7

Implementing Internet Communication

Python includes several built-in modules as well as add-on modules to implement different types of Internet communication. These modules simplify many of the tasks necessary to facilitate socket communication, email, file transfers, data streaming, HTTP requests, and more.

Because the communication possibilities with Python are so vast, this chapter focuses on phrases that implement simple socket servers, socket clients, and FTP clients, as well as POP3 and SMTP mail clients that can be easily incorporated into Python scripts.

Opening a Server-Side Socket for Receiving Data

```
sSock = socket(AF_INET, SOCK_STREAM)
sSock.bind((serverHost, serverPort))
sSock.listen(3)
conn, addr = sSock.accept()
data = conn.recv(1024)
```

The socket module included with Python provides a generic interface to a variety of low-level socket programming. This phrase discusses how to implement a low-level socket server using the socket module.

The first step in implementing a server-side socket interface is to create the server socket by calling `-socket(family, type [, proto])`, which creates and returns a new socket. *family* refers to the address family listed in Table 7.1, *type* refers to the socket types listed in Table 7.2, and *proto* refers to the protocol number, which is typically omitted except when working with raw sockets.

Table 7.1 **Protocol Families for Python Sockets**

Family	Description
AF_INET	Ipv4 protocols (TCP, UDP)
AF_INET6	Ipv6 protocols (TCP, UDP)
AF_UNIX	UNIX domain protocols

Table 7.2 **Socket Types for Python Sockets**

Type	Description
SOCK_STREAM	Opens an existing file for reading.
SOCK_DGRAM	Opens a file for writing. If the file already exists, the contents are deleted. If the file does not already exist, a new one is created.
SOCK_RAW	Opens an existing file for updating, keeping the existing contents intact.

Table 7.2 **Continued**

Type	Description
SOCK_RDM	Opens a file for both reading and writing. The existing contents are kept intact.
SOCK_SEQPACKET	Opens a file for both writing and reading. The existing contents are deleted.

Once the socket has been created, it must be bound to an address and port using the `bind(address)` method, where *`address`* refers to a tuple in the form of (*hostname*, *port*). If the hostname is an empty string, the server will allow connections on any available Internet interface on the system.

NOTE: You can specify *<broadcast>* as the hostname to use the socket to send broadcast messages.

After the socket has been bound to an interface, it can be activated by invoking the `listen(backlog)` method, where *`backlog`* is an integer that indicates how many pending connections the system should queue before rejecting new ones.

Once the socket is active, implement a `while` loop to wait for client connections using the `accept()` method. Once a client connection has been accepted, data can be read from the connection using the `recv(buffsize [,flags])` method. The `send(string [,flags])` method is used to write a response back to the client.

```
from socket import *

serverHost = '' # listen on all interfaces
serverPort = 50007

#Open socket to listen on
sSock = socket(AF_INET, SOCK_STREAM)
sSock.bind((serverHost, serverPort))
sSock.listen(3)

#Handle connections
while 1:

#Accept a connection
    conn, addr = sSock.accept()
    print 'Client Connection: ', addr
    while 1:

#Receive data
        data = conn.recv(1024)
        if not data: break
        print 'Server Received: ', data
        newData = data.replace('Client',
'Processed')

#Send response
        conn.send(newData)

#Close Connection
    conn.close()
```

server_socket.py

```
Client Connection:  ('137.65.77.24', 1678)
Server Received:  Client Message1
Server Received:  Client Message2
```

Output from server_socket.py code

Opening a Client-Side Socket for Sending Data

```
sSock = socket(AF_INET, SOCK_STREAM)
sSock.connect((serverHost, serverPort))
sSock.send(item)
data = sSock.recv(1024)
```

The socket module is also used to create a client-side socket that talks to the server-side socket discussed in the previous phrase.

The first step in implementing a client-side socket interface is to create the client socket by calling `socket(family, type [, proto])`, which creates and returns a new socket. *`family`* refers to the address family listed previously in Table 7.1, *`type`* refers to the socket types listed previously in Table 7.2, and *`proto`* refers to the protocol number, which is typically omitted except when working with raw sockets.

Once the client-side socket has been created, it can connect to the server socket using the `connect(address)` method, where *`address`* refers to a tuple in the form of (*hostname*, *port*).

NOTE: To connect to a server-socket on the local computer, use `localhost` as the hostname in the server address tuple.

