**Introduction to Web Scraping**

Sean Ericson

sean.ericson@colorado.edu

11/20/2019

Overview and setup:

All code can be found at https://github.com/sericson0/Introduction\_To\_Scraping

This document along with the associated code and presentation goes over several tools for collecting data from the internet which can be useful for economists. This will primarily be learning by example, and I will walk through several scripts which tackle various methods of data downloading and web scraping.

Throughout the goal of each code will be to extract relevant data in a format conducive to economic analysis (e.g., csv files which can be loaded into statistics software). I will be using both **R** and **Python**

-Downloading R

R can be downloaded from <https://www.r-project.org/>

Once you have downloaded R, I highly recommend downloading RStudio as well, which can be downloaded from <https://rstudio.com/products/rstudio/download/>

-Downloading and setting up Python

Python is available for download from <https://www.python.org/downloads/>

Make sure you get python 3.X instead of Python 2.7 as the code will not work with Python version 2

I personally find it very useful to run Python from a text editor. My editor of choice is SublimeText <https://www.sublimetext.com/>

If you use SublimeText to run Python then you will want to follow these steps to load the Anaconda Package <http://damnwidget.github.io/anaconda/> Feel free to contact me if you have any issues

Python code is then run with the command ctrl+B

You will need to install a couple of packages as well. With windows this is done through the command prompt with the command:

python -m pip install <package name>

The packages to install are

-pandas

-requests

-numpy

-selenium

Example 1: Bulk Data Downloading.

This section uses the code Download\_Emissions\_Data.R

Say we are interested in looking at the CO2 emissions of power plants over time.

The Energy Information Administration has this data for several years at <https://www.eia.gov/electricity/data/emissions/>

We see that there are 6 files for emissions from 2013-2018. If we right click on some of the links, select **Copy Link Location** and then past the link we get

<https://www.eia.gov/electricity/data/emissions/archive/xls/emissions2013.xlsx>

https://www.eia.gov/electricity/data/emissions/archive/xls/emissions2014.xlsx

https://www.eia.gov/electricity/data/emissions/xls/emissions2018.xlsx

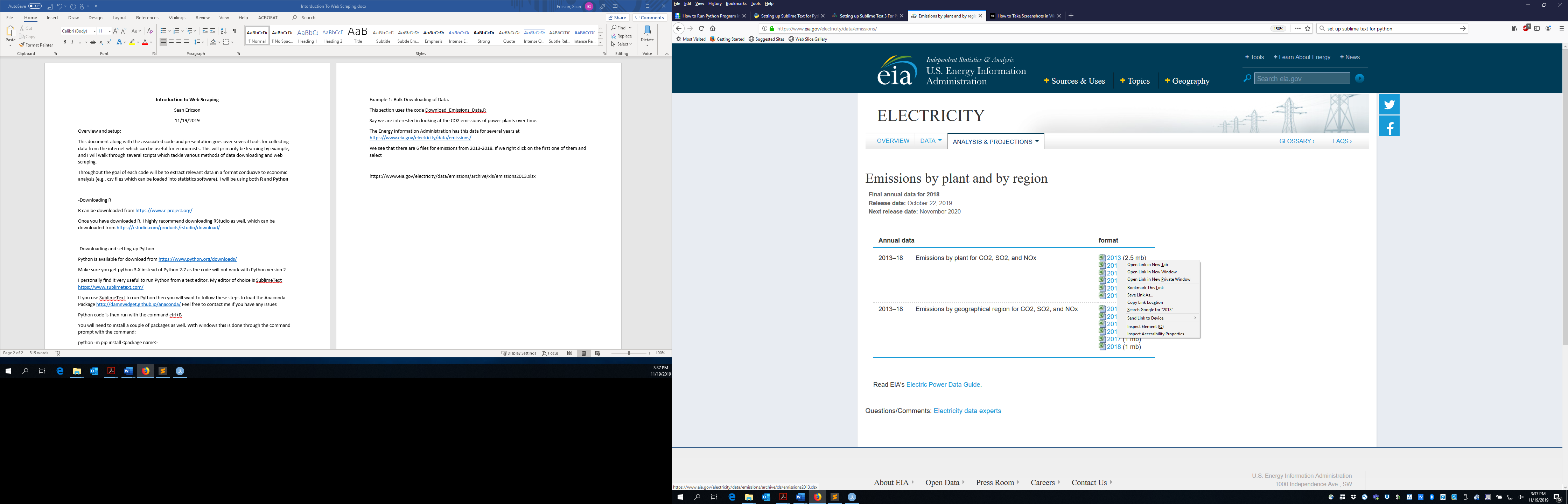


Figure Right click to copy Link

We see that they all have a similar format, only varying by year

(and the last year does not have /archive)

