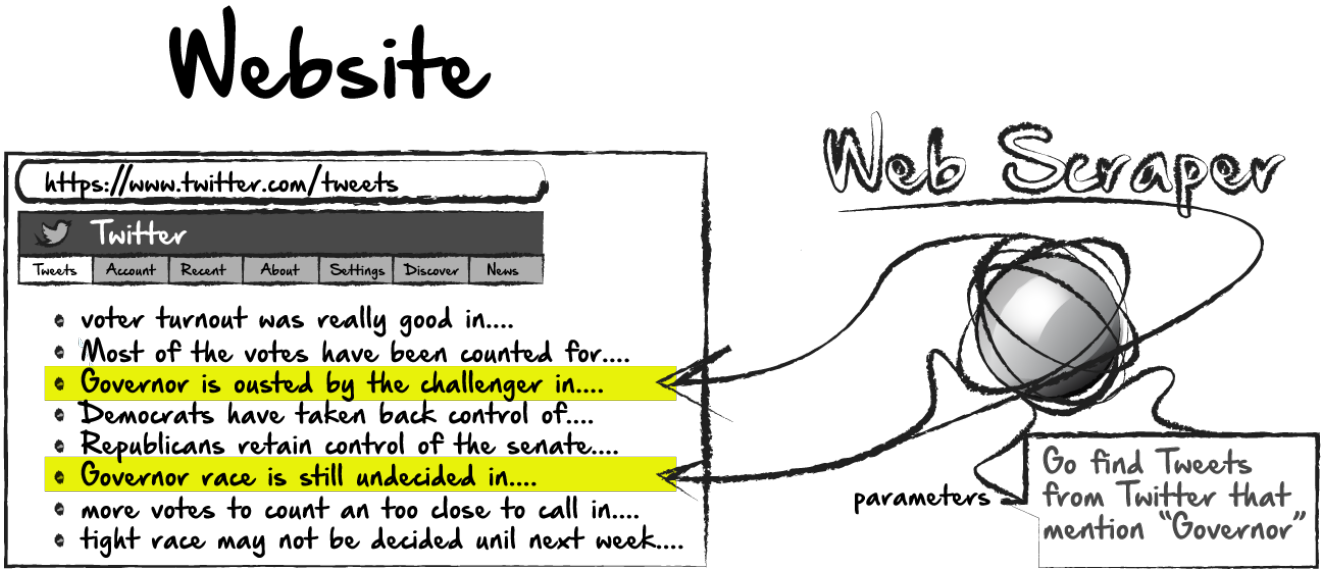
Building a Web Scraper from start to finish

hackernoon.com/building-a-web-scraper-from-start-to-finish-bb6b95388184

What is a Web Scraper?

A Web Scraper is a program that quite literally scrapes or gathers data off of websites. Take the below hypothetical example, where we might build a web scraper that would go to twitter, and gather the content of tweets.

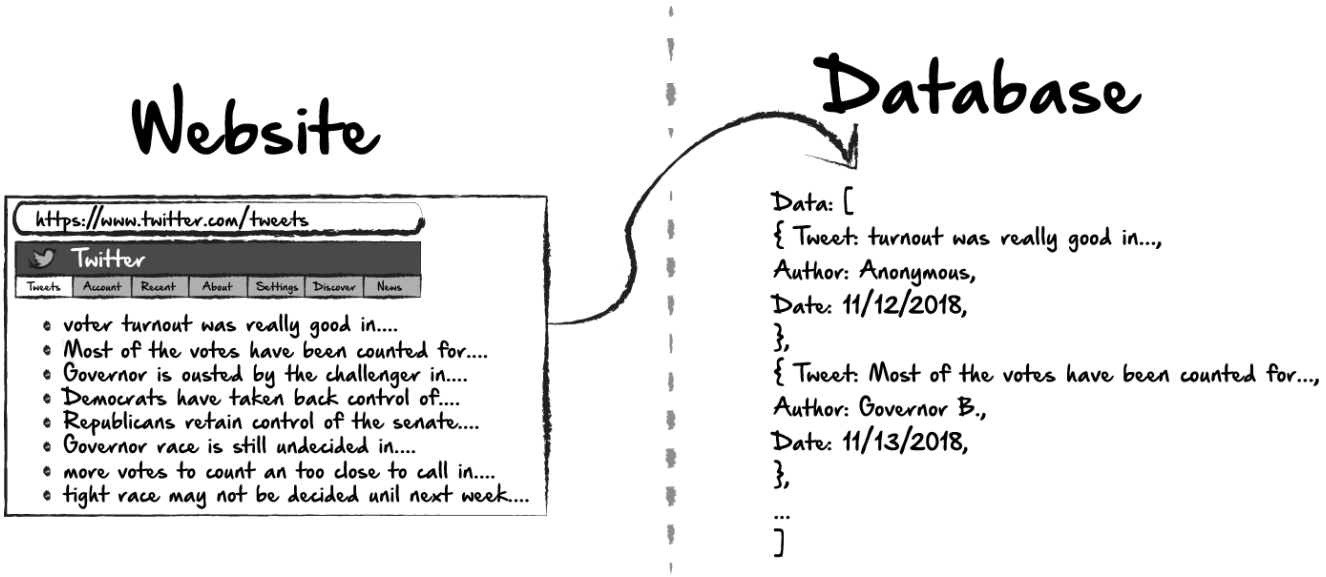


In the above example, we might use a web scraper to gather data from Twitter. We might limit the gathered data to tweets about a specific topic, or by a specific author. As you might imagine, the data that we gather from a web scraper would largely be decided by the parameters we give the program when we build it. At the bare minimum, each web scraping project would need to have a URL to scrape from. In this case, the URL would be

twitter.com. Secondly, a web scraper would need to know which tags to look for the information we want to scrape. In the above example, we can see that we might have a lot of information we wouldn't want to scrape, such as the header, the logo, navigation links, etc. Most of the actual tweets would probably be in a paragraph tag, or have a specific class or other identifying feature. Knowing how to identify where the information on the page is takes a little research before we build the scraper.

