Foundations of Python Network Programming

JOHN GOERZEN

Apress®

Foundations of Python Network Programming

ISBN (pbk): 1-59059-371-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jason Gilmore

Technical Reviewer: Magnus Lie Hetland

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, John Franklin, Jason Gilmore, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Project Manager: Beth Christmas

Copy Edit Manager: Nicole LeClerc

Copy Editor: Mark Nigara

Production Manager: Kari Brooks

Production Editor: Ellie Fountain

Compositor: Susan Glinert Stevens

Proofreader: Elizabeth Berry

Indexer: Kevin Broccoli

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 13

FTP

FTP, THE FILE TRANSFER PROTOCOL, is one of the oldest and most widely used protocols on the Internet. As its name suggests, FTP is designed to transfer files from one location to another. It's bidirectional, meaning that it can be used to both upload and download files. Commands are provided to delete and rename files as well as to create and delete directories. FTP also provides features, benefiting UNIX/Linux servers most, that allow clients to alter permissions of files and directories on the server side. Some FTP servers support other features, such as automatic compression and archiving of directories.

In this chapter, you'll learn how to use Python's FTP interface, `ftplib`. Armed with that knowledge, you'll be able to automatically transfer files to and from remote machines, obtain lists of directories on FTP servers, and obtain information about those servers. You might use this to automatically upload the results of a data-processing job each night or to download a list of products for a user.

Understanding FTP

FTP has a long history, dating back to the early 1970s and predating the arrival of TCP. There are a number of features of FTP that are virtually never encountered today because they're of no use on modern computing platforms. Most of these features relate to systems that cannot support the modern notion of a file as being simply a stream of bytes whose interpretation is up to the application. Many FTP clients and servers don't support these features since they're virtually obsolete all but on paper.

In its present form, FTP is used to exchange files. It can send files in both directions, and with an appropriately intelligent client, two FTP servers can be made to exchange data with each other directly, without requiring the data to be downloaded to a client first.

> **NOTE** The FTP standard is RFC959, available at `www.faqs.org/rfcs/rfc959.html`.

Communication Channels

FTP is unusual because it actually uses two TCP connections to get the job done. One connection is the control or command channel. Commands and brief responses, such as acknowledgments or error codes, are transmitted over that connection. The second connection is the data channel. This connection 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 over it simultaneously. However, in actual practice, this capability is rarely used.

In the traditional sense, 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 as appropriate.

3. The client changes to the appropriate directory on the server.

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 then transmitted, and the data connection is closed.

This worked well in the early days of the Internet, for which every machine worth running FTP on had a public IP address and firewalls were relatively rare. Today, however, the picture is more complicated. Firewalls are the norm, and many people don't have real public IP addresses.

To accommodate this situation, FTP also supports what's known as passive mode. In this scenario, the server opens a port and tells the client where to connect. Other than that, everything behaves the same way.

Passive mode is the default with most FTP clients as well as Python's `ftplib` module.

Authentication and Anonymous FTP

Normally, when you connect to a remote FTP server, you'll provide a username and password to access the system. The FTP Protocol also provides for a third authentication token, called an *account*, which is rarely used. Once logged in,

you would typically have the same permissions on the remote system as if you were sitting at a terminal on the remote machine.

Many have decided that it would be useful to permit people without accounts to access certain areas of FTP servers. To permit this, visitors log in with the username "anonymous" and traditionally send their own e-mail address as a password. Servers configured this way are called *anonymous FTP servers*. Many large FTP sites use anonymous FTP so that anyone who wants files can get them. This is, in concept, similar to web servers for which no authentication is required before viewing documents. To the FTP client, the connection looks exactly the same as any other.

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 some commands.

> **TIP** If you're only interested in downloading files, the `urllib2` module discussed in Chapter 6 also supports FTP and may be easier to use for simple downloading tasks. In this chapter, I describe `ftplib` because it provides FTP-specific features that aren't available with `urllib2`.

