CHAPTER 23

Project 4: In the News

In this project, you see how you go from a simple prototype without any form of abstraction (no functions, no classes) to a generic system in which some important abstractions have been added. Also, you get a brief introduction to the `nntplib` library, which lets you interact with Network News Transfer Protocol (NNTP) servers.

NNTP is a standard network protocol for managing messages posted on Usenet discussion groups. NNTP servers form a global network that collectively manages these newsgroups, and through an NNTP client (also called a *newsreader*) you can post and read messages. Most recent web browsers include NNTP clients, and separate clients exist as well.

The main network of NNTP servers, called Usenet, was established in 1980 (although the NNTP protocol wasn't used until 1985). Compared to current Web 2.0 trends, this is quite "old school," but most of the Internet is based (to some degree) on such old-school technologies,[1] and it probably doesn't hurt to play around with the low-level stuff a bit. Also, you could always replace the NNTP stuff in this chapter with some news-gathering module of your own (perhaps using the web API of some social networking site like Facebook or MySpace).

What's the Problem?

The program you write in this project will be an information-gathering agent, a program that can gather information (more specifically, news) and compile a report for you. Given the network functionality you have already encountered, that might not seem very difficult—and it isn't, really. But in this project you go a bit beyond the simple "download a file with `urllib`" approach. You use another network library that is a bit more difficult to use than `urllib`, namely `nntplib`. In addition, you get to refactor the program to allow many types of news sources and various types of destinations, making a clear separation between the front end and the back end, with the main engine in the middle.

1. Did you know, for example, that the discussion groups at `http://groups.google.com`, such as `sci.math` and `rec.arts.sf.written`, are really Usenet groups under the hood?

The main goals for the final program are as follows:

- The program should be able to gather news from many different sources.
- It should be easy to add new news sources (and even new kinds of sources).
- The program should be able to dispatch its compiled news report to many different destinations, in many different formats.
- It should be easy to add new destinations (and even new kinds of destinations).

Useful Tools

For this project, you don't need to install separate software. However, you do need some standard library modules, including one that you haven't seen before, `nntplib`, which deals with NNTP servers. Instead of explaining all the details of that module, let's examine it through some prototyping.

You will also be using the `time` module (covered in Chapter 10).

Preparations

To be able to use `nntplib`, you need to have access to an NNTP server. If you're not sure whether you do, you could ask your ISP or system administrator for details. In the code examples in this chapter, I use the newsgroup `comp.lang.python.announce`, so you should make sure that your news (NNTP) server has that group, or you should find some other group you would like to use. It is important that the NNTP server support the `NEWNEWS` command. If it doesn't, the programs in this chapter won't work. (If you don't know whether your server supports this command, simply try to execute the programs and see what happens.)

If you don't have access to an NNTP server, or your server's `NEWNEWS` command is disabled, several open servers are available for anyone to use. A quick web search for "free nntp server" should give you some servers to choose from, or you could check out `http://www.newzbot.com` as a starting point.

Assuming that your news server is `news.foo.bar` (this is not a real server name, and won't work), you can test your NNTP server like this:

```
>>> from nntplib import NNTP
>>> server = NNTP('news.foo.bar')
>>> server.group('comp.lang.python.announce')[0]
```

■ **Note** To connect to some servers, you may need to supply additional parameters for authentication. Consult the Python Library Reference (`http://docs.python.org/lib/module-nntplib.html`) for details on the optional parameters of the `NNTP` constructor.

The result of the last line should be a string beginning with `'211'` (basically meaning that the server has the group you asked for) or `'411'` (which means that the server doesn't have the group). It might look something like this:

```
'211 51 1876 1926 comp.lang.python.announce'
```

If the returned string starts with `'411'`, you should use a newsreader to look for another group you might want to use. (You may also get an exception with an equivalent error message.) If an exception is raised, perhaps you got the server name wrong. Another possibility is that you were "timed out" between the time you created the server object and the time you called the `group` method—the server may allow you to stay connected for only a short period of time (such as 10 seconds). If you're having trouble typing that fast, simply put the code in a script and execute it (with an added `print`) or put the server object creation and method call on the same line (separated by a semicolon).

First Implementation

In the spirit of prototyping, let's just tackle the problem head on. The first thing you want to do is download the most recent messages from a newsgroup on an NNTP server. To keep things simple, just print out the result to standard output (with `print`).