At this point, we could build a scraper that would collect all the tweets on a page. This might be useful. Or, we could further filter the scrape, but specifying that we only want to scrape the tweets if it contains certain content. Still looking at our first example, we might be interested in only collecting tweets that mention a certain word or topic, like "Governor." It might be easier to collect a larger group of tweets and parse them later on the back end. Or, we could filter some of the results here beforehand.

Why are web scrapers useful?



We've partially answered this question in the first section. Web scraping could be as simple as identifying content from a large page, or multiple pages of information. However, one of the great things about scraping the web, is that it gives us the ability to not only identify useful and relevant information, but allows us to store that information for later use. In the above example, we might want to store the data we've collected from tweets so that we could see when tweets were the most frequent, what the most common topics were, or what individuals were mentioned the most often.

What prerequisites do we need to build a web scraper?

Before we get into the nuts and bolts of how a web scraper works, let's take a step backward, and talk about where web-scraping fits into the broader ecosystem of web technologies. Take a look at the simple workflow below:

The basic idea of web scraping is that we are taking existing HTML data, using a web scraper to identify the data, and convert it into a useful format. The end stage is to have this data stored as either JSON, or in another useful format. As you can see from the diagram, we could use any technology we'd prefer to build the actual web scraper, such as Python, PHP or even Node, just to name a few. For this example, we'll focus on using Python, and it's accompanying library, Beautiful Soup. It's also important to note here, that in order to build a successful web scraper, we'll need to be at least somewhat familiar with HTML structures, and data formats like JSON.

To make sure that we're all on the same page, we'll cover each of these prerequisites in some detail, since it's important to understand how each technology fits into a web scraping project. The prerequisites we'll talk about next are:

1. **HTML structures**
2. **Python Basics**
3. **Python Libraries**
4. **Storing data as JSON (JavaScript Object Notation)**

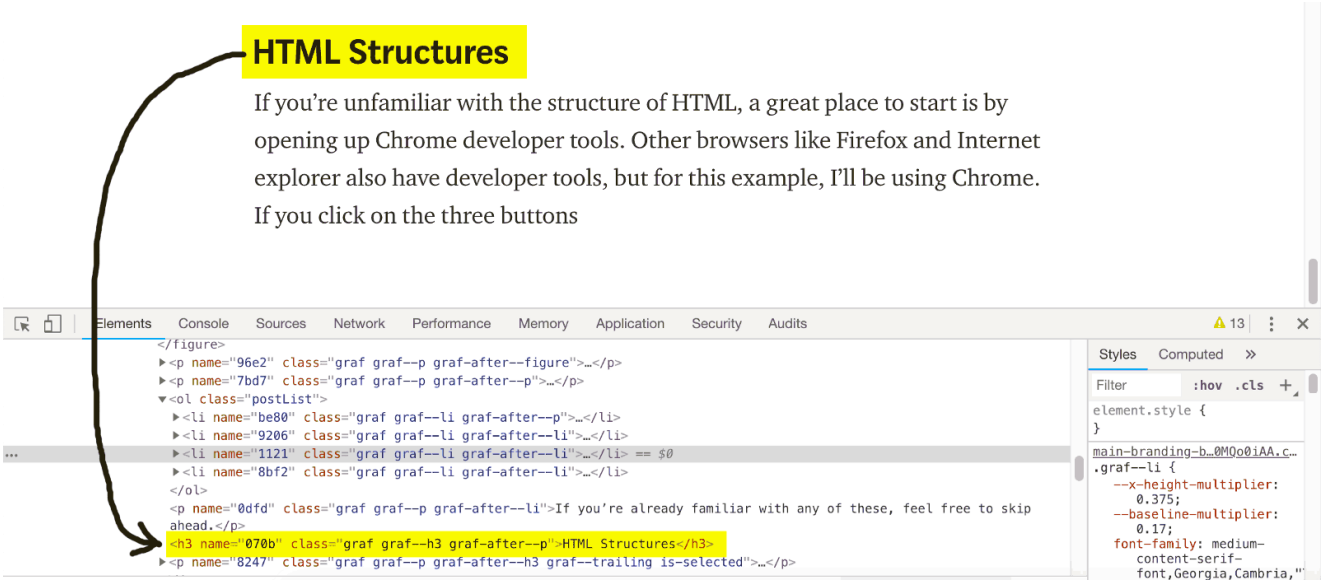
If you're already familiar with any of these, feel free to skip ahead.

