V.PREETHI

[Preethi.v2017@vitstudent.ac.in](mailto:Preethi.v2017@vitstudent.ac.in)

6380073580

Web Scraping Job Postings from Indeed

So you’re in the job market, and you want to work smarter rather than harder at finding new and interesting jobs for yourself? Why not build a web scraper to collect and parse out job posting information for you? Set it, forget it, and come back to your data riches in a nice tidy format! How, you ask? Let’s take a look together!

[A disclaimer before beginning, many websites can restrict or outright bar scraping of data from their pages. Users may be subject to legal ramifications depending on where and how they attempt to scrape information. Many sites have a devoted page to noting restrictions on data scraping at www.[site].com/robots.txt. Be extremely careful if looking at sites that house user data — places like facebook, linkedin, even craigslist, do not take kindly to data being scraped from their pages. Scrape carefully, friends]

For this project, I wanted to explore data science-related jobs posted to a variety of cities on indeed.com, a job aggregator that updates multiple times daily. I conducted my scraping using the “requests” and “BeautifulSoup” libraries in python to gather and parse information from indeed’s pages, before using the “pandas” library to assemble my data into a dataframe for further cleaning and analysis.

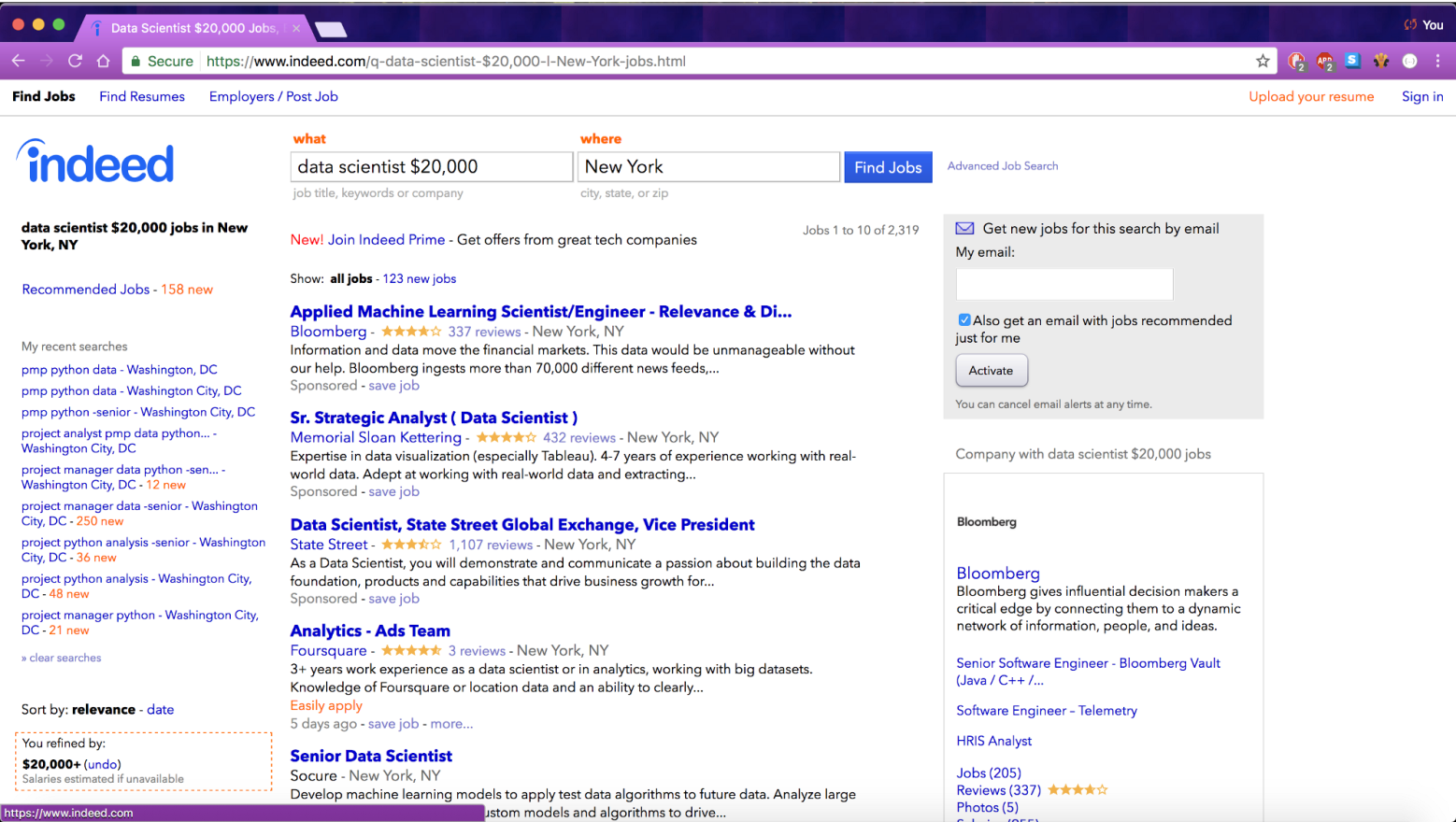
**Examining the URL and Page structure**