Before looking at the details of the implementation, you might want to browse the source code in Listing 23-1 later in this section, and perhaps even execute the program to see how it works.

The program logic isn't very complicated, but you need to figure out how to use `nntplib`. You'll be using one single object of the `NNTP` class. As you saw in the previous section, this class is instantiated with a single constructor argument—the name of an NNTP server. You need to call three methods on this instance:

- `newnews`, which returns a list of articles posted after a certain date and time
- `head`, which gives you various information about the articles (most notably their subjects)
- `body`, which gives you the main text of the articles

The `newnews` method requires a date string (in the form *yymmdd*) and an hour string (in the form *hhmmss*) in addition to the group name. To construct these, you need some functions from the `time` module: `time`, `localtime`, and `strftime`. (See Chapter 10 for more information about the `time` module.)

Let's say you want to download all new messages since yesterday. To do this, you need to construct a date and time 24 hours before the current time. The current time (in seconds) is found with the `time` function. To find the time yesterday, all you need to do is subtract 24 hours (in seconds). To be able to use this time with `strftime`, it must be converted to a time tuple (see Chapter 10) with the `localtime` function. The code for finding "yesterday" then becomes as follows:

```
from time import time, localtime
day = 24 * 60 * 60 # Number of seconds in one day
yesterday = localtime(time() - day)
```

The next step is to format the time correctly, as two strings. For that, you use `strftime`, as in the following example:

```
>>> from time import strftime
>>> strftime('%y%m%d')
'020409'
>>> strftime('%H%M%S')
'141625'
```

The string argument to `strftime` is a *format string*, which specifies the format to use for the time. Most characters are used directly in the resulting time string, but those preceded by a percent sign are replaced with various time-related values. For instance, `%y` is replaced with the last two digits of the year, `%m` with the month (as a two-digit number), and so on. For a full list of these codes, consult the Python Library Reference (`http://docs.python.org/lib/module-time.html`). When supplied only with a format string, `strftime` uses the current time. Optionally, you may supply a time tuple as a second argument:

```
from time import strftime
date = strftime('%y%m%d', yesterday)
hour = strftime('%H%M%S', yesterday)
```

■ **Tip** The `datetime` module gives you a more object-oriented way of dealing with dates and times. Check out the standard library documentation at `http://docs.python.org/lib/module-datetime.html`.

Now that you have the date and time in the correct format for the `newnews` method, you just need to instantiate a server and call the method. Using the same fictitious server name as earlier, the code becomes as follows:

```
servername = 'news.foo.bar'
group = 'comp.lang.python.announce'
server = NNTP(servername)

ids = server.newnews(group, date, hour)[1]
```

Note that I've extracted the second argument of the tuple that is returned from `newnews`. It's sufficient for this example's purposes: a list of *article IDs* of the articles that were posted after the given date and hour.

■ **Note** The `newnews` method sends a `NEWNEWS` command to the NNTP server. As described in the "Preparations" section, this command may not be understood or supported by the server, giving you the error code 500 or 501, respectively, or disabled by the administrator, giving the error code 502. In such cases, you should find another server.

You need the article IDs when you call the `head` and `body` methods later, to tell the server which article you're talking about.

So, you're all set to start using `head` and `body` (for each of the IDs) and printing out the results. Just like `newnews`, `head` and `body` return tuples with various information (such as whether or not the command succeeded), but you care only about the returned data itself, which is the fourth element—a list of strings. The body of the article with a given ID can be fetched like this:

```
body = server.body(id)[3]
```

From the head (a list of lines containing various information about the article, such as the subject, the date it was posted, and so on), you want only the subject. The subject line is in the form `"Subject: Hello, world!"`, so you need to find the line that starts with `"Subject:"` and extract the rest of the line. Because (according to the NNTP standard) `"subject"` can also be spelled as all lowercase, all uppercase, or any kind of combination of uppercase and lowercase letters, you simply call the `lower` method on the line and compare it to `"subject"`. Here is the loop that finds the subject within the data returned by the call to `head`:

```
head = server.head(id)[3]
for line in head:
    if line.lower().startswith('subject:'):
        subject = line[9:]
        break
```

The `break` isn't strictly necessary, but when you've found the subject, there's no need to iterate over the rest of the lines.

