Lab 17

Advanced BeautifulSoup

Lab Objective: *Learn how to use BeautifulSoup to scrape information from the internet and put it into easy-to-access data tables*

The internet is full of information. Sometimes this information is easy to read, sometimes it's not. No matter the case, web scraping is a useful tool used to transform unstructured data into structured data that is easier to analyze.

Throughout this lab, you will use two Python modules in conjunction with `BeautifulSoup4`: `urllib2`, and `selenium`. Make sure these are installed and updated on your machine.

> **Warning**
>
> Web scraping has legal implications. Do not scrape copyrighted information without the consent of the copyright owner. Many websites, in their terms and conmditions agreement, prohibit the practice of crawling parts or all of their website. Be careful and considerate when doing any sort of crawling. Be careful when writing and testing code so that there is no unintended behavior.

Scrapers and Crawlers

If you are on a website and all the information you want is contained on one page, you simply use a web scraper to read through the html code and pick out what you want. An example of this might be your professor's webpage, and you just need to scrape his phone number and email. Suppose instead that your information is found only after clicking through various links. For example, you want to find the email addresses and phone numbers for all the professors in the math deaprtment, but you can only get this information by clicking though an online directory, similar to `http://math.byu.edu/peopleresearch/faculty`. This would require *crawling* through various links to open up each professor's home page, and then scraping their sites. So scrapers are good for getting the information you need from a page,

while crawlers will get you to the various pages from which you want to scrape information.

What Can You Scrape?

There are some websites that permit the practice of web scraping. To ensure that scraping is well-behaved, most websites will tell a crawler where they can and cannot go, and scrapers what they can and cannot scrape. All of this information is included in a text file in the root domain of a website. The file is always titled `robots.txt` and defines considerate behaviors for web crawlers. Each robots file has a set of rules that label parts of a website as disallowed. Parts of a website that are not disallowed are implied to allow access by web crawlers. Many websites will limit crawlers to parts of the sites that will not place a large load on the website's server. It is your duty to honor the rules in `robots.txt` if they exist.

Simple Scraping

In lab 16, BeautifulSoup was used to read short bits of HTML code or a file using the `open()` command. Once the file is loaded, you can navigate through the HTML tree and pick out the data that you want.

Problem 1. Use BeautifulSoup to load the 'Big Bank Info' file. This page is also found at `http://www.federalreserve.gov/releases/lbr/20030930/default.htm`. Using the methods taught, make a 2-D array for bank information. Each row represents a different bank, and each column represents various bank information. In your 2-D array, include the Bank Name, Rank, ID, Domestic Assets, and number of Domestic Branches for the following banks: JPMORGAN, CAPITAL ONE, and DISCOVER.

This is a very slow way to access information from a website. Remember this was only one in a list of over 20 different links to bank information. What if we wanted to get information from every link? Imagine trying to go through HTML trees for a hundred different websites only by copying and pasting HTML code!

Use the `urllib2` library in conjunction with BeautifulSoup to load HTML code from any website. We will go through some simple examples. First import `urllib2` . Use `urllib2`'s `urlopen()` function in conjunction with `read()` to load the HTML code into Python. Then simply use `BeautifulSoup()` to turn the HTML code into a navigable HTML tree.

```
>>> from bs4 import BeautifulSoup
>>> import urllib2

>>> url = "http://csb.stanford.edu/class/public/pages/sykes_webdesign/05_simple.↩
    html"

>>> content = urllib2.urlopen(url).read()
>>> soup = BeautifulSoup(content)
```

```
>>> print(soup.prettify())
<html>
 <head>
  <title>
   A very simple webpage
  </title>
  <basefront size="4">
  </basefront>
 </head>
 <body bgcolor="FFFFFF">
  ...
 </body>
</html>
```

Warning

Since `urllib2` accesses a website server, you may run into problems where you cannot establish a connection to the server. You can create a `try-except` clause to account for this, or just rerun your program.

Problem 2. Using the website `http://www.wunderground.com/history/airport/KSAN/2015/1/1/DailyHistory.html?req_city=San+Diego&req_state=CA&req_statename=California&reqdb.zip=92101&reqdb.magic=1&reqdb.wmo=99999&MR=1` and BeautifulSoup, do the following:

- Return the Actual Max Temperature.
- Return the tag which contains the link for the 'Next Day' button.
- Return the url attached to the link.