Here's a basic `ftplib` example. It connects to a remote server, displays the welcome message, and prints the current working directory.

```
#!/usr/bin/env python
# Basic connection - Chapter 13 - connect.py

from ftplib import FTP

f = FTP('ftp.ibiblio.org')
print "Welcome:", f.getwelcome()
f.login()

print "CWD:", f.pwd()
f.quit()
```

The welcome message generally isn't of interest to a computer program, but may be interesting to a user running the program interactively. The `login()` function can take several parameters: a username, password, and account. If called without parameters, it will log in anonymously, sending the user `anonymous` and a generic anonymous password to the FTP server. That's fine in this case.

The `pwd()` function simply returns the current working directory on the remote site. The `quit()` function logs out and closes the connection. Here's an example session:

```
$ ./connect.py
Welcome: 220 ProFTPD Server (Bring it on...)
CWD: /
```

Downloading ASCII Files

FTP transfers normally use one of two modes: ASCII and binary (or image) mode. In ASCII mode, files are transferred line by line, which allows the client machine to store files with line endings appropriate on its platform. Here's a simple program that downloads a file in ASCII mode:

```
#!/usr/bin/env python
# ASCII download - Chapter 13 - asciidl.py
# Downloads README from remote and writes it to disk.

from ftplib import FTP

def writeline(data):
    fd.write(data + "\n")

f = FTP('ftp.kernel.org')
f.login()

f.cwd('/pub/linux/kernel')
fd = open('README', 'wt')
f.retrlines('RETR README', writeline)
fd.close()

f.quit()
```

The `cwd()` function changes the directory on the remote system. Then the `retrlines()` function begins the transfer. Its first parameter specifies a command to run on the remote system and usually is `RETR`, followed by a filename. Its second

parameter is a function that's called with each line retrieved; if omitted, the data is simply printed to standard output. The lines are passed with the end-of-line character stripped; so the `writeline()` function simply adds it on and writes the data out. To run this program, just run `./asciidl.py`. You'll see a file named `README` show up after it exits.

Downloading Binary Files

Basic binary file transfers work in very much the same way as text file transfers. Here's an example:

```
#!/usr/bin/env python
# Binary download - Chapter 13 - binarydl.py

from ftplib import FTP

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()
```

In this example, a binary file is downloaded—note that after running it, the file `patch8.gz` will appear 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. 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 more low-level interface, but can be useful if you wish to know a little bit more about what's going on. This lets you keep track of the number of bytes transferred, and you can use that information to display status updates for the user. Here's a sample program that uses `ntransfercmd()`:

```
#!/usr/bin/env python
# Advanced binary download - Chapter 13 - advbinarydl.py

from ftplib import FTP
import sys

f = FTP('ftp.kernel.org')
f.login()

f.cwd('/pub/linux/kernel/v1.0')
f.voidcmd("TYPE I")

datasock, estsize = f.ntransfercmd("RETR linux-1.0.tar.gz")
transbytes = 0
fd = open('linux-1.0.tar.gz', 'wb')
while 1:
    buf = datasock.recv(2048)
    if not len(buf):
        break
    fd.write(buf)
    transbytes += len(buf)
    sys.stdout.write("Received %d " % transbytes)

    # This "if" only passes if estsize is nonzero and is not None.
    # That's exactly what we want, since if it's zero, we'd get a
    # divide-by-zero error.
    if estsize:
        sys.stdout.write("of %d bytes (%.1f%%)\r" % \
                (estsize, 100.0 * float(transbytes) / float(estsize)))
    else:
        sys.stdout.write("bytes\r")
    sys.stdout.flush()
sys.stdout.write("\n")
fd.close()
datasock.close()
f.voidresp()

f.quit()
```

There are a few new things to note here. First, there's a call to `voidcmd()`. This passes an FTP command directly to the server, checks for an error, but returns nothing. In this case, you run `TYPE I`. That sets the transfer mode to image, or binary. In the previous example, `retrbinary()` automatically ran that command behind the scenes, but `ntransfercmd()` doesn't.

