scraping_workshop

November 6, 2020

1 Introduction to Web Scraping, Part Two (Python)

- UMN LATIS & Libraries workshop, Nov 15, 2019
- Cody Hennesy (chennesy@umn.edu) and Michael Beckstrand (mjbeckst@umn.edu)

In this part of the workshop, we'll explore reproducible web scraping methods using Python.

Specifically, in this part of the workshop we will: * Use Python 3 in a Jupyter computing environment * Use the Requests and BeautifulSoup Python libraries to access HTML data from the web * Create variables, lists and loops to work with web data in Python

Credits: Content for this workshop was adapted from Rochelle Terman's Web Scraping workshop and from Software Carpentry Python lessons.

1.0.1 Why Python?

- Reproducibility
- Repeatable
- Extensible
- Great for data access and data cleaning

1.0.2 What's Jupyter?

- Web-based, easy to share
- Easy to read, easy to run
- Run code piece by piece

1.1 Python variables

- You can use Python as a calculator.
- To "run" a Jupyter cell hold down shift and select Return/Enter, or choose the "play icon" (right-facing triangle) from the Jupyter menu above.

[ ]:
```
weight_kg = 60
```

[ ]:
```
print(weight_kg)
```

In Python, variable names:

- can include letters, digits, and underscores
- cannot start with a digit

- are case sensitive.

You can do calculations and save text strings in variables too.

[ ]:
```
website = "All the words on a website"
print(website)
```

1.2 Importing Libraries

Importing a library is like getting a piece of lab equipment out of a storage locker and setting it up on the bench. Libraries provide additional functionality to the basic Python package, much like a new piece of equipment adds functionality to a lab space. Just like in the lab, importing too many libraries can sometimes complicate and slow down your programs - so we only import what we need for each program.

Our primary tools will be the Requests library and Beautiful Soup

[ ]:
```
import requests
from bs4 import BeautifulSoup
```

1.2.1 Library functions

The expression `requests.get(...)` is a function call that asks Python to run the function `get` which belongs to the `requests` library.

This dotted notation is used everywhere in Python: the thing that appears before the dot contains the thing that appears after.

As an example, we could use the dot notation to write the relationship between Minneapolis and Minnesota as `Minnesota.Minneapolis`, just as *get* is a function that belongs to the *requests* library.

[ ]:
```
requests.get('http://www.startribune.com/')
```

What did we do above?

1. Created a Python HTTP request object for a GET
2. Send the HTTP request to webserver at http://www.startribune.com/
3. Received the response `[200]` from http://www.startribune.com/ - what's that mean?

In Jupyter notebooks using Python you can explore functions of a library using the *tab* key.

And to understand each function you can get information by putting a question mark after it:

[ ]:
```
requests.get?
```

You can store the data that is returned from the GET request in a variable:

[ ]:
```
star_trib = requests.get('http://www.startribune.com/')
#print(star_trib)
```

1.2.2 Ethical scraping

One way to make sure you're engaging in transparent and ethical scraping practices is to send the website information about *yourself* along with your request.

`requests.get` includes a `headers=` parameter that you can use to send in your name and information about the software we're using to collect data:

[ ]:
```
headers = {'user-agent': 'python-requests/2.22.0; chennesy@umn.edu; Cody␣
 ↪Hennesy'}
star_trib = requests.get('http://www.startribune.com/', headers=headers)
```

Now you can explore the attributes of the data object stored in `star_trib` using the same dot notation.

Use tab to explore the options, and the question mark to read more about the attribute.

`star_trib.text`, for example.

[ ]:
```
src = star_trib.text
```

Let's move the .text content that was returned from the Request into a BeautifulSoup object so we can start to explore the HTML tree.

[ ]:
```
# parse the response into an HTML tree by calling BeautifulSoup
soup = BeautifulSoup(src, 'lxml')

# look at what it looks like now, using the soup.prettify tool
# [:1000] will give us the first 1000 characters in the soup object so it␣
 ↪doesn't fill up the whole screen
print(soup.prettify()[:1000])
```

1.3 Find Elements

BeautifulSoup has a number of functions to find things on a page. Like other webscraping tools, Beautiful Soup lets you find elements by their:

1. HTML tags
2. HTML Attributes
3. CSS Selectors

HTML tags Let's search first for **HTML tags**.

The function `find_all` searches the `soup` tree to find all the elements with an a particular HTML tag, and returns all of those elements.

What does the example below do?

[ ]:
```
# find all elements in a certain tag
soup.find_all("a")
```

[ ]:
```
soup.find_all('p')
```

[ ]:
```
soup.find_all('h3')
```

Because `find_all()` is the most popular method in the Beautiful Soup search API, you can use a shortcut for it. If you treat the BeautifulSoup object as though it were a function, then it's the same as calling `find_all()` on that object.

These two lines of code are equivalent:

[ ]:
```
# soup.find_all("a")
soup("a")
```

HTML Attributes If you search for everything with the `a` tag, you're likely to get a lot of stuff, much of which you don't want. What if we wanted to search for HTML tags ONLY with certain attributes, like particular CSS classes?

We can do this by adding an additional argument to the `find_all` like this: `soup("a", class_="class_name")`

1.3.1 Challenge 1: Find the Most Read and Emailed articles on the Star Tribune homepage

Use Chrome's *Inspect* feature, to find the class name for the Most Read and Most Emailed articles lists. 1. Create a variable called most_read, and use `soup()` to find all of the links with the appropriate class 2. Print out the matches below

In the example below, we are finding all the `a` tags, and then filtering those with `class_="feed-list-link"`.