First, let’s look at a sample page from [indeed](https://www.indeed.com/jobs?q=data+scientist+%2420%2C000&l=New+York&start=10).

Notice a few things about the way the URL is structured:

* note “q=” begins the string for the “what” field on the page, separating search terms with “+” (i.e. searching for “data+scientist” jobs)
* when specifying salary, it will parse by commas in the salary figure, so the beginning of the salary will be preceded by %24 and then the number before the first comma, it will then be broken by %2C and continue with the rest of the number (i.e. %2420%2C000 = $20,000)
* note “&l=” begins the string for city of interest, separating search terms with “+” if city is more than one word (i.e. “New+York”
* note “&start=” notes the search result where you want to begin (i.e., start by looking at the 10th result)

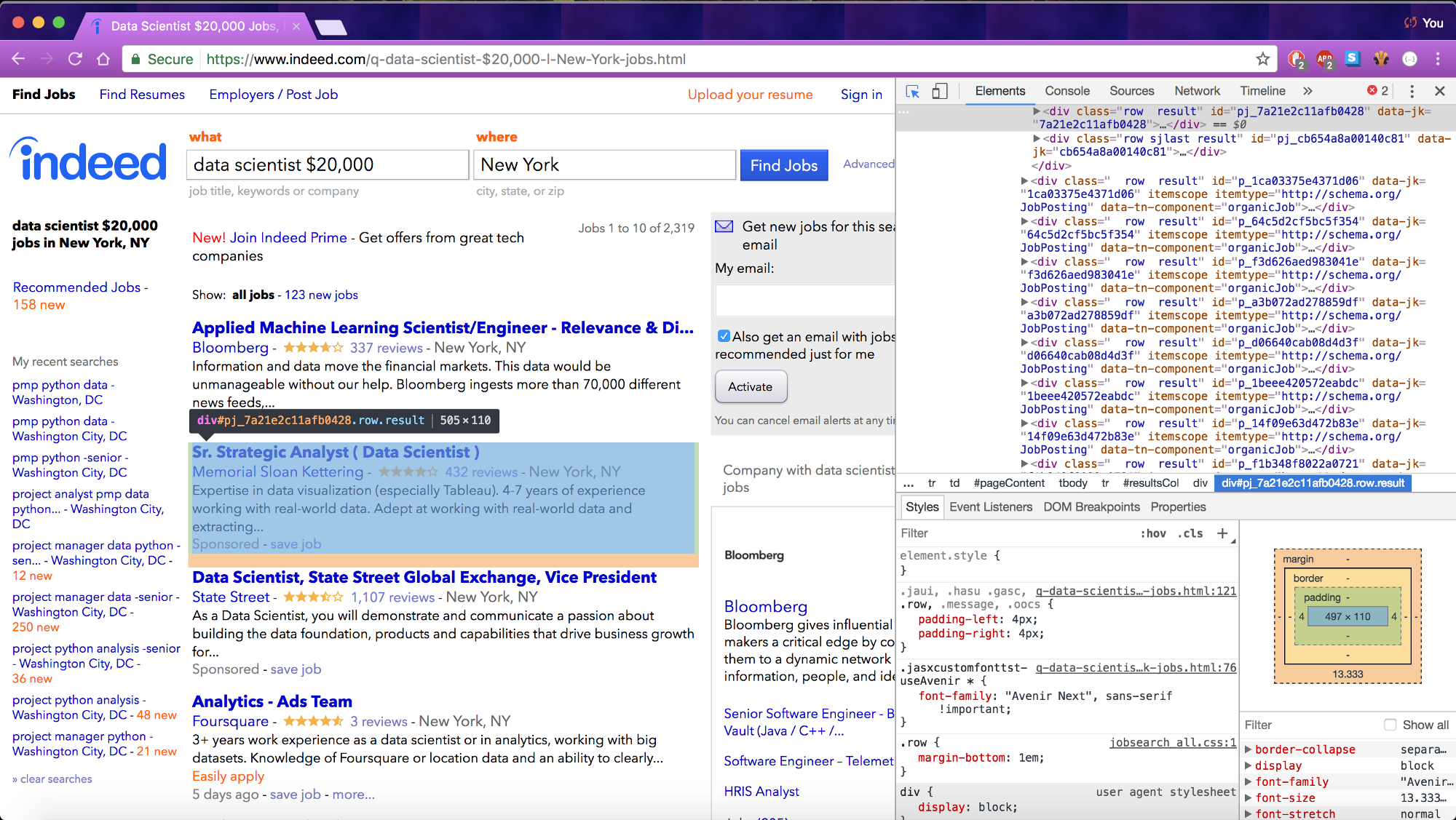
The URL structure will come in handy as we build a scraper to look at and gather data from a series of pages. Keep this in mind for later.



