CHAPTER 22

Project 3: XML for All Occasions

I mentioned XML briefly in Project 1. Now it's time to examine it in more detail. In this project, you see how XML can be used to represent many kinds of data, and how XML files can be processed with the Simple API for XML, or SAX. The goal of this project is to generate a full web site from a single XML file that describes the various web pages and directories.

In this chapter, I assume that you know what XML is and how to write it. If you know some HTML, you're already familiar with the basics. XML isn't really a specific language (such as HTML); it's more like a set of rules that define a *class* of languages. Basically, you still write tags the same way as in HTML, but in XML you can invent tag names yourself. Such specific sets of tag names and their structural relationships can be described in *Document Type Definitions* or *XML Schema*—I won't be discussing those here.

For a concise description of what XML is, see the World Wide Web Consortium's (W3C's) "XML in 10 points" (`http://www.w3.org/XML/1999/XML-in-10-points`). A more thorough tutorial can be found on the W3Schools web site (`http://www.w3schools.com/xml`). For more information about SAX, see the official SAX web site (`http://www.saxproject.org`).

What's the Problem?

The general problem you'll be attacking in this project is to parse (read and process) XML files. Because you can use XML to represent practically anything, and you can do whatever you want with the data when you parse it, the applications are boundless (as the title of this chapter indicates).

The specific problem tackled in this chapter is to generate a complete web site from a single XML file that contains the structure of the site and the basic contents of each page.

Before you proceed with this project, I suggest that you take a few moments to read a bit about XML and to check out its applications. That might give you a better understanding of when it might be a useful file format and when it would just be overkill. (After all, plain-text files can be just fine when they're all you need.)

ANYTHING, YOU SAY?

You may be skeptical about what you can really represent with XML. Well, let me give you just a few examples of the uses of XML:

- To mark up text for ordinary document processing—for example, in the form of XHTML (`http://www.w3.org/TR/xhtml1`) or DocBook XML (`http://www.docbook.org`)
- To represent music (`http://musicxml.org`)
- To represent human moods, emotions, and character traits (`http://xml.coverpages.org/humanML.html`)
- To describe any physical object (`http://xml.coverpages.org/pml-ons.html`)
- To call Python methods across a network (using XML-RPC, demonstrated in Chapter 27)

A sampling of existing applications of XML may be found on the XML Cover Pages (`http://xml.coverpages.org/xml.html#applications`) and at CBEL (`http://www.cbel.com/xml_markup_languages`).

Let's define the specific goals for the project:

- The entire web site should be described by a single XML file, which should include information about individual web pages and directories.
- The program should create the directories and web pages as needed.
- It should be easy to change the general design of the entire web site and regenerate all the pages with the new design.

This last point is perhaps enough to make it all worthwhile, but there are other benefits. By placing all your contents in a single XML file, you could easily write other programs that use the same XML processing techniques to extract various kinds of information, such as tables of contents, indices for custom search engines, and so on. And even if you don't use this for your web site, you could use it to create HTML-based slide shows (or, by using something like ReportLab, discussed in the previous chapter, you could even create PDF slide shows).

Useful Tools

Python has some built-in XML support, but if you're using an old version, you may need to install some extras yourself. In this project, you'll need a functioning SAX parser. To see if you have a usable SAX parser, try to execute the following:

```
>>> from xml.sax import make_parser
>>> parser = make_parser()
```

In all likelihood, no exceptions will be raised when you do this. In that case, you're all set and can continue to the "Preparations" section.

■ **Tip** Plenty of XML tools for Python are out there. One very interesting alternative to the "standard" PyXML framework is Fredrik Lundh's ElementTree (and the C implementation, cElementTree), which is also included in recent versions of the Python standard library, in the package `xml.etree`. If you have an older Python version, you can get ElementTree from `http://effbot.org/zone`. It's quite powerful and easy to use, and may well be worth a look if you're serious about using XML in Python.

If you do get an exception (which may be the case for older Python versions), you must install PyXML. First, download the PyXML package from `http://sf.net/projects/pyxml`. There you can find RPM packages for Linux, binary installers for Windows, and source distributions for other platforms. The RPMs are installed with `rpm --install`, and the binary Windows distribution is installed simply by executing it. The source distribution is installed through the standard Python installation mechanism, Distutils. Simply unpack the `tar.gz` file, change to the unpacked directory, and execute the following:

```
$ python setup.py install
```

You should now be able to use the XML tools.