In R we can download the files of an internet link with **download.file()**

As is seen in the R code, we use download.file() along with the regularity of the link names to download all of the .xls files and then combine them into a single file.

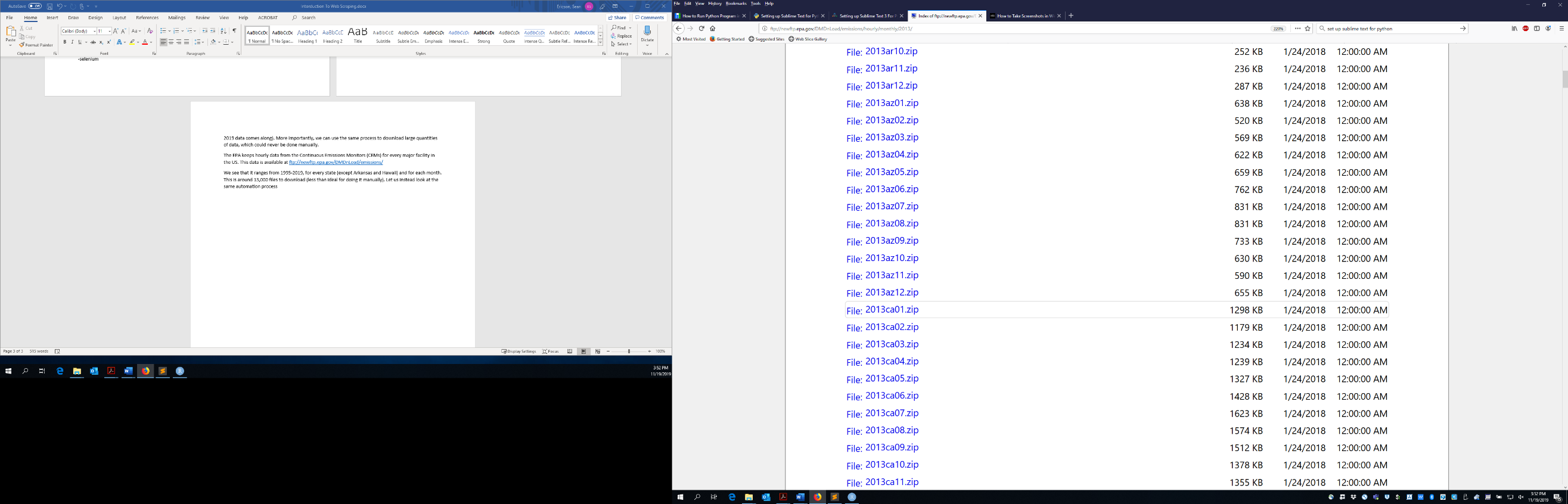
Since there were only 6 files to download, we could have simply done it manually. However, automating the process helps with reproducibility and allows for easy integration of additional data (say when the 2019 data comes along). More importantly, we can use the same process to download large quantities of data, which could never be done manually.

Figure EPA CEMS file names

The EPA keeps hourly data from the Continuous Emissions Monitors (CEMs) for every major facility in the US. This data is available at <ftp://newftp.epa.gov/DMDnLoad/emissions/>

We see that it ranges from 1995-2019, for every state (except Arkansas and Hawaii) and for each month. This is around 13,000 files to download; less than ideal for doing it manually. Let us instead look at the same automation process. We see that the format is

<stuff>/year/<year><state.abb><month>.zip

ftp://newftp.epa.gov/DMDnLoad/emissions/hourly/monthly/2013/2013az04.zip

\*\*\*Remember to check the last year as there are often endcase differences\*\*\*

Because the files are zip files, we use the function unzip() to extract the csv files.