After having extracted the subject and body of an article, you just print it, for instance, like this:

```
print subject
print '-'*len(subject)
print '\n'.join(body)
```

After printing all the articles, you call `server.quit()`, and that's it. In a UNIX shell such as `bash`, you could run this program like this:

```
$ python newsagent1.py | less
```

The use of `less` is useful for reading the articles one at a time. If you have no such pager program available, you could rewrite the `print` part of the program to store the resulting text in a file, which you'll also be doing in the second implementation (see Chapter 11 for more information about file handling). If you don't get any output, try looking further back than yesterday. The source code for the simple news-gathering agent is shown in Listing 23-1.

Listing 23-1. *A Simple News-Gathering Agent (newsagent1.py)*

```
from nntplib import NNTP
from time import strftime, time, localtime
```

```
day = 24 * 60 * 60 # Number of seconds in one day

yesterday = localtime(time() - day)
date = strftime('%y%m%d', yesterday)
hour = strftime('%H%M%S', yesterday)

servername = 'news.foo.bar'
group = 'comp.lang.python.announce'
server = NNTP(servername)

ids = server.newnews(group, date, hour)[1]


for id in ids:
    head = server.head(id)[3]
    for line in head:
        if line.lower().startswith('subject:'):
            subject = line[9:]
            break

    body = server.body(id)[3]

    print subject
    print '-'*len(subject)
    print '\n'.join(body)

server.quit()
```

Second Implementation

The first implementation worked, but was quite inflexible in that it let you retrieve news only from Usenet discussion groups. In the second implementation, you fix that by refactoring the code a bit. You add structure and abstraction by creating some classes and methods to represent the various parts of the code. Once you've done that, some of the parts may be replaced by other classes much more easily than you could replace parts of the code in the original program.

Again, before immersing yourself in the details of the second implementation, you might want to skim (and perhaps execute) the code in Listing 23-2, later in this chapter.

■ **Note** You need to set the `clpa_server` variable to a usable server before the code in Listing 23-2 will work.

So, what classes do you need? Let's just do a quick review of the important nouns in the problem description, as suggested in Chapter 7: information, agent, news, report, network, news source, destination, front end, back end, and main engine. This list of nouns suggests the following main classes (or kinds of classes): `NewsAgent`, `NewsItem`, `Source`, and `Destination`. The various sources will constitute the front end, and the destinations will constitute the back end, with the news agent sitting in the middle.

The easiest of these is `NewsItem`. It represents only a piece of data, consisting of a title and a body (a short text), and can be implemented as follows:

```
class NewsItem:

    def __init__(self, title, body):
        self.title = title
        self.body = body
```

To find out exactly what is needed from the news sources and the news destinations, it could be a good idea to start by writing the agent itself. The agent must maintain two lists: one of sources and one of destinations. Adding sources and destinations can be done through the methods `addSource` and `addDestination`:

```
class NewsAgent:

    def __init__(self):
        self.sources = []
        self.destinations = []

    def addSource(self, source):
        self.sources.append(source)

    def addDestination(self, dest):
        self.destinations.append(dest)
```

The only thing missing now is a method to distribute the news items from the sources to the destinations. During distribution, each destination must have a method that returns all its news items, and each source needs a method for receiving all the news items that are being distributed. Let's call these methods `getItems` and `receiveItems`. In the interest of flexibility, let's just require `getItems` to return an arbitrary iterator of `NewsItems`. To make the destinations easier to implement, however, let's assume that `receiveItems` is callable with a sequence argument (which can be iterated over more than once, to make a table of contents before listing the news items, for example). After this has been decided, the `distribute` method of `NewsAgent` simply becomes as follows:

```
    def distribute(self):
        items = []
        for source in self.sources:
            items.extend(source.getItems())
        for dest in self.destinations:
            dest.receiveItems(items)
```

This iterates through all the sources, building a list of news items. Then it iterates through all the destinations and supplies each of them with the full list of news items.

Now, all you need is a couple of sources and destinations. To begin testing, you can simply create a destination that works like the printing in the first prototype:

```
class PlainDestination:

    def receiveItems(self, items):
        for item in items:
            print item.title
            print '-'*len(item.title)
            print item.body
```

The formatting is the same; the difference is that you have *encapsulated* the formatting. It is now one of several alternative destinations, rather than a hard-coded part of the program. A slightly more complicated destination (`HTMLDestination`, which produces HTML) can be seen in Listing 23-2, later in this chapter. It builds on the approach of `PlainDestination` with a few extra features:

- The text it produces is HTML.
- It writes the text to a specific file, rather than standard output.
- It creates a table of contents in addition to the main list of items.