1. HTML Structures

1.a. Identifying HTML Tags

If you're unfamiliar with the structure of HTML, a great place to start is by opening up Chrome developer tools. Other browsers like Firefox and Internet explorer also have

developer tools, but for this example, I'll be using Chrome. If you click on the three vertical dots in the upper right corner of the browser, and then 'More Tools' option, and then 'Developer Tools', you will see a panel that pops up which looks like the following:



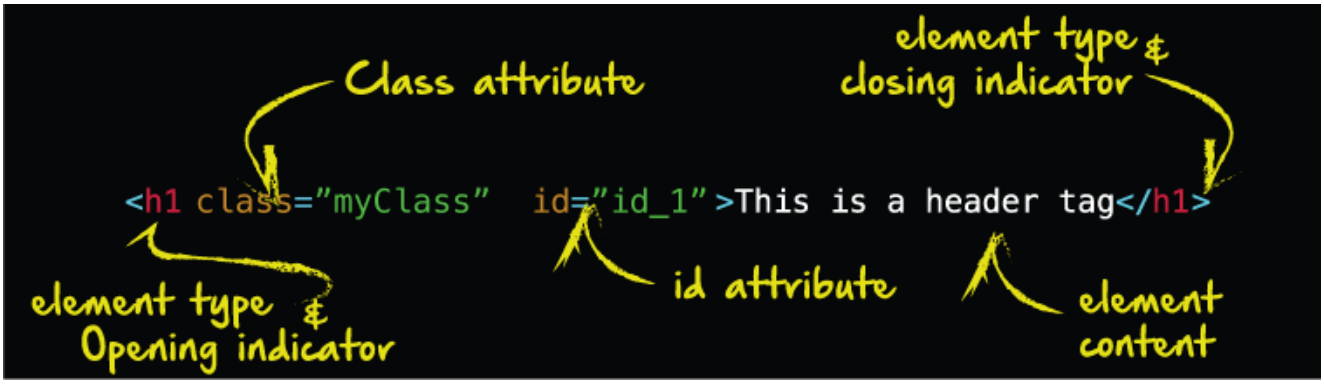
We can quickly see how the current HTML site is structured. All of the content as contained in specific 'tags'. The current heading is in an "<h3>" tag, while most of the paragraphs are in "<p>" tags. Each of the tags also have other attributes like "class" or "name". We don't need to know how to build an HTML site from scratch. In building a web scraper, we only need to know the basic structure of the web, and how to identify specific web elements. Chrome and other browser developer tools allow us to see what tags contain the information we want to scrape, as well as other attributes like "class", that might help us select only specific elements.

Let's look at what a typical HTML structure might look like:

This is similar to what we just looked at in the chrome dev tools. Here, we can see that all the elements in the HTML are contained within the opening and closing 'body' tags. Every element has also has it's own opening and closing tag. Elements that are nested or indented in an HTML structure indicate that the element is a child element of it's container, or parent element. Once we start making our Python web scraper, we can also identify elements that we want to scrape based not only on the tag name, but whether it the element is a child of another element. For example, we can see here that there is a <ul> tag in this structure, indicating an unordered list. Each list element <li> is a child of the parent <ul> tag. If we wanted to select and scrape the entire list, we might want to tell Python to grab all of the child elements of the <ul> tag.

HTML elements

Now, let's take a closer look at HTML elements. Building off of the previous example, here is our <h1> or header element:



Knowing how to specify which elements we want to scrape can be very important. For example, if we told Python we want the <h1> element, that would be fine, unless there are several <h1> elements on the page. If we only want the first <h1> or the last, we might need to be specific in telling Python exactly what we want. Most elements also give us “class” and “id” attributes. If we wanted to select only this <h1> element, we might be able to do so by telling Python, in essence, “Give me the <h1> element that has the class “myClass”. ID selectors are even more specific, so sometimes, if a class attribute returns more elements than we want, selecting with the ID attribute may do the trick.