After the client-side socket has connected to the server-side socket, data can be sent to the server using the `send(string [,flags])` method. The response from the server is received from the connection using the `recv(buffsize [,flags])` method.

```
import sys
from socket import *

serverHost = 'localhost'
serverPort = 50008

message = ['Client Message1', 'Client Message2']

if len(sys.argv) > 1:
    serverHost = sys.argv[1]

#Create a socket
sSock = socket(AF_INET, SOCK_STREAM)

#Connect to server
sSock.connect((serverHost, serverPort))

#Send messages
for item in message:
    sSock.send(item)
    data = sSock.recv(1024)
    print 'Client received: ', 'data'

sSock.close()
```

client_socket.py

```
Client received:  'Processed Message1'
Client received:  'Processed Message2'
```

Output from client_socket.py code

Receiving Streaming Data Using the ServerSocket Module

```
serv= SocketServer.TCPServer(("",50008),myTCPServer)
serv.serve_forever()
. . .
line = self.rfile.readline()
self.wfile.write("%s: %d bytes successfully \
received." % (sck, len(line)))
```

In addition to the socket module, Python includes the SocketServer module to provide you with TCP, UDP, and UNIX classes that implement servers. These classes have methods that provide you with a much higher level of socket control.

To implement a SocketServer to handle streaming requests, first define the class to inherit from the `SocketServer.StreamRequestHandler` class.

To handle the streaming requests, override the `handle` method to read and process the streaming data. The `rfile.readline()` function reads the streaming data until a newline character is encountered, and then returns the data as a string.

To send data back to the client from the streaming server, use the `wfile.write(`*`string`*`)` command to write the string back to the client.

Once you have defined the server class and overridden the `handle` method, create the server object by invoking `SocketServer.TCPServer(`*`address`*`, `*`handler`*`)`, where *`address`* refers to a tuple in the form of (*hostname*, *port*) and *`handler`* refers to your defined server class.

After the server object has been created, you can start handling connections by invoking the server object's `handle_request()` or `serve_forever()` method.

NOTE: In addition to the `TCPServer` method, you can also use the `UDPServer`, `UnixStreamServer`, and `UnixDatagramServer` methods to create other types of servers.

```
import socket
import string

class
myTCPServer(SocketServer.StreamRequestHandler):
    def handle (self):
        while 1:
            peer = self.connection.getpeername()[0]
            line = self.rfile.readline()
            print "%s wrote: %s" % (peer, line)
            sck = self.connection.getsockname()[0]
            self.wfile.write("%s: %d bytes \
                successfuly received." % \
                (sck, len(line)))

#Create SocketServer object
serv =
SocketServer.TCPServer(("",50008),myTCPServer)

#Activate the server to handle clients
serv.serve_forever()
```

stream_server.py

```
137.65.76.8 wrote: Hello
137.65.76.8 wrote: Here is today's weather.
137.65.76.8 wrote: Sunny
137.65.76.8 wrote: High: 75
137.65.76.8 wrote: Low: 58
137.65.76.8 wrote: bye
```

Output from stream_server.py code

Sending Streaming Data

```
sSock = socket(AF_INET, SOCK_STREAM)
sSock.connect((serverHost, serverPort))
line = raw_input("Send to %s: " % (serverHost))
sSock.send(line+'\n')
data = sSock.recv(1024)
```

To send streaming data to the streaming server described in the previous task, first create the client socket by calling `socket(`*`family`*`, `*`type`*` [, `*`proto`*`])`, which creates and returns a new socket.

Once the streaming client-side socket has been created, it can connect to the streaming server using the `connect(`*`address`*`)` method, where *`address`* refers to a tuple in the form of (*hostname*, *port*).

After the streaming client-side socket has connected to the server-side socket, data can be streamed to the server by formatting a stream of data that ends with the newline character and sending it to the server using the `send(`*`string`*` [,`*`flags`*`])` method.

A response from the server is received from the socket using the `recv(`*`buffsize`*` [,`*`flags`*`])` method.

```
import sys
from socket import *

serverHost = 'localhost'
serverPort = 50008

if len(sys.argv) > 1:
    serverHost = sys.argv[1]

#Create socket
sSock = socket(AF_INET, SOCK_STREAM)

#Connect to server
```

```
sSock.connect((serverHost, serverPort))

#Stream data to server.
line = ""
while line != 'bye':
    line = raw_input("Send to %s: " % (serverHost))
    sSock.send(line+'\n')
    data = sSock.recv(1024)
    print 'data'

sSock.shutdown(0)
sSock.close()
```

stream_client.py

```
Send to 137.65.76.28: Hello
'137.65.77.28: 6 bytes received.'
Send to 137.65.76.28: Here is today's weather.
'137.65.77.28: 25 bytes received.'
Send to 137.65.76.28: Sunny
'137.65.77.28: 6 bytes received.'
Send to 137.65.76.28: High: 75
'137.65.77.28: 9 bytes received.'
Send to 137.65.76.28: Low: 58
'137.65.77.28: 8 bytes received.'
Send to 137.65.76.28: bye
'137.65.77.28: 4 bytes received.'
```