Each page of job results will have 15 job posts. Five of these are “sponsored” jobs, which are specially displayed by indeed outside of the normal order of results. The remaining 10 results are specific to the page being viewed.

All of the information on this page is coded with HTML tags. HTML (HyperText Markup Language), is the coding that tells your internet browser how to display a given page’s contents upon accessing it. This includes its basic structure and order. HTML tags also have attributes that are a helpful way of keeping track of what information can be found where within the structure of the page.

Chrome users can examine the HTML structure of a page by right-clicking on a page and choosing “Inspect” from the menu that appears. A menu will appear on the right-hand side of your page, with a long list of nested HTML tags housing the information currently displayed in your browser window. In the upper-left of this menu, there’s a small box with an arrow icon in it. Once clicked, the box will illuminate in blue (notice in the screenshot below). This will allow you to cursor over the elements in the page to display both the tag associated with that item, and to bring your inspection window directly to that item’s place in the HTML for the page.



In the screen show above, I’ve cursored over one of the job postings to show how the entire job’s contents is held within a <div> tag, with attributes including “class = ‘row result’”, “id=’pj\_7a21e2c11afb0428'”, etc,. Luckily, we will not need to know every attribute of every tag to extract our information, but it is helpful to know how to read the basic structure of a page’s HTML.

Now, let’s turn to python to extract the html from the page and look to building our scraper.

**Building the Scraper Components**

Now that we’ve looked at the basic structure of the page and know a little about it’s basic HTML structure, we can see about building code to pull out the information we’re interested in. We’ll import our libraries first. Note, I’m also importing “time”, which can be a helpful way of staggering page requests to not overwhelm a site’s servers when scraping information.

import requests  
import bs4  
from bs4 import BeautifulSoup

import pandas as pd  
import time

Let’s start by pulling a single page, and working out the code to withdraw each piece of information we want:

URL = “https://www.indeed.com/jobs?q=data+scientist+%2420%2C000&l=New+York&start=10"

#conducting a request of the stated URL above:  
page = requests.get(URL)

#specifying a desired format of “page” using the html parser - this allows python to read the various components of the page, rather than treating it as one long string.  
soup = BeautifulSoup(page.text, “html.parser”)

#printing soup in a more structured tree format that makes for easier reading  
print(soup.prettify())

Using prettify makes it much easier to look through a page’s HTML coding, and will provide an output like this:



Now, we know that our variable “soup” has all of the information housed in our page of interest. It is now a matter of writing code to iterate through the various tags (and nested tags therein) to capture the information we want.

While this is not the appropriate place to go over all of the ways in which information can be found or withdrawn from a page’s HTML, the [BeautifulSoup documentation](https://www.crummy.com/software/BeautifulSoup/bs4/doc/" \t "_blank) has a lot of helpful information that can guide one’s searching.

**Withdrawing Basic Elements of Data**

Approaching this task, I wanted to find and extract five key pieces of information from each job posting: Job Title, Company Name, Location, Salary, and Job Summary.

I know from looking at the page, that there must be 15 job postings therein. As such, I know that each function I write to withdraw a piece of information should yield 15 different items. If my output provides fewer than this, I can refer back to the page itself to see what information I’m not capturing.

**Job Title**

As noted above, I could tell that the entirety of each job posting is nested under <div> tags, with an attribute “class” = “row result”.