We also make use of a temporary file with tempfile() so we do not have to save the zipped files.

While the code would take a couple of hours to complete (there is A LOT OF DATA) You can simply let it start, crack open a beer, admire your brilliance, and then in a couple of hours the data set would be yours.

And here is an example of how people approach this sort of problem with Stata <https://www.youtube.com/watch?v=H2uHBhKTSe0>

Example 2: Basic Web Scraping

We will now move over to Python and begin with some simple web scraping. The simplest, but often most powerful form of web scraping is collecting data from an Applications Program Interface (API). In a nutshell, an API is when a website places its data in a format conducive to collecting.

\*\*\*If you are looking to scrape data, always look to see if the website has an API\*\*\*

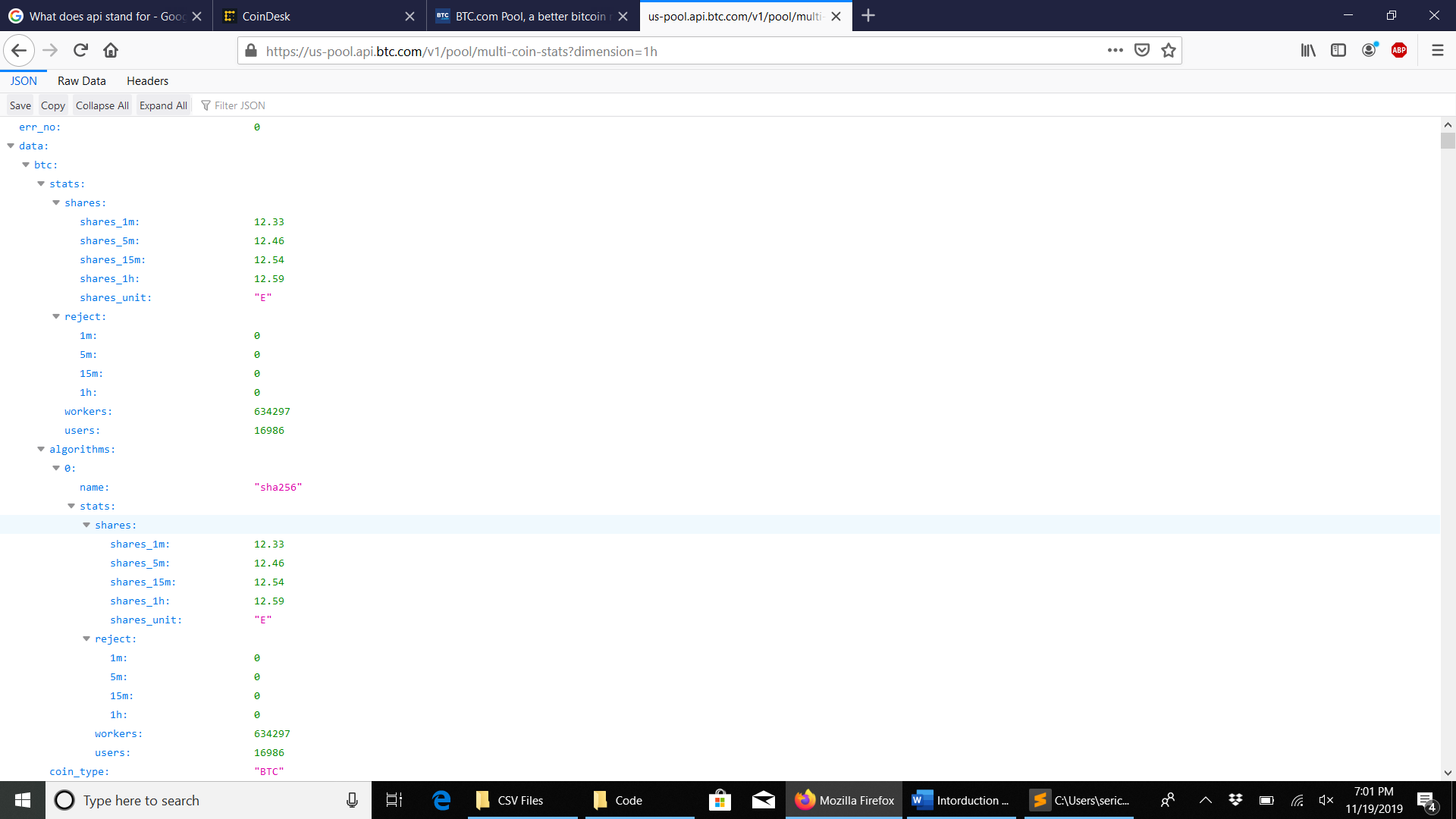
Say we want to scrape some bitcoin mining data to save for later. Specifically, we want to get mining pool data from BTC.com <https://pool.btc.com/> and we want to get price data from <https://www.coindesk.com/> After looking a bit around the websites the bottom we see that they both have an API. Horray!!! API queries use various url’s to return various types of data.

After playing around a bit we find that

<https://us-pool.api.btc.com/v1/pool/multi-coin-stats?dimension=1h>

returns useful data from BTC and

<https://api.coindesk.com/v1/bpi/currentprice.json>

returns the current price from Coindesk.

Now we know where the data we want sits, but how do we get to it? This is where the Python module requests comes in!

https://realpython.com/python-requests/

requests gets the html data from a webpage, which is very useful. Because the API returns a json our job is even easier.

Typing requests.get(api\_url).json() returns a list of all the relevant data we want.

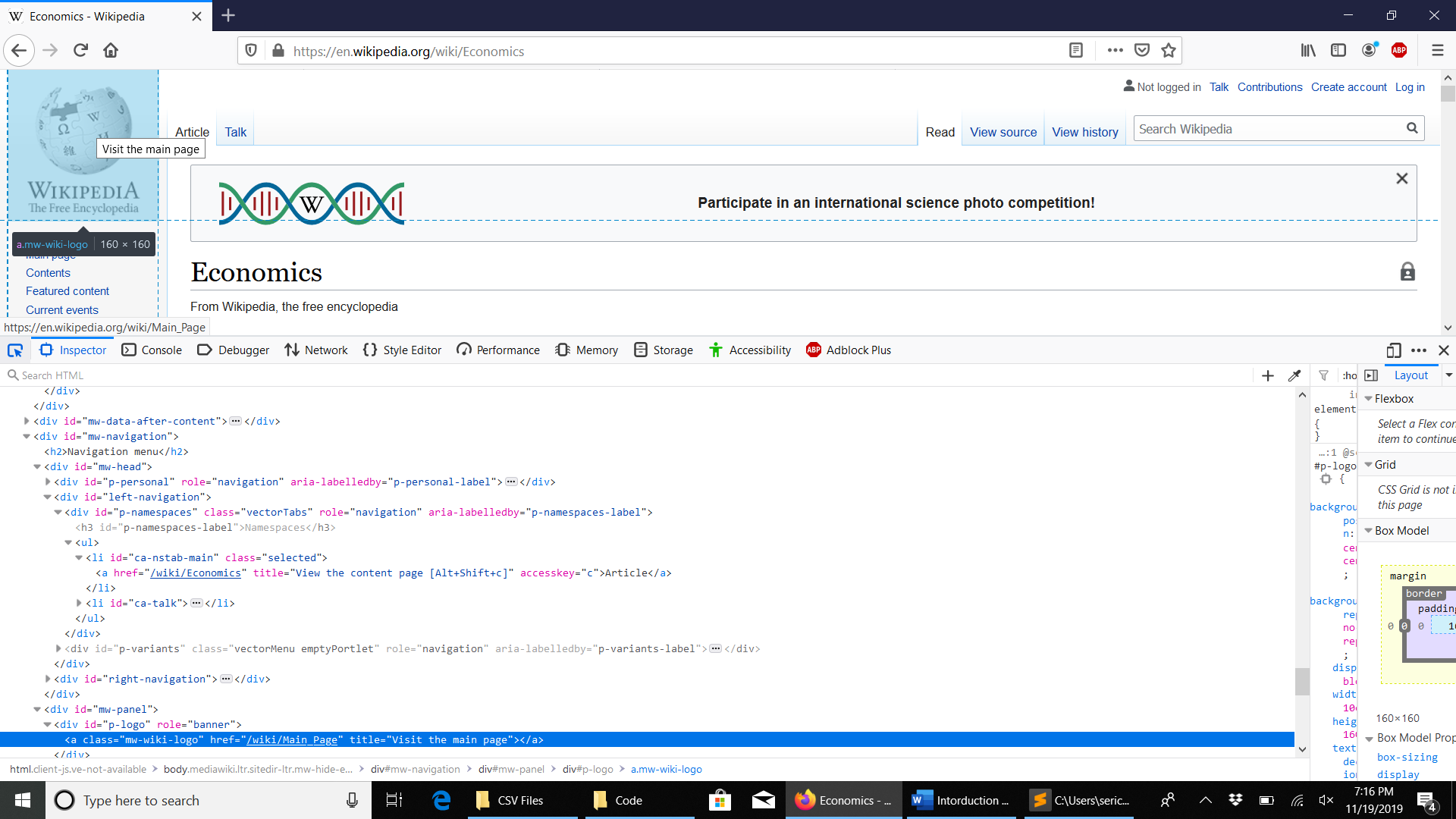
Requests can be used for downloading generic html data. The hardest part is then combing through the html afterwards.