Next, note that the `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 shouldn't be considered authoritative. Also, if no size estimate is available from the FTP server, the estimated size will be `None`.

The data socket is, in fact, a plain socket, so all the socket functions described in chapters 1 through 5 will work on it. In this example, you'll use a simple loop to `recv()` the data from the socket, write it out to disk, and print status updates.

After receiving the data, it's important to close the data socket and call `voidresp()`. The `voidresp()` function reads the response from the server and raises an exception if there was any error. Even if you don't care about detecting errors, if you fail to call `voidresp()` future commands will likely fail in strange ways because their results will be out of sync with the server. Here's an example of running this program:

```
$ ./advbinarydl.py
Received 1259161 of 1259161 bytes (100.0%)
```

Uploading Data

Data can, of course, be uploaded as well. Like downloading, there are two basic functions for uploading: `storbinary()` and `storlines()`. Both take a command to run and a file-like object. The `storbinary()` function will call `read()` on that object, while `storlines()` calls `readline()`. Note that this differs from the download functions for which you supply the function itself. Here's an example that uploads a file in binary mode:

```
#!/usr/bin/env python
# Binary upload - Chapter 13 - binaryul.py
# Arguments: host, username, localfile, remotepath

from ftplib import FTP
import sys, getpass, os.path

host, username, localfile, remotepath = sys.argv[1:]
password = getpass.getpass("Enter password for %s on %s: " % \
        (username, host))
f = FTP(host)
f.login(username, password)
```

```
f.cwd(remotepath)
fd = open(localfile, 'rb')
f.storbinary('STOR %s' % os.path.basename(localfile), fd)
fd.close()

f.quit()
```

This program indeed looks quite similar to earlier versions. Since most anonymous FTP sites don't permit file uploading, this example requires a server and logs in to it with your own account. It then changes to the specified directory, uploads the file, and exits. You can modify this program to upload a file in ASCII mode by simply changing `storbinary()` to `storlines()`.

Here's an example session:

```
$ ./binaryul.py ftp.example.com jgoerzen /bin/sh /tmp
Enter password for jgoerzen on ftp.example.com:
```

This would log in to `ftp.example.com` as `jgoerzen`, then upload the `/bin/sh` from my system to the `/tmp` directory on the remote.

Advanced Binary Uploading

Like the download process, it's also possible to upload files using `ntransfercmd()`, as follows:

```
#!/usr/bin/env python
# Advanced binary upload - Chapter 13 - advbinaryul.py
# Arguments: host, username, localfile, remotepath

from ftplib import FTP
import sys, getpass, os.path

host, username, localfile, remotepath = sys.argv[1:]
password = getpass.getpass("Enter password for %s on %s: " % \
        (username, host))
f = FTP(host)
f.login(username, password)

f.cwd(remotepath)
f.voidcmd("TYPE I")
```

```
fd = open(localfile, 'rb')
datasock, esize = f.ntransfercmd('STOR %s' % os.path.basename(localfile))
esize = os.stat(localfile)[6]
transbytes = 0

while 1:
    buf = fd.read(2048)
    if not len(buf):
        break
    datasock.sendall(buf)
    transbytes += len(buf)
    sys.stdout.write("Sent %d of %d bytes (%.1f%%)\r" % (transbytes, esize,
        100.0 * float(transbytes) / float(esize)))
    sys.stdout.flush()
datasock.close()
sys.stdout.write("\n")
fd.close()
f.voidresp()

f.quit()
```

Compared to the advanced binary download example, this one is very similar. Note the call to `datasock.close()` though. When uploading data, closing the socket is the signal to the server that the upload is complete. If you fail to close the data socket after uploading all your data, the server will be forever stuck waiting for the rest of the data to arrive. Here's an example session:

```
$ ./advbinaryul.py ftp.example.com jgoerzen /bin/sh /tmp
Enter password for jgoerzen on ftp.example.com:
Sent 628684 of 628684 bytes (100.0%)
```

Handling Errors

Like most Python modules, `ftplib` raises exceptions when errors occur. The `ftplib` module defines several exceptions, and it can also raise `socket.error` and `IOError`. As a convenience there's a tuple, called `ftplib.all_errors`, that includes all errors that could be raised by `ftplib`. This is often a useful shortcut. You can enclose your code in a `try:` block, and use `except ftplib.all_errors` to catch any error that may occur.

One of the problems with the basic `retrbinary()` function is that in order to use it easily, you'll usually wind up opening the file on the local end before beginning the transfer from the remote. If the file doesn't exist or the `RETR` command otherwise

fails, you'll have to close and delete the local file, or else wind up with a 0-length file. With the `ntransfercmd()` method, you can check for a problem prior to opening a local file. The previous example already follows these guidelines; if `ntransfercmd()` fails, the exception will cause the program to terminate before the local file is opened.

Scanning Directories

The FTP Protocol provides two ways to discover information about server files and directories. These are implemented in `ftplib` in the `nlst()` and `dir()` functions.

The `nlst()` function returns a list of entries in a given directory. You'll thus receive the names of all files and directories present. However, that is all that's given. There's no information on whether a particular entry is a file or a directory, on its size, or anything else.

The `dir()` function returns a directory listing from the remote. This listing is in a system-defined format, but typically contains a filename, size, modification date, and type. On UNIX servers, it's typically the output of `ls -l` or `ls -la`. Windows servers may use the output of `dir`. Other systems may have their own specific formats. While the output may often be useful to an end user, it's difficult for a program to use it due to the varying output formats. Some clients that need this data implement parsers for the many different types of output formats, or the one type in use in a particular situation.

Here's an example of using `nlst()` to get directory information:

```
#!/usr/bin/env python
#  NLST example - Chapter 13 - nlst.py

from ftplib import FTP

f = FTP('ftp.kernel.org')
f.login()

f.cwd('/pub/linux/kernel')
entries = f.nlst()
entries.sort()

print "%d entries:" % len(entries)
for entry in entries:
    print entry
f.quit()
```

When you run this program, you'll see output like this:

```
$ ./nlst.py
22 entries:
COPYING
CREDITS
Historic
README
SillySounds
crypto
people
ports
projects
testing
uemacs
v1.0
v1.1
v1.2
...
```

If you were to use an FTP client to manually log on to the server, you'd see the same files listed. Notice that the filenames are in a convenient format for automated processing—a list of filenames—but there's no extra information. Now contrast that with the output from this example, which uses `dir()`:

```
#!/usr/bin/env python
#  dir() example - Chapter 13 - dir.py

from ftplib import FTP

f = FTP('ftp.kernel.org')
f.login()

f.cwd('/pub/linux/kernel')
entries = []
f.dir(entries.append)

print "%d entries:" % len(entries)
for entry in entries:
    print entry
f.quit()
```

Notice the call to `f.dir()`. It can take a function that's called for each line, just like `retrlines()`. This capability is handy for adding things to a list; `entries.append()` is called for each line and adds it to the list. The output of this program looks like this:

```
$ ./dir.py
22 entries:
-r--r--r--   1 korg     korg        18458 Mar 13  1994 COPYING
-r--r--r--   1 korg     korg        36981 Sep 16  1996 CREDITS
drwxrwsr-x   4 korg     korg         4096 Mar 20  2003 Historic
-r--r--r--   1 korg     korg        12056 Sep 16  1996 README
drwxrwsr-x   2 korg     korg         4096 Apr 14  2000 SillySounds
drwxrwsr-x   5 korg     korg         4096 Nov 24  2001 crypto
drwxrwsr-x  54 korg     korg         4096 Sep 16 19:52 people
drwxrwsr-x   6 korg     korg         4096 Mar 13  2003 ports
drwxrwsr-x   3 korg     korg         4096 Sep 16  2000 projects
drwxrwsr-x   3 korg     korg         4096 Feb 14  2002 testing
drwxrwsr-x   2 korg     korg         4096 Mar 20  2003 uemacs
drwxrwsr-x   2 korg     korg         4096 Mar 20  2003 v1.0
drwxrwsr-x   2 korg     korg        20480 Mar 20  2003 v1.1
drwxrwsr-x   2 korg     korg         8192 Mar 20  2003 v1.2
...
```

This time the list contains lines of output from `ls -l` on the server. While there's a lot more information for the user, this is a lot more difficult to process for the programmer.