Sometimes, data we are interested in is contained over several different web urls. For example, what if we wanted a graph of the temperature highs over a period of time? For `www.wunderground.com`, the temperature history for a given city will be contained on separate pages for each day, just as in problem 2. In order to access information over several websites, we just need to load each new website into BeautifulSoup and locate the information we're interested in. Consider the following example.

In this example we look at the actual maximum temperature over the year 2014 in San Diego.

```
from bs4 import BeautifulSoup
import urllib2
import re

weather_url = 'https://www.wunderground.com/history/airport/KSAN/2014/1/1/↩
    DailyHistory.html'

weather_content = urllib2.urlopen(weather_url).read()
```

```
weather_soup = BeautifulSoup(weather_content)

actual = []

while('2015' not in weather_soup.find(class_='history-date').string):
    while(len(weather_soup.find_all(string='Actual')) != 1):
        weather_content = urllib2.urlopen(weather_url).read()
        weather_soup = BeautifulSoup(weather_content)
    actual_temp = weather_soup.find(string='Max Temperature').parent.parent.↩
        next_sibling.next_sibling.span.span.text
    actual.append(int(actual_temp))
    next_url = weather_soup.find(string=re.compile('Next Day')).parent['href']
    weather_url = 'https://www.wunderground.com'+next_url
    weather_content = urllib2.urlopen(weather_url).read()
    weather_soup = BeautifulSoup(weather_content)
```

Let's examine the code to see how it works.

After importing the necessary modules and opening up the url in BeautifulSoup, we define the variable `actual` to store the max temperatures in a list. Notice in the url that our dates start in 2014. Since we only want to go through the year 2014 and stop in 2015, we need a way to end our search once we get into 2015. Using `class_='history-date'`, we can find a tag which has the year in the string.

The next while loop is an error check. Sometimes this website does not load properly, so we check that all the information we need is located in the HTML code. In this case, we are looking for the actual max temp for a day, so we need to make sure the 'Actual' column shows up.

The next line of code defines `actual_temp`, which first directs us to the row of 'Max Temperature' and then navigates to the temperature column. This value is then turned to an `int` and stored in the list of temperatures.

Lastly, we find the url reference that is associated with the 'Next Day' link. We manually create the new url, keeping in mind that the new url found in the link is only an extension of the server website. This means that if the server is found at `http://www.wunderground.com` and the `'href'` value for the link is `/history/airport/KSAN/...`, then the new url we load into BeautifulSoup is `http://www.wunderground.com/history/airport/KSAN/...`. Therefore, we add the string representing the base url with the string representing the extension part of the url.

The output is the list of actual max temps over a year, which we can graph.

Problem 3. Adjust the above program to make a list of the average max temperatures over a year. Make a graph of the data points with the day of the year on the x-axis and the average temperatures on the y-axis.

Let's do another example before we turn you lose on the internet. This next example will look at the Commercial Bank data found at `http://www.federalreserve.gov/releases/lbr`. We will look primarily at the Consolidated Assets for JPMorgan over the years from 2003 to 2014.

```
from bs4 import BeautifulSoup
```

```
import urllib2
import re

bank_url = 'http://www.federalreserve.gov/releases/lbr/'
bank_content = urllib2.urlopen(bank_url).read()
bank_soup = BeautifulSoup(bank_content)

assets = []

dates_list = bank_soup.find_all(href=re.compile('((200[3-9])|(201[0-4])).*/default.htm'))

for url in dates_list:
    link_url = str('http://www.federalreserve.gov/releases/lbr/'+url['href'])
    link_content = urllib2.urlopen(link_url).read()
    link_soup = BeautifulSoup(link_content)
    tag = link_soup.find(string=re.compile('JPMORGAN'))
    tag = tag.parent.parent.find_all('td')
    amt = tag[5]
    assets.append(str(amt.string))
```

Now we will examine this code and see how it works.

We first import the necessary modules and load the url into BeautifulSoup. We create the variable `assets` to store the list of asset values.

To get the links we need, we use `find_all()` and select the unique identifiers of the link, namely the links that start with `20**` and end in `/default.htm`. Next we run the `for` loop to iterate through each link.

Just as in the previous example, we need to concatenate the server url `http://www.federalreserve.gov/releases/lbr/` with the extension urls we stored in `dates_list`. This link is loaded into BeautifulSoup.