From there, I could see that job titles are listed under <a> tags, with attribute “title = (title)”. One can find the value of a tag’s attribute with tag[“attribute”], so I could use this to find the job title for each posting.

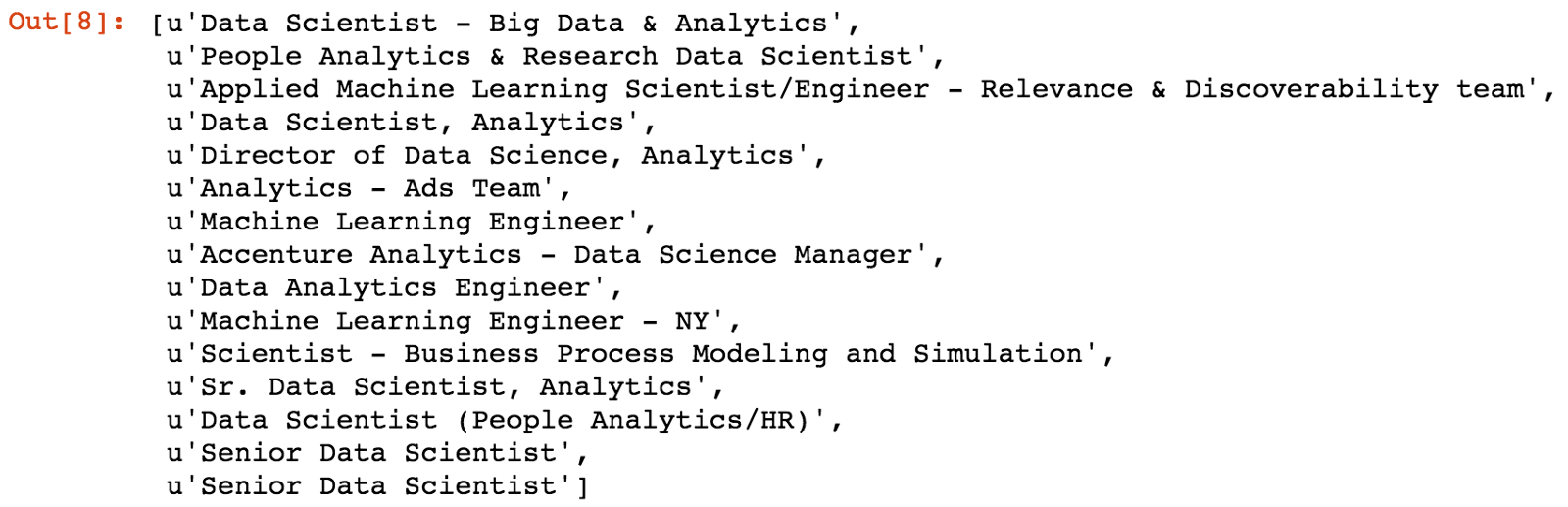
My function for withdrawing job title information involved three steps:

* pulling out all <div> tags with class including “row”
* identifying <a> tags with attribute “data-tn-element”:”jobTitle”
* for each of these <a> tags, find the value of attributes “title”

def extract\_job\_title\_from\_result(soup):   
 jobs = []  
 for div in soup.find\_all(name=”div”, attrs={“class”:”row”}):  
 for a in div.find\_all(name=”a”, attrs={“data-tn-element”:”jobTitle”}):  
 jobs.append(a[“title”])  
 return(jobs)

extract\_job\_title\_from\_result(soup)

This yielded an output like this:



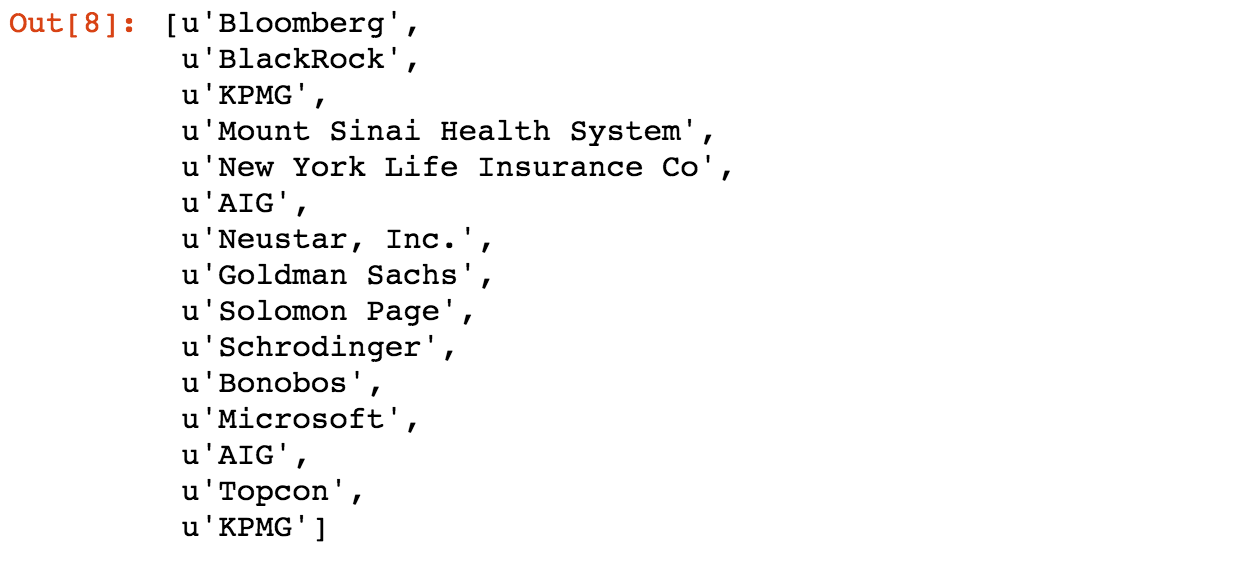
Perfect! All 15 jobs are represented here.

**Company Name**

Company names were a bit tricky, as most would appear in <span> tags, with “class”:”company”. Rarely, however, they will be housed in <span> tags with “class”:”result-link-source”.

I developed if/else statements to extract the company info from either of these places. Company names are output with a lot of white space around them, so inputting .strip() at the end helps to remove this when extracting the information.

def extract\_company\_from\_result(soup):   
 companies = []  
 for div in soup.find\_all(name=”div”, attrs={“class”:”row”}):  
 company = div.find\_all(name=”span”, attrs={“class”:”company”})  
 if len(company) > 0:  
 for b in company:  
 companies.append(b.text.strip())  
 else:  
 sec\_try = div.find\_all(name=”span”, attrs={“class”:”result-link-source”})  
 for span in sec\_try:  
 companies.append(span.text.strip())  
 return(companies)  
   
extract\_company\_from\_result(soup)