2. Python Basics

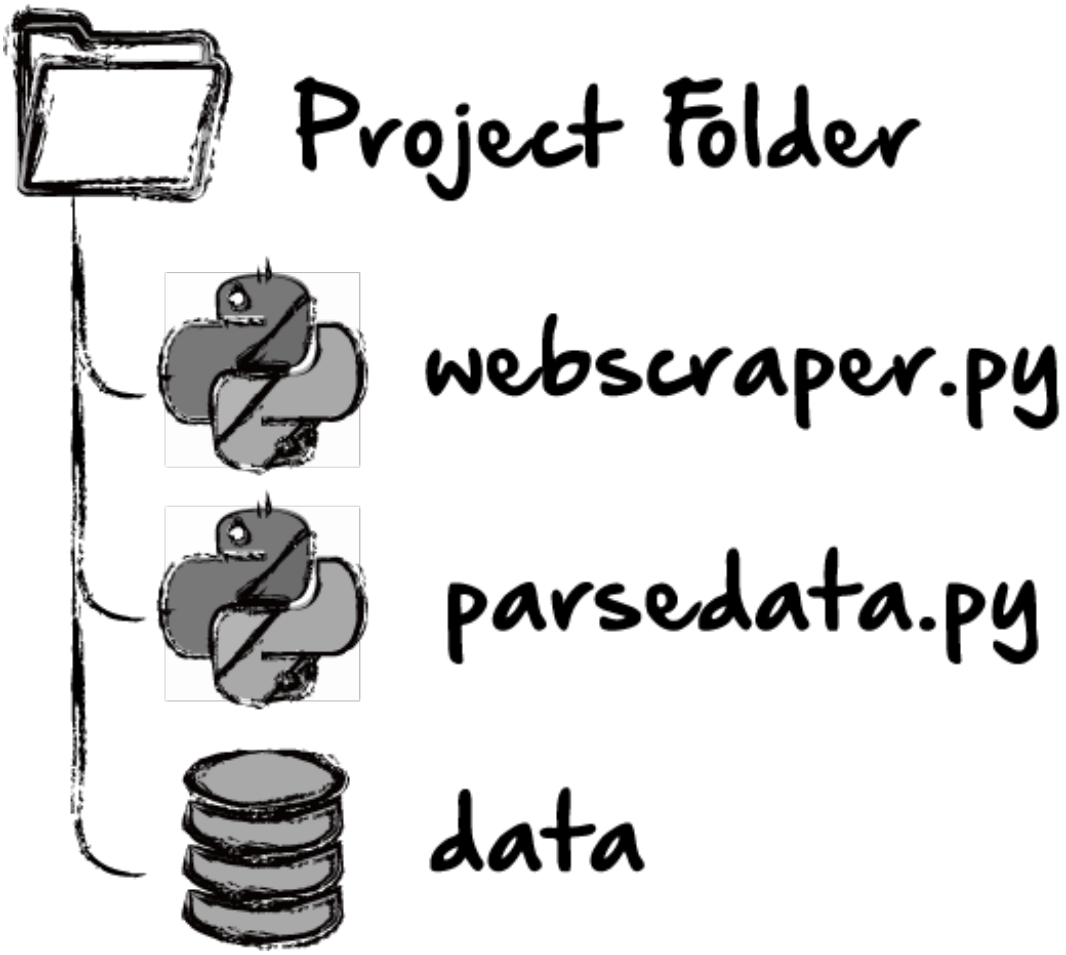
2.a. Setting Up a New Project

One advantage to building a web scraper in Python, is that the syntax of Python is simple and easy to understand. We could be up and running in a matter of minutes with a Python web scraper. If you haven’t already installed Python, go ahead and do that now:

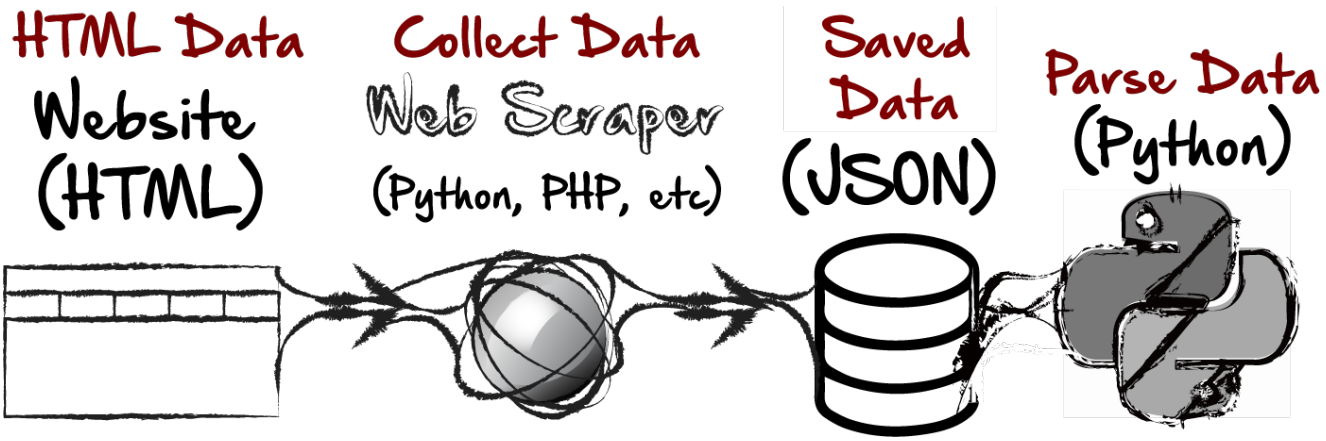
Download Python
The official home of the Python Programming Language www.python.org
We’ll also need to decide on a text editor. I’m using ATOM, but there are plenty of other similar choices, which all do relatively the same thing. Because web scrapers are fairly straight-forward, our choice in which text editor to use is entirely up to us. If you’d like to give ATOM a try, feel free to download it here:

A hackable text editor for the 21st Century
At GitHub, we're building the text editor we've always wanted: hackable to the core, but approachable on the first day... atom.io
Now that we have Python installed, and we’re using a text editor of our choice, let’s create a new Python project folder. First, navigate to wherever we want to create this project. I prefer throwing everything on my already over-cluttered desktop. Then create a new folder, and

inside the folder, create a file. We'll name this file "webscraper.py". We'll also want to make a second file called "parsedata.py" in the same folder. At this point, we should have something similar to this:



One obvious difference is that we don't yet have any data. The data will be what we've retrieved from the web. If we think about what our workflow might be for this project, we might imagine it looking something like this:



First, there's the raw HTML data that's out there on the web. Next, we use a program we create in Python to scrape/collect the data we want. The data is then stored in a format we can use. Finally, we can parse the data to find relevant information. The scraping and the parsing will both be handled by separate Python scripts. The first will collect the data. The second will parse through the data we've collected.

If you're more comfortable setting up this project via the command line, feel free to do that instead.

2.b. Python Virtual Environments

We're not quite done setting up the project yet. In Python, we'll often use libraries as part of our project. Libraries are like packages that contain additional functionality for our project. In our case, we'll use two libraries: Beautiful Soup, and Requests. The Request library allows us to make requests to urls, and access the data on those HTML pages. Beautiful Soup contains some easy ways for us to identify the tags we discussed earlier, straight from our Python script.

If we installed these packages globally on our machines, we could face problems if we continued to develop other applications. For example, one program might use the Requests library, version 1, while a later application might use the Requests library, version 2. This could cause a conflict, making either or both applications difficult to run.

To solve this problem, it's a good idea to set up a virtual environment. These virtual environments are like capsules for the application. This way we could run version 1 of a library in one application and version 2 in another, without conflict, if we created an virtual environment for each application.

First let's bring up the terminal window, as the next few commands are easiest to do from the terminal. On OS X, we'll open the Applications folder, then open the Utilities folder. Open the Terminal application. We may want to add this to our dock as well.

On Windows, we can also find terminal/command line by opening our Start Menu and searching. It's simply an app located at C:\Windows\System32.

