EFM Winter 2015, Week 3: Web Scraping

Jason Phang, Zachry Wang, Allan Zhang

January 28, 2015

Table of Contents

Table of Contents

Introduction

- Web scraping is an immensely useful skill just in general.
- Economics research, pulling information off of terribly designed sites, or just for fun.
- Note though that scraping shouldn't be thought of as a first order solution to the problem of collecting data. Most big data websites will have data available via an API that you can easily call
- Why web scrape then? Because there's a wealth of data that isn't formatted nicely for you.
- Ex. Product prices and descriptions, real estate listings, university courses, sports results, etc

Requisite Tools and Knowledge

- You will need:
- Python with the lxml and Requests libraries
- A web page with nicely structured and formatted data
- Basic HTML knowledge, specifically how tags and content are formatted
- Knowledge or willingness to learn either CSSSelector or XPath

Caveats/Other Notes

- This bears repeating: ALWAYS check for an API before you try to scrape. It's both common courtesy and massively time saving
- Be courteous when scraping. Don't send too many requests too quickly as otherwise you will get blocked
- If the data you need is behind a login, you should use OAuth (or equivalent) if possible. Passing your plaintext username/password is obviously dangerous
- If the data you want is gotten either asynchronously or hidden behind a javascript call (ex. Facebook, although they have an API), then you want to send the request made by the script to the server and scrape the result
- If you're doing large scale data pulling with deep link trees, consider using a web crawler instead. Scrapy for Python uses much of what we'll be covering here

Table of Contents

The Requests Library

- To grab web pages, you will want to use the Requests library from Python
- Python does have a default web page grabber, urllib2, but that library is very clunky to use
- In terms of how to use it, I will actually mostly defer to their documentation. It's very simple to understand, but I will highlight important parts in the following slides
- The quick start documentation for the requests library is here: `http://docs.python-requests.org/en/latest/`

Requesting Web Pages

- Getting pages is very simple:

```
r = requests.get('urlhere')
```

- Note that r is a response object and not the text of the page itself. To get the text, we want to use:

```
r.text
```

- Alternatively, you can just straight up grab the text of the page if you don't care about any other info from the response object:

```
r = requests.get('urlhere').text
```

Parameters in URLs

- Ever come across URLs that look like this?

```
https://www.google.com/search?q=google&rlz=enUS430US430
```

- If you want to crawl a series of web pages that have multiple different changes like:

```
https://www.google.com/search?q=google&rlz=enUS430US430
https://www.google.com/search?q=google+hi&rlz=enUS430US431
https://www.google.com/search?q=google+no&rlz=enUS430US432
```

- Don't attempt to iterate through these URLS manually, because that gets messy when you want to change multiple things.
- Requests will let you pass a dictionary of values instead. Trust me when I say that this syntax is much easier

Parameters in URLs Example

- If you want to crawl:

```
https://www.google.com/search?q=google+no&rlz=enUS430US432
```

- Then pass:

```
dictionary = {'q': 'google+no', 'rlz': 'enUS430US432'}
r = requests.get("https://www.google.com/search", params=dictionary)
```

- And you'll get the URL above

Authentication

- Occasionally you'll need to authenticate yourself via username/password
- Requests has a good tutorial on authentication, which you can find here: `http://docs.python-requests.org/en/latest/user/authentication/`
- Just a few notes on what you should use:
- OAuth should be your first choice. Most large websites will support it such as Google, Yahoo, Amazon, Tumblr, Vimeo, etc, etc
- Try to see if the server supports Digest Authentication next, as it at least provides some security since your password is MD5 encrypted
- If all else fails, use Basic Authentication, but make sure you're sending it to an https url.

Table of Contents

Converting Plaintext

- So now that you have the HTML code from a URL, how do you get the information you want?
- This is where lxml comes in. It's the main tool you will use to actually run through the HTML and find what you want
- You can import lxml in a variety of ways, but the package we want is the html package. It includes all of the functions from the main lxml package etree, but also includes some functions specifically for html:

```
from lxml import html
```

- lxml works by parsing an internal class that's called an Element
- Thankfully, it's very easy to pass a string into an Element class:

```
r = requests.get('urlhere')
parsed = html.fromstring(r.text)
```

Introducing XPath

- Now we have the text of the page set up as an Element, but what can we do with it?
- lxml has many functions that you can use, but the one we want is the xpath function
- With this, we can pull blocks of text with specific html code around it, like so:

```
print parsed.xpath('//a/@href')
```

- The above code will print all href attributes of links in a nice list for us
- Obviously you can exploit this to pull all sorts of things

On XPath

- So you may be wondering what XPath is. In short, it's a specification for navigating XML documents and thus HTML documents by extension
- You should definitely check out the w3schools tutorial on XPath: `http://www.w3schools.com/xpath/`
- I highly recommend that you check out the XPath Syntax page in that tutorial, as that provides everything you need for our purposes
- Here, I'll just provide a bit of context for terms/concepts you may not be familiar with
- In essence, everything in an XML document is called a node. XPath allows you to search through these nodes and pull out information you want
- Attributes are 'modifiers' of sorts to the name of a node. For example, in a node like: <woah option1="dude" option2="Socrates">, then option1 and option2 are the attributes of the woah node
- Note that dude and Socrates are called the values of the attributes

Side Note: XPath vs CSS Selector

- Some of you with more web development experience may be aware of another language used to pull elements: CSS Selector
- You may be more comfortable with this syntax, and if you are, feel free to use it since lxml supports it via the cssselector package:

```
import lxml.cssselector
```