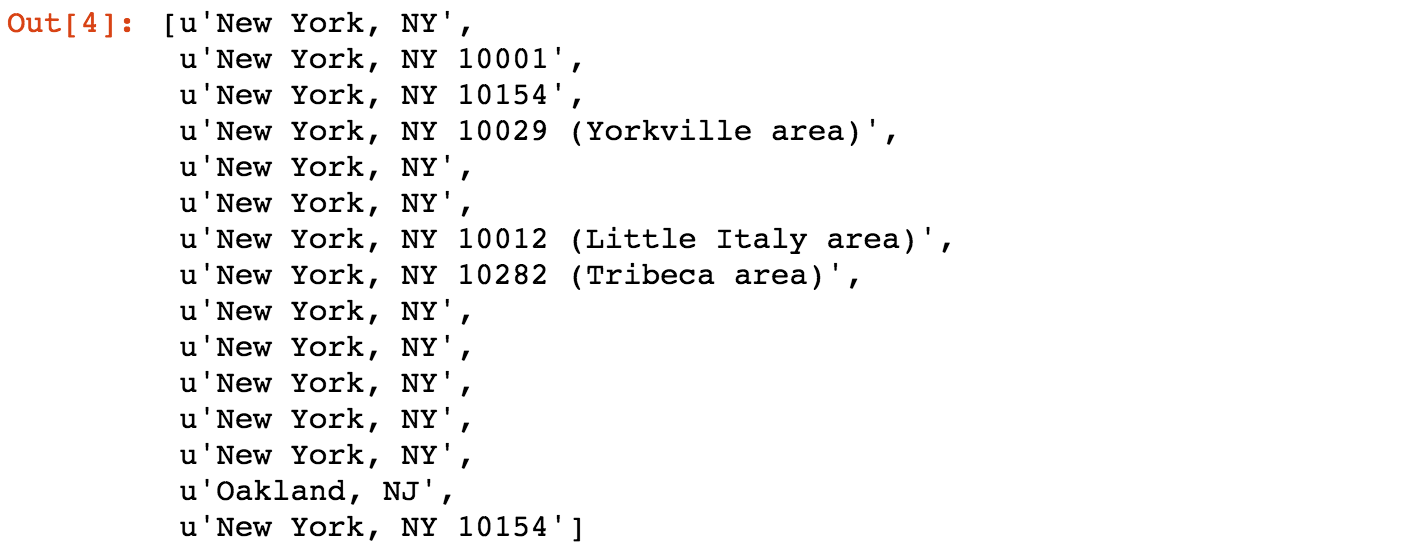


**Location**

Locations are located under the <span> tags. Span tags are sometimes nested within each other, such that the location text may sometimes be within “class” : “location” attributes, or nested in “itemprop” : “addressLocality”. However, a simple for loop can examine all span tags for text wherever it may be and retrieve the necessary information.

def extract\_location\_from\_result(soup):   
 locations = []  
 spans = soup.findAll(‘span’, attrs={‘class’: ‘location’})  
 for span in spans:  
 locations.append(span.text)  
 return(locations)

extract\_location\_from\_result(soup)



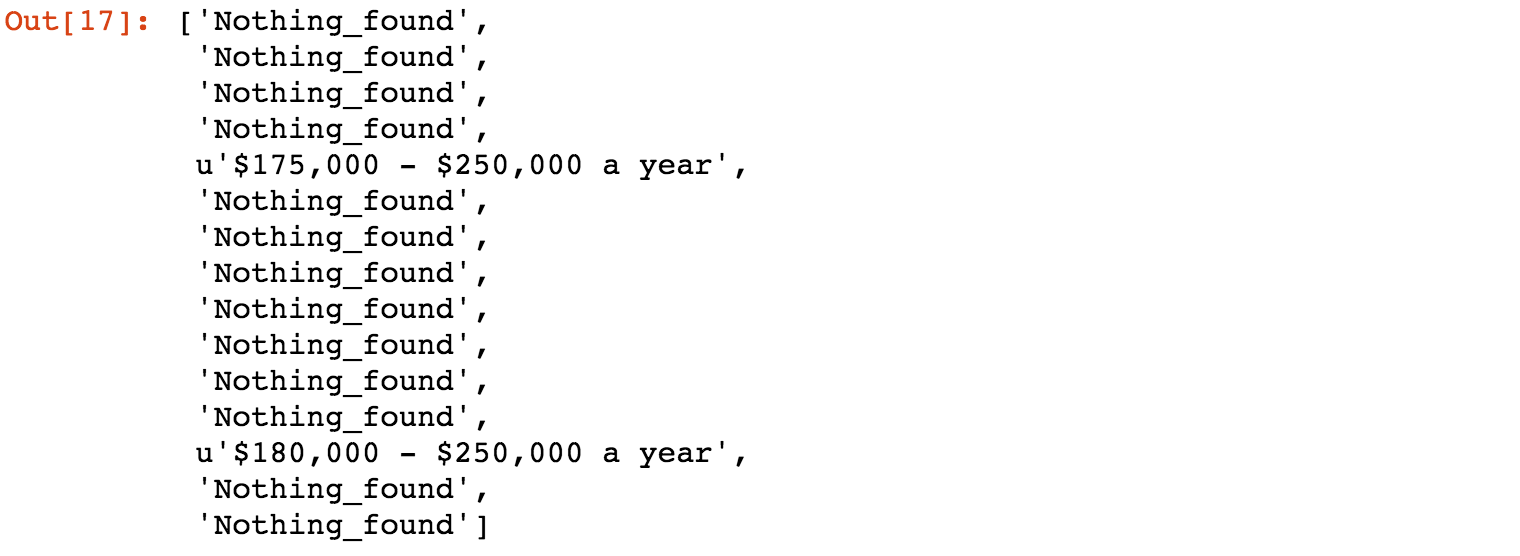
**Salary**

Salary was the most difficult data to extract from job postings. Most postings don’t contain any salary information at all. Among those that do, it can be in one of two different places. So, we need to write a function that can look in multiple places for information, and we need to create a placeholder “Nothing Found” value for any jobs that don’t contain salary. We want this placeholder to ensure that all of the information from any given job post can line up with all of the other pieces of relevant data when we later assemble our data into a single data frame.