Now that we have the terminal open we should navigate to our project folder and use the following command to build the virtual environment:

```
python3 -m venv tutorial-env
```

This step creates the virtual environment, but right now it's just dormant. In order to use the virtual environment, we'll also need to activate it. We can do this by running the following command in our terminal:

On Mac:

```
source tutorial-env/bin/activate
```

Or Windows:

```
tutorial-env\Scripts\activate.bat
```

3 Python Libraries

3.a. Installing Libraries

Now that we have our virtual environment set up and activated, we'll want to install the Libraries we mentioned earlier. To do this, we'll use the terminal again, this time installing the Libraries with the pip installer. Let's run the following commands:

Install Beautiful Soup:

```
pip install bs4
```

Install Requests:

```
pip install requests
```

And we're done. Well, at least we have our environment and Libraries up and running.

3.b. Importing Installed Libraries

First, let's open up our webscraper.py file. Here, we'll set up all of the logic that will actually request the data from the site we want to scrape.

The very first thing that we'll need to do is let Python know that we're actually going to use the Libraries that we just installed. We can do this by importing them into our Python file. It might be a good idea to structure our file so that all of our importing is at the top of the file, and then all of our logic comes afterward. To import both of our libraries, we'll just include the following lines at the top of our file:

```
from bs4 import BeautifulSoup
import requests
```

If we wanted to install other libraries to this project, we could do so through the pip installer, and then import them into the top of our file. One thing to be aware of is that some libraries are quite large, and can take up a lot of space. It may be difficult to to deploy a site we've worked on if it is bloated with too many large packages.

3.c. Python's Requests Library

Requests with Python and Beautiful Soup will basically have three parts:

URL : 'www.twitter.com'

RESPONSE : get(URL)

CONTENT : BeautifulSoup(response.content, "html.parser")

The URL, RESPONSE & CONTENT.

The URL is simply a string that contains the address of the HTML page we intend to scrape.

The RESPONSE is the result of a GET request. We'll actually use the URL variable in the GET request here. If we look at what the response is, it's actually an HTTP status code. If the request was successful, we'll get a successful status code like 200. If there was a problem with the request, or the server doesn't respond to the request we made, the status code could be unsuccessful. If we don't get what we want, we can look up the status code to troubleshoot what the error might be. Here's a helpful resource in finding out what the codes mean, in case we do need to troubleshoot them:

www.restapitutorial.com
Finally, the CONTENT is the content of the response. If we print the entire response content, we'll get all the content on the entire page of the url we've requested.

4. Storing Data as JSON

> If you don't want to spend the time scraping, and want to jump straight to manipulating data, here are several of the datasets I used for this exercise:

https://www.dropbox.com/s/v6vjffuakehjpic/stopwords.json?dl=0
https://www.dropbox.com/s/2wqibsa5fro6gpx/tweetsjson.json?dl=0
https://www.dropbox.com/s/1zwynoyjg15l4gv/twitterData.json?dl=0

4.a. Viewing Scraped Data

Now that we know more or less how our scraper will be set up, it's time to find a site that we can actually scrape. Previously, we looked at some examples of what a twitter scraper might look like, and some of the use cases of such a scraper. However we probably won't actually scraper Twitter here for a couple of reasons. First, whenever we're dealing with dynamically generated content, which would be the case on Twitter, it's a little harder to scrape, meaning

that the content isn't readily visible. In order to do this, we would need to use something like Selenium, which we won't get into here. Secondly, Twitter provides several API's which would probably be more useful in these cases.

Instead, here's a "Fake Twitter" site that's been set up for just this exercise.

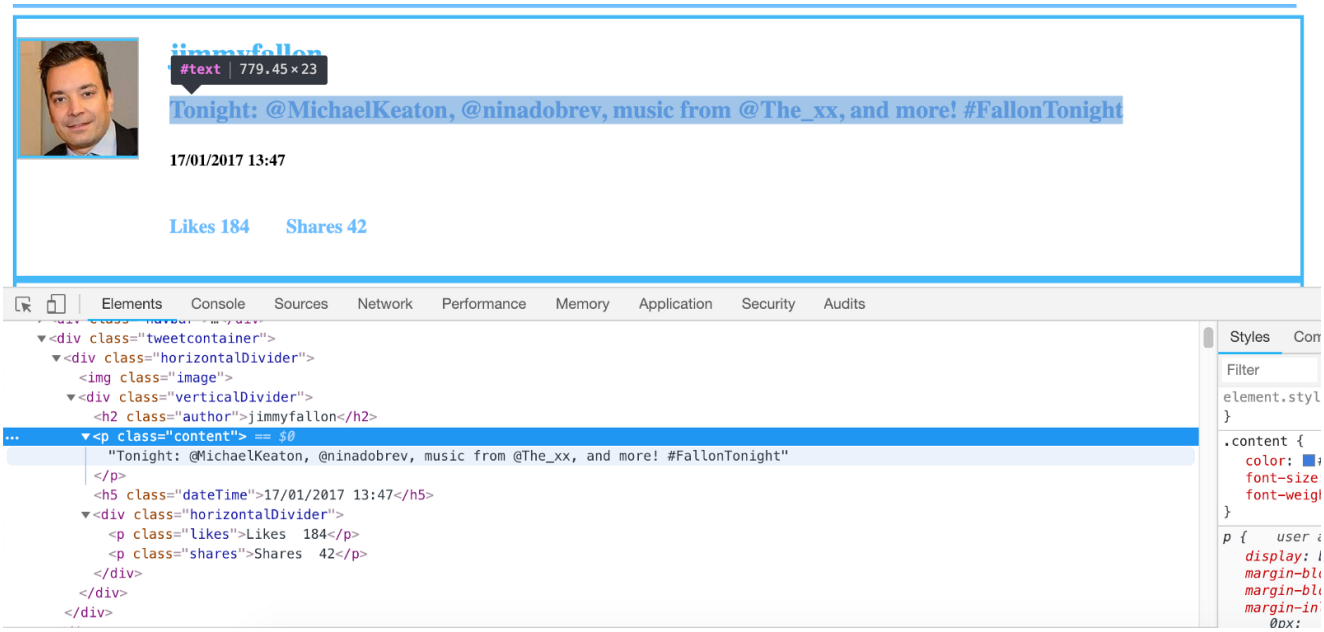
http://ethans_fake_twitter_site.surge.sh/