- For our purposes, I will be using XPath because 1) it's the default in the lxml library, 2) I learned it first, 3) Because it's slightly more powerful than CSS Selectors in certain applications

Finding What You Want: Introduction

- With XPath, we now have a tool to access the text contained within certain nodes. Now how do we determine how to grab our text?
- Manual searching through the HTML
- Now to aid in this, I highly recommend using developer tools in your browser to read the source code
- Why? Well for one not all developers have line breaks in their html, and second, because you get in line element highlighting and nicely collapsed html
- Both Firefox and Chrome have them installed by default. Look in the menus for them, or you can right click on any page and select Inspect Element
- Even better is that these tools have a feature to copy the XPath or CSS Selector of any element by default
- You'll still have to play with the XPath to grab every element you want though, so that will only get you so far

Searching Through HTML

- Let's walk through a very basic example, because I don't think there's a better way to do this. Here's a short sample of HTML from reddit:

```
<div id="siteTable" class="sitetable linklisting">
    <div class=" thing id-t3_2u05rs odd  link " data-fullname="t3_2u05rs">
        <span class="rank">1</span>
        <a class="thumbnail may-blank " href="http://i.imgur.com/sys5WBK.jpg">
        </a>
    </div>
</div>
```

- The first div node marks the entirety of all posts on the front page, the second div node marks the start of the first post, and the span and a are the rank and thumbnail node for the first post
- Now to get the rank and link of the first post, we want:

```
print parsed.xpath('//*[@id="siteTable"]/div[1]/span/text()')
print parsed.xpath('//*[@id="siteTable"]/div[1]/a/@href')
```

Getting All Elements

- Okay so we got what we want for the first post. How about getting it for all posts? Well that's pretty easy:

```
print parsed.xpath('//*[@id="siteTable"]/div/a/@href')
```

- See the difference? Now we're telling it to get the value of the href attribute for ALL div nodes under a node that has an attribute value pair of id="siteTable"
- This will simply return a list of links, which can be easily manipulated in Python

Extensions

- Of course we only scratched the surface here, and you can do much more complicated searches through nodes names, their attributes, and their values in order to get what you want
- I won't be providing code for any of these, but just several examples of extensions
- XPath also natively supports operators like $>, <, |, +$, or, and, etc. You can use these in very creative ways to get what you want.
- Examples: You can get book descriptions only if the value of their price attribute is greater than \$20, reddit links only if the upvotes are within a certain range, etc
- You can of course also do this in Python, but then you'd have to get two lists of prices and descriptions and the size required for your data blows up really fast if you're doing large scale scraping
- XPath also has functions. For example, the contains function checks if a string has another substring within it. This is also really helpful

Web Crawling

- Note that even with just lxml and requests, you can crawl through websites
- Example: Scrape a root page for descriptions, grab all links on that page, and scrape those pages for descriptions
- If the URLs have some kind of pagination, then that's easy as well. Just keep an updated counter of the current page and only look for the link that contains the next page
- Note though that crawling through all pages in a tree is difficult without more tools
- You need to constantly check if you've went through a link already, which actually gets very complex very fast
- If you want to web crawl, then you definitely want to check out Scrapy, which is a Python library designed to make the above task very easy

Pages with Data behind Javascript/AJAX

- First, check to see that your data isn't actually on the page
- Oftentimes, all the JS is doing is making a call to another page on the sever, so if you just see what request it's sending out, have your scraper make that same request, and you should get your data
- Sometimes you just straight up won't be able to get what you want from just working with the html code. At this point you'll want to check out Selenium (for Python) and PhantomJS, which can allow you to essentially scrape something closer to what you see on your browser screen
- Essentially what you're doing is letting a browser assemble what's called a DOM (Document Object Model) from the HTML code, rendering that, and then scraping the DOM instead of the HTML
- Be careful if you go that route though, since sometimes websites will serve up different things to different browsers because of the different layout engines browsers use (Trident in IE, Gecko for Firefox, Blink in Chrome/Opera, WebKit for Safari, etc)

Summary of Steps to Scrape

- Find the page that contains your wanted data
- Use browser developer tools to find the HTML element that contains your data
- Get the XPath for that element, and manipulate it until you make it collect only your wanted data
- Run several small scale experiments over several pages with your data and confirm that you get what you want
- Scale it up and let it run

Final Bits and Pieces

- Definitely check out the links that I referred to throughout the presentation as they're great resources
- I've collected all links in the presentation as well as additional useful ones on the next slide for your convenience
- If you notice, there isn't really that much to lxml and requests. The main struggle comes with learning XPath and figuring out clever ways to get at what you want
- Generally though, most big websites and data sources will have APIs for you to access, so you probably want to check those out first, since reading HTML can get painful
- Definitely try to do the assignment for this one. Practice definitely makes perfect when learning new syntax

Assorted Links

- Requests library guide: `http://docs.python-requests.org/en/latest/`
- Requests library authentication guide: `http://docs.python-requests.org/en/latest/user/authentication/`
- XPath tutorial: `http://www.w3schools.com/xpath/`
- Basic tutorial for scraping: `http://docs.python-guide.org/en/latest/scenarios/scrape/`
- A good resource for scraping. Covers some of the same material as in this presentation as well as some other topics. Best resources are probably the examples on the bottom of the page: `http://jakeaustwick.me/python-web-scraping-resource/`