Some salaries are housed under <nobr> tags, while others are under <div> tags, “class” : “sjcl” and are under separate nested <div> tags with no attributes. Try/except statements were particularly helpful in withdrawing this information.

def extract\_salary\_from\_result(soup):   
 salaries = []  
 for div in soup.find\_all(name=”div”, attrs={“class”:”row”}):  
 try:  
 salaries.append(div.find(‘nobr’).text)  
 except:  
 try:  
 div\_two = div.find(name=”div”, attrs={“class”:”sjcl”})  
 div\_three = div\_two.find(“div”)  
 salaries.append(div\_three.text.strip())  
 except:  
 salaries.append(“Nothing\_found”)  
 return(salaries)

extract\_salary\_from\_result(soup)



As noted above, most jobs did not have any salary information included.

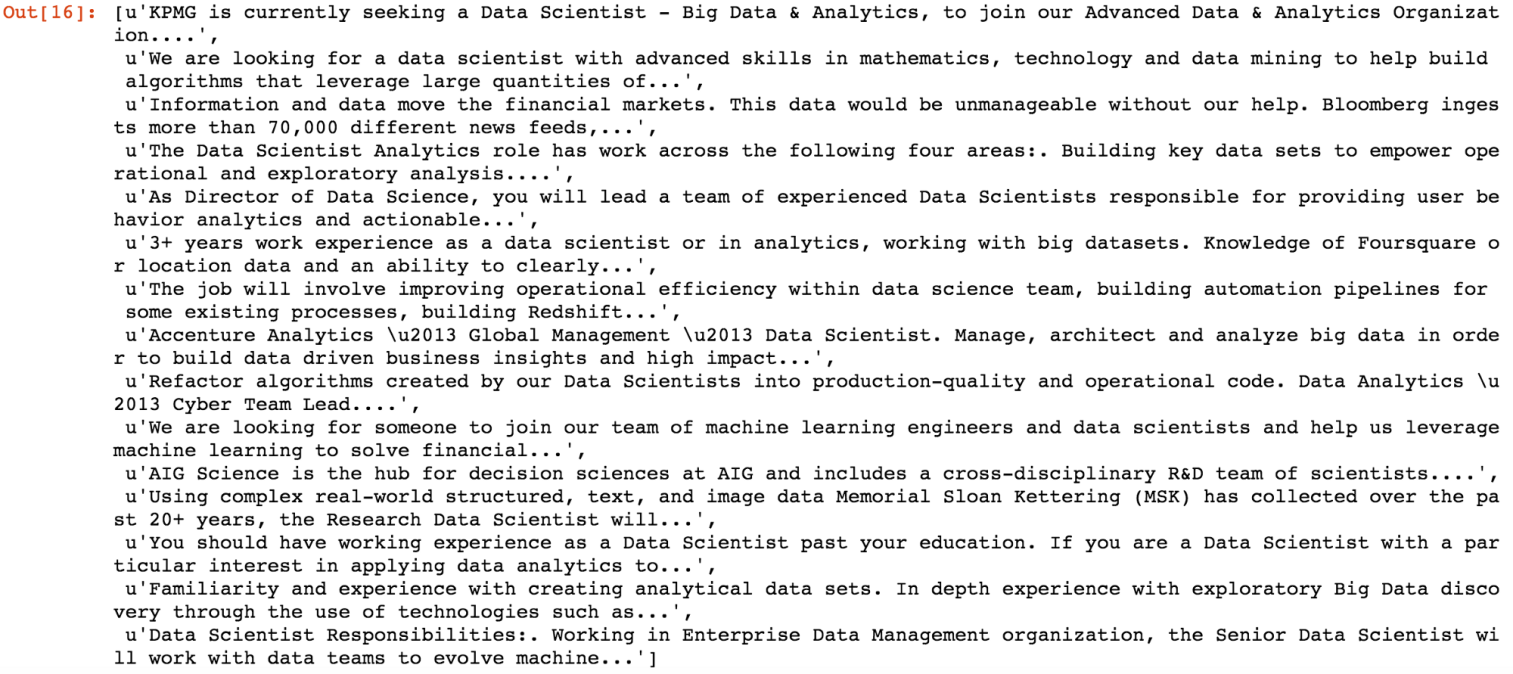
**Job Summary**

Finally, the job summaries. Unfortunately, the entirety of the job summaries are not included in the HTML from a given indeed page, however, we can get some information about each job from what’s provided. Selenium is a suite of tools that could allow a web scraper to click through different links on a page to withdraw this information from the full job postings. However, I did not utilize selenium for this particular effort.

