Screen Scraping

Most web sites are designed first and foremost for human eyes. While well-designed sites offer formal APIs by which you can construct Google maps, upload Flickr photos, or browse YouTube videos, many sites offer nothing but HTML pages formatted for humans. If you need a program to be able to fetch its data, then you will need the ability to dive into densely formatted markup and retrieve the information you need—a process known affectionately as screen scraping.

In one’s haste to grab information from a web page sitting open in your browser in front of you, it can be easy for even experienced programmers to forget to check whether an API is provided for data that they need. So try to take a few minutes investigating the site in which you are interested to see if some more formal programming interface is offered to their services. Even an RSS feed can sometimes be easier to parse than a list of items on a full web page.

Also be careful to check for a “terms of service” document on each site. YouTube, for example, offers an API and, in return, disallows programs from trying to parse their web pages. Sites usually do this for very important reasons related to performance and usage patterns, so I recommend always obeying the terms of service and simply going elsewhere for your data if they prove too restrictive.

Regardless of whether terms of service exist, always try to be polite when hitting public web sites. Cache pages or data that you will need for several minutes or hours, rather than hitting their site needlessly over and over again. When developing your screen-scraping algorithm, test against a copy of their web page that you save to disk, instead of doing an HTTP round-trip with every test. And always be aware that excessive use can result in your IP being temporarily or permanently blocked from a site if its owners are sensitive to automated sources of load.

Fetching Web Pages

Before you can parse an HTML-formatted web page, you of course have to acquire some. Chapter 9 provides the kind of thorough introduction to the HTTP protocol that can help you figure out how to fetch information even from sites that require passwords or cookies. But, in brief, here are some options for downloading content.

- You can use urllib2, or the even lower-level httplib, to construct an HTTP request that will return a web page. For each form that has to be filled out, you will have to build a dictionary representing the field names and data values inside; unlike a real web browser, these libraries will give you no help in submitting forms.

- You can install mechanize and write a program that fills out and submits web forms much as you would do when sitting in front of a web browser. The downside is that, to benefit from this automation, you will need to download the page containing the form HTML before you can then submit it—possibly doubling the number of web requests you perform!
If you need to download and parse entire web sites, take a look at the Scrapy project, hosted at http://scrapy.org, which provides a framework for implementing your own web spiders. With the tools it provides, you can write programs that follow links to every page on a web site, tabulating the data you want extracted from each page.

When web pages wind up being incomplete because they use dynamic JavaScript to load data that you need, you can use the QtWebKit module of the PyQt4 library to load a page, let the JavaScript run, and then save or parse the resulting complete HTML page.

Finally, if you really need a browser to load the site, both the Selenium and Windmill test platforms provide a way to drive a standard web browser from inside a Python program. You can start the browser up, direct it to the page of interest, fill out and submit forms, do whatever else is necessary to bring up the data you need, and then pull the resulting information directly from the DOM elements that hold them.

These last two options both require third-party components or Python modules that are built against large libraries, and so we will not cover them here, in favor of techniques that require only pure Python.

For our examples in this chapter, we will use the site of the United States National Weather Service, which lives here: www.weather.gov/.

Among the better features of the United States government is its having long ago decreed that all publications produced by their agencies are public domain. This means, happily, that I can pull all sorts of data from their web site and not worry about the fact that copies of the data are working their way into this book.

Of course, web sites change, so the source code package for this book available from the Apress web site will include the downloaded pages on which the scripts in this chapter are designed to work. That way, even if their site undergoes a major redesign, you will still be able to try out the code examples in the future. And, anyway—as I recommended previously—you should be kind to web sites by always developing your scraping code against a downloaded copy of a web page to help reduce their load.

**Downloading Pages Through Form Submission**

The task of grabbing information from a web site usually starts by reading it carefully with a web browser and finding a route to the information you need. Figure 10–1 shows the site of the National Weather Service; for our first example, we will write a program that takes a city and state as arguments and prints out the current conditions, temperature, and humidity. If you will explore the site a bit, you will find that city-specific forecasts can be visited by typing the city name into the small “Local forecast” form in the left margin.