Parsing UNIX Directory Listings

If you have a directory listing like the previous one, you'll notice that most UNIX servers provide a directory listing in pretty much the same format. If you have listings like the previous one, you process them with code like this:

```
#!/usr/bin/env python
#  dir() parsing example - Chapter 13 - dirparse.py

from ftplib import FTP

class DirEntry:
    def __init__(self, line):
        self.parts = line.split(None, 8)
```

```
    def isvalid(self):
        return len(self.parts) >= 6

    def gettype(self):
        """Returns - for regular file; d for directory; l for symlink."""
        return self.parts[0][0]

    def getfilename(self):
        if self.gettype() != 'l':
            return self.parts[-1]
        else:
            return self.parts[-1].split(' -> ', 1)[0]

    def getlinkdest(self):
        if self.gettype() == 'l':
            return self.parts[-1].split(' -> ', 1)[1]
        else:
            raise RuntimeError, "getlinkdest() called on non-link item"

class DirScanner(dict):
    """A dictionary object that can load itself up with keys being filenames
    and values being DirEntry objects.  Call addline() from your FTP dir()
    call."""
    def addline(self, line):
        obj = DirEntry(line)
        if obj.isvalid():
            self[obj.getfilename()] = obj

f = FTP('ftp.kernel.org')
f.login()

f.cwd('/pub/linux/kernel')
d = DirScanner()
f.dir(d.addline)

print "%d entries:" % len(d.keys())
for key, value in d.items():
    print "%s: type %s" % (key, value.gettype())

f.quit()
```

The `DirEntry` class represents a single entry in a directory. It defines an `isvalid()` function, which attempts to guess whether the line in question is valid. The check is simplistic, but will catch extra nonentry lines that sometimes occur in directory listings. The remaining functions, such as `gettype()` and `getfilename()`, simply return information by parsing the listing line.

`DirScanner` subclasses the Python built-in dictionary object, adding one additional method: `addline()`, which takes a line of data from `dir()` and, if it's valid, adds it to the dictionary. The key will be the filename listed on that line.

In the program, it's a simple matter to create the `DirScanner` object and have `f.dir()` call `addline()` for each line of information. Finally, the results are printed. They'll look somewhat like this:

```
$ ./dirparse.py
22 entries:
people: type d
testing: type d
COPYING: type -
Historic: type d
v2.6: type d
v2.4: type d
v2.5: type d
v2.2: type d
v2.3: type d
v2.0: type d
v2.1: type d
...
```

Discovering Information Without Parsing Listings

There are caveats to parsing directory listings, especially if the remote server doesn't return UNIX-format listings. By combining the output of `nlst()` with some other commands and error checking, it's possible to at least determine whether a given file is a regular file or a directory. This can all be done without having to use `dir()` results in any way, and it's done in a completely platform-independent manner. So it will work not just with Unix servers, but also with Windows, Mac OS X, and VMS servers. Here's an example:

```
#!/usr/bin/env python
# nlst() with file/directory detection example - Chapter 13
# nlstscan.py

import ftplib

class DirEntry:
    def __init__(self, filename, ftpobj, startingdir = None):
        self.filename = filename
        if startingdir == None:
            startingdir = ftpobj.pwd()
        try:
            ftpobj.cwd(filename)
            self.filetype = 'd'
            ftpobj.cwd(startingdir)
        except ftplib.error_perm:
            self.filetype = '-'

    def gettype(self):
        """Returns - for regular file; d for directory."""
        return self.filetype

    def getfilename(self):
        return self.filename

f = ftplib.FTP('ftp.kernel.org')
f.login()

f.cwd('/pub/linux/kernel')
nitems = f.nlst()
items = [DirEntry(item, f, f.pwd()) for item in nitems]

print "%d entries:" % len(items)
for item in items:
    print "%s: type %s" % (item.getfilename(), item.gettype())
f.quit()
```