On the above "Fake Twitter" site, we can see a selection of actual tweets by Jimmy Fallon between 2013 and 2017. If we follow the above link, we should see something like this:

Here, if we wanted to scrape all of the Tweets, there are several things associated with each Tweet that we could also scrape:

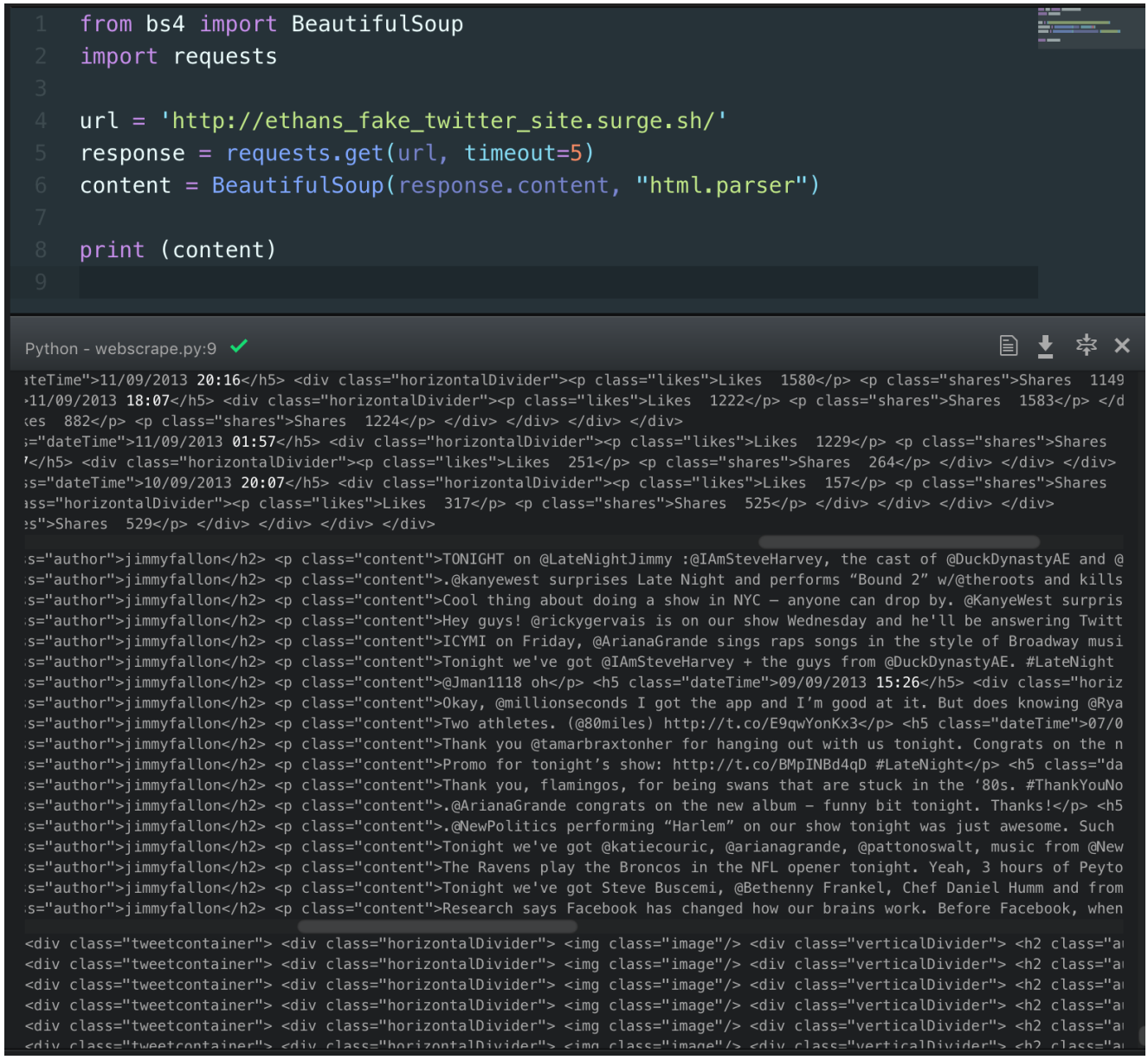
1. The Tweet
2. The Author (JimmyFallon)
3. The Date and Time
4. The Number of Likes
5. The Number of Shares

The first question to ask before we start scraping is what we want to accomplish. For example, if all we wanted to do was know when most of the tweets occured, the only data we actually need to scrape would be the date. Just for ease however, we'll go ahead and scrape the entire Tweet. Let's open up the Developer Tools again in Chrome to take a look at how this is structured, and see if there are any selectors that would be useful in gathering this data:



Under the hood, it looks like each element here is in it's own class. The author is in an <h2> tag with the class named "author". The Tweet is in a <p> tag with a class named "content". Likes and Shares are also in <p> tags with classes named "likes" and "shares". Finally, our Date/Time is in an <h5> tag with a class "dateTime".

