CHAPTER 15

Python and the Web

This chapter tackles some aspects of web programming with Python. This is a really vast area, but I've selected three main topics for your amusement: screen scraping, CGI, and mod_python. In addition, I give you some pointers for finding the proper toolkits for more advanced web application and web service development. For extended examples using CGI, see Chapters 25 and 26. For an example of using the specific web service protocol XML-RPC, see Chapter 27.

Screen Scraping

Screen scraping is a process whereby your program downloads web pages and extracts information from them. This is a useful technique that pops up every time there is a page online that has information you want to use in your program. It is *especially* useful, of course, if the web page in question is dynamic; that is, if it changes over time. Otherwise, you could just download it once and extract the information manually. (The *ideal* situation is, of course, one where the information is available through *web services*, as discussed later in this chapter.)

Conceptually, the technique is very simple. You download the data and analyze it. You could, for example, simply use `urllib`, get the web page's HTML source, and then use regular expressions (see Chapter 10) or another technique to extract the information. Let's say, for example, that you wanted to extract the various employer names and web sites from the Python Job Board, at `http://python.org/community/jobs`. You browse the source and see that the names and URLs can be found as links in `h3` elements, like this (except on one, unbroken line):

```
<h3><a name="google-mountain-view-ca-usa"><a class="reference"
href="http://www.google.com">Google</a> ...
```

Listing 15-1 shows a sample program that uses `urllib` and `re` to extract the required information.

Listing 15-1. *A Simple Screen-Scraping Program*

```
from urllib import urlopen
import re
p = re.compile('<h3><a .*?><a .*? href="(.*?)">(.*?)</a>')
text = urlopen('http://python.org/community/jobs').read()
for url, name in p.findall(text):
    print '%s (%s)' % (name, url)
```

The code could certainly be improved (for example, by filtering out duplicates), but it does its job pretty well. There are, however, at least three weaknesses with this approach:

- The regular expression isn't exactly readable. For more complex HTML code and more complex queries, the expressions can become even more hairy and unmaintainable.
- It doesn't deal with HTML peculiarities like CDATA sections and character entities (such as `&`). If you encounter such beasts, the program will, most likely, fail.
- The regular expression is tied to details in the HTML source code, rather than some more abstract structure. This means that small changes in how the web page is structured can break the program. (By the time you're reading this, it may already be broken.)

The following sections deal with two possible solutions for the problems posed by the regular expression-based approach. The first is to use a program called Tidy (as a Python library) together with XHTML parsing. The second is to use a library called Beautiful Soup, specifically designed for screen scraping.

Note There are other tools for screen scraping with Python. You might, for example, want to check out Ka-Ping Yee's `scrape.py` (found at `http://zesty.ca/python`).

Tidy and XHTML Parsing

The Python standard library has plenty of support for parsing structured formats such as HTML and XML (see the Python Library Reference, Section 8, "Structured Markup Processing Tools," at `http://python.org/doc/lib/markup.html`). I discuss XML and XML parsing in more depth in Chapter 22. In this section, I just give you the tools needed to deal with XHTML, the most up-to-date dialect of HTML, which just happens to be a form of XML.

If every web page consisted of correct and valid XHTML, the job of parsing it would be quite simple. The problem is that older HTML dialects are a bit more sloppy, and some people don't even care about the strictures of those sloppier dialects. The reason for this is, probably, that most web browsers are quite forgiving, and will try to render even the most jumbled and meaningless HTML as best they can. If this happens to look acceptable to the page authors, they may be satisfied. This does make the job of screen scraping quite a bit harder, though.

The general approach for parsing HTML in the standard library is event-based; you write event handlers that are called as the parser moves along the data. The standard library modules `sgmllib` and `htmllib` will let you parse really sloppy HTML in this manner, but if you want to extract data based on document structure (such as the first item after the second level-two heading), you'll need to do some heavy guessing if there are missing tags, for example. You are certainly welcome to do this, if you like, but there is another way: Tidy.

What's Tidy?