Now that we are on the webpage desired, we find the row we want by looking for the tag containing `'JPMORGAN'`. Once we have the tag, we navigate to the appropriate column.

The info is then appended into the data list.

Problem 4. Choose 1 of 3 options.

1. Load `http://www.google.com/finance` into BeautifulSoup. Towards the bottom of the web page, there is a Sector Summary. Go through each sector and locate the top five Gainers. In a SQL table, store the Name, abbreviation, % Change, and Mkt Cap of the top Gainer for each Sector.

2. Load `http://www.espn.go.com/nba/statistics` into BeautifulSoup. Go through the top five offensive leaders. In a SQL table, store the name, career games played, career mins per game, career points per game, and career FG% for each player.

3. Load `http://www.foxsports.com/soccer/united-states-women-team-stats` into BeautifulSoup. Go though each player on the World Cup US

women's team. In a SQL table, store the name, hometown, position, and # of games played in the World Cup.

Advanced Scraping

The examples we have looked at so far have been very basic since the HTML is stored in the source code for the web pages. However, we will look at some examples where the HTML is written dynamically. This means the HTML is brought in from a separate source through javascript or ajax as a `.php` or `.aspx` table. These tables can be difficult to grab data from.

Go to the website `http://www.simplesoccerstats.com/stats/teamstats.php?lge=14&type=goals&season=0`. Open up the page source by right clicking and choosing the option for the page source. There is HTML code, but is it correct? Hit `ctrl+f` and search for 'Chicago,' one of the teams that appears on the actual webpage. It's not there! You can even try the following:

```
>>> from bs4 import BeautifulSoup
>>> import urllib2
>>> import re

>>> soccer_url = 'http://www.simplesoccerstats.com/stats/teamstats.php?lge=14&↩
    type=goals&season=0'
>>> soccer_content = urllib2.urlopen(soccer_url).read()
>>> soccer_soup = BeautifulSoup(soccer_content)

>>> print(soccer_soup.find(string='Chicago'))
None
```

Still nothing. This means the actual table of information is stored somewhere else.

Selenium

Selenium is a great tool to use on simple websites as well as websites with dynamic HTML source code. Basically Selenium will open up a browser and you can see what it is looking at. You can look at the source code of the actual website, and you can have some limited control over navigation, such as clicking links, clicking dropdown menus, pressing the back or forward buttons, etc. To use Selenium, import the following:

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
```

The `webdriver` allows you to use website functionality, while `Keys` allows you to use special keyboard keys like RETURN (i.e. when you want to send information through text boxes). Next, you want to open up a web browser and a website. For these exercises we will use Firefox. You can also use Chrome or IE if those are more familiar; all the commands will be similar.

```
driver = webdriver.Firefox()
```

```
example_url = 'http://www.example.com'
driver.get(example_url)
```

The `.get()` command acts like `urllib2`'s `.urlopen()` and `.read()` combination. We can print out the HTML code using Selenium's `.page_source` attribute.

```
>>> print(driver.page_source)
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"><head>
    <title>Example Domain</title>
    ...
</body></html>
```

Once we have the source code, we can read it into BeautifulSoup.

Remember how we said Selenium reads HTML from a webpage differently than BeautifulSoup? Take a look again at the soccer example.

```
>>> from selenium import webdriver
>>> from selenium.webdriver.common.keys import Keys
>>> from bs4 import BeautifulSoup

>>> browser = webdriver.Firefox()
>>> soccer_url = "http://www.simplesoccerstats.com/stats/teamstats.php?lge=14&↩
    type=goals&season=0"
>>> browser.get(soccer_url)
>>> soccer_soup = BeautifulSoup(browser.page_source)
>>> browser.quit() #closes the web browser
>>> print(soccer_soup.find(string='Chicago').parent)
<td>Chicago</td>
```

This time there is a tag with 'Chicago' contained as text!

Problem 5. Consider the url `http://stats.nba.com/league/team/#!/?sort=W&dir=1`. Make a list of the `a` tags containing each of the 30 NBA teams. Use `.find_all()` in conjunction with whatever unique identifiers get you the correct tags. Hint: class and tag name are a good start.

Problem 6. Use the website from problem 5. Create a SQL table which stores the following information:

- The column titles are Name, Wins, and Losses, where Wins is the number of games won, and Losses is the number of games lost.
- Each row represents a different basketball team, with its win and loss totals.

Hint: Use the tags found in problem 5 as a starting point to find the wins and losses.