Scrape the Web: Take-home cheat sheet

Asheesh Laroia
http://pycon10.asheesh.org/

February 17, 2010

1 Why notes?

These notes serve as a reference companion to the Scrape the Web talk. This is a cheat sheet with higlights of what you need to remember.

1.1 Code samples! Original slides!

Head to http://pycon10.asheesh.org/.

2 Why scrape the web?

It's the world's largest, public-access remote procedure call system. Who can resist?

3 Dealing with the network

3.1 Retrieving documents

- **urllib2.urlopen**: Convenient for easy jobs; bundled with Python. Don't bother with cookielib yourself.
- **mechanize.Browser**: Jump straight to this if you need to set the User-Agent header or deal with cookies.
- **robots.txt**: Try to respect it. mechanize.handle_robots(False) to disable.

3.2 IP address blocking

- SSH tunnels are your first line of defense.

4 Coding strategies

- **IPython and its %edit**: This enhanced interactive Python shell provides a magically convenient %edit built-in. Iterate on your scraper until you're satisfied; re-run your scraping after every save.
- **Save your HTML to disk**: Separate the downloading of a page from its analysis. Preferably, save it to disk first. That way, if your scraper fails, you can recover.

5 Pulling information out of web pages

5.1 "It's text"

If you don't care about the structure of the page, you can just:

- **use string comparisons**: ("eggplant" in urllib2.urlopen(URL).read().lower())
- **use regular expressions**: be careful!
  - If you *must* use regular expressions, don't go alone. Use a regular expression GUI like Kodos to interactively play with your regexp.

  Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems. – Jamie Zawinski.

- Really, be careful with regular expressions. <a href=""> and <a href=''> are not identical. But when machines generate HTML, it might be reasonable.

5.2 "It's HTML"

Web browsers parse web pages. You can, too. Here are some bad ways to do it:

- **XML parsers** (like xml.dom.minidom): Don't bother. Web pages generally don't validate.
- **htmllib**: Event-oriented interface for parsing, like SAX for XML. Gone in Python 3.0. Most document trees aren't long enough to require it, and for those that do, lxml.html should be okay.
- **HTMLParser**: Bad at handling invalid HTML. Stay away unless you know what you're doing.

And here are reasonable choices for parsing:

- **BeautifulSoup**: Convenient, and pure Python. But the latest version doesn't work well. It's time to walk, not run, away from relying on it.
- **html5lib**: A slow but high-quality parser for the busted web pages of the 21st century.
- **lxml.html**: Fast, based on a C core, and high-quality. Comes with a convenient cssselect() feature for finding elements. My personal favorite.

For poking around in the document outside of Python, absolutely use these tools from your favorite browser.

- **View source**: A good quick way to sanity-check the page you're looking at.
- **Inspect element**: *Overwhelmingly* useful. Use Firebug for Firefox, or another browser's built-in DOM inspector, to see a visual representation of the parsed document. Generally fast, easy, and painless.

5.3 “It’s XHTML”

It’s actually nearly never XHTML. See 3.2. Even when it is XHTML, the above tools will work fine.

6 Forms

6.1 Two HTTP methods: GET and POST

<FORM> tags in HTML let browsers submit data. You can find the URL to submit to by checking the <FORM ACTION> attribute. There are two kinds:

- **GET**: The default, this uses a query string (?a=b) to store arguments. These are supposed to be bookmarkable and idempotent.
- **POST**: These are (supposed to be) used for server requests that modify something about the world, like submitting a purchase.

The *name* attribute of the INPUT element drives the form keys.

6.2 Filling out and sending forms

- **urllib2**: To GET, add ’?’ + urllib.urlencode({’name’: ’value’}). To POST, use urllib2.Request.
- **mechanize**: The easiest way is to find the form on a page and select_form() it.

7 Tricks to keep up your sleeve

7.1 Getting around IP address limits

Fundamentally, you can’t. But if you have more IP addresses to use, add an SSH tunnel + tsocks or socks_monkey. Try Tor or Coral CDN if you want to ride on top of others’ addresses, but play nice.

7.2 Solving “Human detection” images (CAPTCHAs)

- Many CAPTCHAs are extremely simple, asking the user to label one of a handful of images. You can label them in advance.
- In a pinch, show them to a human!
- JDownloader has a few CAPTCHA solvers built-in. In a pinch, look at those or try Jython.

7.3 Executing JavaScript

- If it’s easy, just rewrite the JavaScript in Python.
- If it’s not, try SpiderMonkey for a good time.
- If that’s not enough...

7.4 Mechanizing a full web browser

These tools are particularly helpful for "rich" web applications that rely heavily on JavaScript.

- **Selenium Remote Control**: Through Python, remotely command Firefox, Safari, Internet Explorer, and other browsers. I have had a fun time with this! Try the Selenium Recorder to automatically generate Python code to execute various actions.
- **Windmill**: Pure Python version of the above. I have less experience with it, but it looks powerful.

If you are scraping a simple website that lies behind a JavaScript-heavy CAPTCHA (e.g., reCAPTCHA), you might try loading up a full web browser, asking a human to solve the CAPTCHA, and then transferring the cookies over to a Python mechanize bot that does the real work.

7.5 Automatically reverse-engineer website templates

It'd be nice if the computer could simply *learn* the template the website uses. In some cases, it can.

- **templatemaker**: Adrian Holovaty's old Python tool for guessing a decent scraping function given a few examples of a web page.
- **everyblock templatemaker**: everyblock.com uses a new, undocumented version of templatemaker. Dig through their source code dump to find it.

8 How Python-based scrapers can be detected

Your Python code probably doesn't emulate a web browser very well. For example, your Python code:

- may send different HTTP headers than a web browser (particularly User-Agent, but also Accept: and others)
- probably doesn't download tracking images or execute JavaScript
- probably doesn't have a human moving a mouse over DOM elements
- might GET /robots.txt
- and so forth.

The remote web server can probably fingerprint you. Few do, though.

9 Why bother?

Does this seem like a lot of work?

Just remember: the web site *is* the API.