[ ]:
```
# Get only the 'a' tags in 'sidemenu' class
most_read = soup("a", class_="feed-list-link")
print(most_read)
```

CSS Selectors It can be more efficient to search and find things on a website by **CSS selector.** For this we have to use a different method, `select()`. Just pass a string into the `.select()` to get all elements with that string as a valid CSS selector.

In the example above, we can use "a.feed-list-link" as a CSS selector, which returns all `a` tags with class `feed-list-link`.

How to find selectors? A number of browser extensions and other tools exist to help you find HTML and CSS. - Selector Gadget - CSS Selector Helper

[ ]:
```
# get elements with "a.sidemenu" CSS Selector.
most_read_select = soup.select("a.feed-list-link")
```

[ ]:
```
most_read_select
```

1.3.2 Python Lists

The most popular kind of data collection in Python is the list, which takes the place of arrays in programming languages like C and Fortran. Lists have two primary important characteristics: 1. They are mutable, i.e., they can be changed after they are created. 2. They are heterogeneous, i.e., they can store values of many different types.

To create a new list, you can just put some values in square brackets with commas in between.

[ ]:
```
my_list = ['red', 'orange', 'yellow']
my_list
```

To fetch the element at a specific location, put the *index* of that location in square brackets. But keep in mind that Python lists start the index from 0. So the list above has three index values: `my_list[0]` `my_list[1]` `my_list[2]`

[ ]:
```
my_list[1]
my_list[3]
```

Let's go back to all of the a links with the class selector `feed-list-link`.

[ ]:
```
#this is a Python list
most_read = soup.select("a.feed-list-link")
```

You can see how many items are in your list using the `len()` Python function.

[ ]:
```
len(most_read)
```

And you can look at the first element in the list using the syntax variable[0].

Note: [0] refers to the first element in a list in Python, and [1] refers to the second.

[ ]:
```
most_read[0]
```

We can use a built in Python function called type() to explore the results.

[ ]:
```
# save the first element in the list to its own variable to make it easier to␣
↪explore
first_link = most_read[0]

# check out its class
type(first_link)
```

It's a tag! If we look up Tag in the BeautifulSoup documentation, we know that we can use `.text` to look at the text.

[ ]:
```
first_link.text
```

We can also look at the href attribute to check out the URL:

[ ]:
```
first_link['href']
```

1.3.3 Loops

If we want to explore all of the most popular articles, we can loop through each link and only grab the information that we care about.

Note the syntax:

```
for x in y:
    do_something # the code in the loop needs to be indented
    do_another_thing
```

For 'a' tags, we also know there's an 'href' attribute that tells us where the link URL goes.

[ ]:
```
for link in most_read:
    print(link.text, link['href'])
```

Let's clean up the print output in Python by using a built-in .strip() function that removes extra white space from strings, and by adding some line breaks between elements using the `'\n'` escape character.

[ ]:
```
#let's clean that up a bit:
for link in most_read:
    print(link.text.strip(), '\n', link['href'], '\n')
```

1.4 Challenge 2: Find all of the a tags

Remember when we used soup.findall to collect all of the links on the Star Tribune homepage? Create a variable to collect all of the links on the homepage, and then loop over the list to print the text of each link and the URL?

Hint: the results might look pretty messy! You can use `.strip()` to remove whitespace from strings and make the output easier on the eyes.

[ ]:
```
all_links = soup.find_all("a")
for link in all_links:
    print(link.text.strip(), link['href'].strip())
```

1.5 Let's look at a specific article page

[ ]:
```
page = requests.get("http://www.startribune.com/
↪discover-lost-shipwrecks-in-lake-superior/514224542/")
```

[ ]:
```
src = page.text
```

[ ]:
```
page_soup = BeautifulSoup(src, 'lxml')
```

Exploring the HTML in Chrome is a great way to find the right selectors or attributes to scrape, but you can also take sneak peaks at common tags using `.find_all()` to help pinpoint specific elements. For example: `.find_all('h1')` or `.find_all('p')`

[ ]:
```
page_soup.find_all('p')
```

It looks like the p class `Text_Body_mag` would snag the full-text of the article for us:

[ ]:
```
article_text = page_soup.select("p.Text_Body_mag")
```

[ ]:
```
for article in article_text:
    print(article.text)
```

1.5.1 Comments on Star Tribune - How to get them?

Some elements of websites are "hidden" from web scrapers like BeautifulSoup because they appear as part of an iFrame, or because they require other code such as Javascript to load on the page.

If we look at the link to "Show Comments" on the Star Tribune article, for example, there's not a URL, but a call to a Javascript tool called *js-comments* that is visible in the a class selector:

```
<a href="#" class="js-comments-show comments-count-link talk-enabled"><div
class="comments-count">18</div><span class="comments-show
js-comments-show-txt">Show Comments</span><img class="comment-count-image-tracking"
alt="" src="http://apps.startribune.com/circulars/images/blank.gif"
style="display: none;"></a>
```

To capture this "hidden" data, some researchers use browser emulators such as:

- Selenium. This also requires other tools such as ChromeDriver, RSelenium, and/or Selenium with Python.
- Puppeteer

1.6 Challenge 3: Scrape the headline, byline, and date

Explore the HTML for a Star Tribune article to see if you can scrape the headline, byline (author), and the date the article was posted from the page.

- Hint: there are several different routes to get to each element.
- Hint two: The date and bylines can be a little confusing when you use `.select()` because they'll return a Python list, even if there's only one item on the list. To show the `.text` attribute from a list, you can point to the first item on the list using `[0]`.

[ ]:
```
headline = page_soup.h1
headline.text
```

[ ]:
```
byline = page_soup.select('div.article-byline')
byline[0].a.text
```

[ ]:
```
date = page_soup.select('div.article-dateline')
date[0].text.strip()[:-9]
```

1.6.1 More resources

- Programming Historian's Intro to BeautifulSoup