoming files on an internal queue. A machine room full of servers could then read from the queue, each receiving one message for each read it performs, and generate thumbnails for each of the incoming images. The queue might get long during the day and then be short or empty during periods of relatively low use, but either way the front-end web servers are freed to quickly return a page to the waiting customer, telling them that their upload is complete and that their images will soon appear in their photostream.

A publisher-subscriber topology looks very much like a pipeline, but with a key difference. The pipeline makes sure that every queued message is delivered to exactly one consumer—since, after all, it would be wasteful for two thumbnail servers to be assigned the same photograph. But subscribers typically want to receive all of the messages that are being enqueued by each publisher—or else they want to receive every message that matches some particular topic. Either way, a publisher-subscriber model supports messages that fan out to be delivered to every interested subscriber. This kind of queue can be used to power external services that need to push events to the outside world, and also to form a fabric that a machine room full of servers can use to advertise which systems are up, which are going down for maintenance, and that can even publish the addresses of other message queues as they are created and destroyed.



Finally, a request-reply pattern is often the most complex because messages have to make a roundtrip. Both of the previous patterns placed very little responsibility on the producer of a message: they connect to the queue, transmit their message, and are done. But a message queue client that makes a request has to stay connected and wait for the corresponding reply to be delivered back to it. The queue itself, to support this, has to feature some sort of addressing scheme by which replies can be directed to the correct client that is still sitting and waiting for it. But for all of its underlying complexity, this is probably the most powerful pattern of all, since it allows the load of dozens or hundreds of clients to be spread across equally large numbers of servers without any effort beyond setting up the message queue. And since a good message queue will allow servers to attach and detach without losing messages, this topology allows servers to be brought down for maintenance in a way that is invisible to the population of client machines.

# Using Message Queues from Python

There are several AMQP (Advanced Message Queuing Protocol) implementations currently listed in the Python Package Index.

An alternative to using AMQP and having to run a central broker, like RabbitMQ or Apache Qpid, is to use ØMQ, the “Zero Message Queue,” which was invented by the same company as AMQP but moves the messaging intelligence from a centralized broker into every one of your message client programs.

A good summary of the advantages and disadvantages is provided at the ØMQ web site: <http://zeromq.org/docs/welcome-from-amqp>

The next example, `queucrazy.py`, shows some of the patterns that can be supported when message queues are used to connect different parts of an application. It requires ØMQ, which you can most easily make available to Python Index:

```
root@erlerobot:~/Python_files# pip install pyzmq-static
```

The listing uses Python threads to create a small cluster of six different services. One pushes a constant stream of words on to a pipeline. Three others sit ready to receive a word from the pipeline; each word wakes one of them up. The final two are request-reply servers, which resemble remote procedure endpoints and send back a message for each message they receive.

```
import random, threading, time, zmq
zcontext = zmq.Context()

def fountain(url):
 """Produces a steady stream of words."""
 zsock = zcontext.socket(zmq.PUSH)
 zsock.bind(url)
 words = [w for w in dir(__builtins__) if w.islower()]
 while True:
 zsock.send(random.choice(words))
 time.sleep(0.4)

def responder(url, function):
 """Performs a string operation on each word received."""
 zsock = zcontext.socket(zmq.REP)
 zsock.bind(url)
 while True:
 word = zsock.recv()
 zsock.send(function(word)) # send the modified word back

def processor(n, fountain_url, responder_urls):
 """Read words as they are produced; get them processed; print them."""
 zpullsock = zcontext.socket(zmq.PULL)
 zpullsock.connect(fountain_url)

 zreqsock = zcontext.socket(zmq.REQ)
 for url in responder_urls:
 zreqsock.connect(url)

 while True:
 word = zpullsock.recv()
 zreqsock.send(word)
 print n, zreqsock.recv()

def start_thread(function, *args):
 thread = threading.Thread(target=function, args=args)
 thread.daemon = True # so you can easily Control-C the whole program
 thread.start()

start_thread(fountain, 'tcp://127.0.0.1:6700')
start_thread(responder, 'tcp://127.0.0.1:6701', str.upper)
start_thread(responder, 'tcp://127.0.0.1:6702', str.lower)
for n in range(3):
 start_thread(processor, n + 1, 'tcp://127.0.0.1:6700',
```

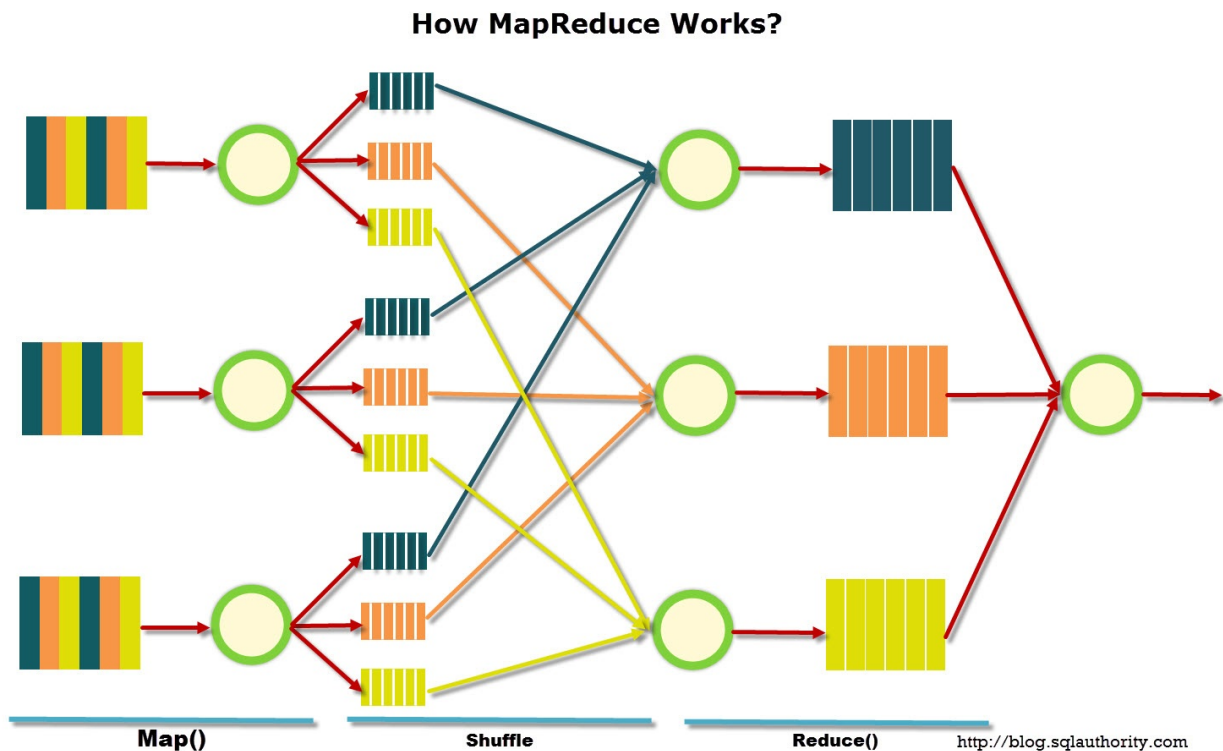
```
time.sleep(30) ['tcp://127.0.0.1:6701', 'tcp://127.0.0.1:6702']
```

The two request-reply servers are different—one turns each word it receives to uppercase, while the other makes its words all lowercase—and you can tell the three processors apart by the fact that each is assigned a different integer.

Finally I would like to add the following to fix the concept of message Queues: Message queues provide a point of coordination and integration for different parts of your application that may require different hardware, load balancing techniques, platforms, or even programming languages. They can take responsibility for distributing messages among many waiting consumers or servers in a way that is not possible with the single point-to-point links offered by normal TCP sockets, and can also use a database or other persistent storage to assure that updates to your service are not lost if the server goes down. Message queues also offer resilience and flexibility, since if some part of your system temporarily becomes a bottleneck, then the message queue can absorb the shock by allowing many messages to queue up for that service. By hiding the population of servers or processes that serve a particular kind of request, the message queue pattern also makes it easy to disconnect, upgrade, reboot, and reconnect servers without the rest of your infrastructure noticing.

# Map-Reduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.



A MapReduce program is composed of a `Map()` procedure that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a `Reduce()` procedure that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The "MapReduce System" (also called "infrastructure" or "framework") orchestrates by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

These two operations bear some resemblance to the Python built-in functions of that name (which Python itself borrowed from the world of functional programming); imagine how one might split across several servers the tasks of summing the squares of many integers:

```
>>> squares = map(lambda n: n*n, range(11))
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> import operator
>>> reduce(operator.add, squares)
385
```

The mapping operation should be prepared to run once on some particular slice of the overall problem or data set, and to produce a tally, table, or response that summarizes its findings for that slice of the input. The reduce operation is then exposed to the outputs of the mapping functions, to combine them together into an ever-accumulating answer. To use the mapreduce cluster's power effectively, frameworks are not content to simply run the reduce function on one node once all of the dozens or hundreds of active machines have finished the mapping stage. Instead, the reduce function is run in parallel on many nodes at once, each considering the output of a handful of map operations, and then these intermediate results are combined again and again in a tree of computations until a final reduce step produces output for the whole input.

In conclusion, the map-reduce pattern provides a cloud-style framework for distributed computation across many

processors and, potentially, across many parts of a large data set.

# HTTP

---

Hypertext is structured text that uses logical links (hyperlinks) between nodes containing text. HTTP (The Hypertext Transfer Protocol) is the protocol to exchange or transfer hypertext.

HTTP is the foundation of data communication for the World Wide Web. As this chapter proceeds to explore the features of HTTP, we are going to illustrate the protocol using several modules that come built-in to the Python Standard Library

# URL Anatomy

Uniform Resource Locators (URLs), are strings that tell your web browser how to fetch resources from the World Wide Web. They are a subclass of the full set of possible Uniform Resource Identifiers (URIs); specifically, they are URIs constructed so that they give instructions for fetching a document, instead of serving only as an identifier.

To understand how they work, consider a very simple URL, for example, like the following: `http://python.org`. If submitted to a web browser, this URL is interpreted as an order to resolve the host name `python.org` to an IP address, make a TCP connection to that IP address at the standard HTTP port 80, and then ask for the root document `/` that lives at that site.

Now imagine another more complicated URL, imagine that we wanted the logo for Nord/LB, a large German bank. The resulting URL might look something like this: `http://example.com:8080/Nord%2FLB/logo?shape=square&dpi=96`

Here, the URL specifies more information than our previous example did:

- The protocol will, again, be HTTP.
- The hostname `example.com` will be resolved to an IP.
- This time, port 8080 will be used instead of 80.
- Once a connection is complete, the remote server will be asked for the resource named: `/Nord%2FLB/logo?shape=square&dpi=96`

Web servers, in practice, have absolute freedom to interpret URLs as they please; however, the intention of the standard is that this URL be parsed into two question-mark-delimited pieces. The first is a path consisting of two elements:

- A Nord/LB path element.
- A logo path element.

The string following the `?` is interpreted as a query containing two terms:

- A shape parameter whose value is square.
- A dpi parameter whose value is 96.

Any characters beyond the alphanumerics, a few punctuation marks—specifically the set `$. _ . ! * ( )`,—and the special delimiter characters themselves (like the slashes) must be percent-encoded by following a percent sign `%` with the two-digit hexadecimal code for the character.

You should note that the following URL paths are not equivalent:

```
Nord%2FLB%2Flogo = A single path component, named Nord/LB/logo.
```

```
Nord%2FLB/logo = Two path components, Nord/LB and logo.
```

```
Nord/LB/logo = Three separate path components Nord, LB, and logo.
```

The most important Python routines for working with URLs live, appropriately enough, in their own module. The [urlparse module](#); this module defines a standard interface to break URL strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a “relative URL” to an absolute URL given a “base URL.”

```
>>> from urlparse import urlparse, urldefrag, parse_qs, parse_qsl
```

With these routines, you can get large and complex URLs like the example given earlier and turn them into their component parts, with RFC-compliant parsing already implemented for you:

```
```python
>>> p = urlparse('http://example.com:8080/Nord%2FLB/logo?shape=square&dpi=96')
>>> p
ParseResult(scheme='http', netloc='example.com:8080', path='/Nord%2FLB/logo',
```

```
» » » params='', query='shape=square&dpi=96', fragment='')
```

The query string that is offered by the `ParseResult` can then be submitted to one of the parsing routines if you want to interpret it as a series of key-value pairs, which is a standard way for web forms to submit them:

```
>>> parse_qs(p.query)
{'shape': ['square'], 'dpi': ['96']}
```

Note that each value in this dictionary is a list, rather than simply a string. This is to support the fact that a given parameter might be specified several times in a single URL; in such cases, the values are simply appended to the list:

```
>>> parse_qs('mode=topographic&pin=Boston&pin=San%20Francisco')
{'mode': ['topographic'], 'pin': ['Boston', 'San Francisco']}
```

This, you will note, preserves the order in which values arrive; of course, this does not preserve the order of the parameters themselves because dictionary keys do not remember any particular order. If the order is important to you, then use the `parse_qs_l()` function instead (the `l` must stand for “list”):

```
>>> parse_qs_l('mode=topographic&pin=Boston&pin=San%20Francisco')
[('mode', 'topographic'), ('pin', 'Boston'), ('pin', 'San Francisco')]
```

Finally, note that an “anchor” appended to a URL after a `#` character is not relevant to the HTTP protocol. This is because any anchor is stripped off and is not turned into part of the HTTP request. Instead, the anchor tells a web client to jump to some particular section of a document after the HTTP transaction is complete and the document has been downloaded. To remove the anchor, use `urldefrag()`:

```
>>> u = 'http://docs.python.org/library/urlparse.html#urlparse.urldefrag'
>>> urldefrag(u)
('http://docs.python.org/library/urlparse.html', 'urlparse.urldefrag')
```

You can turn a `ParseResult` back into a URL by calling its `geturl()` method. When combined with the `urlencode()` function, which knows how to build query strings, this can be used to construct new URLs:

```
>>> import urllib, urlparse
>>> query = urllib.urlencode({'company': 'Nord/LB', 'report': 'sales'})
>>> p = urlparse.ParseResult(
... 'https', 'example.com', 'data', None, query, None)
>>> p.geturl()
'https://example.com/data?report=sales&company=Nord%2FLB'
```

For last, the HTTP request look like this:

```
GET /rfc/rfc2616.txt HTTP/1.1
Accept-Encoding: identity
Host: www.ietf.org
Connection: close
User-Agent: Python-urllib/2.7
```

And the HTTP response that comes back over the socket also starts with a set of headers, but then also includes a body that contains the document itself that has been requested :

```
HTTP/1.1 200 OK
Server: cloudflare-nginx
```


Date: Fri, 11 Jul 2014 07:02:55 GMT
Content-Type: text/plain
Transfer-Encoding: chunked
Connection: close
Set-Cookie: __cfduid=d5be98ff9fbae526f308d478da5bb413e1405062173934; expires=Mon, 23-Dec-2019 23:50:00 GMT; path=/; domain=...
Last-Modified: Fri, 11 Jun 1999 18:46:53 GMT
Vary: Accept-Encoding
CF-RAY: 1483235b13c51043-CDG
<addinfourl at 4341048456 whose fp = <socket._fileobject object at 0x102a13750>>



Relative URLs

Very often, the links used in web pages do not specify full URLs, but relative URLs that are missing several of the usual components. When one of these links needs to be resolved, the client needs to fill in the missing information with the corresponding fields from the URL used to fetch the page in the first place.

The simplest relative links are the names of pages one level deeper than the base page:

```
>>> urlparse.urljoin('http://www.python.org/psf/', 'grants')
'http://www.python.org/psf/grants'
>>> urlparse.urljoin('http://www.python.org/psf/', 'mission')
'http://www.python.org/psf/mission'
```

Note the crucial importance of the trailing slash in the URLs:

```
>>> urlparse.urljoin('http://www.python.org/psf', 'grants')
'http://www.python.org/grants'
```

Like file system paths on the POSIX and Windows operating systems, `.` can be used for the current directory and `..` is the name of the parent:

```
>>> urlparse.urljoin('http://www.python.org/psf/', './mission')
'http://www.python.org/psf/mission'
>>> urlparse.urljoin('http://www.python.org/psf/', '../news/')
'http://www.python.org/news/'
>>> urlparse.urljoin('http://www.python.org/psf/', '/dev/')
'http://www.python.org/dev'
```

And, as illustrated in the last example, a relative URL that starts with a slash is assumed to live at the top level of the same site as the original URL. Happily, the `urljoin()` function ignores the base URL entirely if the second argument also happens to be an absolute URL. This means that you can simply pass every URL on a given web page to the `urljoin()` function, and any relative links will be converted; at the same time, absolute links will be passed through untouched:

```
>>> # Absolute links are safe from change
...
>>> urlparse.urljoin('http://www.python.org/psf/', 'http://yelp.com/')
'http://yelp.com/'
```

Instrumenting urllib2

We now turn to the HTTP protocol itself. Although its on-the-wire appearance is usually an internal detail handled by web browsers and libraries like `urllib2` module. The `urllib2` module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

we are going to adjust its behavior so that we can see the protocol printed to the screen. Take a look at `verbose_http.py` :

```
import StringIO, httplib, urllib2

class VerboseHTTPResponse(httplib.HTTPResponse):
    def _read_status(self):
        s = self.fp.read()
        print '-' * 20, 'Response', '-' * 20
        print s.split('\r\n\r\n')[0]
        self.fp = StringIO.StringIO(s)
        return httplib.HTTPResponse._read_status(self)

class VerboseHTTPConnection(httplib.HTTPConnection):
    response_class = VerboseHTTPResponse
    def send(self, s):
        print '-' * 50
        print s.strip()
        httplib.HTTPConnection.send(self, s)

class VerboseHTTPHandler(urllib2.HTTPHandler):
    def http_open(self, req):
        return self.do_open(VerboseHTTPConnection, req)
```

This customization prints out both the outgoing request and the incoming response instead of keeping them both hidden.

To allow for customization, the `urllib2` library lets you bypass its vanilla `urlopen()` function and instead build an opener full of handler classes of your own devising—a fact that we will use repeatedly as this chapter progresses. Listing 9–1 provides exactly such a handler class by performing a slight customization on the normal HTTP handler. This customization prints out both the outgoing request and the incoming response instead of keeping them both hidden. For many of the following examples, we will use an opener object that we build right here, using the handler from `verbose_http.py` :

```
>>> from verbose_http import VerboseHTTPHandler
>>> import urllib, urllib2
>>> opener = urllib2.build_opener(VerboseHTTPHandler)
```

You can try using this opener against the URL of the RFC that we mentioned at the beginning of this chapter:

```
opener.open('http://www.ietf.org/rfc/rfc2616.txt')
```

The GET Method and The Host Header

When the earliest version of HTTP was first invented, it had a single power: to issue a method called GET that named and returned a hypertext document from a remote server. That method is still the backbone of the protocol today.

The GET method, like all HTTP methods, is the first thing transmitted as part of an HTTP request, and it is immediately followed by the request headers. For simple GET methods, the request simply ends with the blank line that terminates the headers so the server can immediately stop reading and send a response.

```
>>> info = opener.open('http://www.ietf.org/rfc/rfc2616.txt')
GET /rfc/rfc2616.txt HTTP/1.1
Accept-Encoding: identity
Host: www.ietf.org
Connection: close...
```

The opener's `open()` method, like the plain `urlopen()` function at the top level of `urllib2`, returns an information object that lets us examine the result of the GET method. You can see that the HTTP request started with a status line containing the HTTP version, a status code, and a short message. The info object makes these available as object attributes; it also lets us examine the headers through a dictionary-like object:

```
>>> info.code
200
>>> info.msg
'OK'
>>> sorted(info.headers.keys())
['accept-ranges', 'connection', 'content-length', 'content-type',
'date', 'etag', 'last-modified', 'server', 'vary']
>>> info.headers['Content-Type']
'text/plain'
```

Finally, the info object is also prepared to act as a file. The HTTP response status line, the headers, and the blank line that follows them have all been read from the HTTP socket, and now the actual document is waiting to be read. As is usually the case with file objects, you can either start reading the info object in pieces through `read(N)` or `readline()`; or you can choose to bring the entire data stream into memory as a single string:

```
>>> print info.read().strip()
Network Working Group R. Fielding
Request for Comments: 2616 UC Irvine
Obsoletes: 2068 J. Gettys
Category: Standards Track Compaq/W3C
...
```

These are the first lines of the longer text file that you will see if you point your web browser at the same URL.

In a world of six billion people and four billion IP addresses, the need quickly became clear to support servers that might host dozens of web sites at the same IP. And that is why the URL location is now included in every HTTP request. For compatibility, it has not been made part of the GET request line itself, but has instead been stuck into the headers under the name Host.

```
>>> info = opener.open('http://www.google.com/')
----- Response -----
HTTP/1.1 302 Found
Cache-Control: private
...
-----
GET /?gfe_rd=cr&ei=0Y6_U_qjH0eA8QeTg4H4BQ HTTP/1.1
Accept-Encoding: identity
Host: www.google.es
Connection: close
```

```
User-Agent: Python-urllib/2.7
----- Response -----
HTTP/1.1 200 OK
...
```

Depending on how they are configured, servers might return entirely different sites when confronted with two different values for Host; they might present slightly different versions of the same site; or they might ignore the header altogether. But semantically, two requests with different values for Host are asking about two entirely different URLs. When several sites are hosted at a single IP address, those sites are each said to be served by a virtual host, and the whole practice is sometimes referred to as virtual hosting.

Is also important to take care that when handling HTTP different responses can happen, between them codes, errors, and redirection. You can read more about this [here](#).

Payloads and Persistent Connections

By default, HTTP/1.1 servers will keep a TCP connection open even after they have delivered their response. This enables you to make further requests on the same socket and avoid the expense of creating a new socket for every piece of data you might need to download. Keep in mind that downloading a modern web page can involve fetching dozens, if not hundreds, of separate pieces of content. The HTTPConnection class provided by urllib2 lets you take advantage of this feature. In fact, all requests go through one of these objects; when you use a function like `urlopen()` or use the `open()` method on an opener object, an HTTPConnection object is created behind the scenes, used for that one request, and then discarded. When you might make several requests to the same site, use a persistent connection instead:

```
>>> import httplib
>>> c = httplib.HTTPConnection('www.python.org')
>>> c.request('GET', '/')
>>> original_sock = c.sock
>>> content = c.getresponse().read() # get the whole page
>>> c.request('GET', '/about/')
>>> c.sock is original_sock
True
```

Now, if we insert this header manually, then we force the HTTPConnection object to create a second socket when we ask it for a second page:

```
>>> c = httplib.HTTPConnection('www.python.org')
>>> c.request('GET', '/', headers={'Connection': 'close'})
>>> original_sock = c.sock
>>> content = c.getresponse().read()
>>> c.request('GET', '/about/')
>>> c.sock is original_sock
False
```

Note that HTTPConnection does not raise an exception when one socket closes and it has to create another one; you can keep using the same object over and over again. This holds true regardless of whether the server is accepting all of the requests over a single socket, or it is sometimes hanging up and forcing HTTPConnection to reconnect.

POST And Forms

The POST HTTP method was designed to power web forms. When forms are used with the GET method, which is indeed their default behavior, they append the form's field values to the end of the URL: `http://www.google.com/search?q=python+language`

The construction of such a URL creates a new named location that can be saved; bookmarked; referenced from other web pages; and sent in e-mails, Tweets, and text messages. And for actions like searching and selecting data, these features are perfect. But what about a login form that accepts your e-mail address and password? Not only would there be negative security implications to having these elements appended to the form URL—such as the fact that they would be displayed on the screen in the URL bar and included in your browser history—but surely it would be odd to think of your username and password as creating a new location or page on the web site in question: `http://example.com/welcome?email=brandon@rhodesmill.org&pw=aaz9Gog3`

Building URLs in this way would imply that a different page exists on the example.com web site for every possible password that you could try typing. This is undesirable for obvious reasons. And so the POST method should always be used for forms that are not constructing the name of a particular page or location on a web site, but are instead performing some action on behalf of the caller. Forms in HTML can specify that they want the browser to use POST by specifying that method in their

element:

```
<form name="myloginform" action="/access/dummy" method="post">
E-mail: <input type="text" name="e-mail" size="20">
Password: <input type="password" name="password" size="20">
<input type="submit" name="submit" value="Login">
</form>
```

Instead of stuffing form parameters into the URL, a POST carries them in the body of the request. We can perform the same action ourselves in Python by using `urllib2` to format the form parameters, and then supplying them as a second parameter to any of the `urllib2` methods that open a URL. - (From the standard Python library: `urllib.urlencode(query[, doseq])` Convert a mapping object or a sequence of two-element tuples to a “percent-encoded” string, suitable to pass to `urlopen()` above as the optional data argument. This is useful to pass a dictionary of form fields to a POST request.)

```
form = urllib.urlencode({'inputstring': 'Atlanta, GA'})
>>> response = opener.open('http://forecast.weather.gov/zipcity.php', form)
-----
POST /zipcity.php HTTP/1.1
...
Content-Length: 25
Host: forecast.weather.gov
Content-Type: application/x-www-form-urlencoded
...
-----
inputstring=Atlanta%2C+GA
----- Response -----
HTTP/1.1 302 Found
...
Location: http://forecast.weather.gov/MapClick.php?CityName=Atlanta&state=GA
&site=FFC&textField1=33.7629&textField2=-84.4226&e=1
...
-----
GET /MapClick.php?CityName=Atlanta&state=GA&site=FFC&textField1=33.7629&textField2=
-84.4226&e=1 HTTP/1.1
...
----- Response -----
HTTP/1.1 200 OK
...
```

Although our opener object is putting a dashed line between each HTTP request and its payload for clarity (a blank line, you will recall, is what really separates headers and payload on the wire) you are otherwise seeing a raw HTTP POST method here. Note these features of the request-responses shown in example above:

- The request line starts with the string POST.
- Content is provided (and thus, a Content-Length header).
- The form parameters are sent as the body.
- The Content-Type for standard web forms is x-www-form-urlencoded.

The most important thing to grasp is that GET and POST are most emphatically not simply two different ways to format form parameters. Instead, they actually mean two entirely different things. The GET method means, "I believe that there is a document at this URL; please return it." The POST method means, "Here is an action that I want performed."

Successful Form POSTs Should Always Redirect

In the POST example above you can notice that instead of simply returning a status of 200 followed by a page of weather forecast data, it instead returned a 302 redirect that urllib2 obeyed by performing a GET for the page named in the Location: header.

A web site leaves users in a very difficult position if it answers a POST form submission with a literal web page. Well-designed user-facing POST forms always redirect to a page that shows the result of the action, and this page can be safely bookmarked, shared, stored, and reloaded. This is an important feature of modern browsers: if a POST results in a redirect, then pressing the reload button simply refetches the final URL and does not reattempt the whole train of redirects that lead to the current location

REST And More HTTP Methods

Web-based APIs, which fetch documents and data using GET and POST to specific URLs. Therefore, we should immediately note that many modern web services try to integrate their APIs more tightly with HTTP by going beyond the two most common HTTP methods by implementing additional methods like PUT and DELETE.

A design pattern named “Representational State Transfer” has therefore been taking hold in many developer communities. It specifies that the nouns of an API should live at their own URLs. For example, PUT, GET, POST, and DELETE should be used, respectively, to create, fetch, modify, and remove the documents living at these URLs.

By coupling this basic recommendation with further guidelines, the REST methodology guides the creation of web services that make more complete use of the HTTP protocol. Such web services also offer quite clean semantics, and can be accelerated by the same caching proxies that are often used to speed the delivery of normal web pages.

Note that HTTP supports arbitrary method names, even though the standard defines specific semantics for GET and POST and all of the rest. Tradition would dictate using the well-known methods defined in the standard unless you are using a specific framework or methodology that recognizes and has defined other methods.

Identifying User Agents and Web Servers

`User-Agent: Python-urllib/2.6` : This header is optional in the HTTP protocol, and many sites simply ignore or log it. It can be useful when sites want to know which browsers their visitors use most often, and it can sometimes be used to distinguish search engine spiders (bots) from normal users browsing a site.

Many web sites are sensitive to the kinds of browsers that view them. If you need to access such sites with `urllib2`, you can simply instruct it to lie about its identity, and the receiving web site will not know the difference:

```
>>> url = 'https://wca.eclaim.com/'
>>> urllib2.urlopen(url).read()
'<HTML>...The following are...required...Microsoft Internet Explorer...'
>>> agent = 'Mozilla/5.0 (Windows; U; MSIE 7.0; Windows NT 6.0; en-US)'
>>> request = urllib2.Request(url)
>>> request.add_header('User-Agent', agent)
>>> urllib2.urlopen(request).read()
'\r\n<HTML>\r\n<HEAD>\r\n\t<TITLE>Eclaim.com - Log In</TITLE>...'
```

There are databases of possible user agent strings online at several sites that you can reference both when analyzing agent strings that your own servers have received, as well as when concocting strings for your own HTTP requests:

- http://www.zytrax.com/tech/web/browser_ids.htm
- <http://www.useragentstring.com/pages/useragentstring.php>

Content Type Negotiation

It is always possible to simply make an HTTP request and let the server return a document with whatever Content-Type: is appropriate for the information we have requested. Some of the usual content types encountered by a browser include the following: *text/html*, *text/plain*, *text/css*, *image/gif*, *image/jpeg*, *image/x-png*, *application/javascript*, *application/pdf*, *application/zip*.

If the web service is returning a generic data stream of bytes that it cannot describe more specifically, it can always fall back to the content type: *application/octet-stream*.

The four headers that will interest you include the following: *Accept*, *Accept-Charset*, *Accept-Language*, *Accept-Encoding*

Each of these headers supports a comma-separated list of items, where each item can be given a weight between one and zero (larger weights indicate more preferred items) by adding a suffix that consists of a semi-colon and q= string to the item. The result will look something like this (using, for illustration, the Accept: header that my Google Chrome browser seems to be currently using): `Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5`

This indicates that Chrome prefers XML and XHTML, but will accept HTML or even plain text if those are the only document formats available; that Chrome prefers PNG images when it can get them; and that it has no preference between all of the other content types in existence.

Compression

While many documents delivered over HTTP are already fairly heavily compressed, including images and file formats like PDF, web pages themselves are written in verbose SGML dialects that can consume much less bandwidth if subjected to generic textual compression. Similarly, CSS and JavaScript files also contain very stereotyped patterns of punctuation and repeated variable names, which is very amenable to compression.

Web clients can make servers aware that they accept compressed documents by listing the formats they support in a request header, as in this example: ``Accept-Encoding: gzip``

For some reason, many sites seem to not offer compression unless the User-Agent: header specifies something they recognize. Thus, to convince Google to compress its Google News page, you have to use `urllib2` something like this:

```
>>> request = urllib2.Request('http://news.google.com/')
>>> request.add_header('Accept-Encoding', 'gzip')
>>> request.add_header('User-Agent', 'Mozilla/5.0')
>>> info = opener.open(request)
-----
GET / HTTP/1.1
Host: news.google.com
User-Agent: Mozilla/5.0
Connection: close
Accept-Encoding: gzip
----- Response -----
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
...
Content-Encoding: gzip
...
```

Remember that web servers do not have to perform compression, and that many will ignore your `Accept-Encoding: header`. Therefore, you should always check the content encoding of the response, and perform decompression only when the server declares that it is necessary:

```
>>> info.headers['Content-Encoding'] == 'gzip'
True
>>> import gzip, StringIO
>>> gzip.GzipFile(fileobj=StringIO.StringIO(info.read())).read()
'<!DOCTYPE HTML ...<html>...</html>'
```

As you can see, Python does not let us pass the file-like info response object directly to the GzipFile class because, it is not quite file-like enough. Here, we can perform the quick work-around of reading the whole compressed file into memory and then wrapping it in a StringIO object that does support `tell()`.

HTTP Caching

Many elements of a typical web site design are repeated on every page you visit, and your browsing would slow to a crawl if every image and decoration had to be downloaded separately for every page you viewed. Well-configured web servers therefore add headers to every HTTP response that allow browsers, as well as any proxy caches between the browser and the server, to continue using a copy of a downloaded resource for some period of time until it expires.

There are two basic mechanisms by which servers can support client caching. In the first approach, an HTTP response includes an Expires: header that formats a date and time using the same format as the standard Date: header: Expires: Sun, 21 Jan 2010 17:06:12 GMT . However, this requires the client to check its clock—and many computers run clocks that are far ahead of or behind the real current date and time.

This brings us to a second, more modern alternative, the Cache-Control header, that depends only on the client being able to correctly count seconds forward from the present. For example, to allow an image or page to be cached for an hour but then insist that it be refetched once the hour is up, a cache control header could be supplied like this: Cache-Control: max-age=3600, must-revalidate .

The HEAD Method

It's possible that you might want your program to check a series of links for validity or whether they have moved, but you do not want to incur the expense of actually downloading the body that would follow the HTTP headers. In this case, you can issue a HEAD request. This is directly possible through `httplib`, but it can also be performed by `urllib2` if you are willing to write a small request class of your own:

```
>>> class HeadRequest(urllib2.Request):
...     def get_method(self):
...         return 'HEAD'
...
>>> info = urllib2.urlopen(HeadRequest('http://www.google.com/'))
>>> info.read()
''
```

You can see that the body of the response is completely empty.

HTTPS Encryption

An encrypted URL starts with `https:` instead of simply `http:`, uses the default port 443 instead of port 80, and uses TLS.

Encryption has to be negotiated before the user can send his HTTP request, lest all of the information in it be divulged; but until the request is transmitted, the server does not know what Host: the request will specify. Therefore, encrypted web sites still live under the old problem of having to use a different IP address for every domain that must be hosted.

A technique known as “Server Name Indication” (SNI) has been developed to get around this traditional restriction; however, Python does not yet support it. It appears, though, that a patch was applied to the Python 3 trunk with this feature, only days prior to the time of writing. Here is the ticket in case you want to follow the issue:

<http://bugs.python.org/issue5639>.

To use HTTPS from Python, simply supply an `https:` method in your URL:

```
>>> info = urllib2.urlopen('https://www.ietf.org/rfc/rfc2616.txt')
>>>
```

If the connection works properly, then neither your government nor any of the various large and shadowy corporations that track such things should be able to easily determine either the search term you used or the results you viewed.

HTTP Authentication

The HTTP protocol came with a means of authentication that was so poorly thought out and so badly implemented that it seems to have been almost entirely abandoned. When a server was asked for a page to which access was restricted, it was supposed to return a response code: `HTTP/1.1 401 Authorization Required`.

The authentication token was generated by doing base64 encoding on the colon-separated username and password:

```
>>> import base64
>>> print base64.b64encode("guido:van0ranje!")
Z3VpZG86dmFuT3Jhbmp1IQ==
```

This, of course, just protects any special characters in the username and password that might have been confused as part of the headers themselves; it does not protect the username and password at all, since they can very simply be decoded again:

```
>>> print base64.b64decode("Z3VpZG86dmFuT3Jhbmp1IQ==")
guido:van0ranje!
```

Anyway, once the encoded value was computed, it could be included in the second request like this: ``Authorization: Basic QWxhZGRpbjpvYVUHNlc2FtZQ==``

An incorrect password or unknown user would elicit additional 401 errors from the server, resulting in the pop-up box appearing again and again. Finally, if the user got it right, she would either be shown the resource or—if she in fact did not have permission—be shown a response code like the following: `403 Forbidden`.

Python supports this kind of authentication through a handler that, as your program uses it, can accumulate a list of passwords.

```
auth_handler = .HTTPBasicAuthHandler()
auth_handler.add_password(realm='voetbal', uri='http://www.onsoranje.nl/',
                        user='guido', passwd='van0ranje!')
```

The resulting handler can be passed into `build_opener()`.

Cookies

The actual mechanism that powers user identity tracking, logging in, and logging out of modern web sites is the cookie. The HTTP responses sent by a server can optionally include a number of Set-cookie: headers that browsers store on behalf of the user. In every subsequent request made to that site, the browser will include a Cookie: header corresponding to each cookie that has been set.

The most obvious use of cookies is to keep up with user identity. To support logging in, a web site can deploy a normal form that asks for your username and password (or e-mail address and password, or whatever).

Cookies can also be used for feats other than simply identifying users. For example, a site can issue a cookie to every browser that connects, enabling it to track even casual visitors. This approach enables an online store to let visitors start building a shopping cart full of items without ever being forced to create an account.

From the point of view of a web client, cookies are moderately short strings that have to be stored and then divulged when matching requests are made. The Python Standard Library puts this logic in its own module, `cookielib` (The `cookielib` module defines classes for automatic handling of HTTP cookies.), whose `CookieJar` objects can be used as small cookie databases by the `HTTPCookieProcessor` in `urllib2`. To see its effect, you need go no further than the front page of Google, which sets cookies in the mere event of an unknown visitor arriving at the site for the first time. Here is how we create a new opener that knows about cookies:

```
>>> import cookielib
>>> cj = cookielib.CookieJar()
>>> cookie_opener = urllib2.build_opener(VerboseHTTPHandler,
... urllib2.HTTPCookieProcessor(cj))
```

Opening the Google front page will result in two different cookies getting set:

```
>>> response = cookie_opener.open('http://www.google.com/')
-----
GET / HTTP/1.1
...
----- Response -----
HTTP/1.1 200 OK
...
Set-Cookie: PREF=ID=94381994af6d5c77:FF=0:TM=1288205983:LM=1288205983:S=Mtwiv17EB73uL5Ky;
expires=Fri, 26-Oct-2012 18:59:43 GMT; path=/; domain=.google.com
Set-Cookie: NID=40=rWLn_I8_PAhUF62J0yFLtb1-AoftgU0RvGSsa81FhTvd4vXD91iU5D0EdxSVt4otiISY-
3RfEYcGFHZA52w3-85p-hujagtB9akaLnS0QHET2v81kke1EGbp07oWr9u5; expires=Thu, 28-Apr-2011
18:59:43 GMT; path=/; domain=.google.com; HttpOnly
...
```

If you investigate more about `cookielib`, you will find that you can do more than query and modify the cookies that have been set. You can also automatically store them in a file, so that they survive from one Python session to the next. You can even create cookie processors that implement your own custom policies with respect to which cookies to store and which to divulge.

Servers can constrain a cookie to a particular domain and path, in addition to setting a Max-age or expires time. Unfortunately, some browsers ignore this setting, so sites should never base their security on the assumption that the expires time will be obeyed. Therefore, servers can mark cookies as secure; this ensures that such cookies are only transmitted with HTTPS requests to the site and never in unsecure HTTP requests.

HTTP Session Hijacking

A perpetual problem with cookies is that web site designers do not seem to realize that cookies need to be protected as zealously as your username and password. While it is true that well-designed cookies expire and will no longer be accepted as valid by the server, cookies—while they last—give exactly as much access to a web site as a username and password.

Some sites do not protect cookies at all: they might require HTTPS for your username and password, but then return you to normal HTTP for the rest of your session. Other sites are smart enough to protect subsequent page loads with HTTPS, even after you have left the login page, but they forget that static data from the same domain, like images, decorations, CSS files, and JavaScript source code, will also carry your cookie. The better alternatives are to either send all of that information over HTTPS, or to carefully serve it from a different domain or path that is outside the jurisdiction of the session cookie.

Should you happen to observe or capture a Cookie: header from an HTTP request that you observe, remember that there is no need to store it in a CookieJar or represent it as a cookielib object at all. Indeed, you could not do that anyway because the outgoing Cookie: header does not reveal the domain and path rules that the cookie was stored with. Instead, just inject the Cookie: header raw into the requests you make to the web site: `python request = urllib2.Request(url)`
`request.add_header('Cookie', intercepted_value) info = urllib2.urlopen(request)`

Cross-Site Scripting Attacks

The earliest experiments with scripts that could run in web browsers revealed a problem: all of the HTTP requests made by the browser were done with the authority of the user's cookies, so pages could cause quite a bit of trouble by attempting to, say, POST to the online web site of a popular bank asking that money be transferred to the attacker's account. Anyone who visited the problem site while logged on to that particular bank in another window could lose money. To address this, browsers imposed the restriction that scripts in languages like JavaScript can only make connections back to the site that served the web page, and not to other web sites. This is called the "same origin policy."

Today, would-be attackers find ways around this policy by using a constellation of attacks called cross-site scripting (known by the acronym XSS to prevent confusion with Cascading Style Sheets). These techniques include things like finding the fields on a web page where the site will include snippets of user-provided data without properly escaping them, and then figuring out how to craft a snippet of data that will perform some compromising action on behalf of the user or send private information to a third party. Next, the wouldbe attackers release a link or code containing that snippet onto a popular web site, bulletin board, or in spam e-mails, hoping that thousands of people will click and inadvertently assist in their attack against the site. There are a collection of techniques that are important for avoiding cross-site scripting; you can find them in any good reference on web development. The most important ones include the following:

- When processing a form that is supposed to submit a POST request, always carefully disregard any GET parameters.
- Never support URLs that produce some side effect or perform some action simply through being the subject of a GET.
- In every form, include not only the obvious information—such as a dollar amount and destination account number for bank transfers—but also a hidden field with a secret value that must match for the submission to be valid. That way, random POST requests that attackers generate with the dollar amount and destination account number will not work because they will lack the secret that would make the submission valid.

While the possibilities for XSS are not, strictly speaking, problems or issues with the HTTP protocol itself, it helps to have a solid understanding of them when you are trying to write any program that operates safely on the World Wide Web.

A library called WebOb is also available for Python (and listed on the Python Package Index) that contains HTTP request and response classes that were designed from the other direction: that is, they were intended all along as general-purpose representations of HTTP in all of its low-level details. You can learn more about them at the WebOb project web page:

<http://pythonpaste.org/webob/>

Screen Scraping

Most web sites are designed first and foremost for human eyes. While well-designed sites offer formal APIs by which you can construct Google maps, upload Flickr photos, or browse YouTube videos, many sites offer nothing but HTML pages formatted for humans. If you need a program to be able to fetch its data, then you will need the ability to dive into densely formatted markup and retrieve the information you need—a process known affectionately as screen scraping.

Fetching Web Pages

Before you can parse an HTML-formatted web page, you of course have to acquire some. Here are some options for downloading content.

- You can use `urllib2`, or the even lower-level `httplib`, to construct an HTTP request that will return a web page. For each form that has to be filled out, you will have to build a dictionary representing the field names and data values inside; unlike a real web browser, these libraries will give you no help in submitting forms.
- You can to install mechanize and write a program that fills out and submits web forms much as you would do when sitting in front of a web browser. The downside is that, to benefit from this automation, you will need to download the page containing the form HTML before you can then submit it—possibly doubling the number of web requests you perform.
- If you need to download and parse entire web sites, take a look at the Scrapy project, hosted at <http://scrapy.org>, which provides a framework for implementing your own web spiders. With the tools it provides, you can write programs that follow links to every page on a web site, tabulating the data you want extracted from each page.
- When web pages wind up being incomplete because they use dynamic JavaScript to load data that you need, you can use the QtWebKit module of the PyQt4 library to load a page, let the JavaScript run, and then save or parse the resulting complete HTML page.
- Finally, if you really need a browser to load the site, both the Selenium and Windmill test platforms provide a way to drive a standard web browser from inside a Python program. You can start the browser up, direct it to the page of interest, fill out and submit forms, do whatever else is necessary to bring up the data you need, and then pull the resulting information directly from the DOM elements that hold them.

Downloading Pages Through Form Submission

The task of grabbing information from a web site usually starts by reading it carefully with a web browser and finding a route to the information you need.

Figure `fetch_urllib2.py` shows the site of the National Weather Service; for our first example, we will write a program that takes a city and state as arguments and prints out the current conditions, temperature, and humidity.

When using the `urllib2` module from the Standard Library, you will have to read the web page HTML manually to find the form. You can use the View Source command in your browser, search for the words “Local forecast,” and find the following form in the middle of the sea of HTML:

```
<form method="post" action="http://forecast.weather.gov/zipcity.php" ...>
...
<input type="text" id="zipcity" name="inputstring" size="9"
» value="City, St" onfocus="this.value='';" />
<input type="submit" name="Go2" value="Go" />
</form>
```

The only important elements here are the `<form>` itself and the `<input>` fields inside; everything else is just decoration intended to help human readers. This form does a POST to a particular URL with, it appears, just one parameter: an `inputstring` giving the city name and state. `fetch_urllib2.py` shows a simple Python program that uses only the Standard Library to perform this interaction, and saves the result to `phoenix.html`.

```
import urllib, urllib2
data = urllib.urlencode({'inputstring': 'Phoenix, AZ'})
info = urllib2.urlopen('http://forecast.weather.gov/zipcity.php', data)
content = info.read()
open('phoenix.html', 'w').write(content)
```

On the one hand, `urllib2` makes this interaction very convenient; we are able to download a forecast page using only a few lines of code. But, on the other hand, we had to read and understand the form ourselves instead of relying on an actual HTML parser to read it. The approach encouraged by `mechanize` is quite different: you need only the address of the opening page to get started, and the library itself will take responsibility for exploring the HTML and letting you know what forms are present. Here are the forms that it finds on this particular page:

```
>>> import mechanize
>>> br = mechanize.Browser()
>>> response = br.open('http://www.weather.gov/')
>>> for form in br.forms():
... print '%r %r %s' % (form.name, form.attrs.get('id'), form.action)
... for control in form.controls:
... print ' ', control.type, control.name, repr(control.value)
None None http://search.usa.gov/search
» hidden v:project 'firstgov'
» text query ''
» radio affiliate ['nws.noaa.gov']
» submit None 'Go'
None None http://forecast.weather.gov/zipcity.php
» text inputstring 'City, St'
» submit Go2 'Go'
'jump' 'jump' http://www.weather.gov/
» select menu ['http://www.weather.gov/alerts-beta/']
» button None None
```

Once we have determined that we need the `zipcity.php` form, we can write a program like that shown in `etch_mechanize.py`. You can see that at no point does it build a set of form fields manually itself, as was necessary in our previous listing. Instead, it simply loads the front page, sets the one field value that we care about, and then presses the form’s submit button. Note that since this HTML form did not specify a name, we had to create our own filter function—the

lambda function in the listing—to choose which of the three forms we wanted.

```
import mechanize
br = mechanize.Browser()
br.open('http://www.weather.gov/')
br.select_form(predicate=lambda(form): 'zipcity' in form.action)
br['inputstring'] = 'Phoenix, AZ'
response = br.submit()
content = response.read()
open('phoenix.html', 'w').write(content)
```

Many mechanize users instead choose to select forms by the order in which they appear in the page—in which case we could have called `select_form(nr=1)`. But I prefer not to rely on the order, since the real identity of a form is inherent in the action that it performs, not its location on a page.

The Structure of Web Pages

The Hypertext Markup Language (HTML) is one of many markup dialects built atop the Standard Generalized Markup Language (SGML), which bequeathed to the world the idea of using thousands of angle brackets to mark up plain text. Inserting bold and italics into a format like HTML is as simple as typing eight angle brackets:

- The `very` strange book `<i>Tristram Shandy</i>`. The **very** strange book *Tristram Shandy*.

In the terminology of SGML, the strings `` and `` are each tags—they are, in fact, an opening and a closing tag—and together they create an element that contains the text `very` inside it. Elements can contain text as well as other elements, and can define a series of key/value attribute pairs that give more information about the element:

- `<p content="personal">I am reading <i document="play">Hamlet</i>.</p>`
I am reading *Hamlet*.

The problem with SGML languages in this regard—and HTML is one particular example—is that they expect parsers to know the rules about which elements can be nested inside which other elements, and this leads to constructions like this unordered list ``, inside which are several list items `` :

- `FirstSecondThirdFourth`
 - First
 - Second
 - Third
 - Fourth

Since HTML in fact says that

- elements cannot nest, an HTML parser will understand the foregoing snippet to be equivalent to this more explicit XML string:

- `FirstSecondThirdFourth`
 - First
 - Second
 - Third
 - Fourth

And beyond this implicit understanding of HTML that a parser must possess are the twin problems that, first, various browsers over the years have varied wildly in how well they can reconstruct the document structure when given very concise or even deeply broken HTML; and, second, most web page authors judge the quality of their HTML by whether their browser of choice renders it correctly. This has resulted not only in a World Wide Web that is full of sites with invalid and broken HTML markup, but also in the fact that the permissiveness built into browsers has encouraged different flavors of broken HTML among their different user groups.

For more documentation about these topic visit:

- <http://www.w3.org/MarkUp/Guide/>
- <http://www.w3.org/MarkUp/Guide/Advanced.html>
- <http://www.w3.org/MarkUp/Guide/Style>
- <http://werbach.com/barebones/barebones.html>
- <http://www.w3.org/TR/REC-html40/>
- <http://validator.w3.org/>
- <http://tidy.sourceforge.net/>

Three Axes

Parsing HTML with Python requires three choices:

- The parser you will use to digest the HTML, and try to make sense of its tangle of opening and closing tags.
- The API(Application Programming Interface) by which your Python program will access the tree of concentric elements that the parser built from its analysis of the HTML page.
- What kinds of selectors you will be able to write to jump directly to the part of the page that interests you, instead of having to step into the hierarchy one element at a time.

The issue of selectors is a very important one, because a well-written selector can unambiguously identify an HTML element that interests you without your having to touch any of the elements above it in the document tree.

Now, I should pause for a second to explain terms like “deeper,” and I think the concept will be clearest if we reconsider the unordered list that was quoted in the previous section. An experienced web developer looking at that list rearranges it in her head, so that this is what it looks like:

- First
- Second
- Third
- Fourth

```
<ul>
<li>First</li>
<li>Second</li>
<li>Third</li>
<li>Fourth</li>
</ul>
```

Here the `` element is said to be a “parent” element of the individual list items, which “wraps” them and which is one level “above” them in the whole document. The `` elements are “siblings” of one another; each is a “child” of the `` element that “contains” them, and they sit “below” their parent in the larger document tree. This kind of spatial thinking winds up being very important for working your way into a document through an API.

In brief, here are your choices along each of the three axes that were just listed:

- The most powerful, flexible, and fastest parser at the moment appears to be the HTMLParser that comes with lxml; the next most powerful is the longtime favorite BeautifulSoup ; and coming in dead last are the parsing classes included with the Python Standard Library, which no one seems to use for serious screen scraping.
- The best API for manipulating a tree of HTML elements is ElementTree, which has been brought into the Standard Library for use with the Standard Library parsers, and is also the API supported by lxml; BeautifulSoup supports an API peculiar to itself; and a pair of ancient, ugly, event-based interfaces to HTML still exist in the Python Standard Library.
- The lxml library supports two of the major industry-standard selectors: CSS selectors and XPath query language; BeautifulSoup has a selector system all its own, but one that is very powerful and has powered countless web-scraping programs over the years.

Diving into an HTML Document

The tree of objects that a parser creates from an HTML file is often called a Document Object Model, or DOM, even though this is officially the name of one particular API defined by the standards bodies and implemented by browsers for the use of JavaScript running on a web page.

The task we have set for ourselves, you will recall, is to find the current conditions, temperature, and humidity in the phoenix.html page that we have downloaded

There are two approaches to narrowing your attention to the specific area of the document in which you are interested. You can either search the HTML for a word or phrase close to the data that you want, or, as we mentioned previously, use Google Chrome or Firefox with Firebug to “Inspect Element” and see the element you want embedded in an attractive diagram of the document tree.

To see how direct document-object manipulation would work in this case, we can load the raw page directly into both the lxml and BeautifulSoup systems.

```
>>> import lxml.etree
>>> parser = lxml.etree.HTMLParser(encoding='utf-8')
>>> tree = lxml.etree.parse('phoenix.html', parser)
```

The need for a separate parser object here is because, as you might guess from its name, lxml is natively targeted at XML files.

```
>>> from BeautifulSoup import BeautifulSoup
>>> soup = BeautifulSoup(open('phoenix.html'))
Traceback (most recent call last):
...
HTMLParseError: malformed start tag, at line 96, column 720
```

What on earth? Well, look, the National Weather Service does not check or tidy its HTML. Jumping to line 96, column 720 of phoenix.html, we see that there does indeed appear to be some broken HTML:

```
<a href="http://www.weather.gov"<u>www.weather.gov</u></a>
```

You can see that the `<u>` tag starts before a closing angle bracket has been encountered for the `<a>` tag. But why should BeautifulSoup care. I wonder what version I have installed.

```
>>> BeautifulSoup.__version__
'3.1.0'
```

Well, drat. I typed too quickly and was not careful to specify a working version when I ran pip to install BeautifulSoup into my virtual environment. Let's try again:

```
root@erlerobot:~/Python_files# pip install BeautifulSoup==3.0.8.1
```

Now, if we were to take the approach of starting at the top of the document and digging ever deeper until we find the node that we are interested in, we are going to have to generate some very verbose code. Here is the approach we would have to take with lxml:

```
>>> fonttag = tree.find('body').find('div').findall('table')[3] \
... .findall('tr')[1].find('td').findall('table')[1].find('tr') \
... .findall('td')[1].findall('table')[1].find('tr').find('td') \
... .find('table').findall('tr')[1].find('td').find('table') \
... .find('tr').find('td').find('font')
>>> fonttag.text
'\nA Few Clouds'
```

An attractive syntactic convention lets BeautifulSoup handle some of these steps more beautifully:

```
>>> fonttag = soup.body.div('table', recursive=False)[3] \
... ('tr', recursive=False)[1].td('table', recursive=False)[1].tr \
... ('td', recursive=False)[1]('table', recursive=False)[1].tr.td \
... .table('tr', recursive=False)[1].td.table \
... .tr.td.font
>>> fonttag.text
u'A Few Clouds71&deg;F(22&deg;C)'
```

BeautifulSoup lets you choose the first child element with a given tag by simply selecting the attribute `.tagname`, and lets you receive a list of child elements with a given tag name by calling an element like a function with the tag name and a recursive option telling it to pay attention just to the children of an element.

Both `lxml` and BeautifulSoup provide attractive ways to quickly grab a child element based on its tag name and position in the document. We clearly should not be using such primitive navigation to try descending into a real-world web page.

Figuring out how HTML elements are grouped, by the way, is much easier if you either view HTML with an editor that prints it as a tree, or if you run it through a tool like HTML tidy from W3C that can indent each tag to show you which ones are inside which other ones. `tidy` validate, correct, and pretty-print HTML files. You should use this command line:

```
tidy phoenix.html > phoenix-tidied.html
```

Selectors

A selector is a pattern that is crafted to match document elements on which your program wants to operate. Some of them are:

- People who are deeply XML-centric prefer XPath expressions, which are a companion technology to XML itself and let you match elements based on their ancestors, their own identity, and textual matches against their attributes and text content.
- If you are a web developer, then you probably link to CSS selectors as the most natural choice for examining HTML.
- Both lxml and BeautifulSoup, as we have seen, provide a smattering of their own methods for finding document elements.

Here are standards and descriptions for each of the selector styles just described:

- <http://www.w3.org/TR/xpath/>
- <http://codespeak.net/lxml/tutorial.html#using-xpath-to-find-text>
- <http://codespeak.net/lxml/xpathxslt.html>
- <http://www.w3.org/TR/CSS2/selector.html>
- <http://codespeak.net/lxml/cssselect.html>

And, finally, here are links to documentation that looks at selector methods peculiar to lxml and BeautifulSoup:

- <http://codespeak.net/lxml/tutorial.html#elementpath>
- <http://www.crummy.com/software/BeautifulSoup/documentation.html>

Now, here you have a completed weather scraper in the file `weather.py`:

```
import sys, urllib, urllib2
import lxml.etree
from lxml.cssselect import CSSSelector
from BeautifulSoup import BeautifulSoup

if len(sys.argv) < 2:
    print >>sys.stderr, 'usage: weather.py CITY, STATE'
    exit(2)

data = urllib.urlencode({'inputstring': ' '.join(sys.argv[1:])})
info = urllib2.urlopen('http://forecast.weather.gov/zipcity.php', data)
content = info.read()

# Solution #1 using CSSSelector
parser = lxml.etree.HTMLParser(encoding='utf-8')
tree = lxml.etree.fromstring(content, parser)
big = CSSSelector('td.big')(tree)[0]
if big.find('font') is not None:
    big = big.find('font')
print 'Condition:', big.text.strip()
print 'Temperature:', big.findall('br')[1].tail
tr = tree.xpath('..//td[b="Humidity"]')[0].getparent()
print 'Humidity:', tr.findall('td')[1].text
print

# Solution #2 using BeautifulSoup
soup = BeautifulSoup(content) # doctest: +SKIP
big = soup.find('td', 'big')
if big.font is not None:
    big = big.font
print 'Condition:', big.contents[0].string.strip()
temp = big.contents[3].string or big.contents[4].string # can be either
print 'Temperature:', temp.replace('&deg;', ' ')
tr = soup.find('b', text='Humidity').parent.parent.parent
print 'Humidity:', tr('td')[1].string
print
```

Take into account that for running this you also need to have the [lxml module](#) installed.

Web Applications

This chapter focuses on the actual act of programming. Every other issue that we consider will be in the service of this overarching goal: to create a new web service using Python as our language.

Web Servers and Python

Acceptable web site performance generally requires the ability to serve several users concurrently.

To avoid corrupting in-memory data structures, C Python employs a Global Interpreter Lock (GIL), so that only one thread in a multi-threaded program can actually be executing Python code at any given time. Thus Python will let you create as many threads as you want in a given process; however, only one thread can run code at a time, as though your threads were confined to a single processor.

A typical web application receives and parses the user's request, then makes a corresponding request to the database behind it; while that thread is waiting for a response from the database, the GIL is available for any other threads that need to run Python code. Finally the database answers; the waiting thread reacquires the GIL; and, in a quick blaze of CPU activity, the data is turned into an attractive web page, and the response is sent winging its way back to the user.

Thus threads can sometimes at least perform decently. Nevertheless, multiple processes are the more general way to scale. This is because, as a service gets bigger, additional processes can be brought up on additional machines, rather than being confined to a single machine. Threads, no matter their other merits, cannot do that! There are two general approaches to running a Python web application inside of a collector of identical worker processes:

- The Apache web server can be combined with the popular `mod_wsgi` module to host a separate Python interpreter in every Apache worker process.
- The web application can be run inside of either the flup server or the uWSGI server. Both of these servers will manage a pool of worker processes where each process hosts a Python interpreter running your application. The front-end web server can submit requests to flup using either the standard Fast CGI (FCGI) or Simple CGI (SCGI) protocol, while it has to speak to uWSGI in its own special "uwsgi" protocol (whose name is all lowercase to distinguish it from the name of the server).

Choosing a Web Server

All of the popular open source web servers can be used to serve Python web applications, so the full range of modern options is available:

- *Apache HTTP Server*: Since taking the lead as the most popular HTTP server back in 1996. Its stated goal is flexibility and modularity; it is reasonably fast, but it will not win speed records against more recent servers that focus only on speed. Its configuration files can be a bit long and verbose, but through them Apache offers very powerful options for applying different rules and behaviors to different directories and URLs. A variety of extension modules are available (many of which come bundled with it), and user directories can have separate `.htaccess` configuration files that make further adjustments to the main configuration.
- *nginx* (“engine X”): The nginx server has become a great favorite of organizations with a large volume of content that needs to be served quickly. It is considered fairly easy to configure. *lighttpd* (“lighty”): First written to demonstrate an architecture that could support tens of thousands of open client sockets (both nginx and Cherokee are also contenders in this class), this server is known for being very easy to configure. Some system administrators complain about its memory usage, but many others have observed no problems with it.
- *Cherokee*: Not only does this server offer performance that might edge out even nginx and lighttpd, but it lets you configure the server through a built-in web interface.

So to combine each of these servers with Python; for example in the case of Apache: the `mod_wsgi` module has a daemon mode where it internally runs your Python code inside a stack of dedicated server processes that are separate from Apache. Each Web Server Gateway Interface (WSGI) process can even run as a different user. If you really want to use Apache as your front end, this is one of the best options available.

But the most strongly recommended approach today is to set up one of the three fast servers to provide your static content, and then use one of the following three techniques to run your Python code behind them:

- Use HTTP proxying so that your `nginx`, `lighttpd`, or Cherokee front-end server delivers HTTP requests for dynamic web pages to a back-end Apache instance running `mod_wsgi`.
- Use the FastCGI protocol or SCGI protocol to talk to a flup instance running your Python code.
- Use the `uwsgi` protocol to talk to a uWSGI instance running your Python code.

At this point, you understand something of the larger context in which Python web applications are usually run; you are now ready to turn your attention to the task of programming.

WSGI

Integrating Python with web servers was much improved by the creation of PEP 333, which defines the Python Web Server Gateway Interface (WSGI): <http://legacy.python.org/dev/peps/pep-0333/>.

WSGI introduced a single calling convention that every web server could implement, thereby making that web server instantly compatible with all of the Python web applications and web frameworks that also support WSGI.

At the Python library you can get more information about the [wsgiref module](#). This module provides a variety of utility functions for working with WSGI environments. The `wsgiref` package, whose `simple_server` we will use in the example, also contains several utilities for working with WSGI. It includes functions for examining, further unpacking, and modifying the `environ` object; a prebuilt iterator for streaming large files back to the server; and even a `validate` sub-module whose routines can check a WSGI application to see whether it complies with the specification when presented with a series of representative requests.

Developers generally avoid writing raw WSGI applications because the conveniences of even a simple web framework make code so much easier to write and maintain. But, for the sake of illustration, `wsgi_app.py` shows a small WSGI application whose front page asks the user to type a string. Submitting the string takes the user to a second web page, where he can see its base64 encoding. From there, a link will take him back to the first page to repeat the process.

```
import cgi, base64
from wsgiref.simple_server import make_server

def page(content, *args):
    yield '<html><head><title>wsgi_app.py</title></head><body>'
    yield content % args
    yield '</body>'

def simple_app(environ, start_response):
    gohome = '<br><a href="/">Return to the home page</a>'
    q = cgi.parse_qs(environ['QUERY_STRING'])

    if environ['PATH_INFO'] == '/':

        if environ['REQUEST_METHOD'] != 'GET' or environ['QUERY_STRING']:
            start_response('400 Bad Request', [('Content-Type', 'text/plain')])
            return ['Error: the front page is not a form']

        start_response('200 OK', [('Content-Type', 'text/html')])
        return page('Welcome! Enter a string: <form action="encode">'
                    '<input name="mystring"><input type="submit"></form>')

    elif environ['PATH_INFO'] == '/encode':

        if environ['REQUEST_METHOD'] != 'GET':
            start_response('400 Bad Request', [('Content-Type', 'text/plain')])
            return ['Error: this form does not support POST parameters']

        if 'mystring' not in q or not q['mystring'][0]:
            start_response('400 Bad Request', [('Content-Type', 'text/plain')])
            return ['Error: this form requires a "mystring" parameter']

        my = q['mystring'][0]
        start_response('200 OK', [('Content-Type', 'text/html')])
        return page('<tt>%s</tt> base64 encoded is: <tt>%s</tt>' + gohome,
                    cgi.escape(repr(my)), cgi.escape(base64.b64encode(my)))

    else:
        start_response('404 Not Found', [('Content-Type', 'text/plain')])
        return ['That URL is not valid']

print 'Listening on localhost:8000'
make_server('localhost', 8000, simple_app).serve_forever()
```

```
import cgi, base64 from wsgiref.simple_server import make_server
```

```
def page(content, *args): yield ' ' yield content % args yield ' '
```

```
def simple_app(environ, start_response): gohome = 'Return to the home page'
q = cgi.parse_qs(environ['QUERY_STRING'])
```

```
if environ['PATH_INFO'] == '/':

    if environ['REQUEST_METHOD'] != 'GET' or environ['QUERY_STRING']:
        start_response('400 Bad Request', [('Content-Type', 'text/plain')])
        return ['Error: the front page is not a form']

    start_response('200 OK', [('Content-Type', 'text/html')])
    return page('Welcome! Enter a string: <form action="encode">'
                '<input name="mystring"><input type="submit"></form>')

elif environ['PATH_INFO'] == '/encode':

    if environ['REQUEST_METHOD'] != 'GET':
        start_response('400 Bad Request', [('Content-Type', 'text/plain')])
        return ['Error: this form does not support POST parameters']

    if 'mystring' not in q or not q['mystring'][0]:
        start_response('400 Bad Request', [('Content-Type', 'text/plain')])
        return ['Error: this form requires a "mystring" parameter']

    my = q['mystring'][0]
    start_response('200 OK', [('Content-Type', 'text/html')])
    return page('<tt>%s</tt> base64 encoded is: <tt>%s</tt>' + gohome,
                cgi.escape(repr(my)), cgi.escape(base64.b64encode(my)))

else:
    start_response('404 Not Found', [('Content-Type', 'text/plain')])
    return ['That URL is not valid']
```

The first thing to note in this code listing is that two very different objects are being created: a WSGI server that knows how to use HTTP to talk to a web browser and an application written to respond correctly when invoked per the WSGI calling convention. Note that these two pieces—the client and server—could easily be swapped out. This code example should make the calling convention clear enough:

- For each incoming request, the application is called with an environ object, giving it the details of the HTTP request and a live, callable, and named `start_response()`.
- Once the application has decided what HTTP response code and headers need to be returned, it makes a single call to `start_response()`. Its headers will be combined with any headers that the WSGI server might already provide to the client.
- Finally, the application needs only to return the actual content—either a list of strings or a generator yielding strings. Either way, the strings will be concatenated by the WSGI server to produce the response body that is transmitted back to the client. Generators are useful for cases where it would be unwise for an application to try loading all of the content (like large files) into memory at once.

WSGI Middleware

Standard interfaces like WSGI make it possible for developers to create wrappers—a design-patterns person would call these adapters—that accept a request from a server; modify, adjust, or record the request; and then call a normal WSGI application with the modified environment. Such middleware can also inspect and adjust the outgoing data stream; everything, in fact, is up for grabs, and essential arbitrary changes can be made both to the circumstances under which a WSGI application runs, as well as to the content that it returns.

- If several WSGI applications need to live at a single web site under different URLs, then a piece of middleware can be given the URLs. (you can read more in <http://pythonpaste.org/>)
- If each WSGI application on a web site were to keep its own list of passwords and honor only its own session cookies, then users would have to log in again each time they crossed an application boundary. By delegating authentication to WSGI middleware, applications can be relieved even of the duty to provide their own login page; instead, the middleware asks a user who lacks a session cookie to log in; once a user is authenticated, the middleware can pass along the user's identity to the applications by putting the user's information in the `environ` argument. Both `repoze.who` and `repoze.what` can help site integrators assert site-wide control over users and their permissions.
- Theming can be a problem when several small applications are combined to form a larger web site. This is because each application typically has its own approach to theming. This has led to the development of two competing tools, `xdv` and `Deliverance`, that let you build a single HTML theme and then provide simple rules that pull text out of your back-end applications and drop it into your theme in the right places.
- Debuggers can be created that call a WSGI application and, if an uncaught Python exception is raised, display an annotated traceback to support debugging. `WebError` actually provides the developer with a live, in-browser Python command line prompt for every level in a stack trace at which the developer can investigate a failure. Another popular tool is `repoze.profile`, which watches the application as it processes requests and produces a report on which functions are consuming the most CPU cycles.

If you are interested in what WSGI middleware is available, then you can visit this pair of sites to learn more:

- http://wsgi.org/wsgi/Middleware_and_Uutilities
- http://repoze.org/repoze_components.html#middleware

Today there are at least three major competing approaches in the Python community for crafting modular components that can be used to build web sites:

- The WSGI middleware approach thinks that code reuse can often best be achieved through a component stack, where each component uses WSGI to speak to the next. Here, all interaction has to somehow be made to fit the model of a dictionary of strings being handed down and then content being passed back up.
- Everything built atop the `Zope Toolkit` uses formal Design Pattern concepts like interfaces and factories to let components discover one another and be configured for operation. Thanks to adapters, components can often be used with widgets that were not originally designed with a given type of component in mind.
- Several web frameworks have tried to adopt conventions that would make it easy for third-party pieces of functionality to be added to an application easily. The `Django` community seems to have traveled the farthest in this direction, but it also looks as though it has encountered quite serious roadblocks in cases where a component needs to add its own tables to the database that have foreign-key relationships with user tables.

These examples illustrate an important fact: WSGI middleware is a good idea that has worked very well for a small class of problems where the idea of wrapping an application with concentric functionality makes solid sense. However, most web programmers seem to want to use more typical Python mechanisms like APIs, classes, and objects to combine their own code with existing components.

Python Web Frameworks

Now we are going to talk about an entirely different discipline: web application development.

Network programmers think about things like sockets, port numbers, protocols, packet loss, latency, framing, and encodings. Although all of these concepts must also be in the back of a web developer's mind, her actual attention is focused on a set of technologies so intricate and fast-changing that the actual packets and latencies are recalled to mind only when they are causing trouble. The web developer needs to think instead about HTML, GET, POST, forms, REST, CSS, JavaScript, Ajax, APIs, sprites, compression, and emerging technologies like HTML5 and WebSocket. The web site exists in her mind primarily as a series of documents that users will traverse to accomplish goals.

Web frameworks exist to help programmers step back from the details of HTTP—which is, after all, an implementation detail most users never even become aware of—and to write code that focuses on the nouns of web design. `wsgi_app.py` shows how even a very modest Python microframework can be used to reorient the attention of a web programmer.

You can install the framework `bottle` and run the listing once you have activated a virtual environment, like this:

The `bottle_app.py` :

```
import base64, bottle
bottle.debug(True)
app = bottle.Bottle()

@app.route('/encode')
@bottle.view('bottle_template.html')
def encode():
    mystring = bottle.request.GET.get('mystring')
    if mystring is None:
        bottle.abort(400, 'This form requires a "mystring" parameter')
    return dict(mystring=mystring, myb=base64.b64encode(mystring))

@app.route('/')
@bottle.view('bottle_template.html')
def index():
    return dict(mystring=None)

bottle.run(app=app, host='localhost', port=8080)
```

```
root@erlerobot:~/Python_files# pip install bottle
root@erlerobot:~/Python_files# python bottle_app.py
```

The `wsgi_app.py` :

```
import cgi, base64
from wsgiref.simple_server import make_server

def page(content, *args):
    yield '<html><head><title>wsgi_app.py</title></head><body>'
    yield content % args
    yield '</body>'

def simple_app(environ, start_response):
    gohome = '<br><a href="/">Return to the home page</a>'
    q = cgi.parse_qs(environ['QUERY_STRING'])

    if environ['PATH_INFO'] == '/':

        if environ['REQUEST_METHOD'] != 'GET' or environ['QUERY_STRING']:
            start_response('400 Bad Request', [('Content-Type', 'text/plain')])
            return ['Error: the front page is not a form']

        start_response('200 OK', [('Content-Type', 'text/html')])
        return page('Welcome! Enter a string: <form action="encode">'
                    '<input name="mystring"><input type="submit"></form>')
```

```

elif environ['PATH_INFO'] == '/encode':

    if environ['REQUEST_METHOD'] != 'GET':
        start_response('400 Bad Request', [('Content-Type', 'text/plain')])
        return ['Error: this form does not support POST parameters']

    if 'mystring' not in q or not q['mystring'][0]:
        start_response('400 Bad Request', [('Content-Type', 'text/plain')])
        return ['Error: this form requires a "mystring" parameter']

    my = q['mystring'][0]
    start_response('200 OK', [('Content-Type', 'text/html')])
    return page('<tt>%s</tt> base64 encoded is: <tt>%s</tt>' + gohome,
                cgi.escape(repr(my)), cgi.escape(base64.b64encode(my)))

else:
    start_response('404 Not Found', [('Content-Type', 'text/plain')])
    return ['That URL is not valid']

print 'Listening on localhost:8000'
make_server('localhost', 8000, simple_app).serve_forever()

```

In `bottle_app.py` the attention was on the single incoming HTTP request, and the branches in our logic explored all of the possible lifespans for that particular protocol request. `wsgi_app.py` changes the focus to the pages that actually exist on the site and giving each of these pages reasonable behaviors. The same tree of possibilities exists, but the tree exists implicitly thanks to the possible URLs defined in the code, not because the programmer has written a large if statement.

```

## The page template that goes with bottle_app.py.
##
<html><head><title>bottle_app.py</title></head>
<body>
    %if mystring is None:
        Welcome! Enter a string:
        <form action="encode"><input name="mystring"><input type="submit"></form>
    %else:
        <tt>{{mystring}}</tt> base64 encoded is: <tt>{{myb}}</tt><br>
        <a href="/">Return to the home page</a>
    %end
</body>

```

It might seem merely a pleasant convenience that we can use the `Bottle SimpleTemplate` to insert our variables into a web page and know that they will be escaped correctly. But the truth is that templates serve, just like schemes for URL dispatch, to re-orient our attention: instead of the resulting web page existing in our minds as what will result when the strings in our program listing are finally concatenated, we get to lay out its HTML intact, in order, and in a file that can actually take an `.html` extension and be highlighted and indented as HTML in our editor. The Python program will no longer impede our relationship with our markup.

And full-fledged Python frameworks abstract away even more implementation details. A very important feature they typically provide is data abstraction: instead of talking to a database using its raw APIs, a programmer can define models, laying out the data fields so they are easy to instantiate, search, and modify. And some frameworks can provide entire RESTful APIs that allow creation, inspection, modification, and deletion with PUT, GET, POST, and DELETE. The programmer merely needs to define the structure of his data document, and then name the URL at which the tree of REST objects should be based.

When looking for a web framework, you will find that the various frameworks differ on a few major points. The upcoming sections will walk you through what these points are, and how they might affect your development experience.

URL Dispatch Techniques

The various Python web frameworks tend to handle URL dispatch quite differently.

- Some small frameworks like [Bottle](#) and [Flask](#) let you create small applications by decorating a series of callables with URL patterns; small applications can then be combined later by placing them beneath one or more top-level applications.
- Others frameworks, like [Django](#), [Pylons](#), and [Werkzeug](#), encourage each application to define its URLs all in one place. This breaks your code into two levels, where URL dispatch happens in one location and rendering in another. This separation makes it easier to review all of the URLs that an application supports; it also means that you can attach code to new URLs without having to modify the functions themselves.
- Another approach has you define controllers, which are classes that represent some point in the URL hierarchy—say, the path `/cart`—and then write methods on the controller class named `view()` and `edit()` if you want to support sub-pages named `/cart/view` and `/cart/edit`. [CherryPy](#), [TurboGears2](#), and [Pylons](#) (if you use controllers instead of Routes) all support this approach. While determining later what URLs are supported can mean traversing a maze of different connected classes, this approach does allow for dynamic, recursive URL spaces that exist only at runtime as classes hand off dispatch requests based on live data about the site structure.
- A large community with its own conferences exists around the [Zope](#) framework.

The various mechanisms for URL dispatch can all be used to produce fairly clean design, and choosing from among them is largely a matter of taste.

Templates

Almost all web frameworks expect you to produce web pages by combining Python code called a view with an HTML template; you saw this approach in action in `wsgi_app.py`. This approach has gained traction because of its eminent maintainability: building a dictionary of information is best performed in plain Python code, and the items fetched and arranged by the view can then easily be included by the template, so long as the template language supports basic actions like iteration and some form of expression evaluation. (A template is a document consisting of rows and tables, with different ranges and sizes, which facilitates the development of web pages, letters or other content). It is one of the glories of Python that we use views and templates, and one of the shames of traditional PHP development that developers would freely intermix HTML and extensive PHP code to produce a single, unified mess.

Views can also become more testable when their only job is to generate a dictionary of data. A good framework will let you write tests that simply check the raw data returned by the function instead of making you peek repeatedly into fully rendered templates to see if the view corralled its data correctly.

There seem to be two major differences of opinion among the designers and users of the various template languages about what constitutes the best way to use templates:

- Should templates be valid HTML with iteration and expressions hidden in element attributes? Or should the template language use its own style of markup that festoons and wraps the literal HTML of the web page? While the former can let the developer run HTML validation against template files before they are ever rendered and be assured that rendering will not change the validator's verdict, most developers seem to find the latter approach much easier to read and maintain.
- Should templates allow arbitrary Python expressions in template code, or lock down the available options to primitive operations like dictionary get-item and object get-attribute? Many popular frameworks choose the latter option, requiring even lazy programmers to push complex operations into their Python code "where it belongs." But several template languages reason that, if Python programmers do so well without type checking, then maybe they should also be trusted with the choice of which expressions belong in the view and which in the template.

Since many Python frameworks let you plug in your template language of choice, and only a few of them lock you down to one option, you might find that you can pair your favorite approaches.

Pure-Python Web Servers

A fun way to demonstrate that Python comes with “batteries included” is to enter a directory on your system and run the [SimpleHTTPServer](#) Standard Library module as a stand-alone program:

```
root@erlerobot:~/Python_files# python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

If you direct your browser to localhost:8000, you will see the contents of this script's current directory displayed for browsing, such as the listings provided by Apache when a site leaves a directory browsable. Documents and images will load in your web browser when selected, based on the content types chosen through the best guesses of the [mimetypes](#) Standard Library module. The `mimetypes` module converts between a filename or URL and the MIME type associated with the filename extension. Conversions are provided from filename to MIME type and from MIME type to filename extension; encodings are not supported for the latter conversion.

Today, we use namespaces, callables, and duck-typed objects to provide much cleaner forms of extensibility. For example, today an object like `start_response` is provided as an argument (dependency injection), and the WSGI standard specifies its behavior rather than its inheritance tree (duck typing). The Standard Library includes two other HTTP servers:

- [CGIHTTPServer](#) takes the [SimpleHTTPServer](#) and, instead of just serving static files off of the disk, it adds the ability to run CGI scripts .
- [SimpleXMLRPCServer](#) and [DocXMLRPCServer](#) each provide a server endpoint against which client programs can make XML-RPC remote procedure calls. This protocol uses XML files submitted through HTTP requests.

Note that none of the preceding servers is typically intended for production use; instead, they are useful for small internal tasks for which you just need a quick HTTP endpoint to be used by other services internal to a system or subnet. And while most Python web frameworks will provide a way to run your application from the command line for debugging. These pure-Python web servers can be very useful if you are writing an application that users will be installing locally, and you want to provide a web interface without having to ship a separate web server like Apache or nginx.

Common Gateway Interface (CGI)

When the first experiments were taking place with dynamically generated web pages, a calling convention was necessary, and so the Common Gateway Interface (CGI) was defined. It allowed programs in all sorts of languages—C, the various Unix shells, awk, Perl, Python, PHP, and so forth—to be partners in generating dynamic content.

Today, the design of CGI is considered something of a disaster. Running a new process from scratch is just about the most expensive single operation that you can perform on a modern operating system, and requiring that this take place for every single incoming HTTP request is simply madness. You should avoid CGI under all circumstances. But it is possible you might someday have to connect Python code to a legacy HTTP server that does not support at least FastCGI or SCGI, so I will outline CGI's essential features. Three standard lines of communication that already existed between parent and child processes on Unix systems were used by web servers when invoking a CGI script:

- The Unix environment—a list of strings provided to each process upon its invocation that traditionally includes things like TZ=EST (the time zone) and COLUMNS=80 (user's screen width)—was instead stuffed full of information about the HTTP request that the CGI script was being called upon to answer. The various parts of the request's URL; the user agent string; basic information about the web server; and even a cookie could be included in the list of colon-separated keyvalue pairs.
- The standard input to the script could be read to end-of-file to receive whatever data had been submitted in the body of the HTTP request using POST. Whether a request was indeed a POST could be checked by examining the REQUEST_METHOD environment variable.
- Finally, the script would produce content, which it did by writing HTTP headers, a blank line, and then a response body to its standard output. To be a valid response, a Content-Type header was generally necessary at a minimum—though in its absence, some web servers would instead accept a Location header as a signal that they should send a redirect.

Should you ever need to run Python behind an HTTP server that only supports CGI, then I recommend that you use the [CGIHandler](#) module from the `wsgiref` Standard Library package (This is useful when you have a WSGI application and want to run it as a CGI script). This lets you use a normal Python web framework to write your service—or, alternatively, to roll up your sleeves and write a raw WSGI application—and then offer the HTTP server a CGI script, as shown here:

```
import CGIHandler, MyWSGIApp
my_wsgi_app = MyWSGIApp() # configuration necessary here?
CGIHandler().run(my_wsgi_app)
```

Be sure to check whether your web framework of choice already provides a way to invoke it as a CGI script; if so, your web framework will already know all of the steps involved in loading and configuring your application.

mod_python

As it became clear that CGI was both inefficient and inflexible—CGI scripts could not flexibly set the HTTP return code, for example—it became fashionable to start embedding programming languages directly in web servers.

Back in the early days, embedding was also possible, through a somewhat different approach that actually made Python an extension language for much of the internals of Apache itself. The module that supported this was `mod_python`, and for years it was by far the most popular way to connect Python to the World Wide Web. The `mod_python` Apache module put a Python interpreter inside of every worker process spawned by Apache. Programmers could arrange for their Python code to be invoked by writing directives into their Apache configuration.

Today, `mod_python` is mainly of historical interest. I have outlined its features here, not only because you might be called upon to maintain or upgrade a service that is still running on `mod_python`, but because it still provides unique Apache integration points where Python cannot get involved in any other way. If you run into either situation, you can find its documentation at <http://modpython.org/>

E-mail Composition and Decoding

Here, we will learn about the actual payload that is carried by all of the protocols involved in ways as a message is transmitted and received (Authenticated SMTP,POP,IMAP), that is, the format of e-mail messages themselves.

E-mail Messages

Each traditional e-mail message contains two distinct parts: headers and the body. Here is a very simple e-mail message so that you can see what the two sections look like:

```
From: Jane Smith <jsmith@example.com>
To: Alan Jones <ajones@example.com>
Subject: Testing This E-Mail Thing

Hello Alan,
This is just a test message. Thanks.
```

The first section is called the headers, which contain all of the metadata about the message, like the sender, the destination, and the subject of the message —everything except the text of the message itself. The body then follows and contains the message text itself. There are three basic rules of Internet e-mail formatting:

- At least during actual transmission, every line of an e-mail message should be terminated by the two-character sequence carriage return, newline, represented in Python by `'\r\n'`. E-mail clients running on your laptop or desktop machine tend to make different decisions about whether to store messages in this format, or replace these two-character line endings with whatever ending is native to your operating system.
- The first few lines of an e-mail are headers, which consist of a header name, a colon, a space, and a value. A header can be several lines long by indenting the second and following lines from the left margin as a signal that they belong to the header above them.
- The headers end with a blank line (that is, by two line endings back-to-back without intervening text) and then the message body is everything else that follows. The body is also sometimes called the payload.

The headers are there for the benefit of the person who reads the e-mail message, and the most important headers are these:

- **From:** This identifies the message sender. It can also, in the absence of a Reply-to header, be used as the destination when the reader clicks the e-mail client's "Reply" button.
- **Reply-To:** This sets an alternative address for replies, in case they should go to someone besides the sender named in the From header.
- **Subject:** This is a short several-word description of the e-mail's purpose, used by most clients when displaying whole mailboxes full of e-mail messages.
- **Date:** This is a header that can be used to sort a mailbox in the order in which emails arrived.
- **Message-ID and In-Reply-To:** Each ID uniquely identifies a message, and these IDs are then used in e-mail replies to specify exactly which message was being replied to. This can help sophisticated mail readers perform "threading," arranging messages so that replies are grouped directly beneath the messages to which they reply.

Composing Traditional Messages

How can we generate a traditional e-mail in Python without having to implement the formatting details ourselves? The answer is to use the modules within the powerful [email package](#). The email package is a library for managing email messages, including MIME and other RFC 2822-based message documents.

As our first example, `trad_gen_simple.py` shows a program that generates a simple message. Note that when you generate messages this way, manually setting the payload with the Message class, you should limit yourself to using plain 7-bit ASCII text.

```
from email.message import Message
text = """Hello,

This is a test message.

-- Anonymous"""

msg = Message()
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message'
msg.set_payload(text)

print msg.as_string()
```

The program is simple. It creates a Message object, sets the headers and body, and prints the result. When you run this program, you will get a nice formatted message with proper headers:

```
root@erlerobot:~/Python_files# python trad_gen_simple.py
To: recipient@example.com
From: Test Sender <sender@example.com>
Subject: Test Message

Hello,

This is a test message.

-- Anonymous
root@erlerobot:~/Python_files#
```

While technically correct, this message is actually a bit deficient when it comes to providing enough headers to really function in the modern world. For one thing, most e-mails should have a Date header, in a format specific to e-mail messages. Python provides an `email.utils.formatdate()` routine that will generate dates in the right format. You should add a Message-ID header to messages. This header should be generated in such a way that no other e-mail, anywhere in history, will ever have the same Message-ID. This might sound difficult, but Python provides a function to help do that as well: `email.utils.make_msgid()`. So take a look at `trad_gen_newhdrs.py`, which fleshes out our first sample program into a more complete example that sets these additional headers.

```
import email.utils
from email.message import Message

message = """Hello,

This is a test message.

-- Anonymous"""

msg = Message()
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message'
msg['Date'] = email.utils.formatdate(localtime = 1)
msg['Message-ID'] = email.utils.make_msgid()
```

```
msg.set_payload(message)

print msg.as_string()
```

If you run the program, you will notice two new headers in the output.

```
root@erlerobot:~/Python_files# python trad_gen_newhdrs.py
To: recipient@example.com
From: Test Sender <sender@example.com>
Subject: Test Message
Date: Mon, 14 Jul 2014 14:31:50 +0200
Message-ID: <20140714123150.987.14344@root-erlerobot.local>

Hello,

This is a test message.

-- Anonymous
root@erlerobot:~/Python_files#
```

Parsing Traditional Messages

What happens when you receive an incoming message as a raw block of text and want to look inside? Well, the `email` module also provides support for parsing e-mail messages, re-constructing the same `Message` object that would have been used to create the message in the first place. (Of course, it does not matter whether the e-mail you are parsing was originally created in Python through the `Message` class, or whether some other e-mail program created it; the format is standard, so Python's parsing should work either way.) After parsing the message, you can easily access individual headers and the body of the message using the same conventions as you used to create messages: headers look like the dictionary key-values of the `Message`, and the body can be fetched with a function.

A simple example of a parser is shown in `trad_parse.py`. All of the actual parsing takes place in the one-line function `message_from_file()`; everything else in the program listing is simply an illustration of how a `Message` object can be mined for headers and data.

```
import email

banner = '-' * 48
popular_headers = ('From', 'To', 'Subject', 'Date')
msg = email.message_from_file(open('message.txt'))
headers = sorted(msg.keys())

print banner
for header in headers:
    if header not in popular_headers:
        print header + ':', msg[header]
print banner
for header in headers:
    if header in popular_headers:
        print header + ':', msg[header]

print banner
if msg.is_multipart():
    print "This program cannot handle MIME multipart messages."
else:
    print msg.get_payload()
```

The output should be like this

```
root@erlerobot:~/Python_files# python trad_parse.py
-----
Message-ID: <20140714123150.987.14344@root-erlerobot.local>
-----
Date: Mon, 14 Jul 2014 14:33:54 +0200
From: Test Sender <sender@example.com>
Subject: Test Message, Chapter 12
To: recipient@example.com
-----
Hello,
This is a test message.
-- Anonymous
root@erlerobot:~/Python_files#
```

As you can see, the Python Standard Library makes it quite easy both to create and then to parse standard Internet e-mail messages. Note that the email package also offers a `message_from_string()` function that, instead of taking a file, can simply be handed the string containing an e-mail message.

Parsing Dates

The `email` package provides two functions that work together as a team to help you parse the Date field of e-mail messages, whose format you can see in the preceding example: a date and time, followed by a time zone expressed as hours and minutes (two digits each) relative to UTC. Countries in the eastern hemisphere experience sunrise early, so their time zones are expressed as positive numbers, like the following:

```
Date: Sun, 27 May 2007 11:34:43 +1000
```

Those of us in the western hemisphere have to wait longer for the sun to rise, so our time zones lag behind; Eastern Daylight Time, for example, runs four hours behind UTC:

```
Date: Sun, 27 May 2007 08:36:37 -0400
```

To figure out what moment of time is really meant by a Date header, simply call two functions in a row:

- Call `parsedate_tz()` to extract the time and time zone.
- Use `mktime_tz()` to add or subtract the time zone.
- The result will be a standard Unix timestamp.

For example, consider the two Date headers shown previously. If you just compared their bare times, the first date looks later: 11:34 a.m. is, after all, after 8:36 a.m. But the second time is in fact the much later one, because it is expressed in a time zone that is so much farther west. We can test this by using the functions previously named. First, turn the top date into a timestamp:

```
>>> from email.utils import parsedate_tz, mktime_tz
>>> timetuple1 = parsedate_tz('Sun, 27 May 2007 11:34:43 +1000')
>>> print timetuple1
(2007, 5, 27, 11, 34, 43, 0, 1, -1, 36000)
>>> timestamp1 = mktime_tz(timetuple1)
>>> print timestamp1
1180229683.0
Then turn the second date into a timestamp as well, and the dates can be compared directly:
>>> timetuple2 = parsedate_tz('Sun, 27 May 2007 08:36:37 -0400')
>>> timestamp2 = mktime_tz(timetuple2)
>>> print timestamp2
1180269397.0
>>> timestamp1 < timestamp2
True
```

If you have never seen a timestamp value before, they represent time very plainly: as the number of seconds that have passed since the beginning of 1970. You will find functions in Python's old `time` module for doing calculations with timestamps, and you will also find that you can turn them into normal Python `datetime` objects quite easily:

```
>>> from datetime import datetime
>>> datetime.fromtimestamp(timestamp2)
datetime.datetime(2007, 5, 27, 8, 36, 37)
```

In the real world, many poorly written e-mail clients generate their Date headers incorrectly. While the routines previously shown do try to be flexible when confronted with a malformed Date, they sometimes can simply make no sense of it and `parsedate_tz()` has to give up and return `None`. So when checking a real-world e-mail message for a date, remember to do it in three steps: first check whether a Date header is present at all; then be prepared for `None` to be returned when you parse it; and finally apply the time zone conversion to get a real timestamp that you can work with.

Understanding MIME

So far we have discussed e-mail messages that are plain text: the characters after the blank line that ends the headers are to be presented literally to the user as the content of the e-mail message. Today, only a fraction of the messages sent across the Internet are so simple.

The Multipurpose Internet Mail Extensions ([MIME](#)) standard is a set of rules for encoding data, rather than simple plain text, inside e-mails. MIME provides a system for things like attachments, alternative message formats, and text that is stored in alternate encodings. Because MIME messages have to be transmitted and delivered through many of the same old e-mail services that were originally designed to handle plain-text e-mails, MIME operates by adding headers to an e-mail message and then giving it content that looks like plain text to the machine but that can actually be decoded by an e-mail client into HTML, images, or attachments.

The most important features of MIME are, first, that MIME supports multipart messages. A normal e-mail message, as we have seen, contains some headers and a body. But a MIME message can squeeze several different parts into the message body. These parts might be things to be presented to the user in order, like a plain-text message, an image file attachment, and then a PDF attachment. Or, they could be alternative multipart, which represent the same content in different ways — usually, by encoding a message in both plain text and HTML. Second, MIME supports different transfer encodings. Traditional e-mail messages are limited to 7-bit data, which renders them unusable for international alphabets. MIME has several ways of transforming 8-bit data so it fits within the confines of e-mail systems:

- The “plain” encoding is the same as you would see in traditional messages, and passes 7-bit text unmodified.
- “Base-64” is a way of encoding raw binary data that turns it into normal alphanumeric data. Most of the attachments you send and receive —such as images, PDFs, and ZIP files —are encoded with base-64.
- “Quoted-printable” is a hybrid that tries to leave plain English text alone so that it remains readable in old mail readers, while also letting unusual characters be included as well.

MIME also provides content types, which tell the recipient what kind of content is present. For instance, a content type of `text/plain` indicates a plain-text message, while `image/jpeg` is a JPEG image.

How MIME works

You will recall that MIME messages must work within the limited plain-text framework of traditional email messages. To do that, the MIME specification defines some headers and some rules about formatting the body text.

For *non-multipart* messages that are a single block of data, MIME simply adds some headers to specify what kind of content the e-mail contains, along with its character set. But the body of the message is still a single piece, although it might be encoded with one of the schemes already described.

For *multipart* messages, things get trickier: MIME places a special marker in the e-mail body everywhere that it needs to separate one part from the next. Each part can then have its own limited set of headers —which occur at the start of the part —followed by data. By convention, the most basic content in an e-mail comes first (like a plain-text message, if one has been included), so that people without MIME-aware readers will see the plain text immediately without having to scroll down through dozens or hundreds of pages of MIME data.

Composing MIME Attachments

We will start by looking at how to create [MIME](#) messages. To compose a message with attachments, you will generally follow these steps:

1. Create a [MIMEMultipart](#) object and set its message headers.
2. Create a [MIMEText](#) object with the message body text and attach it to the MIMEMultipart object.
3. Create appropriate MIME objects for each attachment and attach them to the MIMEMultipart object.
4. Finally, call `as_string()` on the MIMEMultipart object to write out the resulting message.

Take a look at `mime_gen_basic.py` for a program that implements this algorithm. You can see that parts of the code look similar to logic that we used to generate a traditional e-mail. After creating the message and its text body, the program loops over each file given on the command line and attaches it to the growing message.

```
from email.mime.base import MIMEBase
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email import utils, encoders
import mimetypes, sys

def attachment(filename):
    fd = open(filename, 'rb')
    mimetype, mimeencoding = mimetypes.guess_type(filename)
    if mimeencoding or (mimetype is None):
        mimetype = 'application/octet-stream'
    maintype, subtype = mimetype.split('/')
    if maintype == 'text':
        retval = MIMEText(fd.read(), _subtype=subtype)
    else:
        retval = MIMEBase(maintype, subtype)
        retval.set_payload(fd.read())
        encoders.encode_base64(retval)
        retval.add_header('Content-Disposition', 'attachment',
            filename = filename)
    fd.close()
    return retval

message = """Hello,

This is a test message.

-- Anonymous"""

msg = MIMEMultipart()
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message'
msg['Date'] = utils.formatdate(localtime = 1)
msg['Message-ID'] = utils.make_msgid()

body = MIMEText(message, _subtype='plain')
msg.attach(body)
for filename in sys.argv[1:]:
    msg.attach(attachment(filename))
print msg.as_string()
```

The `attachment()` function does the work of creating a message attachment object. First, it determines the MIME type of each file by using Python's built-in [mimetypes](#) module. If the type can't be determined, or it will need a special kind of encoding, then a type is declared that promises only that the data is made of a "stream of octets" (sequence of bytes) but without any further promise about what they mean. If the file is a text document whose MIME type starts with `text/`, a [MIMEText](#) object is created to handle it; otherwise, a [MIMEBase](#) generic object is created. In the latter case, the contents are assumed to be binary, so they are encoded with base-64. Finally, an appropriate Content-Disposition header is added to that section of the MIME file so that mail readers will know that they are dealing with an attachment.

The result of running this program is shown below :

```

root@erlerobot:~/Python_files# echo "This is a test" > test.txt
root@erlerobot:~/Python_files# gzip < test.txt > test.txt.gz
root@erlerobot:~/Python_files# python mime_gen_basic.py test.txt test.txt.gz
Content-Type: multipart/mixed; boundary="====1623374356=="
MIME-Version: 1.0
To: recipient@example.com
From: Test Sender <sender@example.com>
Subject: Test Message
Date: Mon, 14 Jul 2014 14:36:07 +0200
Message-ID: <20140714123150.987.14344@root-erlerobot.local>
--====1623374356==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Hello,
This is a test message.
-- Anonymous
--====1623374356==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename="test.txt"
This is a test
--====1623374356==
Content-Type: application/octet-stream
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="test.txt.gz"
H4sIAP3o2D8AAwvJyCwWAKJEhZLU4hIuAIwtwPoPAAAA
--====1623374356=== -

```

The message starts off looking quite similar to the traditional ones we created earlier; you can see familiar headers like To, From, and Subject just like before. Note the Content-Type line, however: it indicates multipart/mixed. That tells the mail reader that the body of the message contains multiple MIME parts, and that the string containing equals signs will be the separator between them. Next comes the message's first part. Notice that it has its own Content-Type header! The second part looks similar to the first, but has an additional Content-Disposition header; this will signal most e-mail readers that the part should be displayed as a file that the user can save rather than being immediately displayed to the screen. Finally comes the part containing the binary file, encoded with base-64, which makes it not directly readable.

MIME Alternative Parts

MIME “alternative” parts let you generate multiple versions of a single document. The user’s mail reader will then automatically decide which one to display, depending on which content type it likes best; some mail readers might even show the user radio buttons, or a menu, and let them choose. The process of creating alternatives is similar to the process for attachments, and is illustrated in `mime_gen_alt.py` :

```
from email.mime.base import MIMEBase
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email import utils, encoders

def alternative(data, contenttype):
    maintype, subtype = contenttype.split('/')
    if maintype == 'text':
        retval = MIMEText(data, _subtype=subtype)
    else:
        retval = MIMEBase(maintype, subtype)
        retval.set_payload(data)
        encoders.encode_base64(retval)
    return retval

messagetext = """Hello,

This is a *great* test message.

-- Anonymous"""
messagehtml = """Hello,<P>
This is a <B>great</B> test message from Chapter 12. I hope you enjoy
it!<P>
-- <I>Anonymous</I>"""

msg = MIMEMultipart('alternative')
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message, Chapter 12'
msg['Date'] = utils.formatdate(localtime = 1)
msg['Message-ID'] = utils.make_msgid()

msg.attach(alternative(messagetext, 'text/plain'))
msg.attach(alternative(messagehtml, 'text/html'))
print msg.as_string()
```

Notice the differences between an alternative message and a message with attachments! With the alternative message, no Content-Disposition header is inserted. Also, the MIMEMultipart object is passed the alternative subtype to tell the mail reader that all objects in this multipart are alternative views of the same thing. Note again that it is always most polite to include the plain-text object first for people with ancient or incapable mail readers, which simply show them the entire message as text.

Composing Non-English Headers

Although you have seen how MIME can encode message body parts with base-64 to allow 8-bit data to pass through, that does not solve the problem of special characters in headers. For instance, if your name was Michael Müller (with an umlaut over the “u”), you would have trouble representing your name accurately in your own alphabet. The “u” would come out bare. Therefore, MIME provides a way to encode data in headers. Take a look at `mime_headers.py` for how to do it in Python.

```
from email.mime.text import MIMEText
from email.header import Header

message = """Hello,

This is a test message .

-- Anonymous"""

msg = MIMEText(message)
msg['To'] = 'recipient@example.com'
fromhdr = Header()
fromhdr.append(u"Michael M\xfc1ler")
fromhdr.append('<mmueller@example.com>')
msg['From'] = fromhdr
msg['Subject'] = 'Test Message'

print msg.as_string()
```

The code `\xfc` in the Unicode string (strings in Python source files that are prefixed with `u` can contain arbitrary Unicode characters, rather than being restricted to characters whose value is between 0 and 255).

```
root@erlerobot:~/Python_files# python mime_headers.py
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
To: recipient@example.com
From: =?iso-8859-1?q?Michael_M=Fc1ler?= <mmueller@example.com>
Subject: Test Message
Date: Mon, 14 Jul 2014 14:46:33 +0200
Message-ID: <20140714123150.987.14344@root-erlerobot.local>
Hello,
This is a test message.
-- Anonymous
```

Composing Nested Multiparts

Now that you know how to generate a message with alternatives and one with attachments, you may be wondering how to do both. To do that, you create a standard multipart for the main message. Then you create a multipart/alternative inside that for your body text, and attach your message formats to it. Finally, you attach the various files. Take a look at

`mime_gen_both.py` for the complete solution.

```
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.mime.base import MIMEBase
from email import utils, encoders
import mimetypes, sys

def genpart(data, contenttype):
    maintype, subtype = contenttype.split('/')
    if maintype == 'text':
        retval = MIMEText(data, _subtype=subtype)
    else:
        retval = MIMEBase(maintype, subtype)
        retval.set_payload(data)
        encoders.encode_base64(retval)
    return retval

def attachment(filename):
    fd = open(filename, 'rb')
    mimetype, mimeencoding = mimetypes.guess_type(filename)
    if mimeencoding or (mimetype is None):
        mimetype = 'application/octet-stream'
    retval = genpart(fd.read(), mimetype)
    retval.add_header('Content-Disposition', 'attachment',
                     filename = filename)
    fd.close()
    return retval

messagetext = """Hello,

This is a *great* test message from Chapter 12. I hope you enjoy it!

-- Anonymous"""
messagehtml = """Hello,<P>
This is a <B>great</B> test message <P>
-- <I>Anonymous</I>"""

msg = MIMEMultipart()
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message'
msg['Date'] = utils.formatdate(localtime = 1)
msg['Message-ID'] = utils.make_msgid()

body = MIMEMultipart('alternative')
body.attach(genpart(messagetext, 'text/plain'))
body.attach(genpart(messagehtml, 'text/html'))
msg.attach(body)

for filename in sys.argv[1:]:
    msg.attach(attachment(filename))
print msg.as_string()
```

Parsing MIME Messages

Python's `email` module can read a message from a file or a string, and generate the same kind of inmemory object tree that we were generating ourselves in the aforementioned listings. To understand the e-mail's content, all you have to do is step through its structure. Show an example at `mime_structure.py`:

```
import sys, email

def printmsg(msg, level = 0):
    prefix = "| " * level
    prefix2 = prefix + "|"
    print prefix + "+ Message Headers:"
    for header, value in msg.items():
        print prefix2, header + ":", value
    if msg.is_multipart():
        for item in msg.get_payload():
            printmsg(item, level + 1)

msg = email.message_from_file(sys.stdin)
printmsg(msg)
```

This program is short and simple. For each object it encounters, it checks to see if it is multipart; if so, the children of that object are displayed as well. Individual parts of a message can easily be extracted. You will recall that there are several ways that message data may be encoded; fortunately, the email module can decode them all! `mime_decode.py` shows a program that will let you decode and save any component of a MIME message:

```
import sys, email
counter = 0
parts = []

def printmsg(msg, level = 0):
    global counter
    l = "| " * level
    if msg.is_multipart():
        print l + "Found multipart:"
        for item in msg.get_payload():
            printmsg(item, level + 1)
    else:
        disp = ['%d. Decodable part' % (counter + 1)]
        if 'content-type' in msg:
            disp.append(msg['content-type'])
        if 'content-disposition' in msg:
            disp.append(msg['content-disposition'])
        print l + ", ".join(disp)
        counter += 1
        parts.append(msg)

inputfd = open(sys.argv[1])
msg = email.message_from_file(inputfd)
printmsg(msg)

while 1:
    print "Select part number to decode or q to quit: "
    part = sys.stdin.readline().strip()
    if part == 'q':
        sys.exit(0)
    try:
        part = int(part)
        msg = parts[part - 1]
    except:
        print "Invalid selection."
        continue

    print "Select file to write to:"
    filename = sys.stdin.readline().strip()
    try:
        fd = open(filename, 'wb')
    except:
        print "Invalid filename."
        continue
```

```
fd.write(msg.get_payload(decode = 1))
```

This program steps through the message, like the last example. We skip asking the user about message components that are multipart because those exist only to contain other message objects, like text and attachments; multipart sections have no actual payload of their own.

Decoding Headers

The last trick that we should cover regarding MIME messages is decoding headers that may have been encoded with foreign languages. The function `decode_header()` takes a single header and returns a list of pieces of the header; each piece is a binary string together with its encoding (named as a string if it is something besides 7-bit ASCII, else the value `None`):

```
>>> x = '?iso-8859-1?q?Michael_M=Fc1ler?=' <mmueller@example.com>'
>>> import email.header
>>> pieces = email.header.decode_header(x)
>>> print pieces
[('Michael M\xfc1ler', 'iso-8859-1'), ('<mmueller@example.com>', None)]
```

Of course, this raw information is likely to be of little use to you. To instead see the actual text inside the encoding, use the `decode()` function of each binary string in the list (falling back to an `'ascii'` encoding if `None` was returned) and paste the result together with spaces:

```
>>> print ' '.join( s.decode(enc or 'ascii') for s,enc in pieces )
Michael Müller <mmueller@example.com>
```

It is always good practice to use `decode_header()` on any of the “big three” headers —From, To, and Subject —before displaying them to the user. If no special encoding was used, then the result will simply be a one-element list containing the header string with a `None` encoding.

Simple Mail Transport Protocol (SMTP)

The actual movement of e-mail between systems is accomplished through SMTP: the “Simple Mail Transport Protocol.” In this chapter we will analyze SMTP in depth.

E-mail Clients, Webmail Services

The role of SMTP in message submission, where the user presses “Send” and expects a message to go winging its way across the Internet, will probably be least confusing if we trace the history of how users have historically worked with Internet mail. The key concept to understand as we begin this history is that users have never been asked to sit around and wait for an e-mail message to actually be delivered. This process can often take quite a bit of time—and up to several dozen repeated attempts—before an e-mail message is actually delivered to its destination. Any number of things could cause delays: a message could have to wait because other messages are already being transmitted across a link of limited bandwidth; the destination server might be down for a few hours, or its network might not be currently accessible because of a glitch; and if the mail is destined for a large organization, then it might have to make several different “hops” as it arrives at the big university server, then is directed to a smaller college e-mail machine, and then finally is directed to a departmental e-mail server.

The role of SMTP in message submission, where the user presses “Send” and expects a message to go winging its way across the Internet, will probably be least confusing if we trace the history of how users have historically worked with Internet mail. The key concept to understand as we begin this history is that users have never been asked to sit around and wait for an e-mail message to actually be delivered. This process can often take quite a bit of time—and up to several dozen repeated attempts—before an e-mail message is actually delivered to its destination. Any number of things could cause delays: a message could have to wait because other messages are already being transmitted across a link of limited bandwidth; the destination server might be down for a few hours, or its network might not be currently accessible because of a glitch; and if the mail is destined for a large organization, then it might have to make several different “hops” as it arrives at the big university server, then is directed to a smaller college e-mail machine, and then finally is directed to a departmental e-mail server.

E-mail browsing and submission, therefore, become a black box: your browser interacts with a web API, and on the other end, you will see plain old SMTP connections originating from and going to the large organization as mail is delivered in each direction. But in the world of webmail, client protocols are removed from the equation, taking us back to the old days of pure server-to-server unauthenticated SMTP.

How SMTP Is Used

The foregoing narrative has hopefully helped you structure your thinking about Internet e-mail protocols, and realize how they fit together in the bigger picture of getting messages to and from users. But the subject of this chapter is a narrower one—the Simple Mail Transport Protocol in particular. And we should start by stating the basics:

- SMTP is a TCP/IP-based protocol.
- Connections can be authenticated, or not.
- Connections can be encrypted, or not.

Most e-mail connections across the Internet these days seem to lack any attempt at encryption, which means that whoever owns the Internet backbone routers are theoretically in a position to read simply staggering amounts of other people's mail.

What are the two ways that SMTP is used? First, SMTP can be used for e-mail submission between a client e-mail program like Thunderbird or Outlook, claiming that a user wants to send e-mail, and a server at an organization that has given that user an e-mail address. These connections generally use authentication, so that spammers cannot connect and send millions of messages on a user's behalf without his or her password. Once received, the server puts the message in a queue for delivery (and often makes its first attempt at sending it moments later), and the client can forget about the message and presume the server will keep trying to deliver it. Second, SMTP is used between Internet mail servers as they move e-mail from its origin to its destination. This typically involves no authentication; after all, big organizations like Google, Yahoo!, and Microsoft do not know the passwords of each other's users, so when Yahoo! receives an e-mail from Google claiming that it was sent from an @gmail.com user, Yahoo! just has to believe them (or not—sometimes organizations blacklist each other if too much spam is making it through their servers, as happened to a friend of mine the other day when Hotmail stopped accepting his client's newsletters from GoDaddy's servers because of alleged problems with spam).

So, typically, no authentication takes place between servers talking SMTP to each other—and even encryption against snooping routers seems to be used only rarely. Because of the problem of spammers connecting to e-mail servers and claiming to be delivering mail from another organization's users, there has been an attempt made to lock down who can send email on an organization's behalf. Though controversial, some e-mail servers consult the Sender Policy Framework (SPF), defined in RFC 4408, to see whether the server they are talking to really has the authority to deliver the e-mails it is transmitting. But the SPF and other anti-spam technologies are unfortunately beyond the scope of this book, which must limit itself to the question of using the basic protocols themselves from Python. So we now turn to the more technical question of how you will actually use SMTP from your Python programs.

Sending E-Mail

Successfully sending e-mail generally requires a queue where a message can sit for seconds, minutes, or days until it can be successfully transmitted toward its destination. So you typically do not want your programs using Python's [smtplib](#) to send mail directly to a message's destination—because if your first transmission attempt fails, then you will be stuck with the job of writing a full “mail transfer agent” (MTA), as the RFCs call an e-mail server, and give it a full standards-compliant re-try queue. This is not only a big job, but also one that has already been done well several times, and you will be wise to take advantage of one of the existing MTAs (look at postfix, exim, and qmail) before trying to write something of your own.

So only rarely will you be making SMTP connections out into the world from Python. More usually, your system administrator will tell you one of two things:

- That you should make an authenticated SMTP connection to an existing e-mail server, using a username and password that will belong to your application, and give it permission to use the e-mail server to queue outgoing messages
- That you should run a local binary on the system—like the `sendmail` program— that the system administrator has already gone to the trouble to configure so that local programs can send mail.

Introducing the SMTP Library

Python's built-in SMTP implementation is in the Python Standard Library module `smtplib`. Python's built-in SMTP implementation is in the Python Standard Library module `smtplib`, which makes it easy to do simple tasks with SMTP.

In the examples that follow, the programs are designed to take several command-line arguments: the name of an SMTP server, a sender address, and one or more recipient addresses. Please use them cautiously; name only an SMTP server that you yourself run or that you know will be happy receiving your test messages, lest you wind up getting an IP address banned for sending spam! If you don't know where to find an SMTP server, you might try running a mail daemon like `postfix` or `exim` locally and then pointing these example programs at `localhost`. Many UNIX, Linux, and Mac OS X systems have an SMTP server like one of these already listening for connections from the local machine.

Otherwise, consult your network administrator or Internet provider to obtain a proper hostname and port. Note that you usually cannot just pick a mail server at random; many store or forward mail only from certain authorized clients. So, take a look at `simple.py` for a very simple SMTP program:

```
import sys, smtplib

if len(sys.argv) < 4:
    print "usage: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
    sys.exit(2)

server, fromaddr, toaddrs = sys.argv[1], sys.argv[2], sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,

This is a test message sent to you from the simple.py program.
""" % ('', '.join(toaddrs), fromaddr)

s = smtplib.SMTP(server)
s.sendmail(fromaddr, toaddrs, message)

print "Message successfully sent to %d recipient(s)" % len(toaddrs)
```

So, take a look at `simple.py` for a very simple SMTP program.

```
python
import sys, smtplib

if len(sys.argv) < 4:
    print "usage: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
    sys.exit(2)

server, fromaddr, toaddrs = sys.argv[1], sys.argv[2], sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,

This is a test message sent to you from the simple.py program.
""" % ('', '.join(toaddrs), fromaddr)

s = smtplib.SMTP(server)
s.sendmail(fromaddr, toaddrs, message)

print "Message successfully sent to %d recipient(s)" % len(toaddrs)
```

It starts by generating a simple message from the user's command-line arguments. Then it creates an `smtplib.SMTP` object that connects to the specified server. Finally, all that's required is a call to `sendmail()`. If that returns successfully, then you

know that the message was sent.

When you run the program, it will look like this:

```
root@erlerobot:~/Python_files# python simple.py localhost sender@example.com recipient@example.com
Message successfully sent to 2 recipient(s)
```

Thanks to the hard work that the authors of the Python Standard Library have put into the `sendmail()` method, it might be the only SMTP call you ever need.

Error Handling and Conversation Debugging

There are several different exceptions that might be raised while you're programming with `smtplib`. They are:

- `socket.gaierror` for errors looking up address information.
- `socket.error` for general I/O and communication problems.
- `socket.herror` for other addressing errors.
- `smtplib.SMTPException` or a subclass of it for SMTP conversation problems.

The `smtplib` module also provides a way to get a series of detailed messages about the steps it takes to send an e-mail. To enable that level of detail, you can call `smtpobj.set_debuglevel(1)`. With this option, you should be able to track down any problems. Take a look at `debug.py` for an example program that provides basic error handling and debugging.

```
import sys, smtplib, socket

if len(sys.argv) < 4:
    print "usage: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
    sys.exit(2)

server, fromaddr, toaddrs = sys.argv[1], sys.argv[2], sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,

This is a test message sent to you from the debug.py program.
""" % (' '.join(toaddrs), fromaddr)

try:
    s = smtplib.SMTP(server)
    s.set_debuglevel(1)
    s.sendmail(fromaddr, toaddrs, message)
except (socket.gaierror, socket.error, socket.herror,
        smtplib.SMTPException), e:
    print " *** Your message may not have been sent!"
    print e
    sys.exit(1)
else:
    print "Message successfully sent to %d recipient(s)" % len(toaddrs)
```

This program looks similar to the last one. However, the output will be very different.

```
root@erlerobot:~/Python_files# python debug.py localhost foo@example.com jgoerzen@complete.org
send: 'ehlo localhost\r\n'
reply: '250-localhost\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-SIZE 20480000\r\n'
reply: '250-VRFY\r\n'
reply: '250-ETRN\r\n'
reply: '250-STARTTLS\r\n'
...
Message successfully sent to 1 recipient(s)
```

From this example, you can see the conversation that `smtplib` is having with the SMTP server over the network. Let's look at what's happening: First, the client (the `smtplib` library) sends an EHLO command (an "extended" successor to a more ancient command that was named, more readably, HELO) with your hostname in it. The remote server responds with its hostname, and lists any optional SMTP features that it supports. Next, the client sends the mail from command, which states the "envelope sender" e-mail address and the size of the message. The server at this moment has the opportunity to reject the message (for example, because it thinks you are a spammer); but in this case, it responds with 250 Ok. (Note

that in this case, the code 250 is what matters; the remaining text is just a human-readable comment and varies from server to server.) Then the client sends a `rcpt` command, with the “envelope recipient” that we talked so much about earlier in this chapter; you can finally see that, indeed, it is transmitted separately from the text of the message itself when using the SMTP protocol. If you were sending the message to more than one recipient, they would each be listed on the `rcpt` to line. Finally, the client sends a `data` command, transmits the actual message (using verbose carriage return-linefeed line endings, you will note, per the Internet e-mail standard), and finishes the conversation.

The `smtplib` module is doing all this automatically for you in this example. In the rest of the chapter, we will look at how to take more control of the process so you can take advantage of some more advanced features.

Getting Information from EHLO

Sometimes it is nice to know about what kind of messages a remote SMTP server will accept. For instance, most SMTP servers have a limit on what size message they permit, and if you fail to check first, then you may transmit a very large message only to have it rejected when you have completed transmission.

Some servers do not support ESMTP. On those servers, EHLO will just return an error. In that case, you must send a HELO command instead. In the previous examples, we used `sendmail()` immediately after creating our SMTP object, so `smtplib` had to send its own “hello” message to the server. But if it sees you attempt to send the EHLO or HELO command on your own, then `sendmail()` will no longer attempt to send these commands itself. `ehlo.py` shows a program that gets the maximum size from the server, and returns an error before sending if a message would be too large.

```
import sys, smtplib, socket

if len(sys.argv) < 4:
    print "usage: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
    sys.exit(2)

server, fromaddr, toaddrs = sys.argv[1], sys.argv[2], sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,

This is a test message sent to you from the ehlo.py program.
""" % ('', '.join(toaddrs), fromaddr)

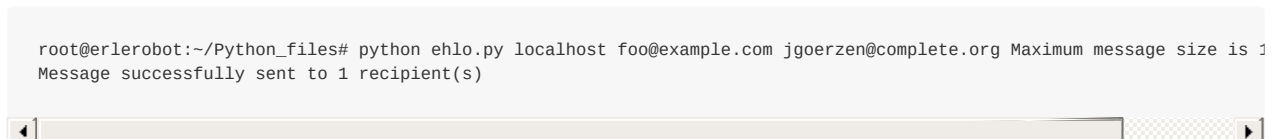
try:
    s = smtplib.SMTP(server)
    code = s.ehlo()[0]
    uses_esmtp = (200 <= code <= 299)
    if not uses_esmtp:
        code = s.helo()[0]
        if not (200 <= code <= 299):
            print "Remote server refused HELO; code:", code
            sys.exit(1)

    if uses_esmtp and s.has_extn('size'):
        print "Maximum message size is", s.esmtp_features['size']
        if len(message) > int(s.esmtp_features['size']):
            print "Message too large; aborting."
            sys.exit(1)

    s.sendmail(fromaddr, toaddrs, message)

except (socket.gaierror, socket.error, socket.herror,
        smtplib.SMTPException), e:
    print " *** Your message may not have been sent!"
    print e
    sys.exit(1)
else:
    print "Message successfully sent to %d recipient(s)" % len(toaddrs)
```

If you run this program, and the remote server provides its maximum message size, then the program will display the size on your screen and verify that its message does not exceed that size before sending. Here is what running this program might look like:



```
root@erlerobot:~/Python_files# python ehlo.py localhost foo@example.com jgoerzen@complete.org Maximum message size is 1
Message successfully sent to 1 recipient(s)
```

Take a look at the part of the code that verifies the result from a call to `ehlo()` or `helo()`. Those two functions return a list; the first item in the list is a numeric result code from the remote SMTP server.

Using Secure Sockets Layer and Transport Layer Security

E-mails sent in plain text over SMTP can be read by anyone with access to an Internet gateway or router across which the packets happen to pass. The best solution to this problem is to encrypt each e-mail with a public key whose private key is possessed only by the person to whom you are sending the e-mail; there are freely available systems such as PGP and GPG for doing exactly this. But regardless of whether the messages themselves are protected, individual SMTP conversations between particular pairs of machines can be encrypted and authenticated using a method known as SSL/TLS.

The general procedure for using TLS in SMTP is as follows:

1. Create the SMTP object, as usual.
2. Send the EHLO command. If the remote server does not support EHLO, then it will not support TLS.
3. Check `s.has_extn()` to see if `starttls` is present. If not, then the remote server does not support TLS and the message can only be sent normally, in the clear.
4. Call `starttls()` to initiate the encrypted channel.
5. Call `ehlo()` a second time; this time, it's encrypted.
6. Finally, send your message.

The first question you have to ask yourself when working with TLS is whether you should return an error if TLS is not available. Depending on your application, you might want to raise an error for any of the following:

- There is no support for TLS on the remote side.
- The remote side fails to establish a TLS session properly.
- The remote server presents a certificate that cannot be validated.

`tls.py` acts as a TLS-capable general-purpose client. It will connect to a server and use TLS if it can; otherwise, it will fall back and send the message as usual. (But it will die with an error if the attempt to start TLS fails while talking to an ostensibly capable server).

```
import sys, smtplib, socket

if len(sys.argv) < 4:
    print "Syntax: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
    sys.exit(2)

server, fromaddr, toaddrs = sys.argv[1], sys.argv[2], sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,

This is a test message sent to you from the tls.py program
in Foundations of Python Network Programming.
""" % ('', '.join(toaddrs), fromaddr)

try:
    s = smtplib.SMTP(server)
    code = s.ehlo()[0]
    uses_esmtp = (200 <= code <= 299)
    if not uses_esmtp:
        code = s.helo()[0]
        if not (200 <= code <= 299):
            print "Remote server refused HELO; code:", code
            sys.exit(1)

    if uses_esmtp and s.has_extn('starttls'):
        print "Negotiating TLS..."
        s.starttls()
        code = s.ehlo()[0]
        if not (200 <= code <= 299):
```

```
        print "Couldn't EHLO after STARTTLS"
        sys.exit(5)
    print "Using TLS connection."
else:
    print "Server does not support TLS; using normal connection."
    s.sendmail(fromaddr, toaddrs, message)

except (socket.gaierror, socket.error, socket.herror,
        smtpplib.SMTPException), e:
    print " *** Your message may not have been sent!"
    print e
    sys.exit(1)
else:
    print "Message successfully sent to %d recipient(s)" % len(toaddrs)
```

If you run this program and give it a server that understands TLS, the output will look like this:

```
root@erlerobot:~/Python_files# python tls.py jgoerzen@complete.org jgoerzen@complete.org
Negotiating TLS...
Using TLS connection.
Message successfully sent to 1 recipient(s)
```

Notice that the call to `sendmail()` in these last few listings is the same, regardless of whether TLS is used.

Authenticated SMTP

We reach the topic of Authenticated SMTP, where your ISP, university, or company e-mail server needs you to log in with a username and password to prove that you are not a spammer before they allow you to send e-mail.

For maximum security, TLS should be used in conjunction with authentication; otherwise your password (and username, for that matter) will be visible to anyone observing the connection. The proper way to do this is to establish the TLS connection first, and then send your authentication information only over the encrypted communications channel.

But using authentication itself is simple; `smtplib` provides a `login()` function that takes a username and a password. `login.py` shows an example. To avoid repeating code already shown in previous listings, this listing does not take the advice of the previous paragraph, and sends the username and password over an un-authenticated connection that will send them in the clear.

```
import sys, smtplib, socket
from getpass import getpass

if len(sys.argv) < 4:
    print "Syntax: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
    sys.exit(2)

server, fromaddr, toaddrs = sys.argv[1], sys.argv[2], sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,

This is a test message sent to you from the login.py program
in Foundations of Python Network Programming.
""" % (' ', '.join(toaddrs), fromaddr)

sys.stdout.write("Enter username: ")
username = sys.stdin.readline().strip()
password = getpass("Enter password: ")

try:
    s = smtplib.SMTP(server)
    try:
        s.login(username, password)
    except smtplib.SMTPException, e:
        print "Authentication failed:", e
        sys.exit(1)
    s.sendmail(fromaddr, toaddrs, message)
except (socket.gaierror, socket.error, socket.herror,
        smtplib.SMTPException), e:
    print " *** Your message may not have been sent!"
    print e
    sys.exit(1)
else:
    print "Message successfully sent to %d recipient(s)" % len(toaddrs)
```

You can run this program just like the previous examples. If you run it with a server that does support authentication, you will be prompted for a username and password. If they are accepted, then the program will proceed to transmit your message.

Post Office Protocol (POP)

The Post Office Protocol, is a simple protocol that is used to download e-mail from a mail server, and is typically used through an e-mail client like Thunderbird or Outlook. POP does not support multiple mailboxes on the remote side, nor does it provide any reliable, persistent message identification. This means that you cannot use POP as a protocol for mail synchronization. The Python Standard Library provides the [poplib](#) module, which provides a convenient interface for using POP. In this chapter, you will learn how to use poplib to connect to a POP server, gather summary information about a mailbox, download messages, and delete the originals from the server.

Connecting and Authenticating

POP supports several authentication methods. The two most common are basic username-password authentication, and APOP, which is an optional extension to POP that helps protect passwords from being sent in plain-text if you are using an ancient POP server that does not support SSL.

The process of connecting and authenticating to a remote server looks like this in Python:

1. Create a `POP3_SSL` or just a plain POP3 object, and pass the remote hostname and port to it.
2. Call `user()` and `pass_()` to send the username and password. Note the underscore in `pass_()`. It is present because `pass` is a keyword in Python and cannot be used for a method name.
3. If the exception `poplib.error_proto` is raised, it means that the login has failed and the string value of the exception contains the error explanation sent by the server.

The choice between POP3 and POP3_SSL is governed by whether your e-mail provider offers—or, in this day and age, even requires—that you connect over an encrypted connection.

`popconn.py` uses the foregoing steps to log in to a remote POP server. Once connected, it calls `stat()`, which returns a simple tuple giving the number of messages in the mailbox and the messages' total size. Finally, the program calls `quit()`, which closes the POP connection.

```
import getpass, poplib, sys

if len(sys.argv) != 3:
    print 'usage: %s hostname user' % sys.argv[0]
    exit(2)

hostname, user = sys.argv[1:]
passwd = getpass.getpass()

p = poplib.POP3_SSL(hostname) # or "POP3" if SSL is not supported
try:
    p.user(user)
    p.pass_(passwd)
except poplib.error_proto, e:
    print "Login failed:", e
else:
    status = p.stat()
    print "You have %d messages totaling %d bytes" % status
finally:
    p.quit()
```

You can test this program if you have a POP account somewhere. The program will then prompt you for your password. Finally, it will display the mailbox status, without touching or altering any of your mail.

When POP servers do not support SSL to protect your connection from snooping, they sometimes at least support an alternate authentication protocol called APOP, which uses a challenge-response scheme to assure that your password is not sent in the clear. (But all of your e-mail will still be visible to any third party watching the packets go by) The Python Standard Library makes this very easy to attempt: just call the `apop()` method, then fall back to basic authentication if the POP server you are talking to does not understand. To use APOP but fall back to plain authentication, you could use a stanza like the one shown below inside your POP program (like `popconn.py`).

```
print "Attempting APOP authentication..."
try:
    p.apop(user, passwd)
except poplib.error_proto:
    print "Attempting standard authentication..."
    try:
        p.user(user)
        p.pass_(passwd)
    except poplib.error_proto, e:
        print "Login failed:", e
```



```
sys.exit(1)
```

Obtaining Mailbox Information

The preceding example showed you `stat()`, which returns the number of messages in the mailbox and their total size. Another useful POP command is `list()`, which returns more detailed information about each message. The most interesting part is the message number, which is required to retrieve messages later. Note that there may be gaps in message numbers: a mailbox may, for example, contain message numbers 1, 2, 5, 6, and 9. Also, the number assigned to a particular message may be different on each connection you make to the POP server. `mailbox.py` shows how to use the `list()` command to display information about each message.

```
import getpass, poplib, sys

if len(sys.argv) != 3:
    print 'usage: %s hostname user' % sys.argv[0]
    exit(2)

hostname, user = sys.argv[1:]
passwd = getpass.getpass()

p = poplib.POP3_SSL(hostname)
try:
    p.user(user)
    p.pass_(passwd)
except poplib.error_proto, e:
    print "Login failed:", e
else:
    response, listings, octet_count = p.list()
    for listing in listings:
        number, size = listing.split()
        print "Message %s has %s bytes" % (number, size)
finally:
    p.quit()
```

The `list()` function returns a tuple containing three items; you should generally pay attention to the second item. Here is its raw output for one of my POP mailboxes at the moment, which has three messages in it:

```
('+OK 3 messages (5675 bytes)', ['1 2395', '2 1626',
'3 1654'], 24)
```

The three strings inside the second item give the message number and size for each of the three messages in my in-box.

Downloading and Deleting Messages

You should now be getting the hang of POP: when using `poplib` you get to issue small atomic commands that always return a tuple inside which are various strings and lists of strings showing you the result. We are now ready to actually manipulate messages! The three relevant methods, which all identify messages using the same integer identifiers that are returned by `list()`, are these:

- `retr(num)`: This method downloads a single message and returns a tuple containing a result code and the message itself, delivered as a list of lines. This will cause most POP servers to set the “seen” flag for the message to “true,” barring you from ever seeing it from POP again (unless you have another way into your mailbox that lets you set messages back to “Unread”).
- `top(num, body_lines)`: This method returns its result in the same format as `retr()` without marking the message as “seen.” But instead of returning the whole message, it just returns the headers plus however many lines of the body you ask for in `body_lines`. This is useful for previewing messages if you want to let the user decide which ones to download.
- `delete(num)`: This method marks the message for deletion from the POP server, to take place when you quit this POP session. Typically you would do this only if the user directly requests irrevocable destruction of the message, or if you have stored the message to disk and used something like `fsync()` to assure the data’s safety.

To put everything together, take a look at `download-and-delete.py`, which is a fairly functional e-mail client that speaks POP. It checks your in-box to determine how many messages there are and to learn what their numbers are; then it uses `top()` to offer a preview of each one; and, at the user’s option, it can retrieve the whole message, and can also delete it from the mailbox.

```
import email, getpass, poplib, sys

if len(sys.argv) != 3:
    print 'usage: %s hostname user' % sys.argv[0]
    exit(2)

hostname, user = sys.argv[1:]
passwd = getpass.getpass()

p = poplib.POP3_SSL(hostname)
try:
    p.user(user)
    p.pass_(passwd)
except poplib.error_proto, e:
    print "Login failed:", e
else:
    response, listings, octets = p.list()
    for listing in listings:
        number, size = listing.split()
        print 'Message', number, '(size is', size, 'bytes):'
        print
        response, lines, octets = p.top(number, 0)
        message = email.message_from_string('\n'.join(lines))
        for header in 'From', 'To', 'Subject', 'Date':
            if header in message:
                print header + ':', message[header]
        print
        print 'Read this message [ny]?'
        answer = raw_input()
        if answer.lower().startswith('y'):
            response, lines, octets = p.retr(number)
            message = email.message_from_string('\n'.join(lines))
            print '-' * 72
            for part in message.walk():
                if part.get_content_type() == 'text/plain':
                    print part.get_payload()
                    print '-' * 72
            print
        print 'Delete this message [ny]?'
        answer = raw_input()
```

```
        if answer.lower().startswith('y'):
            p.dele(number)
            print 'Deleted.'
finally:
    p.quit()
```

If you run this program, you'll see output similar to this:

```
root@erlerobot:~/Python_files# python download-and-delete.py pop.gmail.com my_gmail_acct
Message 1 (size is 1847 bytes):
From: root@server.example.com
To: Brandon Rhodes <brandon.craig.rhodes@gmail.com>
Subject: Backup complete
Date: Tue, 13 Apr 2010 16:56:43 -0700 (PDT)
Read this message [ny]?
n
Delete this message [ny]?
y
Deleted.
```

Internet Message Access Protocol (IMAP)

Such as POP, IMAP is a way that a laptop or desktop computer can connect to a larger Internet server to view and manipulate a user's e-mail. Whereas the capabilities of POP are rather anemic the IMAP protocol offers such a full array of capabilities that many users store their e-mail permanently on the server, keeping it safe from a laptop or desktop hard drive crash.

This chapter will teach just the basics, with a focus on how to best connect from Python.

Understanding IMAP in Python

The Python Standard Library contains an IMAP client interface named `imaplib`, which does offer rudimentary access to the protocol. Unfortunately, it limits itself to knowing how to send requests and deliver their responses back to your code. It makes no attempt to actually implement the detailed rules in the IMAP specification for parsing the returned data.

As an example of how values returned from `imaplib` are usually too raw to be usefully used in a program, take a look at `open_imaplib.py`. It is a simple script that uses `imaplib` to connect to an IMAP account, list the “capabilities” that the server advertises, and then display the status code and data returned by the LIST command.

```
import getpass, sys
from imapclient import IMAPClient

try:
    hostname, username = sys.argv[1:]
except ValueError:
    print 'usage: %s hostname username' % sys.argv[0]
    sys.exit(2)

c = IMAPClient(hostname, ssl=True)
try:
    c.login(username, getpass.getpass())
except c.Error, e:
    print 'Could not log in:', e
    sys.exit(1)

print 'Capabilities:', c.capabilities()
print 'Listing mailboxes:'
data = c.list_folders()
for flags, delimiter, folder_name in data:
    print ' %-30s%s %s' % (' '.join(flags), delimiter, folder_name)
c.logout()
```

If you run this script with appropriate arguments, it will start by asking for your password—IMAP authentication is almost always accomplished through a username and password:

```
root@erlerobot:~/Python_files# python open_imaplib.py imap.example.com brandon@example.com
Password:
```

If your password is correct, it will then print out a response that looks something like the result shown below:

```
Capabilities: ('IMAP4REV1', 'UNSELECT', 'IDLE', 'NAMESPACE', 'QUOTA',
'XLIST', 'CHILDREN', 'XYZZY', 'SASL-IR', 'AUTH=XOAUTH')
Listing mailboxes
Status: 'OK'
Data:
'(\HasNoChildren) "/" "INBOX"'
'(\HasNoChildren) "/" "Personal"'
'(\HasNoChildren) "/" "Receipts"'
'(\HasNoChildren) "/" "Travel"'
'(\HasNoChildren) "/" "Work"'
'(\Noselect \HasChildren) "/" "[Gmail]"'
'(\HasChildren \HasNoChildren) "/" "[Gmail]/All Mail"'
'(\HasNoChildren) "/" "[Gmail]/Drafts"'
'(\HasChildren \HasNoChildren) "/" "[Gmail]/Sent Mail"'
'(\HasNoChildren) "/" "[Gmail]/Spam"'
'(\HasNoChildren) "/" "[Gmail]/Starred"'
'(\HasChildren \HasNoChildren) "/" "[Gmail]/Trash"
```

There are two main problems: First, we have been returned its status code manually and second, `imaplib` gives us no help in interpreting the results.

So unless you want to implement several details of the protocol yourself, you will want a more capable IMAP client library.

IMAPClient

Fortunately, a popular and battle-tested IMAP library for Python does exist, and is available for easy installation from the Python Package Index. The [IMAPClient package](#) is written by a friendly Python programmer named Menno Smits, and in fact uses the Standard Library. Once installed, you can use the python interpreter in the virtual environment to run the program shown in `open_imap.py`.

```
import getpass, sys
from imapclient import IMAPClient

try:
    hostname, username = sys.argv[1:]
except ValueError:
    print 'usage: %s hostname username' % sys.argv[0]
    sys.exit(2)

c = IMAPClient(hostname, ssl=True)
try:
    c.login(username, getpass.getpass())
except c.Error, e:
    print 'Could not log in:', e
    sys.exit(1)

print 'Capabilities:', c.capabilities()
print 'Listing mailboxes:'
data = c.list_folders()
for flags, delimiter, folder_name in data:
    print ' %-30s%s %s' % (' '.join(flags), delimiter, folder_name)
c.logout()
```

You can see immediately from the code that more details of the protocol exchange are now being handled on our behalf. For example, we no longer get a status code back that we have to check every time we run a command; instead, the library is doing that check for us and will raise an exception to stop us in our tracks if anything goes wrong. Second, you can see that each result from the LIST command—which in this library is offered as the `list_folders()` method instead of the `list()` method offered by `imaplib`—has already been parsed into Python data types for us. Each line of data comes back as a tuple giving us the folder flags, folder name delimiter, and folder name, and the flags themselves are a sequence of strings. Take a look at the code below, for what the output of this second script looks like:

```
Capabilities: ('IMAP4REV1', 'UNSELECT', 'IDLE', 'NAMESPACE', 'QUOTA', 'XLIST', 'CHILDREN',
'XYZZY', 'SASL-IR', 'AUTH=XOAUTH')
Listing mailboxes:
\HasNoChildren / INBOX
\HasNoChildren / Personal
\HasNoChildren / Receipts
\HasNoChildren / Travel
\HasNoChildren / Work
\Noselect \HasChildren / [Gmail]
\HasChildren \HasNoChildren / [Gmail]/All Mail
\HasNoChildren / [Gmail]/Drafts
\HasChildren \HasNoChildren / [Gmail]/Sent Mail
\HasNoChildren / [Gmail]/Spam
\HasNoChildren / [Gmail]/Starred
\HasChildren \HasNoChildren / [Gmail]/Trash
```

The standard flags listed for each folder may be zero or more of the following:

- `\Noinferiors`: This means that the folder does not contain any sub-folders and that it is not possible for it to contain sub-folders in the future. Your IMAP client will receive an error if it tries to create a sub-folder under this folder.
- `\Noselect`: This means that it is not possible to run `select_folder()` on this folder—that is, this folder does not and cannot contain any messages. (Perhaps it exists just to allow sub-folders beneath it, as one possibility.)
- `\Marked`: This means that the server considers this box to be interesting in some way; generally, this indicates that new messages have been delivered since the last time the folder was selected. However, the absence of `\Marked` does not

guarantee that the folder does not contain new messages; some servers simply do not implement \Marked at all.

- \Unmarked: This guarantees that the folder doesn't contain new messages.

Message Numbers vs. UIDs

IMAP provides two different ways to refer to a specific message within a folder: by a temporary message number (which typically goes 1, 2, 3, and so forth) or by a UID (unique identifier). The difference between the two lies with persistence. Message numbers are assigned right when you select the folder. This means they can be pretty and sequential, but it also means that if you revisit the same folder later, then a given message may have a different number. For programs such as live mail readers or simple download scripts, this behavior (which is the same as POP) is fine; you do not need the numbers to stay the same. But a UID, by contrast, is designed to remain the same even if you close your connection to the server and do not reconnect again for another week. If a message had UID 1053 today, then the same message will have UID 1053 tomorrow, and no other message in that folder will ever have UID 1053. If you are writing a synchronization tool, this behavior is quite useful! It will allow you to verify with 100% percent certainty that actions are being taken against the correct message. This is one of the things that make IMAP so much more fun than POP.

Most IMAP commands that work with specific messages can take either message numbers or UIDs. Normally, `IMAPClient` always uses UIDs and ignores the temporary message numbers assigned by IMAP. But if you want to see the temporary numbers instead, simply instantiate `IMAPClient` with a `use_uid=False` argument—or, you can even set the value of the class's `use_uid` attribute to `False` and `True` on the fly during your IMAP session.

Summary Information

When you first select a folder, the IMAP server provides some summary information about it—about the folder itself and also about its messages. The summary is returned by `IMAPClient` as a dictionary. Here are the keys that most IMAP servers will return when you run `select_folder()` :

- **EXISTS**: An integer giving the number of messages in the folder.
- **FLAGS**: A list of the flags that can be set on messages in this folder.
- **RECENT**: Specifies the server's approximation of the number of messages that have appeared in the folder since the last time an IMAP client ran `select_folder()` on it.
- **PERMANENTFLAGS**: Specifies the list of custom flags that can be set on messages; this is usually empty.
- **UIDNEXT**: The server's guess about the UID that will be assigned to the next incoming (or uploaded) message
- **UIDVALIDITY**: A string that can be used by clients to verify that the UID numbering has not changed; if you come back to a folder and this is a different value than the last time you connected, then the UID number has started over and your stored UID values are no longer valid.
- **UNSEEN**: Specifies the message number of the first unseen message (one without the `\Seen` flag) in the folder.

Of these flags, servers are only required to return **FLAGS**, **EXISTS**, and **RECENT**, though most will include at least **UIDVALIDITY** as well.

`folder_info.py` shows an example program that reads and displays the summary information of my INBOX mail folder:

```
import getpass, sys
from imapclient import IMAPClient

try:
    hostname, username = sys.argv[1:]
except ValueError:
    print 'usage: %s hostname username' % sys.argv[0]
    sys.exit(2)

c = IMAPClient(hostname, ssl=True)
try:
    c.login(username, getpass.getpass())
except c.Error, e:
    print 'Could not log in:', e
    sys.exit(1)
else:
    select_dict = c.select_folder('INBOX', readonly=True)
    for k, v in select_dict.items():
        print '%s: %r' % (k, v)
    c.logout()
```

When run, this program displays results such as this:

```
...
```

```
root@erlerobot:~/Python_files# python folder_info.py imap.example.com brandon@example.com Password: EXISTS: 3
PERMANENTFLAGS: ('\Answered', '\Flagged', '\Draft', '\Deleted', '\Seen', '\*') READ-WRITE: True UIDNEXT: 2626 FLAGS:
('\Answered', '\Flagged', '\Draft', '\Deleted', '\Seen') UIDVALIDITY: 1 RECENT: 0
```

```
...
```

That shows that my INBOX folder contains three messages, none of which have arrived since I last checked. If your program is interested in using UIDs that it stored during previous sessions, remember to compare the UIDVALIDITY to a stored value from a previous session.

Downloading an Entire Mailbox

With IMAP, the FETCH command is used to download mail, which IMAPClient exposes as its `fetch()` method. The simplest way to fetch involves downloading all messages at once, in a single big gulp. While this is simplest and requires the least network traffic (since you do not have to issue repeated commands and receive multiple responses), it does mean that all of the returned messages will need to sit in memory Download from together as your program examines them. For very large mailboxes whose messages have lots of attachments, this is obviously not practical.

`mailbox_summary.py` downloads all of the messages from my INBOX folder into your computer's memory in a Python data structure, and then displays a bit of summary information about each one.

```
import email, getpass, sys
from imapclient import IMAPClient

try:
    hostname, username, foldername = sys.argv[1:]
except ValueError:
    print 'usage: %s hostname username folder' % sys.argv[0]
    sys.exit(2)

c = IMAPClient(hostname, ssl=True)
try:
    c.login(username, getpass.getpass())
except c.Error, e:
    print 'Could not log in:', e
    sys.exit(1)

c.select_folder(foldername, readonly=True)
msgdict = c.fetch('1:*', ['BODY.PEEK[]'])
for message_id, message in msgdict.items():
    e = email.message_from_string(message['BODY[]'])
    print message_id, e['From']
    payload = e.get_payload()
    if isinstance(payload, list):
        part_content_types = [ part.get_content_type() for part in payload ]
        print 'Parts:', ' '.join(part_content_types)
    else:
        print ' ', ' '.join(payload[:60].split()), '...'

c.logout()
```

Remember that IMAP is stateful: first we use `select_folder()` to put us “inside” the given folder, and then we can run `fetch()` to ask for message content. The range `'1:*` means “the first message through the end of the mail folder”, because message IDs—whether temporary or UIDs—are always positive integers.

Here is what it looks like to run this script:

```
root@erlerobot:~/Python_files# python mailbox_summary.py imap.example.com brandon INBOX
Password:
2590 "Amazon.com" <order-update@amazon.com>
Dear Brandon, Portable Power Systems, Inc. shipped the follo ...
2469 Meetup Reminder <info@meetup.com>
Parts: text/plain text/html
2470 billing@linode.com
Thank you. Please note that charges will appear as "Linode.c ...
```

Downloading Messages Individually

E-mail messages can be quite large, and so can mail folders—many mail systems permit users to have hundreds or thousands of messages, that can each be 10MB or more. That kind of mailbox can easily exceed the RAM on the client machine if its contents are all downloaded at once, as in the previous example. To help network-based mail clients that do not want to keep local copies of every message, IMAP supports several operations besides the big “fetch the whole message” command that we saw in the previous section.

- An e-mail's headers can be downloaded as a block of text, separately from the message.
- Particular headers from a message can be requested and returned.
- The server can be asked to recursively explore and return an outline of the MIME structure of a message.
- The text of particular sections of the message can be returned.

This allows IMAP clients to perform very efficient queries that download only the information they need to display for the user, decreasing the load on the IMAP server and the network, and allowing results to be displayed more quickly to the user. For an example of how a simple IMAP client works, examine `simple_client.py`, which puts together a number of ideas about browsing an IMAP account. Hopefully this provides more context than would be possible if these features were spread out over a half-dozen shorter program listings at this point in the chapter. You can see that the client consists of three concentric loops that each take input from the user as he or she views the list of mail folders, then the list of messages within a particular mail folder, and finally the sections of a specific message.

```
import getpass, sys
from imapclient import IMAPClient

try:
    hostname, username = sys.argv[1:]
except ValueError:
    print 'usage: %s hostname username' % sys.argv[0]
    sys.exit(2)

banner = '-' * 72

c = IMAPClient(hostname, ssl=True)
try:
    c.login(username, getpass.getpass())
except c.Error, e:
    print 'Could not log in:', e
    sys.exit(1)

def display_structure(structure, parentparts=[]):
    """Attractively display a given message structure."""

    # The whole body of the message is named 'TEXT'.

    if parentparts:
        name = '.'.join(parentparts)
    else:
        print 'HEADER'
        name = 'TEXT'

    # Print this part's designation and its MIME type.

    is_multipart = isinstance(structure[0], list)
    if is_multipart:
        parttype = 'multipart/%s' % structure[1].lower()
    else:
        parttype = ('%s/%s' % structure[:2]).lower()
    print '%-9s' % name, parttype,

    # For a multipart part, print all of its subordinate parts; for
    # other parts, print their disposition (if available).

    if is_multipart:
```

```

print
subparts = structure[0]
for i in range(len(subparts)):
    display_structure(subparts[i], parentparts + [ str(i + 1) ])
else:
    if structure[6]:
        print 'size=%s' % structure[6],
    if structure[8]:
        disposition, namevalues = structure[8]
        print disposition,
        for i in range(0, len(namevalues), 2):
            print '%s=%r' % namevalues[i:i+2]
    print

def explore_message(c, uid):
    """Let the user view various parts of a given message."""

    msgdict = c.fetch(uid, ['BODYSTRUCTURE', 'FLAGS'])

    while True:
        print
        print 'Flags:',
        flaglist = msgdict[uid]['FLAGS']
        if flaglist:
            print ' '.join(flaglist)
        else:
            print 'none'
        display_structure(msgdict[uid]['BODYSTRUCTURE'])
        print
        reply = raw_input('Message %s - type a part name, or "q" to quit: '
                          % uid).strip()

        print
        if reply.lower().startswith('q'):
            break
        key = 'BODY[%s]' % reply
        try:
            msgdict2 = c.fetch(uid, [key])
        except c._imap.error:
            print 'Error - cannot fetch section %r' % reply
        else:
            content = msgdict2[uid][key]
            if content:
                print banner
                print content.strip()
                print banner
            else:
                print '(No such section)'

def explore_folder(c, name):
    """List the messages in folder `name` and let the user choose one."""

    while True:
        c.select_folder(name, readonly=True)
        msgdict = c.fetch('1:*', ['BODY.PEEK[HEADER.FIELDS (FROM SUBJECT)]',
                                  'FLAGS', 'INTERNALDATE', 'RFC822.SIZE'])

        print
        for uid in sorted(msgdict):
            items = msgdict[uid]
            print '%6d %20s %6d bytes %s' % (
                uid, items['INTERNALDATE'], items['RFC822.SIZE'],
                ' '.join(items['FLAGS']))
            for i in items['BODY[HEADER.FIELDS (FROM SUBJECT)]'].splitlines():
                print ' ' * 6, i.strip()

        reply = raw_input('Folder %s - type a message UID, or "q" to quit: '
                          % name).strip()
        if reply.lower().startswith('q'):
            break
        try:
            reply = int(reply)
        except ValueError:
            print 'Please type an integer or "q" to quit'
        else:
            if reply in msgdict:
                explore_message(c, reply)

    c.close_folder()

def explore_account(c):
    """Display the folders in this IMAP account and let the user choose one."""

    while True:

```

```

print
folderflags = {}
data = c.list_folders()
for flags, delimiter, name in data:
    folderflags[name] = flags
for name in sorted(folderflags.keys()):
    print '%-30s %s' % (name, ' '.join(folderflags[name]))
print

reply = raw_input('Type a folder name, or "q" to quit: ').strip()
if reply.lower().startswith('q'):
    break
if reply in folderflags:
    explore_folder(c, reply)
else:
    print 'Error: no folder named', repr(reply)

if __name__ == '__main__':
    explore_account(c)

```

You can see that the outer function uses a simple `list_folders()` call to present the user with a list of his or her mail folders, like some of the program listings we have seen already. Each folder's IMAP flags are also displayed. This lets the program give the user a choice between folders:

```

INBOX \HasNoChildren
Receipts \HasNoChildren
Travel \HasNoChildren
Work \HasNoChildren
Type a folder name, or "q" to quit:
` `

```

Once a user has selected a folder, things become more interesting: a summary has to be printed for each message. Note that it is careful to use `BODY.PEEK` instead of `BODY` to fetch these items, since the IMAP server would otherwise mark the messages as `\Seen` merely because they had been displayed in a summary.

The results of this `fetch()` call are printed to the screen once an e-mail folder has been selected:

```

2703 2010-09-28 21:32:13 19129 bytes \Seen From: Brandon Craig Rhodes Subject: Digested Articles
2704 2010-09-28 23:03:45 15354 bytes Subject: Re: [venv] Building a virtual environment for offline testing From: "W. Craig Trader"
2705 2010-09-29 08:11:38 10694 bytes Subject: Re: [venv] Building a virtual environment for offline testing From: Hugo Lopes Tavares
Folder INBOX - type a message UID, or "q" to quit: ` `

```

As you can see, the fact that several items of interest can be supplied to the `IMAP fetch()` command lets us build fairly sophisticated message summaries with only a single round-trip to the server. One final note about the `fetch()` command: it lets you not only pull just the parts of a message that you need at any given moment, but also truncate them in case they are quite long and you just want to provide an excerpt from the beginning to tantalize the user.

Flagging and Deleting Messages

Flagging

You might have noticed, while trying out `simple_client.py` or reading its example output just shown, that IMAP marks messages with attributes called "flags," which typically take the form of a backslashprefixed word, like `\Seen` for one of the messages just cited. Several of these are standard, and are defined in RFC 3501 for use on all IMAP servers. Here is what the most important ones mean:

- `\Answered`: The user has replied to the message.
- `\Draft`: The user has not finished composing the message.
- `\Flagged`: The message has somehow been singled out specially; the purpose and meaning of this flag vary between mail readers.
- `\Recent`: No IMAP client has seen this message before. This flag is unique, in that the flag cannot be added or removed by normal commands; it is automatically removed after the mailbox is selected.
- `\Seen`: The message has been read.

The `IMAPClient` library supports several methods for working with flags. The simplest retrieves the flags as though you had done a `fetch()` asking for 'FLAGS', but goes ahead and removes the dictionary around each answer:

```
>>> c.get_flags(2703)
{2703: ('\Seen',)}
```

There are also calls to add and remove flags from a message:

```
c.remove_flags(2703, ['\Seen'])
c.add_flags(2703, ['\Answered'])
```

In case you want to completely change the set of flags for a particular message without figuring out the correct series of adds and removes, you can use `set_flags()` to unilaterally replace the whole list of message flags with a new one:

```
c.set_flags(2703, ['\Seen', '\Answered'])
```

Any of these operations can take a list of message UIDs instead of the single UID shown in these examples.

Deleting Messages

One last interesting use of flags is that it is how IMAP supports message deletion. The process, for safety, takes two steps: first the client marks one or more messages with the `\Delete` flag; then it calls `expunge()` to perform the deletions as a single operation. The `IMAPClient` library does not make you do this by hand, however (though that would work); instead it hides the fact that flags are involved behind a simple `delete_messages()` routine that marks the messages for you. It still has to be followed by `expunge()` if you actually want the operation to take effect, though:

```
c.delete_messages([2703, 2704])
c.expunge()
```

Searching and Manipulating Messages

Searching is another issue that is very important for a protocol designed to let you keep all your mail on the mail server itself: without search, an e-mail client would have to download all of a user's mail anyway the first time he or she wanted to perform a full-text search to find an e-mail message. The essence of search is simple: you call the `search()` method on an IMAP client instance, and are returned the UIDs (assuming, of course, that you accept the IMAPClient default of `use_uid=True` for your client) of the messages that match your criteria:

```
>>> c.select_folder('INBOX')
>>> c.search('SINCE 20-Aug-2010 TEXT Apress')
[2590L, 2652L, 2653L, 2654L, 2655L, 2699L]
```

There are many criteria that you can combine in order to form a query. Like the rest of IMAP, they are specified in RFC 3501. Some criteria are quite simple, and refer to binary attributes like flags:

```
ALL: Every message in the mailbox
UID (id, ...): Messages with the given UIDs
LARGER n: Messages more than n octets in length
SMALLER m: Messages less than m octets in length
ANSWERED: Have the flag \Answered
DELETED: Have the flag \Deleted
DRAFT: Have the flag \Draft
FLAGGED: Have the flag \Flagged
KEYWORD flag: Have the given keyword flag set
NEW: Have the flag \Recent
OLD: Lack the flag \Recent
UNANSWERED: Lack the flag \Answered
UNDELETED: Lack the flag \Deleted
UNDRAFT: Lack the flag \Draft
UNFLAGGED: Lack the flag \Flagged
UNKEYWORD flag: Lack the given keyword flag
UNSEEN: Lack the flag \Seen
```

There are two sets of criteria for dates, depending on which date you want to query by: the internal Date header (sent date) and the at which arrived at the IMAP server.

Finally, there are two search operations that refer to the text of the message itself—these are the big workhorses that support full-text search of the kind your users are probably expecting when they type into a search field in an e-mail client:

```
BODY string: The message body must contain the string.
TEXT string: The entire message, either body or header, must contain the string somewhere.
```

Manipulating Folders and Messages

Creating or deleting folders is done quite simply in IMAP, by providing the name of the folder:

```
c.create_folder('Personal')
c.delete_folder('Work')
```

Some IMAP servers or configurations may not permit these operations, or may have restrictions on naming; be sure to have error checking in place when calling them. There are two operations that can create new e-mail messages in your IMAP account besides the “normal” means of waiting for people to send them to you. First, you can copy an existing message from its home folder over into another folder. Start by using `select_folder()` to visit the folder where the messages live, and then run the copy method like this:

```
c.select_folder('INBOX')
```

```
c.copy([2653L, 2654L], 'TODO')
```

Finally, it is possible to add a message to a mailbox with IMAP. You do not need to send the message first with SMTP; IMAP is all that is needed. Adding a message is a simple process, though there are a couple of things to be aware of.

You must also be cautious in how carefully you change the line endings, because some messages may use `\r\n` somewhere inside despite using only `\n` for the first few dozen lines, and IMAP clients have been known to fail if a message uses both different line endings! The solution is a simple one, thanks to Python's powerful `splitlines()` string method that recognizes all three possible line endings; simply call the function on your message and then re-join the lines with the standard line ending:

```
>>> 'one\rtwo\nthree\r\nfour'.splitlines()
['one', 'two', 'three', 'four']
>>> '\r\n'.join('one\rtwo\nthree\r\nfour'.splitlines())
'one\r\ntwo\r\nthree\r\nfour'
```

The actual act of appending a message, once you have the line endings correct, is to call the `append()` method on your IMAP client:

```
c.append('INBOX', my_message)
```

You can also supply a list of flags as a keyword argument, as well as a `msg_time` to be used as its arrival time by passing a normal Python datetime object.

Telnet and SSH

The “command line” is the topic of this chapter: how you can access it over the network, together with enough discussion about its typical behavior to get you through any frustrations you might encounter while trying to use it.

Command-Line Automation

Before getting into the details of how the command line works, and how you can access it over the network, we should pause and note that there exist many systems today for automating the entire process. If you have dozens or hundreds of machines to maintain and you need to start sending them all the same commands, then you might find that tools already exist—tools that already provide ways to write command scripts, push them out for execution across a cloud of machines, batch up any error messages or responses for your review, and even save commands in a queue to be re-tried later in case a machine is down and cannot be reached at the moment.

What are the options? First, the [Fabric library](#) is very popular with Python programmers who need to run commands and copy files to remote server machines. As you can see in `fabfile.py`, a Fabric script calls very simple functions with names like `put()`, `cd()`, and `run()` to perform operations on the machines to which it connects. But you can learn more about it at its web site: <http://fabfile.org/>. Although `fabfile.py` is designed to be run by Fabric's own `fab` command-line tool, Fabric can also be used from inside your own Python programs; again, consult their documentation for details.

```
from fabric.api import *

def versions():
    with cd('/usr/bin'):
        with settings(hide('warnings'), warn_only=True):
            for version in '2.4', '2.5', '2.6', '2.7', '3.0', '3.1':
                result = run('python%s -c "None"' % version)
                if not result.failed:
                    print "Host", env.host, "has Python", version
```

Another project to check out is Silver Lining. It is still very immature, but if you are an experienced programmer who needs its specific capabilities, then you might find that it solves your problems well. This library goes beyond batching commands across many different servers: it will actually create and initialize Ubuntu servers through the “libcloud” Python API, and then install your Python web applications there for you. You can learn more about this promising project at <http://cloudsilverlining.org/>.

On the other hand, there is “pexpect.” While it is not, technically, a program that itself knows how to use the network, it is often used to control the system “ssh” or “telnet” command when a Python programmer wants to automate interactions with a remote prompt of some kind. This typically takes place in a situation where no API for a device is available, and commands simply have to be typed each time the command-line prompt appears. Configuring simple network hardware often requires this kind of clunky step-by-step interaction. You can learn more about “pexpect” here: <http://pypi.python.org/pypi/pexpect>.

Finally, there are more specific projects that provide mechanisms for remote systems administration. Red Hat and Fedora users might look at `func`, which uses an SSL-encrypted XML-RPC service that lets you write Python programs that perform system configuration and maintenance: <https://fedorahosted.org/func/>.

Command-Line Expansion and Quoting

If you have ever typed many commands at a Unix command prompt, you will be aware that not every character you type is interpreted literally. Consider this command, for example:

```
root@erlerobot:~# echo *
Hello.txt Python-3.4.1 Python-3.4.1.tgz Python_files build gmapenv hola.txt otro text.txt virtualenv-1.11.6 virtualenv-
root@erlerobot:~#
```

A terminal window showing a shell prompt. The user enters the command 'echo *'. The shell expands this to list all files in the current directory: 'Hello.txt Python-3.4.1 Python-3.4.1.tgz Python_files build gmapenv hola.txt otro text.txt virtualenv-1.11.6 virtualenv-'. The prompt returns to '~#'.

The asterisk `*` in this command was not interpreted to mean “print out an asterisk character to the screen”; instead, the shell thought I was trying to write a pattern that would match all of the file names in the current directory. To actually print out an asterisk, I have to use another special character—an “escape” character, because it lets me “escape” from the shell's normal meaning—to tell it that I just mean the asterisk literally:

```
root@erlerobot:~# echo Here is a lone asterisk: \*
Here is a lone asterisk: *
root@erlerobot:~# echo And here are '*' two '**' more asterisks
And here are * two * more asterisks
root@erlerobot:~#
```

A terminal window showing two commands. The first is 'echo Here is a lone asterisk: \`*`'. The output is 'Here is a lone asterisk: *'. The second is 'echo And here are '`*`' two '`**`' more asterisks'. The output is 'And here are * two * more asterisks'.

The rules by which modern shells interpret the special characters in your command line have become quite complex. Instead, to use the command line effectively, you just have to understand two points:

- Special characters are interpreted as special by the shell you are using, like bash.
- When passing commands to a shell either locally or across the network, you need to escape the special characters you use so that they are not expanded into unintended values on the remote system.

Unix Has No Special Characters

Like many very useful statements, the bold claim of the title of this section is, alas, a lie. There is, in fact, a character that Unix considers special. But, in general, Unix has no special characters, and this is a very important fact for you to grasp.

On the one hand, it makes it very easy to, say, name all of the files in the current directory as arguments to a command; but on the other hand, it can be very difficult to echo a message to the screen that mixes single quotes and double-quotes.

The simple lesson of this section is that the whole set of conventions to which you are accustomed has nothing to do with your operating system; they are simply and entirely a behavior of the bash shell, or of whichever of the other popular (or arcane) shells that you are using. It does not matter how familiar the rules seem, or how difficult it is for you to imagine using a Unix-like system without them. If you take bash away, they are simply not there. You can observe this quite simply by taking control of the operating system's process launcher yourself and trying to throw some special characters at a familiar command:

```
>>> import subprocess
>>> args = ['echo', 'Sometimes an', '*', 'just means an', '*']
>>> subprocess.call(args)
```

Sometimes an *just means an* Here, we are bypassing all of the shell applications that are available for interpreting commands, and we are telling the operating system to start a new process using precisely the list of arguments we have provided. And the process—the echo command, in this case—is getting exactly those characters, instead of having the * turned into a list of file names first. Though we rarely think about it, the most common “special” character is one we use all the time: the space character. Rather than assume that you actually mean each space character to be passed to the command you are invoking, the shell instead interprets it as the delimiter separating the actual text you want the command to see. This causes endless entertainment when people include spaces in Unix file names, and then try to move the file somewhere else:

```
root@erlerobot:~# mv Smith Contract.txt ~/Documents
mv: cannot stat `Smith': No such file or directory
mv: cannot stat `Contract.txt': No such file or directory
```

To make the shell understand that you are talking about one file with a space in its name, not two files, you have to contrive something like one of these possible command lines:

```
root@erlerobot:~# mv Smith\ Contract.txt ~/Documents
root@erlerobot:~# mv "Smith Contract.txt" ~/Documents
root@erlerobot:~# mv Smith*Contract.txt ~/Documents
```

That last possibility obviously means something quite different—since it will match any file name that happens to start with Smith and end with Contract.txt, regardless of whether the text between them is a simple space character or some much longer sequence of text—but I have seen many people type it in frustration who are still learning shell conventions and cannot remember how to type a literal space character for the shell. If you want to convince yourself that none of the characters that the bash shell has taught you to be careful about is special, `shell.py` shows a simple shell, written in Python, that treats only the space as special but passes everything else through literally to the command.

```
import subprocess

while True:
    args = raw_input(' ')
    if not args:
        pass
    elif args == ['exit']:
        break
    elif args[0] == 'show':
        print "Arguments:", args[1:]
```

```
else:
    subprocess.call(args)
```

Running this file, result on:

```
root@erlerobot:~# python shell.py
] echo Hi there!
Hi there!
] echo An asterisk * is not special.
An asterisk * is not special.
] echo The string $HOST is not special, nor are "double quotes".
The string $HOST is not special, nor are "double quotes".
] echo What? No *<>!$ special characters?
What? No *<>!$ special characters?
] show "The 'show' built-in lists its arguments."
Arguments: ['The', "'show'", 'built-in', 'lists', 'its', 'arguments.']]
] exit
```

You can see here absolute evidence that Unix commands—in this case, the `/bin/echo` command that we are calling over and over again—do not generally attempt to interpret their arguments as anything other than strings. The `echo` command happily accepts double-quotes, dollar signs, and asterisks, and treats them all as literal characters. As the foregoing `show` command illustrates, Python is simply reducing our arguments to a list of strings for the operating system to use in creating a new process. What if we fail to split our command into separate arguments?

```
>>> import subprocess
>>> subprocess.call(['echo hello'])
Traceback (most recent call last):
...
OSError: [Errno 2] No such file or directory
```

The operating system does not know that spaces should be special; that is a quirk of shell programs, not of Unix-like operating systems themselves! So the system thinks that it is being asked to run a command literally named `echo [space] hello`, and, unless you have created such a file in the current directory, it fails to find it and raises an exception.

To prevent you from making this mistake, Python stops you in your tracks if you include a null character in a commandline argument:

```
>>> import subprocess
>>> subprocess.call(['echo', 'Sentences can end\0 abruptly.'])
Traceback (most recent call last):
...
TypeError: execv() arg 2 must contain only strings
```

Since every command on the system is designed to live within this limitation, you will generally find there is never any reason to put null characters into command-line arguments anyway.

Quoting Characters for Protection

In the foregoing section, we used routines in Python's [subprocess module](#) to directly invoke commands. (The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.) This was great, and let us pass characters that would have been special to a normal interactive shell. If you have a big list of file names with spaces and other special characters in them, it can be wonderful to simply pass them into a subprocess call and have the command on the receiving end understand you perfectly.

But when you are using remote-shell protocols over the network (which, you will recall, is the subject of this chapter!), you are generally going to be talking to a shell like bash instead of getting to invoke commands directly like you do through the subprocess module. This means that remote-shell protocols will feel more like the `system()` routine from the `os` module, which does invoke a shell to interpret your command line, and therefore involves you in all of the complexities of the Unix command line:

```
>>> import os
>>> os.system('echo *')
Hello.txt Python-3.4.1 Python-3.4.1.tgz Python_files build gmapenv hola.txt otro text.txt virtualenv-1.11.6 virtualenv-
```

Of course, if the other end of a remote-shell connection is using some sort of shell with which you are unfamiliar, there is little that Python can do. The authors of the Standard Library have no idea how, say, a Motorola DSL router's Telnet-based command line might handle special characters, or even whether it pays attention to quotes at all. But if the other end of a network connection is a standard Unix shell of the sh family, like bash or zsh, then you are in luck: the fairly obscure Python [pipes module](#), which is normally used to build complex shell command lines, contains a helper function that is perfect for escaping arguments. It is called `quote`, and can simply be passed a string:

```
>>> from pipes import quote
>>> print quote("filename")
filename
'file with spaces'
>>> print quote("file 'single quoted' inside!")
"file 'single quoted' inside!"
>>> print quote("danger!; rm -r *")
'danger!; rm -r *'
```

So preparing a command line for remote execution generally just involves running `quote()` on each argument and then pasting the result together with spaces. Note that using a remote shell with Python does not involve you in the terrors of two levels of shell quoting! If you have ever tried to build a remote SSH command line that uses fancy quoting, by typing a local command line into your own shell. The attempt tends to generate a series of experiments like this:

```
$ echo $HOST
guinness
$ ssh asaph echo $HOST
guinness
$ ssh asaph echo \ $HOST
asaph
$ ssh asaph echo \\ $HOST
guinness
$ ssh asaph echo \\ \ $HOST
$HOST
$ ssh asaph echo \\ \ \ $HOST
\guinness
```

using a remote-shell protocol through Python does not involve two levels of shell like this. Instead, you get to construct a literal string in Python that then directly becomes what is executed by the remote shell; no local shell is involved. So if using a shell-within-a-shell has you convinced that passing strings and file names safely to a remote shell is a very hard problem, relax: no local shell will be involved in our following examples.

Things Are Different in a Terminal

You will probably talk to more programs than just the shell over your Python-powered remote-shell connection, of course. You will often want to watch the incoming data stream for the information and errors printed out by the commands you are running. And sometimes you will even want to send data back, either to provide the remote programs with input, or to respond to questions and prompts that they present.

When performing tasks like this, you might be surprised to find that programs hang indefinitely without ever finishing the output that you are waiting on, or that data you send seems to not be getting through. To help you through situations like this, a brief discussion of Unix terminals is in order.

A terminal typically names a device into which a user types text, and on whose screen the computer's response can be displayed. If a Unix machine has physical serial ports that could possibly host a physical terminal, then the device directory will contain entries like `/dev/ttyS1` with which programs can send and receive strings to that device. But most terminals these days are, in reality, other programs: an xterm terminal, or a Gnome or KDE terminal program, or a PuTTY client on a Windows machine that has connected via a remote-shell protocol of the kind we will discuss.

But the programs running inside the terminal on your laptop or desktop machine still need to know that they are talking to a person—they still need to feel like they are talking through the mechanism of a terminal device connected to a display. So the Unix operating system provides a set of “pseudoterminal” devices (which might have less confusingly been named “virtual” terminals) with names like `/dev/tty42`. When someone brings up an xterm or connects through SSH, the xterm or SSH daemon grabs a fresh pseudo-terminal, configures it, and runs the user's shell behind it. The shell examines its standard input, sees that it is a terminal, and presents a prompt since it believes itself to be talking to a person.

This is a crucial distinction to understand: the shell presents a prompt because, and only because, it thinks it is connected to a terminal! If you start up a shell and give it a standard input that is not a terminal—like, say, a pipe from another command—then no prompt will be printed, yet it will still respond to commands:

```
root@erlerobot:~# cat | bash
echo Here we are inside of bash, with no prompt!
Here we are inside of bash, with no prompt!
python
print 'Python has not printed a prompt, either.'
import sys
print 'Is this a terminal?', sys.stdin.isatty()
```

You can see that Python, also, does not print its usual startup banner, nor does it present any prompts.

There are even changes in how some commands format their output depending on whether they are talking to a terminal. Some commands with long lines of output—the `ps` command comes to mind—will truncate their lines to your terminal width if used interactively, but produce arbitrarily wide output if connected to a pipe or file. And, entertainingly enough, the familiar column-based output of the `ls` command gets turned off and replaced with a file name on each line (which is, you must admit, an easier format for reading by another program) if its output is a pipe or file:

```
root@erlerobot:~# ls
Hello.txt      Python_files  hola.txt     virtualenv-1.11.6
Python-3.4.1   build         otro         virtualenv-1.11.6.tar.gz
Python-3.4.1.tgz  gmapenv      text.txt
root@erlerobot:~# ls|cat
Hello.txt
Python-3.4.1
Python-3.4.1.tgz
Python_files
build
gmapenv
hola.txt
otro
text.txt
virtualenv-1.11.6
virtualenv-1.11.6.tar.gz
```

```
root@erlerobot:~#
```

A program running behind Telnet, for example, always thinks it is talking to a terminal; so your scripts or programs must always expect to see a prompt each time the shell is ready for input, and so forth. But when you make a connection over the more sophisticated SSH protocol, you will actually have your choice of whether the program thinks that its input is a terminal or just a plain pipe or file. You can test this easily from the command line if there is another computer you can connect to:

```
root@erlerobot:~# ssh -t asaph
asaph$ echo "Here we are, at a prompt."
Here we are, at a prompt.
```

So when you spawn a command through a modern protocol like SSH, you need to consider whether you want the program on the remote end thinking that you are a person typing at it through a terminal, or whether it had best think it is talking to raw data coming in through a file or pipe.

Programs are not actually required to act any differently when talking to a terminal; it is just for our convenience that they vary their behavior:

- Programs that are often used interactively will present a human-readable prompt when they are talking to a terminal. But when they think input is coming from a file, they avoid printing a prompt.
- Sophisticated interactive programs, these days, usually turn on command-line editing when their input is a TTY.
- Many programs read only one line of input at a time when listening to a terminal, because humans like to get an immediate response to every command they type. But when reading from a pipe or file, these same programs will wait until thousands of characters have arrived before they try to interpret their first batch of input.
- It is even more common for programs to adjust their output based on whether they are talking to a terminal.

Both of the last two issues, which involve buffering, cause all sorts of problems when you take a process that you usually do manually and try to automate it—because in doing so you often move from terminal input to input provided through a file or pipe, and suddenly you find that the programs behave quite differently, and might even seem to be hanging because “print” statements are not producing immediate output, but are instead saving up their results to push out all at once when their output buffer is full.

You can see this easily with a simple Python program (since Python is one of the applications that decides whether to buffer its output based on whether it is talking to a terminal) that prints a message, waits for a line of input, and then prints again:

```
root@erlerobot:~# python -c 'print "talk: "; s = raw_input(); print "you said", s'
talk:
hi
you said hi
root@erlerobot:~# python -c 'print "talk: "; s = raw_input(); print "you said", s' | cat
hi
talk:
you said hi
```

You can see that in the first instance, when Python knew its output was a terminal, it printed talk: immediately. But in the second instance, its output was a pipe to the cat command, and so it decided that it could save up the results of that first print statement and batch them together with the rest of the program's output, so that both lines of output appeared only once you had provided your input and the program was ending.

The foregoing problem is why many carefully written programs, both in Python and in other languages, frequently call `flush()` on their output to make sure that anything waiting in a buffer goes ahead and gets sent out, regardless of whether the output looks like a terminal. So those are the basic problems with terminals and buffering: programs change their behavior, often in idiosyncratic ways, when talking to a terminal (think again of the ls example), and they often start heavily

buffering their output if they think they are writing to a file or pipe.

Terminals Do Buffering

Beyond the program-specific behaviors just described, there are additional problems raised by terminals.

For example, what happens when you want a program to be reading your input one character at a time, but the Unix terminal device itself is buffering your keystrokes to deliver them as a whole line? This common problem happens because the Unix terminal defaults to “canonical” input processing, where it lets the user enter a whole line, and even edit it by backspacing and re-typing, before finally pressing “Enter” and letting the program see what he or she has typed. If you want to turn off canonical processing so that a program can see every individual character as it is typed, you can use the `stty` “Set TTY settings” command to disable it:

```
root@erlerobot:~# stty -icanon
```

Another problem is that Unix terminals traditionally supported a pair of keystrokes for pausing the output stream so that the user could read something on the screen before it scrolled off and was replaced by more text. Often these were the characters `Ctrl+S` for “Stop” and `Ctrl+Q` for “Keep going,” and it was a source of great annoyance that if binary data worked its way into an automated Telnet connection that the first `Ctrl+S` that happened to pass across the channel would pause the terminal and probably ruin the session. Again, this setting can be turned off with `stty`:

```
root@erlerobot:~# stty -ixon -ixoff
```

There are plenty of less famous settings that can also cause you grief. Because there are so many—and because they vary between Unix implementations—the `stty` command actually supports two modes, cooked and raw, that turn dozens of settings like `icanon` and `ixon` on and off together:

```
root@erlerobot:~# stty raw
root@erlerobot:~# stty cooked
```

In case you make your terminal settings a hopeless mess after some experimentation, most Unix systems provide a command for resetting the terminal back to reasonable, sane settings (you might need to hit `Ctrl+J` to submit the reset command, since your Return key, whose equivalent is `Ctrl+M`, actually only functions to submit commands because of a terminal setting called `icrnl`):

```
root@erlerobot:~# reset
```

If, instead of trying to get the terminal to behave across a Telnet or SSH session, you happen to be talking to a terminal from Python, check out the [termios module](#) that comes with the Standard Library. This module provides an interface to the POSIX calls for tty I/O control. For a complete description of these calls, see the POSIX or Unix manual pages.

Telnet

Telnet is a network protocol used on the Internet or local area networks to provide a bidirectional interactive text-oriented communication facility using a virtual terminal connection. User data is interspersed in-band with Telnet control information in an 8-bit byte oriented data connection over the Transmission Control Protocol (TCP).

Telnet is insecure: anyone watching your Telnet packets fly by will see your username, password, and everything you do on the remote system. It is clunky. And it has been completely abandoned for most systems administration.

In case you are having to write a Python program that has to speak Telnet to one of these devices, here are a few pointers on using the Python [telnetlib](#). The telnetlib module provides a Telnet class that implements the Telnet protocol.

First, you have to realize that all Telnet does is to establish a channel and to send the things you type, and receive the things the remote system says, back and forth across that channel. This means that Telnet is ignorant of all sorts of things of which you might expect a remote-shell protocol to be aware.

For example, it is conventional that when you Telnet to a Unix machine, you are presented with a login: prompt at which you type your username, and a password: prompt where you enter your password.

The fact that Telnet is ignorant about authentication has an important consequence: you cannot type anything on the command line itself to get yourself pre-authenticated to the remote system, nor avoid the login and password prompts that will pop up when you first connect! If you are going to use plain Telnet, you are going to have to somehow watch the incoming text for those two prompts (or however many the remote system supplies) and issue the correct replies.

Obviously, if systems vary in what username and password prompts they present, then you can hardly expect standardization in the error messages or responses that get sent back when your password fails. That is why Telnet is so hard to script and program from a language like Python and a library like `telnetlib`.

So if you are using Telnet, then you are playing a text game: you watch for text to arrive, and then try to reply with something intelligible to the remote system. To help you with this, the Python `telnetlib` provides not only basic methods for sending and receiving data, but also a few routines that will watch and wait for a particular string to arrive from the remote system.

`telnet_login.py` connects to localhost, which in this case is my Ubuntu laptop, where I have just run `aptitude install telnetd` so that a Telnet daemon is now listening on its standard port 23.

```
import telnetlib

t = telnetlib.Telnet('localhost')
# t.set_debuglevel(1)          # uncomment this for debugging messages

t.read_until('login:')
t.write('brandon\n')
t.read_until('assword:')      # let "P" be capitalized or not
t.write('mypass\n')
n, match, previous_text = t.expect([r'Login incorrect', r'\$'], 10)
if n == 0:
    print "Username and password failed - giving up"
else:
    t.write('exec uptime\n')
    print t.read_all()        # keep reading until the connection closes
```

If the script is successful, it shows you what the simple uptime command prints on the remote system:

```
root@erlerobot:~/Python_files# python telnet_login.py
10:24:43 up 5 days, 12:13, 14 users, load average: 1.44, 0.91, 0.73
```

The listing shows you the general structure of a session powered by `telnetlib`. First, a connection is established, which is represented in Python by an instance of the `Telnet` object. Here only the hostname is specified, though you can also provide a port number to connect to some other service port than standard Telnet. You can call `set_debuglevel(1)` if you want your `Telnet` object to print out all of the strings that it sends and receives during the session. This actually turned out to be important for writing even the very simple script shown in the listing, because in two different cases it got hung up, and I had to re-run it with debugging messages turned on so that I could see the actual output and fix the script. I generally turn off debugging only once a program is working perfectly, and turn it back on whenever I want to do more work on the script.

Note that `Telnet` does not disguise the fact that its service is backed by a TCP socket, and will pass through to your program any `socket.error` and `socket.gaierror` exceptions that are raised. Once the `Telnet` session is established, interaction generally falls into a receive-and-send pattern, where you wait for a prompt or response from the remote end, then send your next piece of information. The listing illustrates two methods of waiting for text to arrive:

- The very simple `read_until()` method watches for a literal string to arrive, then returns a string providing all of the text that it received from the moment it started listening until the moment it finally saw the string you were waiting for.
- The more powerful and sophisticated `expect()` method takes a list of Python regular expressions. Once the text arriving from the remote end finally adds up to something that matches one of the regular expressions, `expect()` returns three items: the index in your list of the pattern that matched, the regular expression `SRE_Match` object itself, and the text that was received leading up to the matching text. For more information on what you can do with a `SRE_Match`, including finding the values of any sub-expressions in your pattern, read the Standard Library documentation for the `re` module.

If the script sees an error message because of an incorrect password—and does not get stuck waiting forever for a login or password prompt that never arrives or that looks different than it was expecting— then it exits:

```
root@erlerobot:~/Python_files# python telnet_login.py
Username and password failed - giving up
```

If you wind up writing a Python script that has to use `Telnet`, it will simply be a larger or more complicated version of the same simple pattern shown here. Both `read_until()` and `expect()` take an optional second argument named `timeout` that places a maximum limit on how long the call will watch for the text pattern before giving up and returning control to your Python script. If they quit and give up because of the timeout, they do not raise an error; instead—awkwardly enough—they just return the text they have seen so far, and leave it to you to figure out whether that text contains the pattern. There are a few odds and ends in the `Telnet` object that we need not cover here. You will find them in the `telnetlib` Standard Library documentation—including an `interact()` method that lets the user “talk” directly over your `Telnet` connection using the terminal! This kind of call was very popular back in the old days, when you wanted to automate login but then take control and issue normal commands yourself.

Normally, each time a `Telnet` server sends an option request, `telnetlib` flatly refuses to send or receive that option. But you can provide a `Telnet` object with your own callback function for processing options; a modest example is shown in `telnet_codes.py`. For most options, it simply re-implements the default `telnetlib` behavior and refuses to handle any options (and always remember to respond to each option one way or another; failing to do so will often hang the `Telnet` session as the server waits forever for your reply). But if the server expresses interest in the “terminal type” option, then this client sends back a reply of “mypython,” which the shell command it runs after logging in then sees as its `$TERM` environment variable.

```
from telnetlib import Telnet, IAC, DO, DONT, WILL, WONT, SB, SE, TTYPE

def process_option(tsocket, command, option):
    if command == DO and option == TTYPE:
        tsocket.sendall(IAC + WILL + TTYPE)
        print 'Sending terminal type "mypython"'
        tsocket.sendall(IAC + SB + TTYPE + '\0' + 'mypython' + IAC + SE)
    elif command in (DO, DONT):
        print 'Will not', ord(option)
        tsocket.sendall(IAC + WONT + option)
    elif command in (WILL, WONT):
```

```
        print 'Do not', ord(option)
        tsocket.sendall(IAC + DONT + option)

t = Telnet('localhost')
# t.set_debuglevel(1)          # uncomment this for debugging messages

t.set_option_negotiation_callback(process_option)
t.read_until('login:', 5)
t.write('brandon\n')
t.read_until('assword:', 5) # so P can be capitalized or not
t.write('mypass\n')
n, match, previous_text = t.expect([r'Login incorrect', r'\$'], 10)
if n == 0:
    print "Username and password failed - giving up"
else:
    t.write('exec echo $TERM\n')
    print t.read_all()
```


SSH: The Secure Shell

The SSH protocol is one of the best-known examples of a secure, encrypted protocol among modern system administrators (HTTPS is probably the very best known).

SSH is descended from an earlier protocol that supported “remote login,” “remote shell,” and “remote file copy” commands named rlogin, rsh, and rcp, which in their time tended to become much more popular than Telnet at sites that supported them. You cannot imagine what a revelation rcp was particular, unless you have spent hours trying to transfer a file between computers armed with only Telnet and a script that tries to type your password for you, only to discover that your file contains a byte that looks like a control character to Telnet or the remote terminal, and have the whole thing hang until you add a layer of escaping (or figure out how to disable both the Telnet escape key and all interpretation taking place on the remote terminal).

But the best feature of the rlogin family was that they did not just echo username and password prompts without actually knowing the meaning of what was going on. Instead, they stayed involved through the process of authentication, and you could even create a file in your home directory that told them “when someone named brandon tries to connect from the asaph machine, just let them in without a password.” Suddenly, system administrators and Unix users alike received back hours of each month that would otherwise have been spent typing their password. Suddenly, you could copy ten files from one machine to another nearly as easily as you could have copied them into a local folder. SSH has preserved all of these great features of the early remote-shell protocol, while bringing bulletproof security and hard encryption that is trusted worldwide for administering critical servers.

An Overview of SSH

At SSH, we reach a protocol so sophisticated that it actually implements its own rules for multiplexing, so that several “channels” of information can all share the same SSH socket. Every block of information SSH sends across its socket is labeled with a “channel” identifier so that several conversations can share the socket. There are at least two reasons sub-channels make sense. First, even though the channel ID takes up a bit of bandwidth for every single block of information transmitted, the additional data is small compared to how much extra information SSH has to transmit to negotiate and maintain encryption anyway. Second, channels make sense because the real expense of an SSH connection is setting it up. Host key negotiation and authentication can together take up several seconds of real time, and once the connection is established, you want to be able to use it for as many operations as possible. Thanks to the SSH notion of a channel, you can amortize the high cost of connecting by performing many operations before you let the connection close. Once connected, you can create several kinds of channels:

- An interactive shell session, like that supported by Telnet.
- The individual execution of a single command.
- A file-transfer session letting you browse the remote filesystem.
- A port-forward that intercepts TCP connections.

SSH Host Keys

When an SSH client first connects to a remote host, they exchange temporary public keys that let them encrypt the rest of their conversation without revealing any information to any watching third parties. Then, before the client is willing to divulge any further information, it demands proof of the remote server's identity. This makes good sense as a first step: if you are really talking to a hacker who has temporarily managed to grab the remote server's IP, you do not want SSH to divulge even your username—much less your password.

There are many problems with this system from the point of view of SSH. While it is true that you can build a public-key infrastructure internal to an organization, where you distribute your own signing authority's certificates to your web browsers or other applications and then can sign your own server certificates without paying a third party, a public-key infrastructure is still considered too cumbersome a process for something like SSH; server administrators want to set up, use, and tear down servers all the time, without having to talk to a central authority first.

So SSH has the idea that each server, when installed, creates its own random public-private key pair that is not signed by anybody. Instead, one of two approaches is taken to key distribution:

- A system administrator writes a script that gathers up all of the host public keys in an organization, creates an `ssh_known_hosts` listing them all, and places this file in the `/etc/ssh` directory on every system in the organization. Now every SSH client will know about every SSH host key before they even connect for the first time.
- Abandon the idea of knowing host keys ahead of time, and instead memorize them at the moment of first connection. Users of the SSH command line will be very familiar with this: the client says it does not recognize the host to which you are connecting, you reflexively answer “yes,” and its key gets stored in your `~/.ssh/known_hosts` file. You actually have no guarantee that you are really talking to the host you think it is; but at least you will be guaranteed that every subsequent connection you ever make to that machine is going to the right place, and not to other servers that someone is swapping into place at the same IP address.

The familiar prompt from the SSH command line when it sees an unfamiliar host looks like this:

```
root@erlerobot:~# ssh asaph.rhodesmill.org
The authenticity of host 'asaph.rhodesmill.org (74.207.234.78)'
can't be established.
RSA key fingerprint is 85:8f:32:4e:ac:1f:e9:bc:35:58:c1:d4:25:e3:c7:8c.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'asaph.rhodesmill.org,74.207.234.78' (RSA)
to the list of known hosts.
```

That “yes” answer buried deep on the next-to-last full line is the answer that I typed giving SSH the go-ahead to make the connection and remember the key for next time.

The [paramiko](#) library has full support for all of the normal SSH tactics surrounding host keys. But its default behavior is rather spare: it loads no host-key files by default, and will then, of course, raise an exception for the very first host to which you connect because it will not be able to verify its key. The exception that it raises is a bit un-informative; it is only by looking at the fact that it comes from inside the `missing_host_key()` function that I usually recognize what has caused the error. (Before doing this, install paramiko module from Python Package Index):

```
>>> import paramiko
>>> client = paramiko.SSHClient()
>>> client.connect('my.example.com', username='test')
Traceback (most recent call last):
...
File ".../paramiko/client.py", line 85, in missing_host_key
>> raise SSHException('Unknown server %s' % hostname)
paramiko.SSHException: Unknown server my.example.com
```

To behave like the normal SSH command, load both the system and the current user's known-host keys before making the

connection:

```
>>> client.load_system_host_keys()
>>> client.load_host_keys('/home/brandon/.ssh/known_hosts')
>>> client.connect('my.example.com', username='test')
```

The `paramiko` library also lets you choose how you handle unknown hosts. Once you have a client object created, you can provide it with a decision-making class that is asked what to do if a host key is not recognized. You can build these classes yourself by inheriting from the `MissingHostKeyPolicy` class:

```
>>> class AllowAnythingPolicy(paramiko.MissingHostKeyPolicy):
...     def missing_host_key(self, client, hostname, key):
...         return
...
>>> client.set_missing_host_key_policy(AllowAnythingPolicy())
>>> client.connect('my.example.com', username='test')
```

Note that, through the arguments to the `missing_host_key()` method, you receive several pieces of information on which to base your decision; you could, for example, allow connections to machines on your own server subnet without a host key, but disallow all others.

Inside `paramiko` there are also several decision-making classes that already implement several basic host-key options:

- `paramiko.AutoAddPolicy` : Host keys are automatically added to your user host-key store (the file `~/.ssh/known_hosts` on Unix systems) when first encountered, but any change in the host key from then on will raise a fatal exception.
- `paramiko.RejectPolicy` : Connecting to hosts with unknown keys simply raises an exception.
- `paramiko.WarningPolicy` : An unknown host causes a warning to be logged, but the connection is then allowed to proceed.

The `AutoAddPolicy` never needs human interaction, but will at least assure you on subsequent encounters that you are still talking to the same machine as before.

SSH Authentication

Since this chapter is primarily about how to “speak SSH” from Python, I will just briefly outline how authentication works. There are generally three ways to prove your identity to a remote server you are contacting through SSH:

- You can provide a username and password.
- You can provide a username, and then have your client successfully perform a public-key challenge-response. This clever operation manages to prove that you are in possession of a secret “identity” key without actually exposing its contents to the remote system.
- You can perform [Kerberos](#) authentication. If the remote system is set up to allow Kerberos, and if you have run the `kinit` command-line tool to prove your identity to one of the master Kerberos servers in the SSH server's authentication domain, then you should be allowed in without a password.

Since option 3 is very rare, we will concentrate on the first two. Using a username and password with `paramiko` is very easy—you simply provide them in your call to the `connect()` method:

```
>>> client.connect('my.example.com', username='brandon', password=mypass)
```

Public-key authentication, where you use `ssh-keygen` to create an “identity” key pair (which is typically stored in your `~/.ssh` directory) that can be used to authenticate you without a password, makes the Python code even easier.

```
>>> client.connect('my.example.com')
```

If your identity key file is stored somewhere other than in the normal `~/.ssh/id_rsa` file, then you can provide its file name—or a whole Python list of file names—to the `connect()` method manually:

```
>>> client.connect('my.example.com', key_filename='/home/brandon/.ssh/id_sysadmin')
```

Once the `connect()` method has succeeded, you are now ready to start performing remote operations, all of which will be forwarded over the same physical socket without requiring re-negotiation of the host key, your identity, or the encryption that protects the SSH socket itself.

Shell Sessions and Individual Commands

Once you have a connected SSH client, the entire world of SSH operations is open to you. Simply by asking, you can access remote-shell sessions, run individual commands, commence file-transfer sessions, and set up port forwarding.

First, SSH can set up a raw shell session for you, running on the remote end inside a pseudoterminal so that programs act like they normally do when they are interacting with the user at a terminal. This kind of connection behaves very much like a Telnet connection; take a look at `ssh_simple.py` for an example, which pushes a simple echo command at the remote shell, and then asks it to exit.

```
import paramiko

class AllowAnythingPolicy(paramiko.MissingHostKeyPolicy):
    def missing_host_key(self, client, hostname, key):
        return

client = paramiko.SSHClient()
client.set_missing_host_key_policy(AllowAnythingPolicy())
client.connect('127.0.0.1', username='test') # password=''

channel = client.invoke_shell()
stdin = channel.makefile('wb')
stdout = channel.makefile('rb')

stdin.write('echo Hello, world\nexit\r')
print stdout.read()

client.close()
```

If you actually run this command, you will see that the commands you type are actually echoed to you twice, and that there is no obvious way to separate these command echoes from the actual command output.

Because of quirky terminal-dependent behaviors, you should generally avoid ever using `invoke_shell()` unless you are actually writing an interactive terminal program where you let a live user type commands. A much better option for running remote commands is to use `exec_command()`, which, instead of starting up a whole shell session, just runs a single command, giving you control of its standard input, output, and error streams just as though you had run it using the [subprocess module](#) in the Standard Library. As we have seen this module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

A script demonstrating its use is shown in `ssh_commands.py`. The difference between `exec_command()` and a local subprocess is that you do not get the chance to pass command-line arguments as separate strings; instead, you have to pass a whole command line for interpretation by the shell on the remote end.

```
import paramiko

class AllowAnythingPolicy(paramiko.MissingHostKeyPolicy):
    def missing_host_key(self, client, hostname, key):
        return

client = paramiko.SSHClient()
client.set_missing_host_key_policy(AllowAnythingPolicy())
client.connect('127.0.0.1', username='test') # password=''

for command in 'echo "Hello, world!"', 'uname', 'uptime':
    stdin, stdout, stderr = client.exec_command(command)
    stdin.close()
    print repr(stdout.read())
    stdout.close()
    stderr.close()

client.close()
```

Every time you start a new SSH shell session with `invoke_shell()`, and every time you kick off a command with `exec_command()`, a new SSH “channel” is created behind the scenes, which is what provides the file-like Python objects that let you talk to the remote command's standard input, output, and error. Channels, as just explained, can run in parallel, and SSH will cleverly interleave their data on your single SSH connection so that all of the conversations happen simultaneously without ever becoming confused.

Take a look at `ssh_threads.py` for a very simple example of what is possible. Here, two “commands” are kicked off remotely, which are each a simple shell script with some echo commands interspersed with pauses created by calls to `sleep`. The [threading module](#) constructs higher-level threading interfaces on top of the lower level thread module.

```
import threading
import paramiko

class AllowAnythingPolicy(paramiko.MissingHostKeyPolicy):
    def missing_host_key(self, client, hostname, key):
        return

client = paramiko.SSHClient()
client.set_missing_host_key_policy(AllowAnythingPolicy())
client.connect('127.0.0.1', username='test') # password=''

def read_until_EOF(fileobj):
    s = fileobj.readline()
    while s:
        print s.strip()
        s = fileobj.readline()

out1 = client.exec_command('echo One;sleep 2;echo Two;sleep 1;echo Three')[1]
out2 = client.exec_command('echo A;sleep 1;echo B;sleep 2;echo C')[1]
thread1 = threading.Thread(target=read_until_EOF, args=(out1,))
thread2 = threading.Thread(target=read_until_EOF, args=(out2,))
thread1.start()
thread2.start()
thread1.join()
thread2.join()

client.close()
```

In order to be able to process these two streams of data simultaneously, we are kicking off two threads, and are handing each of them one of the channels from which to read. They each print out each line of new information as soon as it arrives, and finally exit when the `readline()` command indicates end-of-file by returning an empty string. When run, this script should return something like this:

```
root@erlerobot:~/Python_files# python ssh_threads.py
One
A
B
Two
Three
C
```

SSH channels over the same TCP connection are completely independent, can each receive (and send) data at their own pace, and can close independently when the particular command that they are talking to finally terminates.

SFTP: File Transfer Over SSH

Version 2 of the SSH protocol includes a sub-protocol called the “SSH File Transfer Protocol” (SFTP) that lets you walk the remote directory tree, create and delete directories and files, and copy files back and forth from the local to the remote machine. The capabilities of SFTP are so complex and complete, in fact, that they support not only simple file-copy operations, but can power graphical file browsers and can even let the remote filesystem be mounted locally.

When talking about SFTP commands than is provided by the bare `paramiko` documentation for the Python SFTP client (<http://www.lag.net/paramiko/docs/paramiko.SFTPClient-class>); here are the main things to remember when doing SFTP:

- The SFTP protocol is stateful, just like FTP, and just like your normal shell account. So you can either pass all file and directory names as absolute paths that start at the root of the filesystem, or use `getcwd()` and `chdir()` to move around the filesystem and then use paths that are relative to the directory in which you have arrived.
- You can open a file using either the `file()` or `open()` method and you get back a file-like object connected to an SSH channel that runs independently of your SFTP channel.
- Because each open remote file gets an independent channel, file transfers can happen asynchronously; you can open many remote files at once and have them all streaming down to your disk drive, or open new files and be sending data the other way.
- Finally, keep in mind that no shell expansion is done on any of the file names you pass across SFTP. If you try using a file name like `*` or one that has spaces or special characters, they are simply interpreted as part of the file name. This means that any support for pattern-matching that you want to provide to the user has to be through fetching the directory contents yourself and then checking their pattern against each one, using a routine like those provided in `fnmatch` in the Python Standard Library. `fnmatch` module provides support for Unix shell-style wildcards, which are not the same as regular expressions.

A very modest example SFTP session is shown in `sftp.py`. It does something simple that system administrators might often need: it connects to the remote system and copies messages log files out of the `/var/log` directory, perhaps for scanning or analysis on the local machine. The `functools` module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module, as shown in the `sftp.py`:

```
import functools
import paramiko

class AllowAnythingPolicy(paramiko.MissingHostKeyPolicy):
    def missing_host_key(self, client, hostname, key):
        return

client = paramiko.SSHClient()
client.set_missing_host_key_policy(AllowAnythingPolicy())
client.connect('127.0.0.1', username='test') # password=''

def my_callback(filename, bytes_so_far, bytes_total):
    print 'Transfer of %r is at %d/%d bytes (%.1f%%)' % (
        filename, bytes_so_far, bytes_total, 100. * bytes_so_far / bytes_total)

sftp = client.open_sftp()
sftp.chdir('/var/log')
for filename in sorted(sftp.listdir()):
    if filename.startswith('messages.'):
        callback_for_filename = functools.partial(my_callback, filename)
        sftp.get(filename, filename, callback=callback_for_filename)

client.close()
```

Note that, although I made a big deal of talking about how each file that you open with SFTP uses its own independent channel, the simple `get()` and `put()` convenience functions provided by `paramiko`— which are really lightweight wrappers

for an `open()` followed by a loop that reads and writes—do not attempt any asynchrony, but instead just block and wait until each whole file has arrived. This means that the foregoing script calmly transfers one file at a time, producing output that looks something like this:

```
root@erlerobot:~/Python_files# python sftp.py
Transfer of 'messages.1' is at 32768/128609 bytes (25.5%)
Transfer of 'messages.1' is at 65536/128609 bytes (51.0%)
Transfer of 'messages.1' is at 98304/128609 bytes (76.4%)
Transfer of 'messages.1' is at 128609/128609 bytes (100.0%)
Transfer of 'messages.2.gz' is at 32768/40225 bytes (81.5%)
Transfer of 'messages.2.gz' is at 40225/40225 bytes (100.0%)
Transfer of 'messages.3.gz' is at 28249/28249 bytes (100.0%)
Transfer of 'messages.4.gz' is at 32768/71703 bytes (45.7%)
Transfer of 'messages.4.gz' is at 65536/71703 bytes (91.4%)
Transfer of 'messages.4.gz' is at 71703/71703 bytes (100.0%)
```


File Transfer Protocol (FTP)

The File Transfer Protocol (FTP) was once among the most widely used protocols on the Internet, invoked whenever a user wanted to transfer files between Internet-connected computers.

In this chapter we will examine this protocol and study the possible alternatives.

What to Use Instead of FTP

Today, there are better alternatives than the FTP protocol for pretty much anything you could want to do with it.

The biggest problem with the protocol is its lack of security: not only files, but usernames and passwords are sent completely in the clear and can be viewed by anyone observing network traffic.

A second issue is that an FTP user tends to make a connection, choose a working directory, and do several operations all over the same network connection. Modern Internet services, with millions of users, prefer protocols like HTTP that consist of short, completely self-contained requests, instead of long-running FTP connections that require the server to remember things like a current working directory.

A final big issue is filesystem security. The early FTP servers, instead of showing users just a sliver of the host filesystem that the owner wanted exposed, tended to simply expose the entire filesystem, letting users `cd` to `/` and snoop around to see how the system was configured.

So what are the alternatives?

- For file download, HTTP is the standard protocol on today's Internet, protected with SSL when necessary for security. Instead of exposing systemspecific file name conventions like FTP, HTTP supports system-independent URLs.
- Anonymous upload is a bit less standard, but the general tendency is to use a form on a web page that instructs the browser to use an HTTP POST operation to transmit the file that the user selects.
- File synchronization has improved immeasurably since the days when a recursive FTP file copy was the only common way to get files to another computer. Instead of wastefully copying every file, modern commands like `rsync` or `rdist` efficiently compare files at both ends of the connection and copy only the ones that are new or have changed.
- Full filesystem access is actually the one area where FTP can still commonly be found on today's Internet: thousands of cut-rate ISPs continue to support FTP, despite its insecurity, as the means by which users copy their media and (typically) PHP source code into their web account. A much better alternative today is for service providers to support SFTP instead.

Communication Channels

FTP is unusual because, by default, it actually uses two TCP connections during operation. One connection is the control channel, which carries commands and the resulting acknowledgments or error codes. The second connection is the data channel, which is used solely for transmitting file data or other blocks of information, such as directory listings. Technically, the data channel is full duplex, meaning that it allows files to be transmitted in both directions simultaneously. However, in actual practice, this capability is rarely used.

The process of downloading a file from an FTP server ran mostly like this:

1. First, the FTP client establishes a command connection by connecting to the FTP port on the server.
2. The client authenticates itself, usually with username and password.
3. The client changes directory on the server to where it wants to deposit or retrieve files.
4. The client begins listening on a new port for the data connection, and then informs the server about that port.
5. The server connects to the port the client requested.
6. The file is transmitted.
7. The data connection is closed.

FTP also supports what is known as *passive mode*. In this scenario, the data connection is made backward: the server opens an extra port, and tells the client to make the second connection. Other than that, everything behaves the same way.

Using FTP in Python

The Python module `ftplib` is the primary interface to FTP for Python programmers. It handles the details of establishing the various connections for you, and provides convenient ways to automate common commands. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other ftp servers. It is also used by the module `urllib` to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet [RFC 959](#).

`connect.py` shows a very basic `ftplib` example. The program connects to a remote server, displays the welcome message, and prints the current working directory.

```
from ftplib import FTP

f = FTP('ftp.ibiblio.org')
print "Welcome:", f.getwelcome()
f.login()

print "Current working directory:", f.pwd()
f.quit()
```

Recall that an FTP session can visit different directories, just like a shell prompt can move between locations with `cd`. Here, the `pwd()` function returns the current working directory on the remote site of the connection. Finally, the `quit()` function logs out and closes the connection. Here is what the program outputs when run:

```
root@erlerobot:~/Python_files# python connect.py
Welcome: 220 ProFTPD Server
Current working directory: /
```

ASCII and Binary Files

When making an FTP transfer, you have to decide whether you want the file treated as a monolithic block of binary data, or whether you want it parsed as a text file so that your local machine can paste its lines back together using whatever end-of-line character is native to your platform. A file transferred in so-called “ASCII mode” is delivered one line at a time, so that you can glue the lines back together on the local machine using its own line-ending convention. Take a look at `asciidl.py` for a Python program that downloads a well-known text file and saves it in your local directory.

```
import os
from ftplib import FTP

if os.path.exists('README'):
    raise IOError('refusing to overwrite your README file')

def writeline(data):
    fd.write(data)
    fd.write(os.linesep)

f = FTP('ftp.kernel.org')
f.login()
f.cwd('/pub/linux/kernel')

fd = open('README', 'w')
f.retrlines('RETR README', writeline)
fd.close()

f.quit()
```

In the example, the `cwd()` function selects a new working directory on the remote system. Then the `retrlines()` function begins the transfer. Its first parameter specifies a command to run on the remote system, usually `RETR`, followed by a file name. Its second parameter is a function that is called, over and over again, as each line of the text file is retrieved; if omitted, the data is simply printed to standard output. The lines are passed with the end-of-line character stripped, so the homemade `writeline()` function simply appends your system’s standard line ending to each line as it is written out. Try running this program; there should be a file in your current directory named `README` after the program is done. Basic binary file transfers work in much the same way as text-file transfers; `binarydl.py` shows an example.

```
import os
from ftplib import FTP

if os.path.exists('patch8.gz'):
    raise IOError('refusing to overwrite your patch8.gz file')

f = FTP('ftp.kernel.org')
f.login()
f.cwd('/pub/linux/kernel/v1.0')

fd = open('patch8.gz', 'wb')
f.retrbinary('RETR patch8.gz', fd.write)
fd.close()

f.quit()
```

When run, it deposits a file named `patch8.gz` in your current working directory. The `retrbinary()` function simply passes blocks of data to the specified function. This is convenient, since a file object’s `write()` function expects just such data—so in this case, no custom function is necessary.

Advanced Binary Downloading

The `ftplib` module provides a second function that can be used for binary downloading: `ntransfercmd()`. This command provides a lower-level interface, but can be useful if you want to know a little bit more about what's going on during the download. In particular, this more advanced command lets you keep track of the number of bytes transferred, and you can use that information to display status updates for the user. `advbinarydl.py` shows a sample program that uses `ntransfercmd()`.

```
import os, sys
from ftplib import FTP

if os.path.exists('linux-1.0.tar.gz'):
    raise IOError('refusing to overwrite your linux-1.0.tar.gz file')

f = FTP('ftp.kernel.org')
f.login()

f.cwd('/pub/linux/kernel/v1.0')
f.voidcmd("TYPE I")

datasock, size = f.ntransfercmd("RETR linux-1.0.tar.gz")
bytes_so_far = 0
fd = open('linux-1.0.tar.gz', 'wb')

while 1:
    buf = datasock.recv(2048)
    if not buf:
        break
    fd.write(buf)
    bytes_so_far += len(buf)
    print "\rReceived", bytes_so_far,
    if size:
        print "of %d total bytes (%.1f%%)" % (
            size, 100 * bytes_so_far / float(size)),
    else:
        print "bytes",
    sys.stdout.flush()

print
fd.close()
datasock.close()
f.voidresp()
f.quit()
```

There are a few new things to note here. First comes the call to `voidcmd()`. This passes an FTP command directly to the server, checks for an error, but returns nothing. In this case, the raw command is TYPE I. That sets the transfer mode to “image,” which is how FTP refers internally to binary files. In the previous example, `retrbinary()` automatically ran this command behind the scenes, but the lower-level `ntransfercmd()` does not. Next, note that `ntransfercmd()` returns a tuple consisting of a data socket and an estimated size. Always bear in mind that the size is merely an estimate, and should not be considered authoritative; the file may end sooner, or it might go on much longer, than this value. Also, if a size estimate from the FTP server is simply not available, then the estimated size returned will be `None`.

After receiving the data, it is important to close the data socket and call `voidresp()`, which reads the command response code from the server, raising an exception if there was any error during transmission. Even if you do not care about detecting errors, failing to call `voidresp()` will make future commands likely to fail because the server's output socket will be blocked waiting for you to read the results. Here is an example of running this program:

```
root@erlerobot:~/Python_files# python advbinarydl.py
Received 1259161 of 1259161 bytes (100.0%)
```

Uploading Data

File data can also be uploaded through FTP. As with downloading, there are two basic functions for uploading:

`storbinary()` and `storlines()`. Both take a command to run, and a file-like object to transmit. The `storbinary()` function will call the `read()` method repeatedly on that object until its content is exhausted, while `storlines()`, by contrast, calls the `readline()` method. Unlike the corresponding download functions, these methods do not require you to provide a callable function of your own. (But you could, of course, pass a file-like object of your own crafting whose `read()` or `readline()` method computes the outgoing data as the transmission proceeds. `binaryu1.py` shows how to upload a file in binary mode.

```
from ftplib import FTP
import sys, getpass, os.path

if len(sys.argv) != 5:
    print "usage: %s <host> <username> <localfile> <remotedir>" % (
        sys.argv[0])
    exit(2)

host, username, localfile, remotedir = sys.argv[1:]
password = getpass.getpass(
    "Enter password for %s on %s: " % (username, host))

f = FTP(host)
f.login(username, password)
f.cwd(remotedir)

fd = open(localfile, 'rb')
f.storbinary('STOR %s' % os.path.basename(localfile), fd)
fd.close()

f.quit()
```

This program looks quite similar to our earlier efforts. Since most anonymous FTP sites do not permit file uploading, you will have to find a server somewhere to test it against; I simply installed the old, venerable `ftpd` on my laptop for a few minutes and ran the test like this:

```
root@erlerobot:~/Python_files# python binaryu1.py localhost brandon test.txt /tmp
```

You can modify this program to upload a file in ASCII mode by simply changing `storbinary()` to `storlines()`.

Advanced Binary Uploading

Just like the download process had a complicated raw version, it is also possible to upload files “by hand” using `ntransfercmd()`, as shown in `advbinaryul.py`.

```
from ftplib import FTP
import sys, getpass, os.path

BLOCKSIZE = 8192 # chunk size to read and transmit: 8 kB

if len(sys.argv) != 5:
    print "usage: %s <host> <username> <localfile> <remotedir>" % (
        sys.argv[0])
    exit(2)

host, username, localfile, remotedir = sys.argv[1:]
password = getpass.getpass("Enter password for %s on %s: " % \
    (username, host))
f = FTP(host)
f.login(username, password)

f.cwd(remotedir)
f.voidcmd("TYPE I")

fd = open(localfile, 'rb')
datasock, esize = f.ntransfercmd('STOR %s' % os.path.basename(localfile))
size = os.stat(localfile)[6]
bytes_so_far = 0

while 1:
    buf = fd.read(BLOCKSIZE)
    if not buf:
        break
    datasock.sendall(buf)
    bytes_so_far += len(buf)
    print "\rSent", bytes_so_far, "of", size, "bytes", \
        "\r" + "(%.1f%%)\r" % (100 * bytes_so_far / float(size))
    sys.stdout.flush()

print
datasock.close()
fd.close()
f.voidresp()
f.quit()
```

Now we can perform an upload that continuously displays its status as it progresses:

```
root@erlerobot:~/Python_files# python binaryul.py localhost brandon patch8.gz /tmp
Enter password for brandon on localhost:
Sent 6408 of 6408 bytes (100.0%)
```


Handling Errors

Like most Python modules, `ftplib` will raise an exception when an error occurs. It defines several exceptions of its own, and it can also raise `socket.error` and `IOError`. As a convenience, it offers a tuple, named `ftplib.all_errors`, that lists all of the exceptions that can possibly be raised by `ftplib`. This is often a useful shortcut for writing a `try...except` clause.

One of the problems with the basic `retrbinary()` function is that, in order to use it easily, you will usually wind up opening the file on the local end before beginning the transfer on the remote side. If your command aimed at the remote side retorts that the file does not exist, or if the `RETR` command otherwise fails, then you will have to close and delete the local file you have just created (or else wind up littering the filesystem with zero-length files).

With the `ntransfercmd()` method, by contrast, you can check for a problem prior to opening a local file. `nlst.py` already follows these guidelines: if `ntransfercmd()` fails, the exception will cause the program to terminate before the local file is opened. Scanning DirectoriesFTP provides two ways to discover information about server files and directories. These are implemented in `ftplib` as the `nlst()` and `dir()` methods.

- The `nlst()` method returns a list of entries in a given directory—all of the files and directories inside. However, the bare names are all that is returned. There is no other information about which particular entries are files or are directories, on the sizes of the files present, or anything else.
- The more powerful `dir()` function returns a directory listing from the remote. This listing is in a system-defined format, but typically contains a file name, size, modification date, and file type. On UNIX servers, it is typically the output of one of these two shell commands:

```
root@erlerobot:~# ls -l
root@erlerobot:~# ls -la
```

`nlst.py` shows an example of using `nlst()` to get directory information.

```
from ftplib import FTP

f = FTP('ftp.ibiblio.org')
f.login()
f.cwd('/pub/academic/astronomy/')
entries = f.nlst()
entries.sort()
print len(entries), "entries:"
for entry in entries:
    print entry
f.quit()
```

`nlst.py` shows an example of using `nlst()` to get directory information. When you run this program, you will see output like this:

```
root@erlerobot:~/Python_files# python nlst.py
13 entries:
INDEX
README
ephem_4.28.tar.Z
hawaii_scope
incoming
jupiter-moons.shar.Z
lunar.c.Z
lunisolar.shar.Z
moon.shar.Z
planetary
sat-track.tar.Z
stars.tar.Z
xephem.tar.Z
```

If you were to use an FTP client to manually log on to the server, you would see the same files listed. Notice that the file names are in a convenient format for automated processing—a bare list of file names—but that there is no extra information. The result will be different when we try another file listing command in `dir.py` :

```
from ftplib import FTP

f = FTP('ftp.ibiblio.org')
f.login()
f.cwd('/pub/academic/astronomy/')
entries = []
f.dir(entries.append)
print "%d entries:" % len(entries)
for entry in entries:
    print entry
f.quit()
```

Notice that the filenames are in a convenient format for automated processing — a bare list of filenames — but that is no extra information. Contrast the bare list of file names we saw earlier with the output from `dir.py` , which uses `dir()` :

```
root@erlerobot:~/Python_files# python dir.py
13 entries:
-rw-r--r-- 1 (?) » (?) » » 750 Feb 14 1994 INDEX
-rw-r--r-- 1 root » bin » » 135 Feb 11 1999 README
-rw-r--r-- 1 (?) » (?) » » 341303 Oct 2 1992 ephem_4.28.tar.Z
drwxr-xr-x 2 (?) » (?) » » 4096 Feb 11 1999 hawaii_scope
drwxr-xr-x 2 (?) » (?) » » 4096 Feb 11 1999 incoming
-rw-r--r-- 1 (?) » (?) » » 5983 Oct 2 1992 jupiter-moons.shar.Z
-rw-r--r-- 1 (?) » (?) » » 1751 Oct 2 1992 lunar.c.Z
-rw-r--r-- 1 (?) » (?) » » 8078 Oct 2 1992 lunisolar.shar.Z
-rw-r--r-- 1 (?) » (?) » » 64209 Oct 2 1992 moon.shar.Z
drwxr-xr-x 2 (?) » (?) » » 4096 Jan 6 1993 planetary
-rw-r--r-- 1 (?) » (?) » » 129969 Oct 2 1992 sat-track.tar.Z
-rw-r--r-- 1 (?) » (?) » » 16504 Oct 2 1992 stars.tar.Z
-rw-r--r-- 1 (?) » (?) » » 410650 Oct 2 1992 xephem.tar.Z
```

The `dir()` method takes a function that it calls for each line, delivering the directory listing in pieces just like `retrlines()` delivers the contents of particular files. Here, we simply supply the `append()` method of our plain old Python entries list.

Detecting Directories and Recursive Download

If you cannot guarantee what information an FTP server might choose to return from its `dir()` command, how are you going to tell directories from normal files—an essential step to downloading entire trees of files from the server? The answer, shown in `recursed1.py`, is to simply try a `cwd()` into every name that `nlst()` returns and, if you succeed, conclude that the entity is a directory. This sample program does not do any actual downloading; instead, to keep things simple, it simply prints out the directories it visits to the screen.

```
import os, sys
from ftplib import FTP, error_perm

def walk_dir(f, dirpath):
    original_dir = f.pwd()
    try:
        f.cwd(dirpath)
    except error_perm:
        return # ignore non-directories and ones we cannot enter
    print dirpath
    names = f.nlst()
    for name in names:
        walk_dir(f, dirpath + '/' + name)
    f.cwd(original_dir) # return to cwd of our caller

f = FTP('ftp.kernel.org')
f.login()
walk_dir(f, '/pub/linux/kernel/Historic/old-versions')
f.quit()
```

This sample program will run a bit slow—there are, it turns out, quite a few files in the old-versions directory on the Linux Kernel Archive— but within a few dozen seconds, you should see the resulting directory tree displayed on the screen:

```
root@erlerobot:~/Python_files# python recursed1.py
/pub/linux/kernel/Historic/old-versions
/pub/linux/kernel/Historic/old-versions/impure
/pub/linux/kernel/Historic/old-versions/old
/pub/linux/kernel/Historic/old-versions/old/corrupt
/pub/linux/kernel/Historic/old-versions/tytso
```

Creating Directories, Deleting Things

Finally, FTP supports file deletion, and supports both the creation and deletion of directories. These more obscure calls are all described in the `ftplib` documentation:

- `delete(filename)` will delete a file from the server.
- `mkd(dirname)` attempts to create a new directory.
- `rmd(dirname)` will delete a directory; note that most systems require the directory to be empty first.
- `rename(oldname, newname)` works, essentially, like the Unix command `mv`: if both names are in the same directory, the file is essentially re-named; but if the destination specifies a name in a different directory, then the file is actually moved.

Doing FTP Securely

To use TLS, create your FTP connection with the `FTP_TLS` class instead of the plain `FTP` class; simply by doing this, your username and password and, in fact, the entire FTP command channel will be protected from prying eyes. If you then additionally run the class's `prot_p()` method (it takes no arguments), then the FTP data connection will be protected as well. Should you for some reason want to return to using an un-encrypted data connection during the session, there is a `prot_c()` method that returns the data stream to normal. Again, your commands will continue to be protected as long as you are using the `FTP_TLS` class.

Check the Python Standard Library documentation for more details (they include a small code sample) if you wind up needing this extension to FTP: http://docs.python.org/library/ftplib.html#ftplib.FTP_TLS

Remote Procedure Call (RPC)

Remote Procedure Call (RPC) systems let you call a remote function using the same syntax that you would use when calling a routine in a local API or library. This tends to be useful in two situations: First, when your program has a lot of work to do, and you want to spread it across several machines by making calls across the network; and second, when you need data or information that is only available on another hard drive or network.

In this chapter we will try to know RCP better and learn how we can use it in combination con Python.

Features of RPC

Besides serving their the essential purpose of letting you make what appear to be local function or method calls that are in fact passing across the network to a different server, RPC protocols have several key features, and also some differences, that you should keep in mind when choosing and then deploying an RPC client or server.

First, every RPC mechanism has limits on the kind of data you can pass. The most popular protocols, therefore, support only a few kinds of numbers and strings; one sequence or list data type; and then something like a struct or associative array.

A second common feature is the ability of the server to signal that an exception occurred while it was running the remote function. In such cases, the client RPC library will typically raise an exception itself to tell the client that something has gone wrong.

Third, many RPC mechanisms provide introspection, which is a way for clients to list the calls that are supported and perhaps to discover what arguments they take.

Fourth, each RPC mechanism needs to support some addressing scheme whereby you can reach out and connect to a particular remote API. Some such mechanisms are quite complicated, and they might even have the ability to automatically connect you to the correct server on your network for performing a particular task, without your having to know its name beforehand. Other mechanisms are quite simple and just ask you for the IP address, port number, or URL of the service you want to access. These mechanisms expose the underlying network addressing scheme, rather than creating a scheme of their own.

Finally, some RPC mechanisms support authentication, access control, and even full impersonation of particular user accounts when RPC calls are made by several different client programs wielding different credentials.

XML-RPC

XML-RPC has native support in Python precisely because it was one of the first RPC protocols of the Internet age, operating natively over HTTP instead of insisting on its own on-the-wire protocol. This means our examples will not even require any third-party modules. While we will see that this makes our RPC server somewhat less capable than if we moved to a third-party library, this will also make the examples good ones for an initial foray into RPC.

If you have ever used raw XML, then you are familiar with the fact that it lacks any data-type semantics; it cannot represent numbers, for example, but only elements that contain other elements, text strings, and text-string attributes. Thus the XML-RPC specification has to build additional semantics on top of the plain XML document format in order to specify how things like numbers should look when converted into marked-up text. The Python Standard Library makes it easy to write either an XML-RPC client or server, though more power is available when writing a client. For example, the client library supports HTTP basic authentication, while the server does not support this. Therefore, we will begin at the simple end, with the server.

`xmlrpc_server.py` shows a basic server that starts a web server on port 7001 and listens for incoming Internet connections. Here we will use the [operator module](#), which exports a set of efficient functions corresponding to the intrinsic operators of Python.

```
import operator, math
from SimpleXMLRPCServer import SimpleXMLRPCServer

def addtogether(*things):
    """Add together everything in the list `things`."""
    return reduce(operator.add, things)

def quadratic(a, b, c):
    """Determine `x` values satisfying: `a` * x*x + `b` * x + c == 0"""
    b24ac = math.sqrt(b*b - 4.0*a*c)
    return list(set([ (-b-b24ac) / 2.0*a,
                     (-b+b24ac) / 2.0*a ]))

def remote_repr(arg):
    """Return the `repr()` rendering of the supplied `arg`."""
    return arg

server = SimpleXMLRPCServer(('127.0.0.1', 7001))
server.register_introspection_functions()
server.register_multicall_functions()
server.register_function(addtogether)
server.register_function(quadratic)
server.register_function(remote_repr)
print "Server ready"
server.serve_forever()
```

You can see that the three sample functions that the server offers over XML-RPC — the ones that are added to the RPC service through the `register_function()` calls — are quite typical Python functions. And that, again, is the whole point of XML-RPC: it lets you make routines available for invocation over the network without having to write them any differently than if they were normal functions offered inside of your program.

Note that two additional configuration calls are made in addition to the three calls that register our functions. Each of them turns on an additional service that is optional, but often provided by XML-RPC servers: an introspection routine that a client can use to ask which RPC calls are supported by a given server; and the ability to support a multicall function that lets several individual function calls be bundled together into a single network round-trip. This server will need to be running before we can try any of the next three program listings, so bring up a command window and get it started:

```
root@erlerobot:~/Python_files# python xmlrpc_server.py
Server ready
```

This means that the server is now waiting for connections on localhost port 7001.

Now, open another command window and get ready to try out the next three listings as we review them. First, we will try out the introspection capability that we turned on in this particular server. Note that this ability is optional, and it may not be available on many other XML-RPC services that you use online or that you deploy yourself. `xmlrpc_introspect.py` shows how introspection happens from the client's point of view.

```
import xmlrpclib
proxy = xmlrpclib.ServerProxy('http://127.0.0.1:7001')

print 'Here are the functions supported by this server:'
for method_name in proxy.system.listMethods():

    if method_name.startswith('system.'):
        continue

    signatures = proxy.system.methodSignature(method_name)
    if isinstance(signatures, list) and signatures:
        for signature in signatures:
            print '%s(%s)' % (method_name, signature)
    else:
        print '%s(...)' % (method_name,)

    method_help = proxy.system.methodHelp(method_name)
    if method_help:
        print ' ', method_help
```

The introspection mechanism is an optional extension that is not actually defined in the XML-RPC specification itself. The client is able to call a series of special methods that all begin with the string `system.` to distinguish them from normal methods. These special methods give information about the other calls available. We start by calling `listMethods()`. If introspection is supported at all, then we will receive back a list of other method names; for this example listing, we ignore the system methods and only proceed to print out information about the other ones. In the `xmlrpc_introspect.py` we use the `xmlrpc` module, this module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

```
root@erlerobot:~/Python_files# python xmlrpc_introspect.py
Here are the functions supported by this server:
addtogether(...)
    Add together everything in the list `things`.
quadratic(...)
    Determine `x` values satisfying: `a` * x*x + `b` * x + c == 0
remote_repr(...)
    Return the `repr()` rendering of the supplied `arg`.
```

You will recall that the whole point of an RPC service is to make function calls in a target language look as natural as possible. And as you can see in `xmlrpc_client.py` the Standard Library's `xmlrpclib` gives you a proxy object for making function calls against the server. These calls look exactly like local function calls.

```
import xmlrpclib
proxy = xmlrpclib.ServerProxy('http://127.0.0.1:7001')
print proxy.addtogether('x', 'y', 'z')
print proxy.addtogether(20, 30, 4, 1)
print proxy.quadratic(2, -4, 0)
print proxy.quadratic(1, 2, 1)
print proxy.remote_repr((1, 2.0, 'three'))
print proxy.remote_repr([1, 2.0, 'three'])
print proxy.remote_repr({'name': 'Arthur', 'data': {'age': 42, 'sex': 'M'}})
print proxy.quadratic(1, 0, 1)
```

Note how almost all of the calls work without a hitch, and how both of the calls in this listing and the functions themselves back in `xmlrpc_server.py` look like completely normal Python; there is with nothing about them that is particular to a network:


```

root@erlerobot:~/Python_files# python xmlrpc_client.py
xyz
55
[0.0, 8.0]
[-1.0]
[1, 2.0, 'three']
[1, 2.0, 'three']
{'data': {'age': [42], 'sex': 'M'}, 'name': 'Arthur'}
Traceback (most recent call last):
...
xmlrpclib.Fault: <Fault 1: "<type 'exceptions.ValueError':math domain error">

```

Note that XML-RPC function calls, like those of Python and many other languages in its lineage, can take several arguments, but can only return a single result value. That value might be a complex data structure, but it will be returned as a single result. And the protocol does not care whether that result has a consistent shape or size; the list returned by `quadratic()` varies in its number of elements returned without any complaint from the network logic. Note, also, that the rich variety of Python data types must be reduced to the smaller set that XMLRPC itself happens to support. In particular, XML-RPC only supports a single sequence type: the list.

Thus far we have covered the general features and restrictions of XML-RPC. If you consult the documentation for either the client or the server module in the Standard Library, you can learn about a few more features. In particular, you can learn how to use TLS and authentication by supplying more arguments to the `ServerProxy` class. But one feature is important enough to go ahead and cover here: the ability to make several calls in a network round-trip when the server supports it, as shown in `xmlrpc_multicall.py`.

```

import xmlrpclib
proxy = xmlrpclib.ServerProxy('http://127.0.0.1:7001')
multicall = xmlrpclib.MultiCall(proxy)
multicall.addtogether('a', 'b', 'c')
multicall.quadratic(2, -4, 0)
multicall.remote_repr([1, 2.0, 'three'])
for answer in multicall():
    print answer

```

When you run this script, you can carefully watch the server's command window to confirm that only a single HTTP request is made in order to answer all three function calls that get made.

Three final points are worth mentioning before we move on to examining another RPC mechanism:

- There are two additional data types that sometimes prove hard to live without, so many XML-RPC mechanisms support them: dates and the value that Python calls `None`. Python's client and server both support options that will enable the transmission and reception of these nonstandard types.
- Keyword arguments are, alas, not supported by XML-RPC, because few languages are sophisticated enough to include them and XML-RPC wants to interoperate with those languages. Some services get around this by allowing a dictionary to be passed as a function's final argument.
- Finally, keep in mind that dictionaries can only be passed if all of their keys are strings, whether normal or Unicode. See the "Self-documenting Data" section later in this chapter for more information on how to think about this restriction.

JSON-RPC

The bright idea behind JSON is to serialize data structures to strings that use the syntax of the JavaScript programming language. This means that JSON strings can be turned back into data in a web browser simply by using the `eval()` function. By using a syntax specifically designed for data rather than adapting a verbose document markup language like XML, this remote procedure call mechanism can make your data much more compact while simultaneously simplifying your parsers and library code.

JSON-RPC is not supported in the Python Standard Library, so you will have to choose one of the several third-party distributions available. You can find these distributions on the Python Package Index. My own favorite is [lovely.jsonrpc](#). If you install it in a virtual environment, then you can try out the server and client shown in Listings `jsonrpc_server.py` and `jsonrpc_client.py`.

```
from wsgiref.simple_server import make_server
import lovely.jsonrpc.dispatcher, lovely.jsonrpc.wsgi

def lengths(*args):
    results = []
    for arg in args:
        try:
            arglen = len(arg)
        except TypeError:
            arglen = None
        results.append((arglen, arg))
    return results

dispatcher = lovely.jsonrpc.dispatcher.JSONRPCDispatcher()
dispatcher.register_method(lengths)
app = lovely.jsonrpc.wsgi.WSGIJSONRPCApplication({'': dispatcher})
server = make_server('localhost', 7002, app)
print "Starting server"
while True:
    server.handle_request()
```

The server code is quite simple, as an RPC mechanism should be. As with XML-RPC, we merely need to name the functions that we want offered over the network, and they become available for queries.

```
from lovely.jsonrpc import proxy
proxy = proxy.ServerProxy('http://localhost:7002')
print proxy.lengths((1,2,3), 27, {'Sirius': -1.46, 'Rigel': 0.12})
```

First, note that the protocol allowed us to send as many arguments as we wanted; it was not bothered by the fact that it could not introspect a static method signature from our function. Second, note that the `None` value in the server's reply passes back to us unhindered.

```
root@erlerobot:~/Python_files# python jsonrpc_server.py Starting server [In another command window:] $ python
jsonrpc_client.py [[3, [1, 2, 3]], [None, 27], [2, {'Rigel': 0.12, 'Sirius': -1.46}]]
```

Self-documenting Data

You have just seen that both XML-RPC and JSON-RPC appear to support a data structure very much like a Python dictionary, but with an annoying limitation. In XML-RPC, the data structure is called a struct, whereas JSON calls it an object. To the Python programmer, however, it looks like a dictionary, and your first reaction will probably be annoyance that its keys cannot be integers, floats, or tuples. Let us look at a concrete example. Imagine that you have a dictionary of physical element symbols indexed by their atomic number:

```
{1: 'H', 2: 'He', 3: 'Li', 4: 'Be', 5: 'B', 6: 'C', 7: 'N', 8: 'O'}
```

If you need to transmit this dictionary over an RPC mechanism, simply put, the struct and object RPC data structures are not designed to pair keys with values in containers of an arbitrary size. Instead, they are designed to associate a small set of pre-defined attribute names with the attribute values that they happen to carry for some particular object. If you try to use a struct to pair random keys and values, you might inadvertently make it very difficult to use for people unfortunate enough to be using statically-typed programming languages. Instead, you should think of dictionaries being sent across RPCs as being like the `__dict__` attributes of your Python objects, which you should generally not find yourself using to associate an arbitrary set of keys with values.

All of this means that the dictionary that I showed a few moments ago should actually be serialized as a list of explicitly labelled values if it is going to be used by a general-purpose RPC mechanism:

```
{'number': 1, 'symbol': 'H'},  
{'number': 2, 'symbol': 'He'},  
{'number': 3, 'symbol': 'Li'},  
{'number': 4, 'symbol': 'Be'},  
{'number': 5, 'symbol': 'B'},  
{'number': 6, 'symbol': 'C'},  
{'number': 7, 'symbol': 'N'},  
{'number': 8, 'symbol': 'O'}}
```

Note that the preceding examples show the Python dictionary as you will pass it into your RPC call, not the way it would be represented on the wire.

If you have a Python dictionary like the one we are discussing here, you can turn it into an RPCappropriate data structure, and then change it back with code like this:

```
>>> elements = {1: 'H', 2: 'He'}  
>>> t = [ {'number': key, 'symbol': elements[key]} for key in elements ]  
>>> t  
[{'symbol': 'H', 'number': 1}, {'symbol': 'He', 'number': 2}]  
>>> dict( (obj['number'], obj['symbol']) for obj in t )  
{1: 'H', 2: 'He'}
```

Using named tuples might be an even better way to marshal such values before sending them if you find yourself creating and destroying too many dictionaries to make this transformation appealing.

Talking About Objects: Pyro and RPyC

If the idea of RPC was to make remote function calls look like local ones, then the two basic RPC mechanisms we have looked at actually fail pretty spectacularly. If the functions we were calling happened to only use basic data types in their arguments and return values, then XML-RPC and JSONRPC would work fine. But think of all of the occasions when you use more complex parameters and return values instead! What happens when you need to pass live objects?

When all you have are Python programs that need to talk to each other, there is at least one excellent reason to look for an RPC service that knows about Python objects and their ways: Python has a number of very powerful data types, so it can simply be unreasonable to try “talking down” to the dialect of limited data formats like XML-RPC and JSON-RPC. This is especially true when Python dictionaries, sets, and datetime objects would express exactly what you want to say. There are two Python-native RPC systems that we should mention: Pyro and RPyC. The Pyro project lives here:

<http://www.xs4all.nl/~irmen/pyro3/>

This well-established RPC library is built on top of the Python pickle module, and it can send any kind of argument and response value that is inherently pickle-able. Basically, this means that, if an object (and its attributes) can be reduced to its basic types, then it can be transmitted. However, if the values you want to send or receive are ones that the [pickle module](#) chokes on, then Pyro will not work for your situation. The `pickle` module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream is converted back into an object hierarchy.

An RPyC Example

The RPyC project lives here: <http://rpyc.wikidot.com/>

This project takes a much more sophisticated approach toward objects. Indeed, where what actually gets passed across the network is a reference to an object that can be used to call back and invoke more of its methods later if the receiver needs to. The most recent version also seems to have put more thought into security, which is important if you are letting other organizations use your RPC mechanism. After all, if you let someone give you some data to un-pickle, you are essentially letting them run arbitrary code on your computer.

You can see an example client and server in Listings `rpyc_client.py` and `rpyc_server.py`. If you want an example of the incredible kinds of things that a system like RPyC makes possible, you should study these listings closely.

```
import rpyc

def noisy(string):
    print 'Noisy:', repr(string)

proxy = rpyc.connect('localhost', 18861, config={'allow_public_attrs': True})
fileobj = open('testfile.txt')
linecount = proxy.root.line_counter(fileobj, noisy)
print 'The number of lines in the file was', linecount
```

At first the client might look like a rather standard program using an RPC service. After all, it calls a generically-named `connect()` function with a network address, and then accesses methods of the returned proxy object as though the calls were being performed locally.

The server exposes a single method that takes the proffered file object and callable function. It uses these exactly as you would in a normal Python program that was happening inside a single process. It calls the file object's `readlines()` and expects the return value to be an iterator over which a for loop can repeat. Finally, the server calls the function object that has been passed in without any regard for where the function actually lives (namely, in the client).

```
import rpyc

class MyService(rpyc.Service):
    def exposed_line_counter(self, fileobj, function):
        for linenum, line in enumerate(fileobj.readlines()):
            function(line)
        return linenum + 1

from rpyc.utils.server import ThreadedServer
t = ThreadedServer(MyService, port = 18861)
t.start()
```

It is especially instructive to look at the output generated by running the client, assuming that a small `testfile.txt` indeed exists in the current directory and that it has a few words of wisdom inside:

```
root@erlerobot:~/Python_files# python rpyc_client.py
Noisy: 'Simple\n'
Noisy: 'is\n'
Noisy: 'better\n'
Noisy: 'than\n'
Noisy: 'complex.\n'
The number of lines in the file was 5
```

Equally startling here are two facts. First, the server was able to iterate over multiple results from `readlines()`, even though this required the repeated invocation of file-object logic that lived on the client. Second, the server didn't somehow

copy the `noisy()` function's code object so it could run the function directly; instead, it repeatedly invoked the function, with the correct argument each time, on the client side of the connection.

RPyC takes exactly the opposite approach from the other RPC mechanisms we have looked at. Whereas all of the other techniques try to serialize and send as much information across the network as possible, and then leave the remote code to either succeed or fail with no further information from the client, the RPyC scheme only serializes completely immutable items such as Python integers, floats, strings, and tuples. For everything else, it passes across an object name that lets the remote side reach back into the client to access attributes and invoke methods on those live objects.

RPC, Web Frameworks, Message Queues

Be willing to explore alternative transmission mechanisms for your work with RPC services. The classes provided in the Python Standard Library for XML-RPC, for example, are not even used by many Python programmers who need to speak that protocol.

There are three useful ways that you can look into moving beyond overly simple example code that makes it look as though you have to bring up a new web server for every RPC service you want to make available from a particular site.

First, look into whether you can use the pluggability of WSGI to let you install an RPC service that you have incorporated into a larger web project that you are deploying. Implementing both your normal web application and your RPC service as WSGI servers beneath a filter that checks the incoming URL enables you to allow both services to live at the same hostname and port number.

Second, instead of using a dedicated RPC library, you may find that your web framework of choice already knows how to host an XML-RPC, JSON-RPC, or some other flavor of RPC call.

Third, you might want to try sending RPC messages over an alternate transport that does a better job than the protocol's native transport of routing the calls to servers that are ready to handle them. Message queues are often an excellent vehicle for RPC calls when you want a whole rack of servers to stay busy sharing the load of incoming requests.

Recovering From Network Errors

Of course, there is one reality of life on the network that RPC services cannot easily hide: the network can be down or even go down in the middle of a particular RPC call. You will find that most RPC mechanisms simply raise an exception if a call is interrupted and does not complete. Note that an error, unfortunately, is no guarantee that the remote end did not process the request — maybe it actually did finish processing it, but then the network went down right as the last packet of the reply was being sent. In this case, your call would have technically happened and the data would have been successfully added to the database or written to a file or whatever the RPC call does. However, you will think the call failed and want to try it again — possibly storing the same data twice.

Binary Options: Thrift and Protocol Buffers

It is possible you will want both features: a compact and efficient binary format and support across several different languages. Here are a few options:

- Some JSON-RPC libraries support the BSON protocol, which provides a tight binary transport format and also an expanded range of data types beyond those supported by JSON.
- The Apache Foundation is now incubating Thrift, an RPC system developed several years ago at Facebook and released as open source.
- Google Protocol Buffers are popular with many programmers, but strictly speaking they are not a full RPC system; instead, they are a binary data serialization protocol.