Preparations

Before you can write the program that processes your XML files, you must design your XML format. What tags do you need, what attributes should they have, and which tags should go where? To find out, let's first consider what it is you want your XML to describe.

The main concepts are web site, directory, page, name, title, and contents:

- You won't be storing any information about the web site itself, so the *web site* is just the top-level element enclosing all the files and directories.
- A *directory* is mainly a container for files and other directories.
- A *page* is a single web page.
- Both directories and web pages need *names*. These will be used as directory names and file names, as they will appear in the file system and the corresponding URLs.
- Each web page should have a *title* (not the same as its file name).
- Each web page will also have some *contents*. You'll just use plain XHTML to represent the contents here. That way, you can simply pass it through to the final web pages and let the browsers interpret it.

In short, your document will consist of a single `website` element, containing several `directory` and `page` elements, each of the directory elements optionally containing more pages and directories. The `directory` and `page` elements will have an attribute called `name`, which will contain their name. In addition, the `page` tag has a `title` attribute. The `page` element contains XHTML code (of the type found inside the XHTML `body` tag). A sample file is shown in Listing 22-1.

Listing 22-1. *A Simple Web Site Represented As an XML File (website.xml)*

```
<website>
  <page name="index" title="Home Page">
    <h1>Welcome to My Home Page</h1>

    <p>Hi, there. My name is Mr. Gumby, and this is my home page. Here
    are some of my interests:</p>

    <ul>
      <li><a href="interests/shouting.html">Shouting</a></li>
      <li><a href="interests/sleeping.html">Sleeping</a></li>
      <li><a href="interests/eating.html">Eating</a></li>
    </ul>
  </page>
  <directory name="interests">
    <page name="shouting" title="Shouting">
      <h1>Mr. Gumby's Shouting Page</h1>

      <p>...</p>
    </page>
    <page name="sleeping" title="Sleeping">
      <h1>Mr. Gumby's Sleeping Page</h1>

      <p>...</p>
    </page>
    <page name="eating" title="Eating">
      <h1>Mr. Gumby's Eating Page</h1>

      <p>...</p>
    </page>
  </directory>
</website>
```

First Implementation

At this point, we haven't yet looked at how XML parsing works. The approach we are using here (called SAX) consists of writing a set of event handlers (just as in GUI programming) and then letting an existing XML parser call these handlers as it reads the XML document.

WHAT ABOUT DOM?

There are two common ways of dealing with XML in Python (and other programming languages, for that matter): SAX and the Document Object Model (DOM). A SAX parser reads through the XML file and tells you what it sees (text, tags, and attributes), storing only small parts of the document at a time. This makes SAX simple, fast, and memory-efficient, which is why I have chosen to use it in this chapter. DOM takes another approach: it constructs a data structure (the *document tree*), which represents the entire document. This is slower and requires more memory, but can be useful if you want to manipulate the structure of your document, for example.

For information about using DOM in Python, check out the Python Library Reference (`http://www.python.org/doc/current/lib/module-xml.dom.html`). In addition to the standard DOM handling, the standard library contains two other modules: `xml.dom.minidom` (a simplified DOM) and `xml.dom.pulldom` (a cross between SAX and DOM, which reduces memory requirements).

A very fast and simple XML parser (which doesn't really use DOM, but creates a complete document tree from your XML document) is pyRXP (`http://www.reportlab.org/pyrxp.html`). And then there is ElementTree, which is flexible and easy to use.

Creating a Simple Content Handler

Several event types are available when parsing with SAX, but let's restrict ourselves to three: the beginning of an element (the occurrence of an opening tag), the end of an element (the occurrence of a closing tag), and plain text (characters). To parse the XML file, let's use the `parse` function from the `xml.sax` module. This function takes care of reading the file and generating the events, but as it generates these events, it needs some event handlers to call. These event handlers will be implemented as methods of a *content handler* object. You'll subclass the `ContentHandler` class from `xml.sax.handler` because it implements all the necessary event handlers (as dummy operations that have no effect), and you can override only the ones you need.

Let's begin with a minimal XML parser (assuming that your XML file is called `website.xml`):

```
from xml.sax.handler import ContentHandler
from xml.sax import parse

class TestHandler(ContentHandler): pass
parse('website.xml', TestHandler())
```

If you execute this program, seemingly nothing happens, but you shouldn't get any error messages either. Behind the scenes, the XML file is parsed, and the default event handlers are called, but because they don't do anything, you won't see any output.