If we use the same format we used above to scrape this site, and print the results, we will probably see something that looks similar to this:

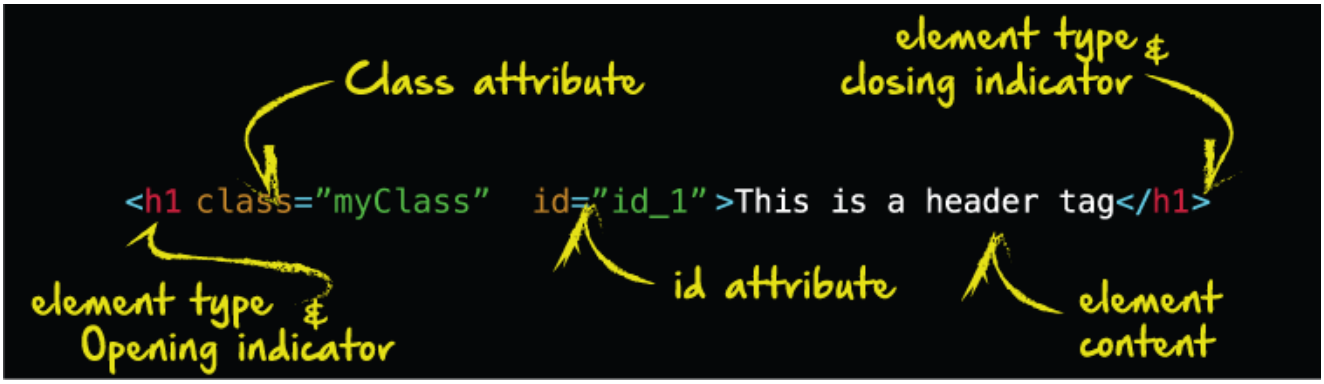


What we've done here, is simply followed the steps outlined earlier. We've started by importing bs4 and requests, and then set URL, RESPONSE and CONTENT as variables, and printed the content variable. Now, the data we've printed here isn't very useful. We've simply printed the entire, raw HTML structure. What we would prefer is to get the scraped data into a useable format.

4.b Selectors in Beautiful Soup

In order to get a tweet, we'll need to use the selectors that beautiful soup provides. Let's try this:

Instead of printing the entire content, we'll try to get the tweets. Let's take another look at our example html from earlier, and see how it relates to the above code snippet:



The previous code snippet is using the class attribute “content” as a selector. Basically the `'p', attrs={"class": "content"}` is saying, “we want to select the all of the paragraph tags , but only the ones which have the class named “content”.

Now, if we stopped there, and printed the results, we would get the entire tag, the ID, the class and the content. The result would look like:

<p class="content> Tonight: @MichaelKeaton, @ninadobrev, music from @The_xx, and more! #FallonTonight

But, all we really want is the content, or text of the tag:

Tonight: @MichaelKeaton, @ninadobrev, music from @The_xx, and more! #FallonTonight

So the `.text` tells Python that if we find a tag, with the class “content”, we’ll only select the text content of this tag.

When we run this line of code though, we only get the very first tweet, and not all the tweets. It seems a little counter-intuitive, since we did use the `findAll()` method. In order to get all the tweets, and not just the first one, we’ll need to loop over the content and select it in a loop, like this:

Now, when we loop over the content, we’ll be able to view all of the tweets. Awesome!

4.c Converting Scraped Data to JSON

The next step in this process, before we actually store the data, is to convert it to JSON. JSON stands for JavaScript Object Notation. In Python the terminology is Dicts. In either case, this data will be in the form of key/value pairs. In our case, this data might look like the following:

Each Tweet would have this format, and could be stored in an array. This would allow us to more easily parse the data later. We could quickly ask Python for all of the dates, or all the likes, or count the number of times “show” is used in all “tweets”. Storing the data in a usable way like this will be a key to doing something fun with the data later. If we scroll back up and look again at the HTML structure, we might notice that each tweet is in a

element with the class name “tweetcontainer”. Each author, tweet, date, etc, will be inside of one of these containers. We previously looped over all the data and selected the tweets from each element. Why don’t we do the same thing, but instead loop over each container, so we can select the individual date, author & tweet from each one. Our code might look something like this:

```
for tweet in content.findAll('div', attrs={"class": "tweetcontainer"}):
    tweetObject = {
        "author": "JimmyFallon",
        "date": "02/28/2018",
        "tweet": "Don't miss tonight's show!",
        "likes": "250",
        "shares": "1000"
    }
```

However, instead of this data, we want to select the individual data from each tweet to build our object. The end result would instead be:

```
from bs4 import BeautifulSoup
import requests

url = 'http://ethans_fake_twitter_site.surge.sh/'
response = requests.get(url, timeout=5)
content = BeautifulSoup(response.content, "html.parser")

tweetArr = []
for tweet in content.findAll('div', attrs={"class": "tweetcontainer"}):
    tweetObject = {
        "author": tweet.find('h2', attrs={"class": "author"}).text.encode('utf-8'),
        "date": tweet.find('h5', attrs={"class": "dateTime"}).text.encode('utf-8'),
        "tweet": tweet.find('p', attrs={"class": "content"}).text.encode('utf-8'),
        "likes": tweet.find('p', attrs={"class": "likes"}).text.encode('utf-8'),
        "shares": tweet.find('p', attrs={"class": "shares"}).text.encode('utf-8')
    }
    print tweetObject
```

Awesome! All of our data is in a nice, easy to use format. Although, up to this point, all we’ve done is printed the results. Let’s add one final step, and save the data as a JSON file.