And that's it, really. The table of contents is created using hyperlinks that link to parts of the page. You accomplish this by using links of the form `<a href="#nn">...</a>` (where `nn` is some number), which leads to the headline with the enclosing anchor tag `<a name="nn">...</a>` (where `nn` should be the same number as in the table of contents). The table of contents and the main listing of news items are built in two different `for` loops. You can see a sample result (using the upcoming `NNTPSource`) in Figure 23-1.

When thinking about the design, I considered using a generic superclass to represent news sources and one to represent news destinations. As it turns out, the sources and destinations don't really share any behavior, so there is no point in using a common superclass. As long as they implement the necessary methods (`getItems` and `receiveItems`) correctly, the `NewsAgent` will be happy. (This is an example of using a *protocol*, as described in Chapter 9, rather than requiring a specific, common superclass.)

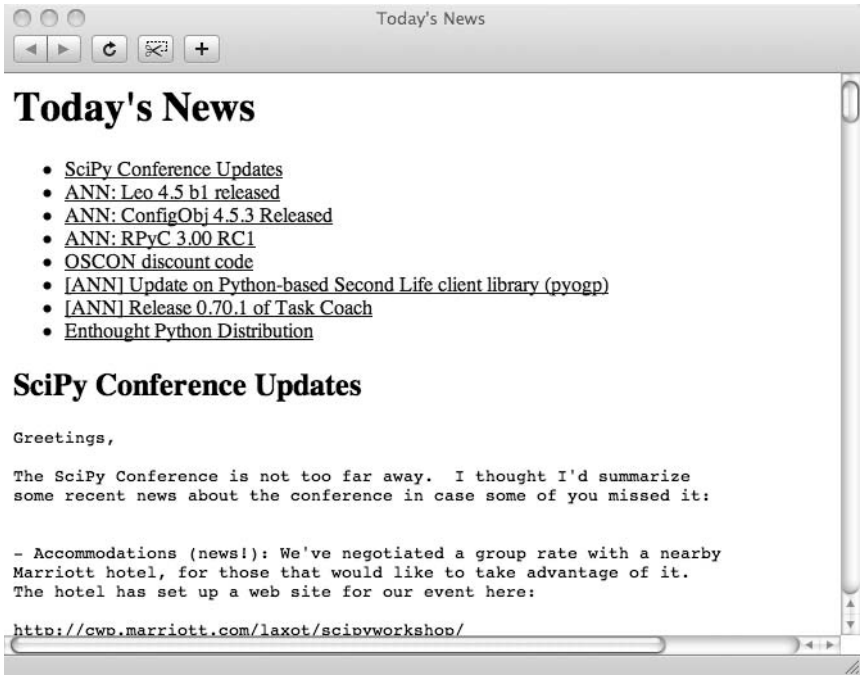

Figure 23-1. *An automatically generated news page*

When creating an `NNTPSource`, much of the code can be snipped from the original prototype. As you will see in Listing 23-2, the main differences from the original are the following:

- The code has been encapsulated in the `getItems` method. The `servername` and `group` variables are now arguments to the constructor. Also a `window` (a time window) is added, instead of assuming that you want the news since yesterday (which is equivalent to setting `window` to 1).
- To extract the subject, a `Message` object from the `email` module is used (constructed with the `message_from_string` function). This is the sort of thing you might add to later versions of your program as you thumb through the documentation (to see if features that can do what you need already exist).
- Instead of printing each news item directly, a `NewsItem` object is yielded (making `getItems` a generator).

To show the flexibility of the design, let's add another news source—one that can extract news items from web pages (using regular expressions; see Chapter 10 for more information). `SimpleWebSource` (see Listing 23-2) takes a URL and two regular expressions (one representing titles and one representing bodies) as its constructor arguments. In `getItems`, it uses the regular expression methods `findall` to find all the occurrences (titles and bodies) and `zip` to combine these. It then iterates over the list of `(title, body)` pairs, yielding a `NewsItem` for each. As you can see, adding new kinds of sources (or destinations, for that matter) isn't very difficult.