Let's try a simple extension. Add the following method to the `TestHandler` class:

```
def startElement(self, name, attrs):
    print name, attrs.keys()
```

This overrides the default `startElement` event handler. The parameters are the relevant tag name and its attributes (kept in a dictionary-like object). If you run the program again (using `website.xml` from Listing 22-1), you see the following output:

```
website []
page [u'name', u'title']
h1 []
p []
ul []
li []
a [u'href']
li []
a [u'href']
li []
a [u'href']
directory [u'name']
page [u'name', u'title']
h1 []
p []
page [u'name', u'title']
h1 []
p []
page [u'name', u'title']
h1 []
p []
```

How this works should be pretty clear. In addition to `startElement`, you'll use `endElement` (which takes only a tag name as its argument) and `characters` (which takes a string as its argument).

The following is an example that uses all these three methods to build a list of the headlines (the `h1` elements) of the web site file:

```
from xml.sax.handler import ContentHandler
from xml.sax import parse

class HeadlineHandler(ContentHandler):

    in_headline = False
```

```
    def __init__(self, headlines):
        ContentHandler.__init__(self)
        self.headlines = headlines
        self.data = []


    def startElement(self, name, attrs):
        if name == 'h1':
            self.in_headline = True

    def endElement(self, name):
        if name == 'h1':
            text = ''.join(self.data)
            self.data = []
            self.headlines.append(text)
            self.in_headline = False

    def characters(self, string):
        if self.in_headline:
            self.data.append(string)

headlines = []
parse('website.xml', HeadlineHandler(headlines))

print 'The following <h1> elements were found:'
for h in headlines:
    print h
```

Note that the `HeadlineHandler` keeps track of whether it's currently parsing text that is inside a pair of `h1` tags. This is done by setting `self.in_headline` to `True` when `startElement` finds an `h1` tag, and setting `self.in_headline` to `False` when `endElement` finds an `h1` tag. The `characters` method is automatically called when the parser finds some text. As long as the parser is between two `h1` tags (`self.in_headline` is `True`), `characters` will append the string (which may be just a part of the text between the tags) to `self.data`, which is a list of strings. The task of joining these text fragments, appending them to `self.headlines` (as a single string), and resetting `self.data` to an empty list also befalls `endElement`. This general approach (of using Boolean variables to indicate whether you are currently "inside" a given tag type) is quite common in SAX programming.

Running this program (again, with the `website.xml` file from Listing 22-1), you get the following output:

```
The following <h1> elements were found:
Welcome to My Home Page
Mr. Gumby's Shouting Page
Mr. Gumby's Sleeping Page
Mr. Gumby's Eating Page
```

Creating HTML Pages

Now you're ready to make the prototype. For now, let's ignore the directories and concentrate on creating HTML pages. You need to create a slightly embellished event handler that does the following:

- At the start of each `page` element, opens a new file with the given name, and writes a suitable HTML header to it, including the given title
- At the end of each `page` element, writes a suitable HTML footer to the file, and closes it
- While inside the `page` element, passes through all tags and characters without modifying them (writes them to the file as they are)
- While not inside a `page` element, ignores all tags (such as `website` and `directory`)

Most of this is pretty straightforward (at least if you know a bit about how HTML documents are constructed). There are two problems, however, which may not be completely obvious:

- You can't simply "pass through" tags (write them directly to the HTML file you're building) because you are given their names only (and possibly some attributes). You must reconstruct the tags (with angle brackets and so forth) yourself.
- SAX itself gives you no way of knowing whether you are currently "inside" a `page` element. You must keep track of that sort of thing yourself (as you did in the `HeadlineHandler` example). For this project, you're interested only in whether or not to pass through tags and characters, so you'll use a Boolean variable called `passthrough`, which you'll update as you enter and leave the pages.

See Listing 22-2 for the code for the simple program.

Listing 22-2. *A Simple Page Maker Script (pagemaker.py)*

```
from xml.sax.handler import ContentHandler
from xml.sax import parse

class PageMaker(ContentHandler):
    passthrough = False
    def startElement(self, name, attrs):
        if name == 'page':
            self.passthrough = True
            self.out = open(attrs['name'] + '.html', 'w')
            self.out.write('<html><head>\n')
            self.out.write('<title>%s</title>\n' % attrs['title'])
            self.out.write('</head><body>\n')
```

```
        elif self.passthrough:
            self.out.write('<' + name)
            for key, val in attrs.items():
                self.out.write(' %s="%s"' % (key, val))
            self.out.write('>')


    def endElement(self, name):
        if name == 'page':
            self.passthrough = False
            self.out.write('\n</body></html>\n')
            self.out.close()
        elif self.passthrough:
            self.out.write('</%s>' % name)
    def characters(self, chars):
        if self.passthrough: self.out.write(chars)

parse('website.xml', PageMaker ())
```

You should execute this in the directory in which you want your files to appear. Note that even if two pages are in two different directory elements, they will end up in the same real directory. (That will be fixed in our second implementation.)

Again, using the file `website.xml` from Listing 22-1, you get four HTML files. The file called `index.html` contains the following:

```
<html><head>
<title>Home Page</title>
</head><body>

    <h1>Welcome to My Home Page</h1>

    <p>Hi, there. My name is Mr. Gumby, and this is my home page. Here
    are some of my interests:</p>

    <ul>
      <li><a href="interests/shouting.html">Shouting</a></li>
      <li><a href="interests/sleeping.html">Sleeping</a></li>
      <li><a href="interests/eating.html">Eating</a></li>
    </ul>

</body></html>
```

Figure 22-1 shows how this page looks when viewed in a browser.

Looking at the code, two main weaknesses should be obvious:

- It uses `if` statements to handle the various event types. If you need to handle many such event types, your `if` statements will get large and unreadable.
- The HTML code is hard-wired. It should be easy to replace.

Both of these weaknesses will be addressed in the second implementation.

Figure 22-1. *A generated web page*

Second Implementation

Because the SAX mechanism is so low level and basic, you may often find it useful to write a mix-in class that handles some administrative details such as gathering character data, managing Boolean state variables (such as `passthrough`), or dispatching the events to your own custom event handlers. The state and data handling are pretty simple in this project, so let's focus on the handler dispatch.

A Dispatcher Mix-In Class

Rather than needing to write large `if` statements in the standard generic event handlers (such as `startElement`), it would be nice to just write your own specific ones (such as `startPage`) and have them called automatically. You can implement that functionality in a mix-in class, and then subclass the mix-in along with `ContentHandler`.

■ **Note** As mentioned in Chapter 7, a *mix-in* is a class with limited functionality that is meant to be subclassed along with some other more substantial class.

You want the following functionality in your program:

- When `startElement` is called with a name such as `'foo'`, it should attempt to find an event handler called `startFoo` and call it with the given attributes.
- Similarly, if `endElement` is called with `'foo'`, it should try to call `endFoo`.
- If, in any of these methods, the given handler is not found, a method called `defaultStart` (or `defaultEnd`, respectively) will be called, if present. If the default handler isn't present either, nothing should be done.

In addition, some care should be taken with the parameters. The custom handlers (for example, `startFoo`) do not need the tag name as a parameter, while the custom default handlers (for example, `defaultStart`) do. Also, only the start handlers need the attributes.

Confused? Let's begin by writing the simplest parts of the class:

```
class Dispatcher:

    # ...

    def startElement(self, name, attrs):
        self.dispatch('start', name, attrs)
    def endElement(self, name):
        self.dispatch('end', name)
```

Here, the basic event handlers are implemented, and they simply call a method called `dispatch`, which takes care of finding the appropriate handler, constructing the argument tuple, and then calling the handler with those arguments. Here is the code for the `dispatch` method:

```
    def dispatch(self, prefix, name, attrs=None):
        mname = prefix + name.capitalize()
        dname = 'default' + prefix.capitalize()
        method = getattr(self, mname, None)
        if callable(method): args = ()
        else:
            method = getattr(self, dname, None)
            args = name,
        if prefix == 'start': args += attrs,
        if callable(method): method(*args)
```

The following is what happens:

1. From a prefix (either `'start'` or `'end'`) and a tag name (for example, `'page'`), construct the method name of the handler (for example, `'startPage'`).
2. Using the same prefix, construct the name of the default handler (for example, `'defaultStart'`).
3. Try to get the handler with `getattr`, using `None` as the default value.
4. If the result is callable, assign an empty tuple to `args`.
5. Otherwise, try to get the default handler with `getattr`, again using `None` as the default value. Also, set `args` to a tuple containing only the tag name (because the default handler needs that).
6. If you are dealing with a start handler, add the attributes to the argument tuple (`args`).
7. If your handler is callable (that is, it is either a viable specific handler or a viable default handler), call it with the correct arguments.