I suggest you look at the Python package bs4 (Beautiful Soup) for returning information from html.

Figure BTC.com API output

Example 3: Advanced Web Scraping

Lets take a moment to see the structure of a web page. When we right-click we can open “inspect element”, this looks similar to the screenshot below



The html has a nested structure, with each element identified by some combination of id, class names, titles, and class type.

Websites are also often dynamic. This means that there are buttons to click, places to type in keys, and JavaScript to be run. All of this is taken care of behind the scenes through your web browser. At the same time, dynamic data cannot be scraped by using requests, and requests cannot navigate web pages. To do this we use the package **selenium.**

With selenium we use a web browser with automated actions, so anything that you see or do on the internet we can get Python to do for us. To use selenium, we need to have a webdriver next to wherever the python code is saved at. I use a chrome webdriver, but all can be accessed here <https://pypi.org/project/selenium/>

Selenium creates a web browser just like when you open Chrome or Firefox. This means that any action you can do on your browser you can do with selenium. A nice cheat sheet for relevant commands can be found here <http://allselenium.info/python-selenium-commands-cheat-sheet-frequently-used/>

Lets see how this works with scraping some reviews from the website Glassdoor.

We create a browser with:

**browser = wd.Chrome()**

Glassdoor requires us to sign, which we can do by going to the login page with

browser.get(“https://www.glassdoor.com/profile/login\_input.htm”)

We then find the input elements for the username and password, which we can do with

**userename\_field = browser.find\_element\_by\_name('username')**

**password\_field = browser.find\_element\_by\_name('password')**

we use the .send\_keys command to input our username and password:

**username\_field.send\_keys(“sean.ericson@colorado.edu”)**

**password\_field.send\_keys(“password123”)**

Finally, we click the button to sign in with click()

**browser.find\_element\_by\_xpath('//button[@type="submit"]').click()**

A list of really useful functions to find elements (all are of the form <element>.<function> ) are:

find\_element\_by\_name()

find\_element\_by\_xpath()

find\_element\_by\_id()

find\_element\_by\_class\_name()

find\_element\_by\_ccs\_selector()

All of these are various ways of selecting elements. If you use the plural (find\_element**s**) then it returns a list of all of the elements that match

We can then manipulate elements with:

click() #Clicks a button

send\_keys() #writes text to input

clear() #clears input box

is\_selected() #returns True/False

Finally, we can output element data with

.text

get\_attribute(“attribute”) #particularly useful is get\_attribute(“innerHTML”)

Using these, we are able to navigate through the review pages and download information we may find valuable.

Summary

Web scraping is not an exact science, and every project is unique. It can be easy to get carried away as well. Remember that the economists are judged not by what data we collect but by what we do with it. At the same time, you now have some powerful tools to collect whatever data you need. Between being able to bulk download, scrape html and use selenium to navigate a web browser, there is little if any data which is outside of your reach. Happy scraping!

Please let me know if you have any questions or issues. Feel free to email me at sean.ericson@colorado.edu