Sockets programming in Python

Develop networking applications using Python's basic sockets features

M. Tim Jones (mtj@mtjones.com)
Senior Principal Software Engineer
Emulex Corp.

04 October 2005

This tutorial shows how to develop sockets-based networking applications using Python. In this tutorial, you first learn a few Python basics and see why Python makes a good network programming language. Then you move on to the basic sockets features of Python, using a sample chat application as a guide, and look at several other, high-level, classes that provide asynchronous communications.

Before you start

About this tutorial

Python is a popular object-oriented scripting language with a simple syntax and a large developer base. It is a general purpose language and can be used in a variety of settings. It's also popular as a beginner's programming language, much like the BASIC language of the 1970s.

This tutorial demonstrates the Python language with emphasis on networking programming. I define the basic sockets features of Python in addition to some of Python's other classes that provide asynchronous sockets. I also detail Python's application-layer protocol classes, showing how to build Web clients, mail servers and clients, and more.

I also demonstrate a simple chat server to illustrate the power of Python for sockets applications.

You should have a basic understanding of the standard BSD Sockets API and some experience with the GNU/Linux® environment. Some familiarity with object-oriented concepts is also beneficial.

Prerequisites

This tutorial and the examples demonstrated in it rely on version 2.4 of Python. You can download this version from the Python Web site (see Resources for a link). To build the Python interpreter,

you need the GNU C compiler (gcc) and the `configure/make` utilities (which are part of any standard GNU/Linux distribution).

You should have a basic understanding of the standard BSD Sockets API and some experience with the GNU/Linux environment. Some familiarity with object-oriented concepts is also beneficial.

Introducing Python

First, I'll provide a taste of Python.

What is Python?

Python is a general purpose object-oriented scripting language that can be applied to a large variety of problems. It was created in the early 1990s at CWI in Amsterdam and continues to evolve today under the Python Software Foundation.

Python is amazingly portable and can be found in almost all operating systems.

Python is interpreted and is easy to extend. You can extend Python by adding new modules that include functions, variables, or types through compiled C or C++ functions.

You can also easily embed Python within C or C++ programs, allowing you to extend an application with scripting capabilities.

One of the most useful aspects of Python is its massive number of extension modules. These modules provide standard functions such as string or list processing, but there are also application-layer modules for video and image processing, audio processing, and yes, networking.

A taste of Python

I'll give you a sense of what Python is all about.

As an interpreted language, it's easy to use Python to try out ideas and quickly prototype software. Python programs can be interpreted as a whole or line by line.

You can test the following snippets of Python code by first running Python, then typing each line one at a time. After Python is invoked, a prompt (>>>) appears to allow you to type commands. Note that indentation is important in Python, so the preceding spaces in a line must not be ignored:

Listing 1. Some Python samples to try out

```
# Open a file, read each line, and print it out
for line in open('file.txt'):
  print line


# Create a file and write to it
file = open("test.txt", "w")
file.write("test line\n")
file.close()
```

```
# Create a small dictionary of names and ages and manipulate
family = {'Megan': 13, 'Elise': 8, 'Marc': 6}

# results in 8
family['Elise']

# Remove the key/value pair
del family['Elise']


# Create a list and a function that doubles its input.  Map the
#    function to each of the elements of the list (creating a new
#    list as a result).
arr = [1, 2, 3, 4, 5]
def double(x): return x*x
map(double, arr)


# Create a class, inherit by another, and then instantiate it and
#    invoke its methods.
class Simple:
  def __init__(self, name):
    self.name = name

  def hello(self):
    print self.name+" says hi."

class Simple2(Simple):
  def goodbye(self):
    print self.name+" says goodbye."

me = Simple2("Tim")
me.hello()
me.goodbye()
```

Why use Python?

The number one reason to learn and use Python is its popularity. The size of its user base and the growing number of applications built with Python make it a worthwhile investment.

You find Python in several development areas -- it's used to build system utilities, as a glue language for program integration, for Internet applications, and for rapid prototyping.

Python also has some advantages over other scripting languages. It has a simple syntax and is conceptually clear, making it easy to learn. Python is also easier and more descriptive when using complex data structures (such as lists, dictionaries, and tuples). Python can also extend languages and be extended by languages in turn.

I find that Python's syntax makes it more readable and maintainable than Perl but less so than Ruby. Python's advantage over Ruby is the large number of libraries and modules that are available. Using these, you can build feature-rich programs with little custom code.

Python's use of indentation to identify block scope can be rather annoying, but its simplicity tends to make up for this minor flaw.

Now, let's dig into sockets programming in Python.

Python sockets modules

Basic Python sockets modules

Python offers two basic sockets modules. The first, `Socket`, provides the standard BSD Sockets API. The second, `SocketServer`, provides a server-centric class that simplifies the development of network servers. It does this in an asynchronous way in which you can provide plug-in classes to do the application-specific jobs of the server. Table 1 lists the classes and modules that this section covers.

Table 1. Python classes and modules

Class/module	Description
Socket	Low-level networking interface (per the BSD API)
SocketServer	Provides classes that simplify the development of network servers

Let's look at each of these modules to understand what they can do for you.

The `Socket` module

The `Socket` module provides the basic networking services with which UNIX® programmers are most familiar (otherwise known as the *BSD API*). This module has everything you need to build socket servers and clients.

The difference between this API and the standard C API is in its object orientation. In C, a socket descriptor is retrieved from the socket call then used as a parameter to the BSD API functions. In Python, the `socket` method returns a socket object to which the socket methods can be applied. Table 2 provides some of the class methods and Table 3 shows a subset of the instance methods.

Table 2. Class methods for the Socket module

Class method	Description
`Socket`	Low-level networking interface (per the BSD API)
`socket.socket(family, type)`	Create and return a new socket object
`socket.getfqdn(name)`	Convert a string quad dotted IP address to a fully qualified domain name
`socket.gethostbyname(hostname)`	Resolve a hostname to a string quad dotted IP address
`socket.fromfd(fd, family, type)`	Create a socket object from an existing file descriptor

Table 3. Instance methods for the Socket module

Instance method	Description
`sock.bind( (adrs, port) )`	Bind the socket to the address and port
`sock.accept()`	Return a client socket (with peer address information)
`sock.listen(backlog)`	Place the socket into the listening state, able to pend *backlog* outstanding connection requests
`sock.connect( (adrs, port) )`	Connect the socket to the defined host and port

`sock.recv( buflen[, flags] )`	Receive data from the socket, up to `buflen` bytes
`sock.recvfrom( buflen[, flags] )`	Receive data from the socket, up to `buflen` bytes, returning also the remote host and port from which the data came
`sock.send( data[, flags] )`	Send the data through the socket
`sock.sendto( data[, flags], addr )`	Send the data through the socket
`sock.close()`	Close the socket
`sock.getsockopt( lvl, optname )`	Get the value for the specified socket option
`sock.setsockopt( lvl, optname, val )`	Set the value for the specified socket option

The difference between a *class method* and an *instance method* is that instance methods require a socket instance to be performed (returned from socket), where class methods do not.

The SocketServer module

The `SocketServer` module is an interesting module that simplifies the development of socket servers. A discussion of its use is far beyond the scope of this tutorial, but I'll demonstrate its basic use and then refer you to the Resources section for links to a more detailed discussion.

Consider the simple case shown in Listing 2. Here, I implement a simple "Hello World" server which emits the message when a client connects. I first create a request handler that inherits the `SocketServer.StreamRequestHandler` class. I define a method called `handle` that handles the requests for the server. Everything the server does must be handled within the context of this function (at the end, the socket is closed). This process works in simple cases, but you can implement even simple HTTP servers with this class. In the `handle` method, I emit my salutation and then exit.

Now that the connection handler is ready, all that's left is to create the socket server. I use the `SocketServer.TCPServer` class, providing the address and port number (to which the server will be bound) and my request-handler method. The result is a `TCPServer` object. Calling the `serve_forever` method starts the server and makes it available for connections.

Listing 2. Implementing a simple server with the SocketServer module

```
import SocketServer

class hwRequestHandler( SocketServer.StreamRequestHandler ):
  def handle( self ):
    self.wfile.write("Hello World!\n")


server = SocketServer.TCPServer( ("", 2525), hwRequestHandler )
server.serve_forever()
```

That's it! Python permits a number of variations on this theme, including `UDPServers` and forking and threading servers.

Sockets programming in Python

In languages with sockets, the socket is universally the same -- it's a conduit between the two applications that can communicate with one another.

Preliminaries

Whether you're writing a sockets application in Python, Perl, Ruby, Scheme, or any other useful language (and by *useful* I mean languages that have a sockets interface), the socket is universally the same. It's a conduit between the two applications that can communicate with one another (either locally on a single machine or between two machines in separate locations).

The difference with sockets programming in a language like Python is in the helper classes and methods that can simplify sockets programming. In this section I'll demonstrate the Python `socket` API. You can execute the Python interpreter with a script or, if you execute Python by itself, you can interact with it one line at a time. In this way, you can see the result of each method invoked.

The following example illustrates interacting with the Python interpreter. Here, I use the `socket` class method `gethostbyname` to resolve a fully qualified domain name (www.ibm.com) to a string quad-dotted IP address ('129.42.19.99'):

Listing 3. Using the socket API from the interpreter command line

```
[camus]$ python
Python 2.4 (#1, Feb 20 2005, 11:25:45)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-5)] on linux2
Type "help", "copyright", "credits" or "license" for more
   information.
>>> import socket
>>> socket.gethostbyname('www.ibm.com')
'129.42.19.99'
>>>
```

After the `socket` module is imported, I invoke the `gethostbyname` class method to resolve the domain name to an IP address.

Now, I'll discuss the basic `socket` methods and communicating through sockets. Feel free to follow along with your Python interpreter.

Creating and destroying sockets

To create a new socket, you use the `socket` method of the `socket` class. This is a class method because you don't yet have a `socket` object from which to apply the methods. The `socket` method is similar to the BSD API, as demonstrated in the creation of a stream (TCP) and datagram (UDP) socket:

Listing 4. Creating stream and datagram sockets

```
streamSock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )

dgramSock = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
```

In each case, a `socket` object is returned. The `AF_INET` symbol -- argument one -- indicates that you're requesting an Internet Protocol (IP) socket, specifically *IPv4.* The second argument is the transport protocol type (`SOCK_STREAM` for TCP sockets and `SOCK_DGRAM` for UDP sockets). If your underlying operating system supports IPv6, you can also specify `socket.AF_INET6` to create an IPv6 socket.

To close a connected socket, you use the `close` method:

```
streamSock.close()
```

Finally, you can delete a socket with the `del` statement:

```
del streamSock
```

This statement permanently removes the `socket` object. Attempting to reference the socket thereafter produces an error.

Socket addresses

An endpoint address for a socket is a tuple consisting of an interface address and a port number. Because Python can represent tuples easily, the address and port are represented as such. This illustrates an endpoint for interface address 192.168.1.1 and port 80:

```
( '192.168.1.1', 80 )
```

You can also use a fully qualified domain name here, such as:

```
( 'www.ibm.com', 25 )
```

This example is simple and certainly beats the `sockaddr_in` manipulation that's necessary in C. The following discussion provides examples of addresses in Python.

Server sockets

Server sockets are typically those that expose a service on a network. Because server and client sockets are created in different ways, I discuss them independently.

After you create the socket, you use the `bind` method to bind an address to it, the `listen` method to place it in the listening state, and finally the `accept` method to accept a new client connection. This is demonstrated below:

Listing 5. Using server sockets

```
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
sock.bind( ('', 2525) )
sock.listen( 5 )
newsock, (remhost, remport) = sock.accept()
```

For this server, the address `('', 2525)` is used which means that the wildcard is used for the interface address `('')`, allowing incoming connections from any interface on the host. You also bind to port number 2525.

Note here that the `accept` method returns not only the new `socket` object that represents the client connection (`newsock`) but also an address tuple (the remote address and port number of the peer end of the socket). Python's `SocketServer` module can simplify this process even further, as demonstrated above.

You can also create datagram servers, but they are connectionless and therefore have no associated `accept` method. The following example creates a datagram server socket:

Listing 6. Creating a datagram server socket

```
sock = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
sock.bind( ('', 2525) )
```

The upcoming discussion of sockets I/O shows how I/O works for both stream and datagram sockets.

Now, let's explore how a client creates a socket and connects it to a server.

Client sockets

The mechanisms for creating and connecting client sockets are similar to the setup of server sockets. Upon creating a socket, an address is needed -- not to locally bind the socket (as is the case with a server) but rather to identify where the socket should attach. Say there's a server on a host with an interface IP address of '192.168.1.1' and port 2525. The following code creates a new socket and connects it to the defined server:

Listing 7. Creating a stream socket and connecting to the server

```
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
sock.connect( ('192.168.1.1', 2525) )
```

For datagram sockets, the process is a bit different. Recall that datagram sockets are by nature disconnected. One way to the think about it is as follows: Whereas stream sockets are pipes between two endpoints, datagram sockets are message-based, able to communicate with multiple peers at the same time. Here's an example of a datagram client.

Listing 8. Creating a datagram socket and connecting to the server

```
sock = socket.socket( socket.AF_INET, sock.sock_DGRAM )
sock.connect( ('192.168.1.1', 2525) )
```

What's different here is that even though I've used the `connect` method, there's no real *connection* between the client and server. The connect here is a simplification for later I/O. Typically in datagram sockets, you must provide the destination information with the data that you want to send. By using `connect`, I've cached this information with the client and `send` methods can occur much like stream socket versions (no destination address is necessary). You can call `connect` again to re-specify the target of the datagram client's messages.

Stream sockets I/O

Sending or receiving data through stream sockets is simple in Python. Several methods exist to move data through a stream socket (such as `send`, `recv`, `read`, and `write`).

This first example demonstrates a server and client for stream sockets. In this demonstration, the server echoes whatever it receives from the client.

The echo stream server is presented in Listing 9. Upon creating a new stream socket, an address is bound to it (accept connections from any interface and port 45000) and then the `listen` method is invoked to enable incoming connections. The echo server then goes into a loop for client connections. The `accept` method is called and blocks (that is, does not return) until a new client connects, at which point the new client socket is returned along with address information for the remote client. With this new client socket, I call `recv` to get a string from the peer, then write this string back out to the socket. I then immediately close the socket.

Listing 9. Simple Python stream echo server

```
import socket

srvsock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
srvsock.bind( ('', 23000) )
srvsock.listen( 5 )

while 1:

  clisock, (remhost, remport) = srvsock.accept()
  str = clisock.recv(100)
  clisock.send( str )
  clisock.close()
```

Listing 10 shows the echo client that corresponds with the server in Listing 9. Upon creating a new stream socket, the `connect` method is used to attach this socket to the server. When connected (when the `connect` method returns), the client emits a simple text message with the `send` method, then awaits the echo with the `recv` method. The `print` statement is used to emit what's read. When this is done, the `close` method is performed to close the socket.

Listing 10. Simple Python stream echo server

```
import socket

clisock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )

clisock.connect( ('', 23000) )

clisock.send("Hello World\n")
print clisock.recv(100)

clisock.close()
```

Datagram sockets I/O

Datagram sockets are disconnected by nature which means that communication requires that a destination address be provided. Similarly, when a message is received through a socket, the source of the data must be returned. The `recvfrom` and `sendto` methods support the additional address information as you can see in the datagram echo server and client implementations.

Listing 11 shows the datagram echo server. A socket is first created and then bound to an address using the `bind` method. An infinite loop is then entered for serving client requests. The `recvfrom` method receives a message from the datagram socket and returns not only the message but also the address of the source of the message. This information is then turned around with the `sendto` method to return the message to the source.

Listing 11. Simple Python datagram echo server

```
import socket

dgramSock = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
dgramSock.bind( ('', 23000) )

while 1:

  msg, (addr, port) = dgramSock.recvfrom( 100 )
  dgramSock.sendto( msg, (addr, port) )
```

The datagram client is even simpler. After creating a datagram socket, I use the `sendto` method to send a message to a specific address. (Remember: Datagrams have no connection.) After `sendto` finishes, I await the echo response with `recv`, then print it. Note that I don't use `recvfrom` here because I'm not interested in the peer address information.

Listing 12. Simple Python datagram echo client

```
import socket

dgramSock = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )

dgramSock.sendto( "Hello World\n", ('', 23000) )
print dgramSock.recv( 100 )
dgramSock.close()
```

Socket options

Sockets default to a set of standard behaviors, but it's possible to alter the behavior of a socket using options. You manipulate socket options with the `setsockopt` method and capture them with the `getsockopt` method.

Using socket options is simple in Python, as demonstrated in Listing 13. In the first example, I read the size of the socket send buffer. In the second example, I get the value of the `SO_REUSEADDR` option (reuse the address within the `TIME_WAIT` period) and then enable it.

Listing 13. Using socket options

```
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )

# Get the size of the socket's send buffer
bufsize = sock.getsockopt( socket.SOL_SOCKET, socket.SO_SNDBUF )

# Get the state of the SO_REUSEADDR option
state = sock.getsockopt( socket.SOL_SOCKET, socket.SO_REUSEADDR )

# Enable the SO_REUSEADDR option
sock.setsockopt( socket.SOL_SOCKET, socket.SO_REUSEADDR, 1 )
```

The `SO_REUSEADDR` option is most often used in socket server development. You can increase the socket send and receive buffers for greater performance, but given that you're operating here in an interpreted scripting language, it may not provide you with much benefit.

Asynchronous I/O

Python offers asynchronous I/O as part of the `select` module. This feature is similar to the C select mechanism but has some simplifications. I'll first introduce `select` and then show you how to use it in Python.

The `select` method allows you to multiplex events for several sockets and for several different events. For example, you can instruct `select` to notify you when a socket has data available, when it's possible to write data through a socket, and when an error occurs on a socket; and you can perform these actions for many sockets at the same time.

Where C works with bitmaps, Python uses lists to represent the descriptors to monitor and also the return descriptors whose constraints are satisfied. Consider the following example in which you await some input from standard input:

Listing 14. Awaiting input from stdin

```
rlist, wlist, elist = select.select( [sys.stdin], [], [] )

print sys.stdin.read()
```

The arguments passed to `select` are lists representing read events, write events, and error events. The `select` method returns three lists containing the objects whose events were satisfied (read, write, exception). In this example, upon return `rlist` should be `[sys.stdin]`, indicating that data are available to read on stdin. This data are then read with the `read` method.

The `select` method also works on socket descriptors. In the following example (see Listing 15), two client sockets are created and connected to a remote peer. The `select` method is then used to identify which socket has data available for reading. The data are then read and emitted to stdout.

Listing 15. Demonstrating the `select` method with multiple sockets

```
import socket
import select

sock1 = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
sock2 = socket.socket( socket.AF_INET, socket.SOCK_STREAM )

sock1.connect( ('192.168.1.1', 25) )
sock2.connect( ('192.168.1.1', 25) )

while 1:

  # Await a read event
  rlist, wlist, elist = select.select( [sock1, sock2], [], [], 5 )

  # Test for timeout
  if [rlist, wlist, elist] == [ [], [], [] ]:

    print "Five seconds elapsed.\n"

  else:

    # Loop through each socket in rlist, read and print the available data
    for sock in rlist:
```

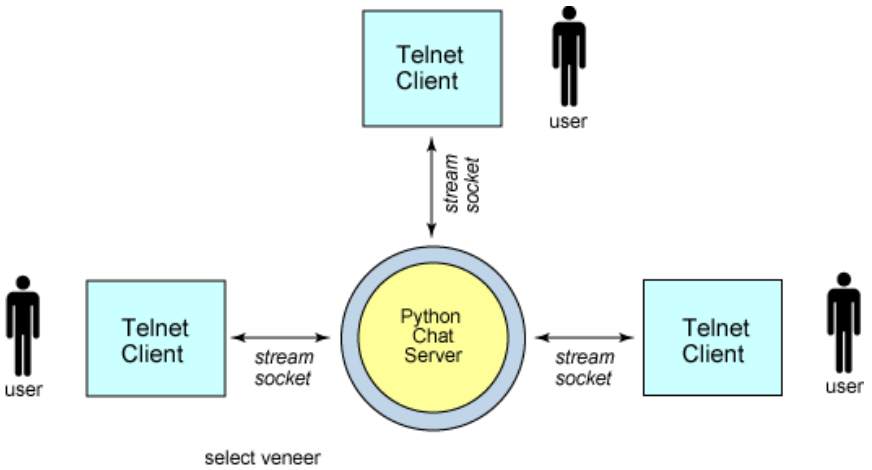
```
print sock.recv( 100 )
```

Building a Python chat server

A simple chat server

You've explored the basic networking APIs for Python; now you can put this knowledge to use in a simple application. In this section, you'll build a simple chat server. Using Telnet, clients can connect to your Python chat server and globally communicate with one another. Messages submitted to the chat server are viewed by others (in addition to management information, such as clients joining or leaving the chat server). This model is shown graphically in Figure 1.

Figure 1. The chat server uses the select method to support an arbitrary number of clients



An important requirement to place on your chat server is that it must be scalable. The server must be able to support an arbitrary number of stream (TCP) clients.

To support an arbitrary number of clients, you use the `select` method to asynchronously manage your client list. But you also use a feature of `select` for your server socket. The read event of `select` determines when a client has data available for reading, but it also can be used to determine when a server socket has a new client trying to connect. You exploit this behavior to simplify the development of the server.

Next, I'll explore the source of the Python chat server and identify the ways in which Python helps simplify its implementation.

The `ChatServer` class

Let's start by looking at the Python chat server class and the `__init__` method -- the constructor that's invoked when a new instance is created.

The class is made up of four methods. The `run` method is invoked to start the server and permit client connections. The `broadcast_string` and `accept_new_connection` methods are used internally in the class and will be discussed shortly.

The `__init__` method is a special method that's invoked when a new instance of the class is created. Note that all methods take the `self` argument, a reference to the class instance itself (much like the `this` parameter in C++). You'll see the `self` parameter, part of all instance methods, used here to access instance variables.

The `__init__` method creates three instance variables. The `port` is the port number for the server (passed in the constructor). The `srvsock` is the socket object for this instance, and `descriptors` is a list that contains each `socket` object for the class. You use this list within the `select` method to identify the read event list.

Finally, Listing 16 shows the code for the `__init__` method. After creating a stream socket, the `SO_REUSEADDR` socket option is enabled so that the server can be quickly restarted, if necessary. The wildcard address is bound with the defined port number. Then the `listen` method is invoked to permit incoming connections. The server socket is added to the `descriptors` list (the only element at present), but all client sockets will be added as they arrive (see `accept_new_connection`). A salutation is provided to stdout indicating that the server has started.

Listing 16. The ChatServer class with the `init` method

```
import socket
import select

class ChatServer:

  def __init__( self, port ):
    self.port = port;

    self.srvsock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
    self.srvsock.setsockopt( socket.SOL_SOCKET, socket.SO_REUSEADDR, 1 )
    self.srvsock.bind( ("", port) )
    self.srvsock.listen( 5 )

    self.descriptors = [self.srvsock]
    print 'ChatServer started on port %s' % port

  def run( self ):
    ...

  def broadcast_string( self, str, omit_sock ):
    ...

  def accept_new_connection( self ):
    ...
```

The `run` method

The `run` method is the server loop for your chat server (see Listing 17). When called, it enters an infinite loop, providing communication between connected clients.

The core of the server is the `select` method. I pass the `descriptor` list (which contains all the server's sockets) as the read event list to `select` (and null lists for write and exception). When a read event is detected, it's returned as `sread`. (I ignore the `swrite` and `sexc` lists.) The `sread` list contains the `socket` objects that will be serviced. I iterate through the `sread` list, checking each socket object found.

The first check in the iterator loop is if the `socket` object is the server. If it is, a new client is trying to connect and the `accept_new_connection` method is called. Otherwise, the client socket is read. If a null is returned from `recv`, the peer socket closed.

In this case, I construct a message and send it to all connected clients, close the peer socket, and remove the corresponding object from the `descriptor` list. If the `recv` return is not null, a message is available and stored in `str`. This message is distributed to all other clients using `broadcast_string`.

Listing 17. The chat server `run` method is the core of the chat server

```
def run( self ):

  while 1:

    # Await an event on a readable socket descriptor
    (sread, swrite, sexc) = select.select( self.descriptors, [], [] )

    # Iterate through the tagged read descriptors
    for sock in sread:

      # Received a connect to the server (listening) socket
      if sock == self.srvsock:
        self.accept_new_connection()
      else:

        # Received something on a client socket
        str = sock.recv(100)

        # Check to see if the peer socket closed
        if str == '':
          host,port = sock.getpeername()
          str = 'Client left %s:%s\r\n' % (host, port)
          self.broadcast_string( str, sock )
          sock.close
          self.descriptors.remove(sock)
        else:
          host,port = sock.getpeername()
          newstr = '[%s:%s] %s' % (host, port, str)
          self.broadcast_string( newstr, sock )
```

Helper methods

The two helper methods in the chat server class provide methods for accepting new client connections and broadcasting messages to the connected clients.

The `accept_new_connection` method (see Listing 18) is called when a new client is detected on the incoming connection queue. The `accept` method is used to accept the connection, which returns the new `socket` object and remote address information. I immediately add the new socket to the `descriptors` list, then send a salutation to the new client welcoming the client to the chat. I create a string identifying that the client has connected and broadcast this information to the group using `broadcast_string` (see Listing 19).

Note that in addition to the string being broadcast, a socket object is also passed. The reason is that I want to selectively omit some sockets from getting certain messages. For example, when a client sends a message to the group, the message goes to the group but not back to itself. When

I generate the status message identifying a new client joining the group, it goes to the group but not the new client. This task is performed in `broadcast_string` with the `omit_sock` argument. This method walks through the `descriptors` list and sends the string to all sockets that are not the server socket and not `omit_sock`.

Listing 18. Accepting a new client connection on the chat server

```
def accept_new_connection( self ):

  newsock, (remhost, remport) = self.srvsock.accept()
  self.descriptors.append( newsock )

  newsock.send("You're connected to the Python chatserver\r\n")
  str = 'Client joined %s:%s\r\n' % (remhost, remport)
  self.broadcast_string( str, newsock )
```

Listing 19. Broadcasting a message to the chat group

```
def broadcast_string( self, str, omit_sock ):

  for sock in self.descriptors:
    if sock != self.srvsock and sock != omit_sock:
      sock.send(str)

  print str,
```

Instantiating a new `ChatServer`

Now that you've seen the Python chat server (under 50 lines of code), let's see how to instantiate a new chat server object in Python.

Start the server by creating a new `ChatServer` object (passing the port number to be used), then calling the `run` method to start the server and allow incoming connections:

Listing 20. Instantiating a new chat server

```
myServer = ChatServer( 2626 )
myServer.run()
```

At this point, the server is running and you can connect to it from one or more clients. You can also chain methods together to simplify this process (as if it needs to be simpler):

Listing 21. Chaining methods

```
myServer = ChatServer( 2626 ).run()
```

which achieves the same result. I'll show the `ChatServer` class in operation.

Demonstrating the `ChatServer`

Here's the `ChatServer` in action. I show the output of the `ChatServer` (see Listing 22 and the dialog between two clients (see Listing 23 and Listing 24). The user-entered text appears in bold.

Listing 22. Output from the `ChatServer`

```
[plato]$ python pchatsrvr.py
ChatServer started on port 2626
Client joined 127.0.0.1:37993
Client joined 127.0.0.1:37994
[127.0.0.1:37994] Hello, is anyone there?
[127.0.0.1:37993] Yes, I'm here.
[127.0.0.1:37993]  Client left 127.0.0.1:37993
```

Listing 23. Output from Chat Client #1

```
[plato]$ telnet localhost 2626
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
You're connected to the Python chatserver
Client joined 127.0.0.1:37994
[127.0.0.1:37994] Hello, is anyone there?
Yes, I'm here.^]
telnet> close
Connection closed.
[plato]$
```

Listing 24. Output from Chat Client #2

```
[plato]$ telnet localhost 2626
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
You're connected to the Python chatserver
Hello, is anyone there?
[127.0.0.1:37993] Yes, I'm here.
[127.0.0.1:37993] Client left 127.0.0.1:37993
```

As you see in Listing 22, all dialog between all clients is emitted to stdout, including client connect and disconnect messages.

High-level networking classes

Networking modules

Python includes several specialized modules for application-layer protocols (built on the standard socket module). The available modules are wide and varied and they provide module implementations of the Hypertext Transfer Protocol (HTTP), the Simple Mail Transfer Protocol (SMTP), Internet Message Access Protocol (IMAP) and Post Office Protocol (POP3), the Network News Transfer Protocol (NNTP), XML-RPC (remote procedure call), FTP, and many others.

This section demonstrates the modules shown in Table 4.

Table 4. Useful application-layer protocol modules

Module	Protocol implemented
`httplib`	HTTP client

`smtplib`	SMTP client
`poplib`	POP3 client

The `httplib` (HTTP client)

The HTTP client interface can be useful when developing Web robots or other Internet scraping agents. The Web protocol is request/response in nature over stream sockets. Python makes it easy to build Web robots through a simple Web interface.

Listing 25 demonstrates the `httplib` module. You create a new HTTP client instance with `HTTPConnection`, providing the Web site to which you want to connect. With this new object (`httpconn`), you can request files with the `request` method. Within `request`, you specify the HTTP `GET` method (which requests a file from the server, compared to `HEAD` which simply retrieves information about the file). The `getresponse` method parses the HTTP response header to understand if an error was returned. If the file was successfully retrieved, the `read` method on the new response object returns and prints the text.

Listing 25. Building a simple (non-rendering) HTTP client with `httplib`

```
import httplib

httpconn = httplib.HTTPConnection("www.ibm.com")

httpconn.request("GET", "/developerworks/index.html")

resp = httpconn.getresponse()

if resp.reason == "OK":

  resp_data = resp.read()

  print resp_data

httpconn.close()
```

The `smptlib` (SMTP client)

SMTP allows you to send e-mail messages to a mail server which can be useful in networking systems to relay status about the operation of a device. The Python module for sending e-mail messages is simple and consists of creating an `SMTP` object, sending an e-mail message using the `sendmail` method, then closing the connection with the `quit` method.

The example in Listing 26 demonstrates sending a simple e-mail message. The `msg` string contains the body of the message (which should include the subject line).

Listing 26. Sending a short e-mail message with `smtplib`

```
import smtplib

fromAdrs = 'mtj@mtjones.com'
toAdrs = 'you@mail.com'
msg = 'From: me@mail.com\r\nTo: you@mail.com\r\nSubject:Hello\r\nHi!\r\n'

mailClient = smtplib.SMTP('192.168.1.1')
mailClient.sendmail( fromAdrs, toAdrs, msg )
mailClient.quit
```

The `poplib` (POP3 client)

POP3 is another useful application-layer protocol for which a module exists within Python. The POP3 protocol allows you to connect to a mail server and download new mail, which can be useful for remote commanding -- embedding commands within the body of an e-mail message. After executing the embedded command, you can use `smptlib` to return a response e-mail message to the source.

This demonstration in Listing 27 shows a simple application that connects to a mail server and emits the subject lines for all pending e-mail for the user.

The `poplib` is relatively simple but offers several methods for gathering and managing e-mail at the server. In this example, I create a new `POP3` object with the `POP3` method, specifying the mail server. The `user` and `pass_` methods authenticate the application to the server; the `stat` method returns the number of messages waiting for the user and the total number of bytes taken up by all messages.

Next, I loop through each available message and use the `retr` method to grab the next e-mail message. This method returns a list of the form:

```
(response, ['line, ...], octets)
```

where *response* is the `POP3` response for the particular message, the *line*`list` represents the individual lines of the e-mail message, and the final element, *octets*, is the number of bytes for the e-mail message. The inner loop simply iterates over the second element ([1]) of the list which is the list of the e-mail message body. For each line, I test whether 'Subject:' is present; if so, I print this line.

After all e-mail messages have been checked, a call to the `quit` method ends the POP3 session.

Instead of using the `retr` method, you could also use the `top` method to extract the header for the e-mail message. This step would be faster and minimize the amount of data transferred to this client.

Listing 27. Retrieving e-mail messages from a POP3 mail server and emitting the subject line

```
import poplib
import re

popClient = poplib.POP3('192.168.1.1')

popClient.user('user')
popClient.pass_('password')

numMsgs, mboxSize = popClient.stat()

print "Number of messages ", numMsgs
print "Mailbox size", mboxSize
print
```

```
for id in range (numMsgs):
  for mail in popClient.retr(id+1)[1]:
    if re.search( 'Subject:', mail ):
      print mail

  print


popClient.quit()
```

Summary

This tutorial reviewed the basics of the sockets API and showed how to build networking applications in Python. The standard sockets module was introduced as a way to build both client and server applications, as well as the SocketServer module which simplifies the construction of simple socket servers. I presented a simple chat server implemented in Python that offered support for a scalable number of clients using the `select` method. In closing, I previewed some of Python's high-level networking classes that simplify the development of applications requiring application-layer networking protocols.

Python is an interesting and useful language that is well worth your time to learn.

RELATED TOPICS: Series: Programming Linux sockets Series: Charming Python Download Python

About the author

M. Tim Jones

M. Tim Jones is a senior principal software engineer with Emulex Corp. in Longmont, Colorado, where he architects and designs networking and storage products. Tim's design activities have ranged from real-time kernels for communication satellites to networking protocols and embedded firmware. He is the author of many articles on subjects from artificial intelligence (AI) to application-layer protocol development. He has also the author of *AI Application Programming* (now in its second edition), *GNU/Linux Application Programming, BSD Sockets Programming from a Multilanguage Perspective*, and *TCP/IP Application Layer Protocols for Embedded Systems* (all through Charles River Media).