Got that? This basically means that you can now write content handlers like this:

```
class TestHandler(Dispatcher, ContentHandler):
    def startPage(self, attrs):
        print 'Beginning page', attrs['name']
    def endPage(self):
        print 'Ending page'
```

Because the dispatcher mix-in takes care of most of the plumbing, the content handler is fairly simple and readable. (Of course, you'll add more functionality in a little while.)

Factoring Out the Header, Footer, and Default Handling

This section is much easier than the previous one. Instead of doing the calls to `self.out.write` directly in the event handler, you'll create separate methods for writing the header and footer. That way, you can easily override these methods by subclassing the event handler. Let's make the default header and footer really simple:

```
    def writeHeader(self, title):
        self.out.write("<html>\n  <head>\n    <title>")
        self.out.write(title)
        self.out.write("</title>\n  </head>\n  <body>\n")

    def writeFooter(self):
        self.out.write("\n  </body>\n</html>\n")
```

Handling of the XHTML contents was also linked a bit too intimately with the original handlers. The XHTML will now be handled by `defaultStart` and `defaultEnd`:

```
def defaultStart(self, name, attrs):
    if self.passthrough:
        self.out.write('<' + name)
        for key, val in attrs.items():
            self.out.write(' %s="%s"' % (key, val))
        self.out.write('>')

def defaultEnd(self, name):
    if self.passthrough:
        self.out.write('</%s>' % name)
```

This works just like before, except that I've moved the code to separate methods (which is usually a good thing). Now, on to the last piece of the puzzle.

Support for Directories

To create the necessary directories, you need a couple of useful functions from the `os` and `os.path` modules. One of these functions is `os.makedirs`, which makes all the necessary directories in a given path. For example, `os.makedirs('foo/bar/baz')` creates the directory `foo` in the current directory, then creates `bar` in `foo`, and finally, `baz` in `bar`. If `foo` already exists, only `bar` and `baz` are created, and similarly, if `bar` also exists, only `baz` is created. However, if `baz` exists as well, an exception is raised.

To avoid this exception, you need the function `os.path.isdir`, which checks whether a given path is a directory (that is, whether it exists already). Another useful function is `os.path.join`, which joins several paths with the correct separator (for example, `/` in UNIX and so forth).

At all times during the processing, keep the current directory path as a list of directory names, referenced by the variable `directory`. When you enter a directory, append its name; when you leave it, pop the name off. Assuming that `directory` is set up properly, you can define a function for ensuring that the current directory exists:

```
def ensureDirectory(self):
    path = os.path.join(*self.directory)
    if not os.path.isdir(path): os.makedirs(path)
```

Notice how I've used argument splicing (with the star operator, *) on the directory list when supplying it to `os.path.join`.

The base directory of our web site (for example, `public_html`) can be given as an argument to the constructor, which then looks like this:

```
def __init__(self, directory):
    self.directory = [directory]
    self.ensureDirectory()
```

The Event Handlers

Finally we've come to the event handlers. You need four of them: two for dealing with directories, and two for pages. The directory handlers simply use the `directory` list and the `ensureDirectory` method:

```
    def startDirectory(self, attrs):
        self.directory.append(attrs['name'])
        self.ensureDirectory()

    def endDirectory(self):
        self.directory.pop()
```

The page handlers use the `writeHeader` and `writeFooter` methods. In addition, they set the `passthrough` variable (to pass through the XHTML), and—perhaps most important—they open and close the file associated with the page:

```
    def startPage(self, attrs):
        filename = os.path.join(*self.directory+[attrs['name']+'.html'])
        self.out = open(filename, 'w')
        self.writeHeader(attrs['title'])
        self.passthrough = True


    def endPage(self):
        self.passthrough = False
        self.writeFooter()
        self.out.close()
```

The first line of `startPage` may look a little intimidating, but it is more or less the same as the first line of `ensureDirectory`, except that you add the file name (and give it an `.html` suffix).

The full source code of the program is shown in Listing 22-3.

