scraping_fa20

November 6, 2020

1 Introduction to Web Scraping, Part Two (Python)

- UMN LATIS & Libraries workshop, Nov 6, 2020
- 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.

Note: Another popular tool for web scraping with Python is Scrapy. The general consensus is that it's a faster tool, and it does *more* than Beautiful Soup, but it might be more complex to learn.

[ ]:
```
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): Finding the Most Read and Emailed articles on the Star Tribune homepage

We can use Chrome's *Inspect* feature, to find the class name for the Most Read and Most Emailed articles lists (feed-list-link). 1. Let's create a variable called most_read, and use `soup()` to find all of the links with the appropriate class 2. Then we'll 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.

We'll 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.

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

While printing content can provide a useful output as we code, it's usually much more useful to store the data so that we can operate on it later on (save it, clean it, etc.). In this case let's create an empty Python list and then extract the URLs from our most_read list to save there.

[ ]:
```
# you can create an empty list using open and closed square brackets without␣
↪content
most_read_urls = []

for link in most_read:
    # we can append new items to a list "in place" (without using an equals␣
↪sign) using the append() function
    most_read_urls.append(link['href'])
```

[ ]:
```
print(most_read_urls)
```

Now we have a list of URLs that we can later use as part of a modular approach to scraping. If we figure out how to scrape the content from an article page, we can re-use that code and say "scrape the article content for every page in this list of URLs." To do that, let's figure out how to scrape content from an article page.

1.4 Scrape an article page

[ ]:
```
page = requests.get("https://www.startribune.com/
↪former-vikings-great-matt-blair-dies-at-age-70-likely-linked-to-cte/
↪572832852/")
```

[ ]:
```
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` would snag the full-text of the article for us:

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

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

1.4.1 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

1.5 Challenge (3): Scrape the headline, byline, and date

Let's explore the HTML for a Star Tribune article to see if we can scrape the headline, byline (author), and the date the article was posted from the page.

[ ]:
```
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 Functions

Now let's define a function `scrape_articles` that cycles through our `most_read_urls` list, and grabs all of the data we care about from each page.

The function definition opens with the keyword **def** followed by the name of the function (format_articles) and a parenthesized list of parameter names (unformatted_docs). The body of the function — the statements that are executed when it runs — is indented below the definition line. The body concludes with a return keyword followed by the value we want to take from the function.

[ ]:
```
def add_things(x, y):
    z = x + y
    return z
```

We can run the function, passing values to the x and y parameters, and save the return value to a new variable.

[ ]:
```
z = add_things(10,25)
print(z)
```

1.7 Putting it all together: Scraping list function

So let's use a function to scrape a few key elements from article pages. The input that the function will take is a list of URLs. Then for each url, it will scrape the headline, byline, and article text. Since we'll be hitting the Star Tribune server pretty rapidly, let's build in a timer using the time library to pause between each page.

[ ]:
```
import time
```

[ ]:
```
# new function that accepts one parameter, a list.
def scrape_strib(url_list):
    #empty list to hold each page
    results_list = []

    for url in url_list:
        print('Scraping:', url)
        time.sleep(5) # wait for 5 seconds

        # request the page content and convert it to a soup object
        page = requests.get(url)
        src = page.text
        page_soup = BeautifulSoup(src, 'lxml')

        #find the text on the page
        article_text = page_soup.select("p") # because different kinds of␣
↪articles use different p classes, we'll make this as generic as possible

        # save all of the p_tags to a long string
        article_string = ''.join(article.text.strip() for article in␣
↪article_text)

        # save the headline (there should only be one!)
        headline = page_soup.h1
```

```
        # we haven't talked about if/else statements, but before assigning a␣
↪value we want to check that headline exists
        # we can do that with the handy statement if x is not None:
        # whatever is indented will only take place if the 'if' statement␣
↪evaluates as true; otherwise we'll skip it, and go to the else
        if headline is not None:
            headline = headline.text.strip()
        else:
            headline = ''

        # bylines have a lot of variation from page to page so we're going to␣
↪do a somewhat complex if/else check
        byline = page_soup.select('div.article-byline')
        if len(byline) > 0:
            if byline[0].a is not None:
                byline = byline[0].a.text
            else:
                byline = ''
        else:
            byline = ''

        # this is a new kind of container object, known as a tuple. It's a good␣
↪way to package these lists and strings together for each page, and then␣
↪assign them as one object to the overall results_list.
        article_tuple = (url, headline, byline, article_string)
        results_list.append(article_tuple)

    #return results_list after the for loop concludes
    return results_list
```

Now we can call our function, passing the most_read_urls list as a parameter, and saving the return value to a variable called scraped_articles.

[ ]:
```
scraped_articles = scrape_strib(most_read_urls)
```

We can explore the content by referring to the index for each item in the scraped_articles list. Also, looking over the URLs above, we can guess that some of these (like the videos) are probably not going to fit our articles list very well. When building a scraper, you'll want to spend a fair amount of time testing different kinds of content and building if/else statements to only collect the data you need.

[ ]:
```
# we can look at the full tuple for each article
print(scraped_articles[0])
```

[ ]:
```
# we can look at specific items by adding referring to the index of the tuple
print('First article:')
```

```
print('url:', scraped_articles[0][0])
print('title:', scraped_articles[0][1])
print('byline:', scraped_articles[0][2])
print('First 250 chars from the text:', scraped_articles[0][3][0:250])
```

2 Data view and export

When you've collected a lot of data like this it can be helpful to save it for later use. Which format you want to use to save data in Python depends on the structure of the data, but common formats are CSVs, JSON, and pickle files.

In the case of more complex data structures, like our list of lists, we can use a tabular data tool called Pandas to store each article in a row, and then save that Pandas dataframe to a CSV file.

[ ]:
```
import pandas as pd

# pandas has a function called DataFrame() that accepts a list as its input,␣
↪and then we'll define the names of our column names
df = pd.DataFrame(scraped_articles, columns=['url', 'title', 'byline', 'text'])

# now let's view the table
df
```

And we can save a dataframe to a csv file using the pandas function to_csv().

[ ]:
```
df.to_csv('scraped_sites.csv')
```

2.0.1 More resources

- Programming Historian's Intro to BeautifulSoup