Output from stream_client.py code

Sending Email Using SMTP

```
mMessage = ('From: %s\nTo: %s\nDate: %s\nSubject:\
            %s\n%s\n' % \
            (From, To, Date, Subject, Text))
s = smtplib.SMTP('mail.sfcn.org')
rCode = s.sendmail(From, To, mMessage)
s.quit()
```

The smtplib module included with Python provides simple access to SMTP servers that allow you to connect and quickly send mail messages from your Python scripts.

Mail messages must be formatted properly for the To, From, Date, Subject, and text fields to be processed properly by the SMTP mail server. The code in send_smtp.py shows the proper formatting for the mail message, including the item headers and newline characters.

Once the mail message is properly formatted, connect to the SMTP server using the `smtplib.SMTP(`*`host`*`[,`*`port`*`])` method. If it is necessary to log in to the SMTP server, use the `login(`*`user`*`, `*`password`*`)` method to complete an authentication.

Once connected to the SMTP server, the formatted message can be sent using `sendmail(`*`from`*`, `*`to`*`, `*`message`*`)`, where *`from`* is the sending email address string, *`to`* specifies a list of destination email address strings, and *`message`* is the formatted message string.

After you are finished sending messages, use the `quit()` method to close the connection to the SMTP server.

```
import smtplib
import time

From = "bwdayley@sfcn.org"
To = ["bwdayley@novell.com"]
Date = time.ctime(time.time())
Subject = "New message from Brad Dayley."
Text = "Message Text"
#Format mail message
mMessage = ('From: %s\nTo: %s\nDate: \
            %s\nSubject: %s\n%s\n' %
```

```
            (From, To, Date, Subject, Text))

print 'Connecting to Server'
s = smtplib.SMTP('mail.sfcn.org')

#Send mail
rCode = s.sendmail(From, To, mMessage)
s.quit()

if rCode:
    print 'Error Sending Message'
else:
    print 'Message Sent Successfully'
```

send_smtp.py

```
Connecting to Server
Message Sent Successfully
```

Output from send_smtp.py code

Also see Figure 7.1.

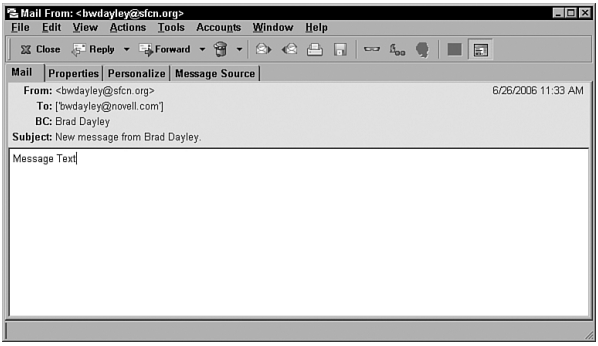

Figure 7.1 Email message sent by send_smtp.py code.

Retrieving Email from a POP3 Server

```
mServer = poplib.POP3('mail.sfcn.org')
mServer.user(getpass.getuser())
mServer.pass_(getpass.getpass())
numMessages = len(mServer.list()[1])
for msg in mServer.retr(mList+1)[1]:
```

The poplib module included with Python provides simple access to POP3 mail servers that allow you to connect and quickly retrieve messages using your Python scripts.

Connect to the POP3 mail server using the `poplib.POP3(`*`host`* `[,`*`port`* `[,`*`keyfile`* `[,`*`certfile`*`]]])` method, where *`host`* is the address of the POP3 mail server. The optional *`port`* argument defaults to 995. The other optional arguments, *`keyfile`* and *`certfile`*, refer to the PEM-formatted private key and certificate authentication files, respectively.

To log in to the POP3 server, the code in pop3_mail.py calls the `user(`*`username`*`)` and `pass_(`*`password`*`)` methods of the POP3 server object to complete the authentication.