Tidy (`http://tidy.sf.net`) is a tool for fixing ill-formed and sloppy HTML. It can fix a range of common errors in a rather intelligent manner, doing a lot of work that you would probably rather not do yourself. It's also quite configurable, letting you turn various corrections on or off.

Here is an example of an HTML file filled with errors, some of them just Old Skool HTML, and some of them plain wrong (can you spot all the problems?):

```
<h1>Pet Shop
<h2>Complaints</h3>

<p>There is <b>no <i>way</b> at all</i> we can accept returned
parrots.

<h1><i>Dead Pets</h1>

<p>Our pets may tend to rest at times, but rarely die within the
warranty period.

<i><h2>News</h2></i>

<p>We have just received <b>a really nice parrot.

<p>It's really nice.</b>

<h3><hr>The Norwegian Blue</h3>

<h4>Plumage and <hr>pining behavior</h4>
<a href="#norwegian-blue">More information<a>

<p>Features:
<body>
<li>Beautiful plumage
```

Here is the version that is fixed by Tidy:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title></title>
</head>
<body>
<h1>Pet Shop</h1>
<h2>Complaints</h2>
<p>There is <b>no <i>way</i> at all</b> we can accept returned
parrots.</p>
<h1><i>Dead Pets</i></h1>
<p>Our pets may tend to rest at times, but rarely die within the
warranty period.</p>
<h2><i>News</i></h2>
<p>We have just received <b>a really nice parrot.</b></p>
<p><b>It's really nice.</b></p>
<hr>
```

```
<h3>The Norwegian Blue</h3>
<h4>Plumage and</h4>
<hr>
<h4>pining behavior</h4>
<a href="#norwegian-blue">More information</a>
<p>Features:</p>
<ul class="noindent">
<li>Beautiful plumage</li>
</ul>
</body>
</html>
```

Of course, Tidy can't fix all problems with an HTML file, but it does make sure it's well-formed (that is, all elements nest properly), which makes it much easier for you to parse it.

Getting a Tidy Library

You can get Tidy and the library version of Tidy, Tidylib, from `http://tidy.sf.net`. You should also get a Python wrapper. You can get µTidyLib from `http://utidylib.berlios.de`, or mxTidy from `http://egenix.com/products/python/mxExperimental/mxTidy`.

At the time of writing, µTidyLib seems to be the most up-to-date of the two, but mxTidy is a bit easier to install. In Windows, simply download the installer for mxTidy, run it, and you have the module `mx.Tidy` at your fingertips. There are also RPM packages available. If you want to install the source package (presumably in a UNIX or Linux environment), you can simply run the Distutils script, using `python setup.py install`.

Using Command-Line Tidy in Python

You don't *have* to install either of the libraries, though. If you're running a UNIX or Linux machine of some sort, it's quite possible that you have the command-line version of Tidy available. And no matter what operating system you're using, you can probably get an executable binary from the TidyLib web site (`http://tidy.sf.net`).

Once you have the binary version, you can use the `subprocess` module (or some of the `popen` functions) to run the Tidy program. Assuming, for example, that you have a messy HTML file called `messy.html`, the following program will run Tidy on it and print the result.

```
from subprocess import Popen, PIPE

text = open('messy.html').read()
tidy = Popen('tidy', stdin=PIPE, stdout=PIPE, stderr=PIPE)

tidy.stdin.write(text)
tidy.stdin.close()

print tidy.stdout.read()
```

In practice, instead of printing the result, you would, most likely, extract some useful information from it, as demonstrated in the following sections.

But Why XHTML?

The main difference between XHTML and older forms of HTML (at least for our current purposes) is that XHTML is quite strict about closing all elements explicitly. So in HTML you might end one paragraph simply by beginning another (with a tag), but in XHTML, you first need to close the paragraph explicitly (with a tag). This makes XHTML much easier to parse, because you can tell directly when you enter or leave the various elements. Another advantage of XHTML (which I won't really capitalize on in this chapter) is that it is an XML dialect, so you can use all kinds of nifty XML tools on it, such as XPath. For example, the links to the forms extracted by the program in Listing 15-1 could also be extracted by the XPath expression `//h3/a/@href`. (For more about XML, see Chapter 22; for more about the uses of XPath, see, for example, `http://www.w3schools.com/xpath`.)

A very simple way of parsing the kind of well-behaved XHTML you get from Tidy is using the standard library module (and class) `HTMLParser`.[1]

Using HTMLParser

Using `HTMLParser` simply means subclassing it and overriding various event-handling methods such as `handle_starttag` and `handle_data`. Table 15-1 summarizes the relevant methods and when they're called (automatically) by the parser.

Table 15-1. *The HTMLParser Callback Methods*

Callback Method	When Is It Called?
`handle_starttag(tag, attrs)`	When a start tag is found, `attrs` is a sequence of (`name`, `value`) pairs.
`handle_startendtag(tag, attrs)`	For empty tags; default handles start and end separately.
`handle_endtag(tag)`	When an end tag is found.
`handle_data(data)`	For textual data.
`handle_charref(ref)`	For character references of the form `&#ref;`.
`handle_entityref(name)`	For entity references of the form `&name;`.
`handle_comment(data)`	For comments; called with only the comment contents.
`handle_decl(decl)`	For declarations of the form <!…>.
`handle_pi(data)`	For processing instructions.

For screen-scraping purposes, you usually won't need to implement all the parser callbacks (the event handlers), and you probably won't need to construct some abstract representation of the entire document (such as a document tree) to find what you want. If you just keep track of the minimum of information needed to find what you're looking for, you're in business. (See Chapter 22 for more about this topic, in the context of XML parsing with SAX.) Listing 15-2 shows a program that solves the same problem as Listing 15-1, but this time using `HTMLParser`.

1. This is not to be confused with the class `HTMLParser` from the `htmllib` module, which you can also use, of course, if you're so inclined. It's more liberal in accepting ill-formed input.

Listing 15-2. *A Screen-Scraping Program Using the HTMLParser Module*

```
from urllib import urlopen
from HTMLParser import HTMLParser

class Scraper(HTMLParser):

    in_h3 = False
    in_link = False

    def handle_starttag(self, tag, attrs):
        attrs = dict(attrs)
        if tag == 'h3':
            self.in_h3 = True

        if tag == 'a' and 'href' in attrs:
            self.in_link = True
            self.chunks = []
            self.url = attrs['href']

    def handle_data(self, data):
        if self.in_link:
            self.chunks.append(data)

    def handle_endtag(self, tag):
        if tag == 'h3':
            self.in_h3 = False
        if tag == 'a':
            if self.in_h3 and self.in_link:
                print '%s (%s)' % (''.join(self.chunks), self.url)
            self.in_link = False

text = urlopen('http://python.org/community/jobs').read()
parser = Scraper()
parser.feed(text)
parser.close()
```

A few things are worth noting. First of all, I've dropped the use of Tidy here, because the HTML in the web page is well behaved enough. If you're lucky, you may find that you don't need to use Tidy either. Also note that I've used a couple of Boolean *state variables* (attributes) to keep track of whether I'm inside `h3` elements and links. I check and update these in the event handlers. The `attrs` argument to `handle_starttag` is a list of `(key, value)` tuples, so I've used `dict` to turn them into a dictionary, which I find to be more manageable.

The `handle_data` method (and the `chunks` attribute) may need some explanation. It uses a technique that is quite common in event-based parsing of structured markup such as HTML and XML. Instead of assuming that I'll get all the text I need in a single call to `handle_data`, I assume that I may get several chunks of it, spread over more than one call. This may happen for several reasons—buffering, character entities, markup that I've ignored, and so on—and I just need to

make sure I get all the text. Then, when I'm ready to present my result (in the `handle_endtag` method), I simply `join` all the chunks together. To actually run the parser, I call its `feed` method with the text, and then call its `close` method.

This solution is, most likely, more robust to any changes in the input data than the version using regular expressions (Listing 15-1). Still, you may object that it is too verbose (it's *certainly* more verbose than the XPath expression, for example) and perhaps almost as hard to understand as the regular expression. For a more complex extraction task, the arguments in favor of this sort of parsing might seem more convincing, but one is still left with the feeling that there must be a better way. And, if you don't mind installing another module, there is . . .

Beautiful Soup

Beautiful Soup is a spiffy little module for parsing and dissecting the kind of HTML you often find on the Web—the sloppy and ill-formed kind. To quote the Beautiful Soup web site (`http://crummy.com/software/BeautifulSoup`):

> *You didn't write that awful page. You're just trying to get some data out of it. Right now, you don't really care what HTML is supposed to look like.*
>
> *Neither does this parser.*

Downloading and installing Beautiful Soup is a breeze. Download the file `BeautifulSoup.py` and put it in your Python path (for example, in the `site-packages` directory of your Python installation). If you want, you can instead download a `tar` archive with installer scripts and tests. With Beautiful Soup installed, the running example of extracting Python jobs from the Python Job Board becomes really, really simple *and* readable, as shown in Listing 15-3.

Listing 15-3. *A Screen-Scraping Program Using Beautiful Soup*

```
from urllib import urlopen
from BeautifulSoup import BeautifulSoup

text = urlopen('http://python.org/community/jobs').read()
soup = BeautifulSoup(text)

jobs = set()
for header in soup('h3'):
    links = header('a', 'reference')
    if not links: continue
    link = links[0]
    jobs.add('%s (%s)' % (link.string, link['href']))

print '\n'.join(sorted(jobs, key=lambda s: s.lower()))
```

I simply instantiate the `BeautifulSoup` class with the HTML text I want to scrape, and then use various mechanisms to extract parts of the resulting parse tree. For example, I call `soup('h3')` to get a list of all `h3` elements. I iterate over these, binding the `header` variable to each one in turn, and call `header('a', 'reference')` to get a list of a child elements of the

reference class (I'm talking CSS classes here). I could also have followed the strategy from previous examples, of retrieving the a elements that have href attributes; in Beautiful Soup, using class attributes like this is easier.

As I'm sure you noticed, I added the use of set and sorted (with a key function set to ignore case differences) in Listing 15-3. This has nothing to do with Beautiful Soup; it was just to make the program more useful, by eliminating duplicates and printing the names in sorted order.

If you want to use your scrapings for an RSS feed (discussed later in this chapter), you can use another tool related to Beautiful Soup, called Scrape 'N' Feed (at http://crummy.com/software/ScrapeNFeed).

Dynamic Web Pages with CGI

While the first part of this chapter dealt with client-side technology, now we switch gears and tackle the server side. This section deals with a basic web programming technology: the Common Gateway Interface (CGI). CGI is a standard mechanism by which a web server can pass your queries (typically supplied through a web form) to a dedicated program (for example, your Python program) and display the result as a web page. It is a simple way of creating web applications without writing your own special-purpose application server. For more information about CGI programming in Python, see the Web Programming topic guide on the Python web site (http://wiki.python.org/moin/WebProgramming).

The key tool in Python CGI programming is the cgi module. You can find a thorough description of it in the Python Library Reference (http://python.org/doc/lib/module-cgi.html). Another module that can be very useful during the development of CGI scripts is cgitb—more about that later, in the section "Debugging with cgitb."

Before you can make your CGI scripts accessible (and runnable) through the Web, you need to put them where a web server can access them, add a *pound bang* line, and set the proper file permissions. These three steps are explained in the following sections.

Step 1. Preparing the Web Server

I'm assuming that you have access to a web server—in other words, that you can put stuff on the Web. Usually, that is a matter of putting your web pages, images, and so on in a particular directory (in UNIX, typically called public_html). If you don't know how to do this, you should ask your Internet service provider (ISP) or system administrator.

■ **Tip** If you are running Mac OS X, you have the Apache web server as part of your operating system installation. It can be switched on through the Sharing preference pane of System Preferences, by checking the Web Sharing option.