The key to this program is in DirEntry's __init__() method. For each result from nlst(), the program attempts to change to a directory with that name. If this succeeds, the program knows that the result is the name of an actual directory, and it changes back to the starting directory. If the cwd() attempt fails, it will raise ftplib.error_perm, which is a good indication that the item in question isn't a directory. In some cases, this logic may be fooled; for instance, if the user doesn't have permission to use cwd() to change to the directory, this code may assume it has a file to deal with; but a subsequent attempt to download the file will fail. If you're concerned about this rare case, you could also attempt to start downloading a file. If that fails, you know you have something to which you don't have permission, but still don't know what.

This program will usually be slower than parsing the output from dir() since it has to issue so many different cwd() commands. However, the output from this program looks just like the output from the program that parsed the dir() result.

Downloading Recursively

Now that you're able to obtain directory listings and differentiate files from directories, it's possible to download entire directory trees from an FTP server. The following example downloads an entire directory tree from the Linux kernel repository.

```
#!/usr/bin/env python
# Recursive downloader - Chapter 13 - recursedl.py

from ftplib import FTP
import os, sys

class DirEntry:
    def __init__(self, line):
        self.parts = line.split(None, 8)

    def isvalid(self):
        return len(self.parts) >= 6

    def gettype(self):
        """Returns - for regular file; d for directory; l for symlink."""
        return self.parts[0][0]
```

```
    def getfilename(self):
        if self.gettype() != 'l':
            return self.parts[-1]
        else:
            return self.parts[-1].split(' -> ', 1)[0]

    def getlinkdest(self):
        if self.gettype() == 'l':
            return self.parts[-1].split(' -> ', 1)[1]
        else:
            raise RuntimeError, "getlinkdest() called on non-link item"

class DirScanner(dict):
    def addline(self, line):
        obj = DirEntry(line)
        if obj.isvalid():
            self[obj.getfilename()] = obj

def downloadfile(ftpobj, filename):
    ftpobj.voidcmd("TYPE I")
    datasock, estsize = ftpobj.ntransfercmd("RETR %s" % filename)
    transbytes = 0
    fd = open(filename, 'wb')
    while 1:
        buf = datasock.recv(2048)
        if not len(buf):
            break
        fd.write(buf)
        transbytes += len(buf)
        sys.stdout.write("%s: Received %d " % (filename, transbytes))
        if estsize:
            sys.stdout.write("of %d bytes (%.1f%%)\r" % (estsize,
                100.0 * float(transbytes) / float(estsize)))
        else:
            sys.stdout.write("bytes\r")
    fd.close()
    datasock.close()
    ftpobj.voidresp()
    sys.stdout.write("\n")
```

```
def downloaddir(ftpobj, localpath, remotepath):
    print "*** Processing directory", remotepath
    localpath = os.path.abspath(localpath)
    oldlocaldir = os.getcwd()
    if not os.path.isdir(localpath):
        os.mkdir(localpath)
    olddir = ftpobj.pwd()
    try:
        os.chdir(localpath)
        ftpobj.cwd(remotepath)
        d = DirScanner()
        f.dir(d.addline)

        for filename, entryobj in d.items():
            if entryobj.gettype() == '-':
                downloadfile(ftpobj, filename)
            elif entryobj.gettype() == 'd':
                downloaddir(ftpobj, localpath + '/' + filename,
                        remotepath + '/' + filename)
                # Re-display directory info
                print "*** Processing directory", remotepath
    finally:
        os.chdir(oldlocaldir)
        ftpobj.cwd(olddir)

f = FTP('ftp.kernel.org')
f.login()

downloaddir(f, 'old-versions', '/pub/linux/kernel/Historic/old-versions')

f.quit()
```

The `DirEntry` and `DirScanner` classes are the same as in the earlier `dir()`-based scanner. The `downloadfile()` function works like the advanced binary download example. The `downloaddir()` function is the heart of the program. It takes a local directory and a remote directory. It then scans the remote directory and looks at each entry.