NOTE: The example uses `getuser()` and `getpass()` from the getpass module to retrieve the username and password. The username and password can also be passed in as clear text strings.

After it's authenticated to the POP3 server, the poplib module provides several methods to manage the mail messages. The example uses the `list()` method to retrieve a list of messages in the tuple format (*response*, *msglist*, *size*), where *response* is the server's response code, *msglist* is a list of messages in string format, and *size* is the size of the response in bytes.

To retrieve only a single message, use `retr(msgid)`. The `retr` method returns the message numbered msgid in the form of a tuple (*response*, *lines*, *size*), where *response* is the server response, *lines* is a list of strings that compose the mail message, and *size* is the total size in bytes of the message.

NOTE: The *lines* list returned by the `retr` method includes all lines of the messages, including the header. To retrieve specific information, such as the recipient list, the *lines* list must be parsed.

When you are finished managing the mail messages, use the `quit()` method to close the connection to the POP3 server.

```
import poplib
import getpass

mServer = poplib.POP3('mail.sfcn.org')

#Login to mail server
mServer.user(getpass.getuser())
mServer.pass_(getpass.getpass())

#Get the number of mail messages
numMessages = len(mServer.list()[1])

print "You have %d messages." % (numMessages)
print "Message List:"

#List the subject line of each message
for mList in range(numMessages) :
    for msg in mServer.retr(mList+1)[1]:
        if msg.startswith('Subject'):
            print '\t' + msg
```

```
            break

mServer.quit()
```

pop3_mail.py

```
password:
You have 10 messages.
Message List:
    Subject: Static IP Info
    Subject: IP Address Change
    Subject: Verizon Wireless Online Statement
    Subject: New Static IP Address
    Subject: Your server account has been created
    Subject: Looking For New Home Projects?
    Subject: PDF Online - cl_scr_sheet.xls
    Subject: Professional 11 Upgrade Offer
    Subject: #1 Ball Played at the U.S. Open
    Subject: Chapter 3 submission
```

Output from pop3_mail.py code

Using Python to Fetch Files from an FTP Server

```
ftp = ftplib.FTP('ftp.novell.com', 'anonymous', \
                 'bwdayley@novell.com')
gFile = open("readme.txt", "wb")
ftp.retrbinary('RETR Readme', gFile.write)
gFile.close()
ftp.quit()
```

A common and extremely useful function of Python scripts is to retrieve files to be processed using the FTP protocol. The ftplib module included in Python allows you to use Python scripts to quickly attach to an FTP

server, locate files, and then download them to be processed locally.

To open a connection to the FTP server, create an FTP server object using the `ftplib.FTP([host [, user [, passwd]]])` method.

Once the connection to the server is opened, the methods in the ftplib module provide most of the FTP functionality to navigate the directory structure, manage files and directories, and, of course, download files.

The example shows connecting to an FTP server, listing the files and directories in the FTP server root directory using the `dir()` method, and then changing the directory using the `cwd(path)` method. In the example, the contents of the file Readme are downloaded from the FTP server and written to the local file readme.txt using the `retrbinary(command, callback [, blocksize [, reset]])` method.

After you are finished downloading/managing the files on the FTP server, use the `quit()` method to close the connection.

```
import ftplib

#Open ftp connection
ftp = ftplib.FTP('ftp.novell.com', 'anonymous',
'bwdayley@novell.com')

#List the files in the current directory
print "File List:"
files = ftp.dir()
print files


#Get the readme file
```

```
ftp.cwd("/pub")
gFile = open("readme.txt", "wb")
ftp.retrbinary('RETR Readme', gFile.write)
gFile.close()
ftp.quit()

#Print the readme file contents
print "\nReadme File Output:"
gFile = open("readme.txt", "r")
buff = gFile.read()
print buff
gFile.close()
```

ftp_client.py

```
File List:
-rw-r--r-- 1 root root  720 Dec 15  2005 README.html
-rw-r--r-- 1 root root  1406 Dec 15  2005 Readme
drwxrwxrwx 2 root root  53248 Jun 26 18:10 incoming
drwxrwxrwx 2 root root  16384 Jun 26 17:53 outgoing
drwxr-xr-x 3 root root  4096 May 12 16:12 partners
drwxr-xr-x 2 root root  4096 Apr  4 18:24 priv
drwxr-xr-x 4 root root  4096 May 25 22:20 pub
None

Readme File Output:
************************************************

 Before you download any software product you must
 read and agree to the following:
. . .
```

Output from ftp_client.py code