Listing 22-3. *The Web Site Constructor (website.py)*

```
from xml.sax.handler import ContentHandler
from xml.sax import parse
import os

class Dispatcher:

    def dispatch(self, prefix, name, attrs=None):
        mname = prefix + name.capitalize()
        dname = 'default' + prefix.capitalize()
        method = getattr(self, mname, None)
        if callable(method): args = ()
```

```
        else:
            method = getattr(self, dname, None)
            args = name,
        if prefix == 'start': args += attrs,
        if callable(method): method(*args)

    def startElement(self, name, attrs):
        self.dispatch('start', name, attrs)

    def endElement(self, name):
        self.dispatch('end', name)

class WebsiteConstructor(Dispatcher, ContentHandler):

    passthrough = False

    def __init__(self, directory):
        self.directory = [directory]
        self.ensureDirectory()

    def ensureDirectory(self):
        path = os.path.join(*self.directory)
        if not os.path.isdir(path): os.makedirs(path)


    def characters(self, chars):
        if self.passthrough: self.out.write(chars)

    def defaultStart(self, name, attrs):
        if self.passthrough:
            self.out.write('<' + name)
            for key, val in attrs.items():
                self.out.write(' %s="%s"' % (key, val))
            self.out.write('>')

    def defaultEnd(self, name):
        if self.passthrough:
            self.out.write('</%s>' % name)

    def startDirectory(self, attrs):
        self.directory.append(attrs['name'])
        self.ensureDirectory()

    def endDirectory(self):
        self.directory.pop()
```

```
    def startPage(self, attrs):
        filename = os.path.join(*self.directory+[attrs['name']+'.html'])
        self.out = open(filename, 'w')
        self.writeHeader(attrs['title'])
        self.passthrough = True

    def endPage(self):
        self.passthrough = False
        self.writeFooter()
        self.out.close()

    def writeHeader(self, title):
        self.out.write('<html>\n  <head>\n    <title>')
        self.out.write(title)
        self.out.write('</title>\n  </head>\n  <body>\n')

    def writeFooter(self):
        self.out.write('\n  </body>\n</html>\n')

parse('website.xml', WebsiteConstructor('public_html'))
```

Listing 22-3 generates the following files and directories:

- public_html/
- public_html/index.html
- public_html/interests
- public_html/interests/shouting.html
- public_html/interests/sleeping.html
- public_html/interests/eating.html

ENCODING BLUES

If your XML file contains special characters (those with ordinal numbers above 127), you may be in trouble. The XML parser uses Unicode strings during its processing, and returns those to you (for example, in the `characters` event handler). Unicode handles the special characters just fine. However, if you want to convert this Unicode string to an ordinary string (which is what happens when you print it, for example), an exception is raised (assuming that your default encoding is ASCII):

```
>>> some_string = u'Mööõse'
>>> some_string
u'M\xf6\xf6\xf6se'
>>> print some_string
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
```

As you can see, the error message is "ASCII encoding error," which actually means that Python has tried to *encode* the Unicode string with the ASCII encoding, which isn't possible when it contains special characters like this. (You can find the default encoding of your installation using the `sys.getdefaultencoding` function. You can also change it with the `sys.setdefaultencoding`, but only in the site-wide customization file called `site.py`.) Encoding is done with the `encode` method:

```
>>> some_string.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
```

To solve this problem, you need to use another encoding, such as ISO8859-1 (which is fine for most European languages):

```
>>> print some_string.encode('iso8859-1')
Möööse
```

(The actual appearance of the output will depend on your terminal emulator.)

Note that if you're using non-ASCII characters directly in your source code, you need to mark that, so that the interpreter knows what to do with the file. For Latin 1 (another name for ISO8859-1), you could simply put the following comment into your file (directly after the pound bang line):

```
# -*- coding: latin-1 -*-
```

You can find more information about such encodings at the W3C web site (`http://www.w3.org/International/O-charset.html`).

Further Exploration

Now you have the basic program. What can you do with it? Here are some suggestions:

- Create a new `ContentHandler` for generating a table of contents or a menu (with links) for the web site.
- Add navigational aids to the web pages that tell the users where (in which directory) they are.
- Create a subclass of `WebsiteConstructor` that overrides `writeHeader` and `writeFooter` to provide customized design.
- Create another `ContentHandler` that constructs a single web page from the XML file.
- Create a `ContentHandler` that summarizes your web site somehow, for example in RSS.

- Check out other tools for transforming XML, especially XML Transformations (XSLT).
- Create one or more PDF documents based on the XML file, using a tool such as ReportLab's Platypus (`http://www.reportlab.org`).
- Make it possible to edit the XML file through a web interface (see Chapter 25).

What Now?

After this foray into the world of XML parsing, let's do some more network programming. In the next chapter, you create a program that can gather news items from various network sources (such as web pages and Usenet groups) and generate custom news reports for you.