To put the code to work, let's instantiate an agent, some sources, and some destinations. In the function `runDefaultSetup` (which is called if the module is run as a program), several such objects are instantiated:

- A `SimpleWebSource` for the BBC News web site, which uses two simple regular expressions to extract the information it needs

---

■ **Note** The layout of the HTML on the BBC News pages might change, in which case you need to rewrite the regular expressions. This also applies if you are using some other page. Just view the HTML source and try to find a pattern that applies.

- An `NNTPSource` for `comp.lang.python`, with the time window set to 1, so it works just like the first prototype
- A `PlainDestination`, which prints all the news gathered
- An `HTMLDestination`, which generates a news page called `news.html`

When all of these objects have been created and added to the `NewsAgent`, the `distribute` method is called.

You can run the program like this:

```
$ python newsagent2.py
```

The resulting `news.html` page is shown in Figure 23-2.

The full source code of the second implementation is found in Listing 23-2.

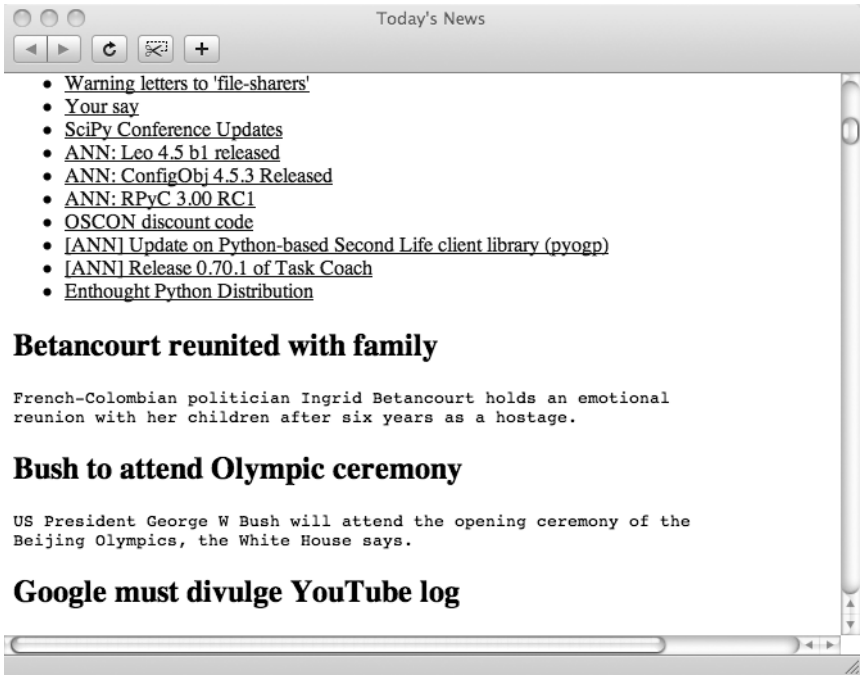

Figure 23-2. *A news page with more than one source*

Listing 23-2. *A More Flexible News-Gathering Agent (newsagent2.py)*

```
from nntplib import NNTP
from time import strftime, time, localtime
from email import message_from_string
from urllib import urlopen
import textwrap
import re

day = 24 * 60 * 60 # Number of seconds in one day

def wrap(string, max=70):
    """
    Wraps a string to a maximum line width.
    """
```

```
    return '\n'.join(textwrap.wrap(string)) + '\n'

class NewsAgent:
    """
    An object that can distribute news items from news
    sources to news destinations.
    """
    def __init__(self):
        self.sources = []
        self.destinations = []

    def addSource(self, source):
        self.sources.append(source)

    def addDestination(self, dest):
        self.destinations.append(dest)

    def distribute(self):
        """
        Retrieve all news items from all sources, and
        Distribute them to all destinations.
        """
        items = []
        for source in self.sources:
            items.extend(source.getItems())
        for dest in self.destinations:
            dest.receiveItems(items)

class NewsItem:
    """
    A simple news item consisting of a title and body text.
    """
    def __init__(self, title, body):
        self.title = title
        self.body = body

class NNTPSource:
    """
    A news source that retrieves news items from an NNTP group.
    """
    def __init__(self, servername, group, window):
        self.servername = servername
        self.group = group
        self.window = window

    def getItems(self):
```

```
        start = localtime(time() - self.window*day)
        date = strftime('%y%m%d', start)
        hour = strftime('%H%M%S', start)

        server = NNTP(self.servername)


        ids = server.newnews(self.group, date, hour)[1]

        for id in ids:
            lines = server.article(id)[3]
            message = message_from_string('\n'.join(lines))

            title = message['subject']
            body = message.get_payload()
            if message.is_multipart():
                body = body[0]

            yield NewsItem(title, body)

        server.quit()

class SimpleWebSource:
    """
    A news source that extracts news items from a web page using
    regular expressions.
    """
    def __init__(self, url, titlePattern, bodyPattern):
        self.url = url
        self.titlePattern = re.compile(titlePattern)
        self.bodyPattern = re.compile(bodyPattern)

    def getItems(self):
        text = urlopen(self.url).read()
        titles = self.titlePattern.findall(text)
        bodies = self.bodyPattern.findall(text)
        for title, body in zip(titles, bodies):
            yield NewsItem(title, wrap(body))

class PlainDestination:
    """
    A news destination that formats all its news items as
    plain text.
    """
    def receiveItems(self, items):
        for item in items:
            print item.title
```

```
            print '-'*len(item.title)
            print item.body


class HTMLDestination:
    """
    A news destination that formats all its news items
    as HTML.
    """
    def __init__(self, filename):
        self.filename = filename

    def receiveItems(self, items):

        out = open(self.filename, 'w')
        print >> out, """
        <html>
          <head>
            <title>Today's News</title>
          </head>
          <body>
          <h1>Today's News</h1>
        """

        print >> out, '<ul>'
        id = 0
        for item in items:
            id += 1
            print >> out, '  <li><a href="#%i">%s</a></li>' % (id, item.title)
        print >> out, '</ul>'

        id = 0
        for item in items:
            id += 1
            print >> out, '<h2><a name="%i">%s</a></h2>' % (id, item.title)
            print >> out, '<pre>%s</pre>' % item.body

        print >> out, """
          </body>
        </html>
        """
```

```
def runDefaultSetup():
    """
    A default setup of sources and destination. Modify to taste.
    """
    agent = NewsAgent()


    # A SimpleWebSource that retrieves news from the
    # BBC News site:
    bbc_url = 'http://news.bbc.co.uk/text_only.stm'
    bbc_title = r'(?s)a href="[^"]*">\s*<b>\s*(.*?)\s*</b>'
    bbc_body = r'(?s)</a>\s*<br />\s*(.*?)\s*<'
    bbc = SimpleWebSource(bbc_url, bbc_title, bbc_body)

    agent.addSource(bbc)

    # An NNTPSource that retrieves news from comp.lang.python.announce:
    clpa_server = 'news.foo.bar' # Insert real server name
    clpa_group = 'comp.lang.python.announce'
    clpa_window = 1
    clpa = NNTPSource(clpa_server, clpa_group, clpa_window)

    agent.addSource(clpa)

    # Add plain-text destination and an HTML destination:
    agent.addDestination(PlainDestination())
    agent.addDestination(HTMLDestination('news.html'))

    # Distribute the news items:
    agent.distribute()

if __name__ == '__main__': runDefaultSetup()
```

Further Exploration

Because of its extensible nature, this project invites much further exploration. Here are some ideas:

- Create a more ambitious `WebSource`, using the screen-scraping techniques discussed in Chapter 15.
- Create an `RSSSource`, which parses RSS, also discussed briefly in Chapter 15.
- Improve the layout for the `HTMLDestination`.

- Create a page monitor that gives you a news item if a given web page has changed since the last time you examined it. (Just download a copy when it has changed and compare that later. Take a look at the standard library module `filecmp` for comparing files.)
- Create a CGI version of the news script (see Chapter 15).
- Create an `EmailDestination`, which sends you an email message with news items. (See the standard library module `smtplib` for sending email.)
- Add command-line switches to decide which news formats you want. (See the standard library modules `getopt` and `optparse` for some techniques.)
- Give the information about where the news comes from, to allow a fancier layout.
- Try to categorize your news items (by searching for keywords, perhaps).
- Create an `XMLDestination`, which produces XML files suitable for the site builder in Project 3 (Chapter 22). *Voilà*—you have a news web site.

What Now?

You've done a lot of file creation and file handling (including downloading the required files), and although that is very useful for a lot of things, it isn't very interactive. In the next project, you create a chat server, where you can chat with your friends online. You can even extend it to create your own virtual (textual) environment.