4.d Saving the Data

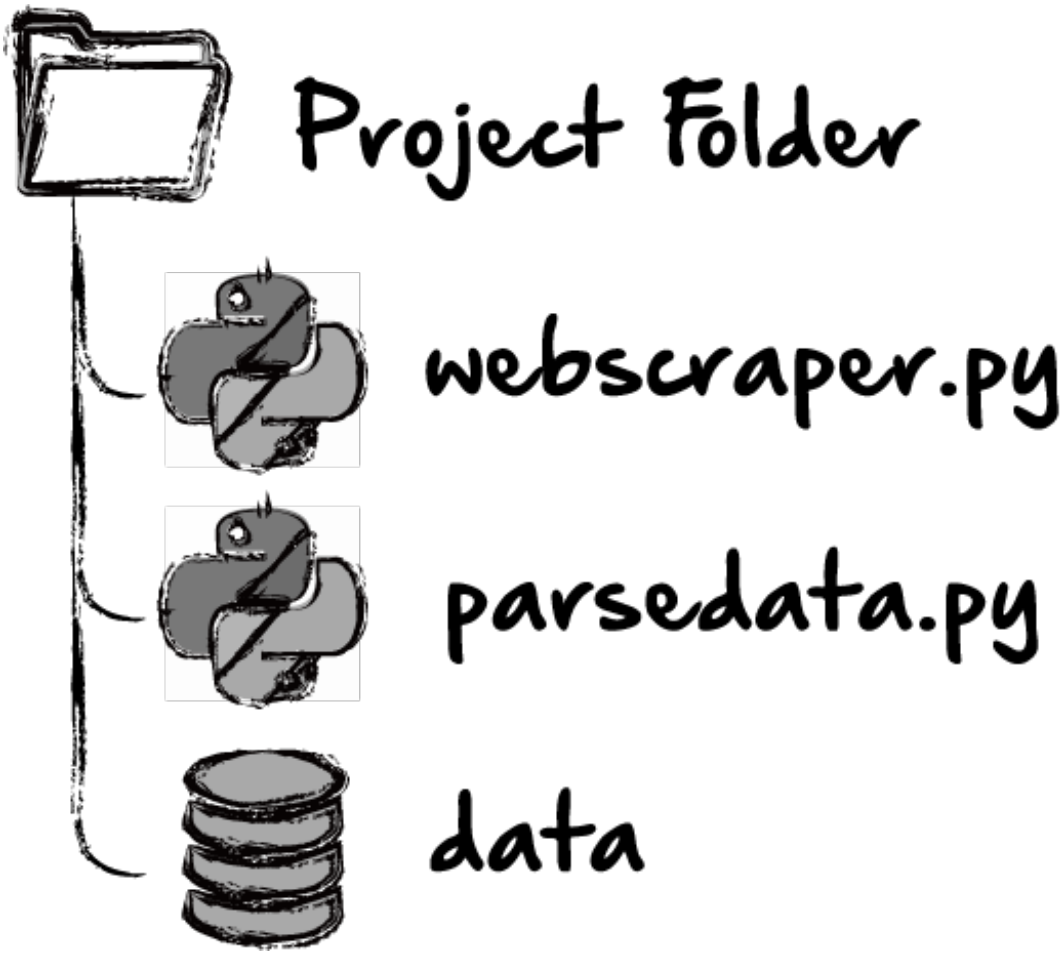
In order to do this, will add one more import to our code at the top and import json. This is a core library, so we don’t need to install it via pip like we did the other packages. Then, after looping through our data, and building the tweetobject from each element, we’ll append that object, or dict to our tweetArr, which will be an array of tweets. Finally, we’ll take advantage of the json library and write a json file, using our tweet array as the data to write. The final code might look like this:

```
from bs4 import BeautifulSoup
import requests
import json

url = 'http://ethans_fake_twitter_site.surge.sh/'
response = requests.get(url, timeout=5)
content = BeautifulSoup(response.content, "html.parser")

tweetArr = []
for tweet in content.findAll('div', attrs={"class": "tweetcontainer"}):
    tweetObject = {
        "author": tweet.find('h2', attrs={"class": "author"}).text.encode('utf-8'),
        "date": tweet.find('h5', attrs={"class": "dateTime"}).text.encode('utf-8'),
        "tweet": tweet.find('p', attrs={"class": "content"}).text.encode('utf-8'),
        "likes": tweet.find('p', attrs={"class": "likes"}).text.encode('utf-8'),
        "shares": tweet.find('p', attrs={"class": "shares"}).text.encode('utf-8')
    }
    tweetArr.append(tweetObject)
with open('twitterData.json', 'w') as outfile:
    json.dump(tweetArr, outfile)
```

When running this, Python should have generated and written a new file called twitterData. Now let's try to parse that data!

5. Parsing JSON Data

Let's go back to our file tree and open up our parsing file (parsedata.py), which should be blank.



Just like we called json, and opened a json file in the previous step, we'll do the same thing in this step. However, now instead of writing to a json file, we'll want to read from the json file we just created.

```
import json

with open('twitterData.json') as json_data:
    jsonData = json.load(json_data)
```

Now, we can use the variable jsonData. This should contain all of the information we scraped, but in JSON format. Let's start with something simple, printing all of the dates of all the tweets:

```
for i in jsonData:
    print i['date']
```

Running this command, we should see a generated list of all the dates of all the tweets.

Another fun thing to do would be to see how often certain words appear in the tweets. For example, we could run a query that would see how often "Obama" appears in the tweets:

Doing this will show us the entire tweet object for each tweet where “obama” is mentioned. Pretty cool right? Obviously, the possibilities are endless. It should be clear now, how easy it is to efficiently scrape data from the web, and then convert it to a usable format to be parsed through. Fantastic!

If you have any feedback or questions, feel free to reach out!