If a given entry is a file, `downloadfile()` is called to download it. If the entry is a directory, then `downloaddir()` calls itself to process the new directory. At the end of processing, the `finally` clause ensures that working directories are back where they were when the function was first invoked. When you run this program, you'll see output somewhat like this:

```
$ ./recursedl.py
*** Processing directory /pub/linux/kernel/Historic/old-versions
linux-0.96a.tar.bz2: Received 174003 of 174003 bytes (100.0%)
linux-0.96b.patch2.bz2.sign: Received 248 of 248 bytes (100.0%)
linux-0.96b.patch2.gz: Received 5825 of 5825 bytes (100.0%)
RELNOTES-0.95a: Received 6257 of 6257 bytes (100.0%)
*** Processing directory /pub/linux/kernel/Historic/old-versions/tytso
linux-0.10.tar.sign: Received 248 of 248 bytes (100.0%)
linux-0.10.tar.bz2.sign: Received 248 of 248 bytes (100.0%)
linux-0.10.tar.bz2: Received 90032 of 90032 bytes (100.0%)
linux-0.10.tar.gz.sign: Received 248 of 248 bytes (100.0%)
linux-0.10.tar.gz: Received 123051 of 123051 bytes (100.0%)
*** Processing directory /pub/linux/kernel/Historic/old-versions
linux-0.96a.patch4.bz2.sign: Received 248 of 248 bytes (100.0%)
linux-0.96a.patch3.gz.sign: Received 248 of 248 bytes (100.0%)
linux-0.95.tar.bz2.sign: Received 248 of 248 bytes (100.0%)
...
```

Here, you can see the program began processing with the `old-versions` directory. It downloaded four files from that directory, then it encountered the `tytso` subdirectory. The program downloaded all files from `tytso`, then resumed processing other files in `old-versions`.

Manipulating Server Files and Directories

The `ftplib` module provides several simple methods for manipulating files and directories on the server. With these methods, you can do things like delete files and directories and do renames.

Deleting Files and Directories

You can call `delete()` to delete a file. You can also use `rmd()` to delete a directory, though most systems require the directory to be empty before it can be deleted. Both functions take a single parameter—a file or directory name—and will raise an exception if an error occurs.

Creating Directories

The `mkd()` function takes a single parameter and creates a new directory that has that name. There must not be any existing file or directory with the name. If there is, or any other error occurs, an exception will be raised.

Moving and Renaming Files

The `rename()` function takes two parameters: the name of an existing file and a new name for the file. It works essentially the same as the UNIX command `mv`. If both names are in the same directory, the file is essentially renamed. If the destination specifies a name in a different directory, the file is moved.

Summary

FTP lets you transfer files between your machine and a remote FTP server. In Python, the `ftplib` library is used to talk to FTP servers.

FTP supports binary and ASCII transfers. ASCII transfers are usually used for text files, and permit line endings to be adjusted as the file is transferred. Binary transfers are used for everything else. The `retrlines()` function is used to download a file in ASCII mode, while `retrbinary()` downloads a file in binary mode.

You can also upload files to a remote server. The `storlines()` function uploads a file in ASCII mode, and `storbinary()` uploads a file in binary mode.

The `ntransfercmd()` function can be used for binary uploads and downloads. It gives you more control over the transfer process and is often used to present status updates for the user.

The `ftplib` module raises exceptions on errors. The special tuple `ftplib.all_errors` can be used to catch any error that it might raise.

You can use `cwd()` to change to a particular directory on the remote end. The `nlst()` command returns a simple list of all entries (files or directories) in a given directory. The `dir()` command returns a list in server-specific format, which often includes more details. Even with `nlst()`, you can usually detect whether an entry is a file or directory by attempting to use `cwd()` to change to it and noting whether you get an error.