Summaries are located under <span> tags. Span tags are sometimes nested within each other, such that the location text may sometimes be within “class” : “location” tags, or nested in “itemprop” : “addressLocality”. However, a simple for loop can examine all span tags for text wherever it may be and retrieve the necessary information.

def extract\_summary\_from\_result(soup):   
 summaries = []  
 spans = soup.findAll(‘span’, attrs={‘class’: ‘summary’})  
 for span in spans:  
 summaries.append(span.text.strip())  
 return(summaries)

extract\_summary\_from\_result(soup)



**Putting all the Pieces Together**

We’ve got all of the various pieces of our scraper. Now, we need to assemble them into the final scraper that will withdraw the appropriate information for each job post, keep it separate from all other job posts, and assemble all of my job posts into a single dataframe one at a time.

We can set up the initial conditions for each scrape by specifying a few pieces of information:

* We can detail how many results we want to scrape from each city of interest
* We can assemble a list of all of the cities for which we want to scrape job postings
* We can create an empty dataframe to house the scraped data for each posting. We can specify in advance the names of our columns for where we expect each piece of information to be located.

max\_results\_per\_city = 100

city\_set = [‘New+York’,’Chicago’,’San+Francisco’, ‘Austin’, ‘Seattle’, ‘Los+Angeles’, ‘Philadelphia’, ‘Atlanta’, ‘Dallas’, ‘Pittsburgh’, ‘Portland’, ‘Phoenix’, ‘Denver’, ‘Houston’, ‘Miami’, ‘Washington+DC’, ‘Boulder’]

columns = [“city”, “job\_title”, ”company\_name”, ”location”, ”summary”, ”salary”]

sample\_df = pd.DataFrame(columns = columns)

It goes without saying that the more results you want, and the more cities you look at, the longer the scraping process will take. This isn’t a huge issue if you start your scraper before you go out or go to sleep, but it’s still something to consider.

Assembling the actual scraper relates back to the patterns we noticed in the URL structure above. Because we know how the URLs will be patterned for each page, we can exploit this when building a loop to visit each page in a specific order to extract data.

#scraping code:

for city in city\_set:  
 for start in range(0, max\_results\_per\_city, 10):  
 page = requests.get(‘[http://www.indeed.com/jobs?q=data+scientist+%2420%2C000&l='](http://www.indeed.com/jobs?q=data+scientist+%2420%2C000&l=%27) + str(city) + ‘&start=’ + str(start))  
 time.sleep(1) #ensuring at least 1 second between page grabs  
 soup = BeautifulSoup(page.text, “lxml”, from\_encoding=”utf-8")  
 for div in soup.find\_all(name=”div”, attrs={“class”:”row”}):   
 #specifying row num for index of job posting in dataframe  
 num = (len(sample\_df) + 1)   
 #creating an empty list to hold the data for each posting  
 job\_post = []   
 #append city name  
 job\_post.append(city)   
 #grabbing job title  
 for a in div.find\_all(name=”a”, attrs={“data-tn-element”:”jobTitle”}):  
 job\_post.append(a[“title”])   
 #grabbing company name  
 company = div.find\_all(name=”span”, attrs={“class”:”company”})   
 if len(company) > 0:   
 for b in company:  
 job\_post.append(b.text.strip())   
 else:   
 sec\_try = div.find\_all(name=”span”, attrs={“class”:”result-link-source”})  
 for span in sec\_try:  
 job\_post.append(span.text)   
 #grabbing location name  
 c = div.findAll(‘span’, attrs={‘class’: ‘location’})   
 for span in c:   
 job\_post.append(span.text)   
 #grabbing summary text  
 d = div.findAll(‘span’, attrs={‘class’: ‘summary’})   
 for span in d:  
 job\_post.append(span.text.strip())   
 #grabbing salary  
 try:  
 job\_post.append(div.find(‘nobr’).text)   
 except:  
 try:  
 div\_two = div.find(name=”div”, attrs={“class”:”sjcl”})   
 div\_three = div\_two.find(“div”)   
 job\_post.append(div\_three.text.strip())  
 except:  
 job\_post.append(“Nothing\_found”)   
 #appending list of job post info to dataframe at index num  
 sample\_df.loc[num] = job\_post

#saving sample\_df as a local csv file — define your own local path to save contents   
sample\_df.to\_csv(“[filepath].csv”, encoding=’utf-8')

And that should do it! After a short while (maybe longer), you’ll have you’re very own dataframe of scraped job postings. Here’s a quick look at an example output:

