1. **MOTIVATION**
    a.
2. **Pre-requisites and requirements**
    a. There are no prerequisites, although you might find this workshop goes a little fast if you don't have any experience. I'd like to emphasize that this is an introductory workshop, so you will be exposed to a lot of new information in a short amount of time – you should keep in mind that some material is central to web scraping while other material is auxiliary. Keep in mind that while this is a web scraping workshop, we will need to use supplemental code and techniques around the actual act of scraping, and you should not go through this workshop expecting to memorize every little function that we use. The important takeaways will be the general process, workflow, and technique of web scraping with Python.
3. Ethics
    a. The ethics of web scraping are extremely important, so please pay close attention to this section. By attending this workshop and learning about these easy and powerful tools, you also have a responsibility to use them appropriately and legally.
    b. First, let's talk about when we're allowed to take data from websites, since not all websites allow the kind of automated access techniques we're using.
        i. Almost all websites publish a "robots.txt" file that specifies what parts of the website we're allowed to scrape, if any. These files are the standard way for websites to inform web-crawlers what parts of the website they can access. Let's look at an example:
            1. CKTG
            2. https://www.chefknivestogo.com/robots.txt
            3. This page makes no sense to the average human eye, so let's break it down:
                a. The "User-agent" line specifies the specific entity that the permissions on the next lines apply to. So, in this example, the first user-agent specified is "asterisk", which represents a "wildcard" here, which means that it refers to everybody (except Roverbot, whose permission s are specified at the bottom). So, since we are included in "everybody", we aren't allowed to access "/cgi-bin/", "/RT/", etc.
                b. At the bottom, we can see Roverbot's permissions. I assume that Roverbot was some kind of web-crawler, and apparently it didn't respect the rules of the chefknivestogo website, so it's no longer allowed to access anything on there – that's what this "Disallow: /" means – every URL that starts with chefknivestogo.com is not accessible to Roverbot.
                c. To clarify, being disallowed on "robots.txt" does not *necessarily* mean that you cannot access the site or this particular part of the site – it means that the site's owners do not consent to you accessing the site. In other words, it is *unethical* to scrape the site, and if that isn't enough motivation, it's also illegal. That being said, some sites will ban users or automated requests from certain

sections or the entire site (one example is craigslist, who prevent scraping as much as possible). If it's ever unclear about whether you can access a certain part of a website, you should contact the owners before accessing it! You can also contact the UBC office of the university counsel for advice on whether you can scrape a website.

c. Next, we need to discuss how we can know whether a given website can handle our information requests. This is an important consideration, since if we send too many requests to a website, it could crash. In fact, this is a form of cyber-attack called a Denial-of-Service attack, where the perpetrator attacks a site by sending too many requests for its servers to handle.
    i. Some websites publish a specific "crawl-delay" on their "robots.txt" page, which indicates how many seconds a crawler has to wait between information requests. For example, we can go to https://www.nrcs.usda.gov/robots.txt, where we see that they limited the request rate for "msnbot" and "Slurp". This means that these two user-agents have to wait 10 seconds between sending requests.
    ii. However, these are the only two user-agents that are given an explicit request rate limit, so it's still unclear how quickly we're allowed to request information from the website.
    iii. There is no general answer for how many requests per second a server can handle, since it depends on the server, what you're requesting, and other factors, but in general you should aim to scrape as slowly as possible! For most sites, a delay of 1-5 seconds between requests is enough.
    iv. One more timing-related consideration is that if you're going to send a lot of requests to a website, it's best to send them at the time of day when the site receives the least traffic. This depends on the website, due to time differences and other considerations, but in general it's something to consider.
d. The last ethical concern we'll cover is identification – it's good practice to identify ourselves to websites. It turns out that our browser alreadys automatically identify us to websites when we visit, and we can modify this identification to make us contactable. For example, we can add an email address so that the webmaster can email us if our scraping is problematic. More on this later.
e. ADD API SECTION (JUST SAY SOME WEBSITES PROVIDE API'S WHICH THEY PREFER WE USE TO GET DATA)
f. ADD SECTION ON HEADERS (SHOULD PROBABLY ADD HEADERS FOR WIKI SQUIRREL AND LICHESS EXERCISES SINCE THEY USE REQUESTS – PD.READ_HTML DOESN'T HAVE A "HEADERS" PARAMETER)

4. Functions, Data Types, and Pandas
    a. First, I'm going to bring everyone up to speed on the basic datatypes in Python.
    b. List
        i. Lists store multiple items (called elements) in one ordered variable.
        ii. Elements are separated by commas.
        iii. List comprehension examples
    c. Dict

i. A dictionary is a correspondence of keys to values. Here is an example of a dictionary:
ii. **test_dict = {'a':1, 'second element':2}**
iii. In this example, the letter "a" is a key, and the value 1 is its corresponding value. So, if we type **test_dict["a"]**, we see the output is 1. Note that in this case the key and value are different values "a" is a string, and "1" is an integer, but they are still a valid key:value pairing. This is part of the beauty of dictionaries – you can have just about any key:value pairing.
iv. One thing to note about dictionaries is that they are **one-way** correspondences. By that, I mean that we cannot enter the *value* to get the key. For example, if we try to access **test_dict[1]**, we get a "KeyError", since 1 is a value and not a key.
v. Dictionaries *can* be ordered, but in general, order is not important to them like it is for lists.
vi. Some of you may recognize that this is very similar to a JSON file structure – in fact, you can save Python dictionaries as JSON files, and you can import JSON files as dictionaries!

d. Pandas DataFrame
   i. **INCOMPLETE**

5. Web Scraping
   a. Pandas read_html()
      i. read_html() is a simple tool that can't be used for every web scraping job, but it's powerful, easy to use, and shouldn't be overlooked!
         1. After you give read_html() a url, it returns a list containing every table it can detect on the page.
         2. Documentation: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_html.html
      ii. **USDA FIPS**
         1. Motivation/Background
            a. FIPS county codes are numerical ID codes for American counties and county equivalents. They were officially replaced by the US gov in 2009, but lots of data sources still go by FIPS, and I actually need FIPS for a project I'm working on right now.
            b. I use FIPS for my project because I need to match text strings of US locations to geographical data which is labeled by FIPS and not by text. For example, I need to convert "Baldwin County, Alabama" to FIPS code "01003".
            c. A quick google search for "US FIPS codes" leads me to a USDA website which lists county FIPS codes next to corresponding state abbreviations and county names. While we can download this data from other sources such as census.gov, we can also scrape this data in just a few lines of code with pandas.read_html():
         2. Coding

a. usda_fips_page = pd.read_html("https://www.nrcs.usda.gov/wps/portal/nrcs/detail/national/home/?cid=nrcs143_013697")
    i. As mentioned previously, this returns a list of tables, where the tables have been converted to pandas dataframes.
b. Let's examine the output… okay, strange. It looks like there are multiple copies of the FIPS table we wanted (and this explains why reading the page took so long). This isn't a problem, we can just grab the first one and ignore the rest.
c. Let's print the dataframe to see if there are any problems.
d. It looks like all of the rows came out okay except for the last one, so we'll just delete that manually
e. Save to .csv (or other file)

b. BeautifulSoup + Requests
    i. Wikipedia red-bellied snake
    ii. Note that wikipedia.org has an API, so if we were actually interested in getting data from their site we'd use the API. However, this is a learning exercise, so we'll scrape the front-end.
        1. In this exercise we'll be getting all of the text from a Wikipedia article, analyzing their frequencies, and saving to a .csv file.
        2. We start by using the "requests" package to "get" the html document which displays the red bellied snake Wikipedia article.
        3. Let's look at the text of the output… ok it's a total mess. We need to significantly process this before it's usable.
        4. All you need to know about HTML for now is that it's a bunch of elements , and each **element** has a **tag** and a **class**. For example, the "a" tag is used for URLs, so if we want to find all URLS in the document, we would use BeautifulSoup to find all **elements** with "a" tags.
        5. In our example, we are looking to extract all text elements from the document, so we will find all elements with "p" tags ("p" stands for "paragraph")
            a. This definitely looks better – if you look closely, it has more English text than the previous things we printed.
        6. This is where the BeautifulSoup library comes in handy – we can use it to get all of the text out of these paragraphs.
        7. **Print .text output**
        8. The next step is to save all of this text together as one string variable. We don't absolutely *have* to do this, but it will make our final goal of word-frequency analysis a lot easier. Let's look at this new variable:
        9. **Print wiki_snake_text**
        10. It's less readable to the eye than printing out each individual paragraph, but all of the text is in one place and we can analyze frequencies now. There are a few erroneous characters, such as "\n", which makes a new line, and "\xa0", which is a space that prevents a line break. I'm not too concerned

about these for our informal analysis, but I've provided a source explaining how to remove these from text.
    a. https://docs.python.org/2/library/unicodedata.html#unicodedata.normalize
11. Now we can do our brief word frequency analysis! We start by import NLTK, which stands for "Natural language toolkit" – this is one of the most popular Python packages for processing human language data.
12. NLTK has a function called "FreqDist", which stands for "frequency distribution." The function takes in a **list** of words and returns a **dictionary** where each word is paired with the number of times it appears in the document. Let's apply the function and see what the frequencies are.
13. First, remember that our text is stored as one long string, and we need to convert it to a list of words.
14. **split_wiki_snake_text = wiki_snake_text.split()**
15. Examining our output, it looks like exactly what we wanted. Let's put it into the FreqDist() function and see how the output looks!
16. **wiki_snake_word_freqs = nltk.FreqDist(split_wiki_snake_text)**
17. Looks pretty good – the pairing of words and their frequencies seems believable at least. However, it's currently saved as a dictionary datatype, and we want to save it as a .csv, which is more a tabular format. Specifically, I think that having two columns – one containing words, the other containing their associated frequencies – is the ideal format for storage.
18. We'll first convert it to a Pandas dataframe in order to save it as a .csv file. So, we're going to initialize a Pandas dataframe from the dictionary's keys and values.
19. **wiki_snake_output = pd.DataFrame({'Word': wiki_snake_word_freqs.keys(), 'Frequency': wiki_snake_word_freqs.values()})**
20. Printing the output, we see that everything seems good, although the dataframe is not sorted in order – we can do that with **.sort_values('Frequency', ascending=False)**.
21. Everything looks good, except I'm more interested in these values as percentages – let's make a new column to reflect that.
22. Finally, since everything is ready, we'll save this as a .csv file and we're done here!
23. **wiki_snake_output.to_csv("Red Bellied Snake Words & Frequencies.csv")**

iii. Lichess top player info

1. *Note for presenter*
    a. ***This exercise takes too much time to fit into the 2 hour workshop. It may be worth using this lichess.org example as material in an intermediate web scraping w/ python workshop.***
    b. ***If there are 15 minutes left, I recommend just showing the attendees how to use selector gadget with Requests + BeautifulSoup (i.e., very briefly make the functions to get the***

players's # games played, etc., but less focus on the overall workflow and more on using selectorgadget + BeautifulSoup to select specific elements from a page).

   c. ***I recommend skipping the first part of this exercise – where we get the table of the top 200 players from lichess.org – this table requires a decent amount of cleaning, which is not the point of the workshop! You can read the pre-processed .csv file of the top 200 players and then jump to the second half of this exercise :)***

   d. ***One more tiny detail: the "top 200 players.csv" file will be out of date when you next run the workshop. This won't have any impact on the workshop, the code, or anything really – it just might throw you or attendees off a little so here's a heads-up.***

2. For our final exercise, we'll be going to my favorite open source chess website and getting the rating of top players, along with a few other statistics.
3. First, let's inspect the robots.txt file and make sure we're allowed to get this data: https://lichess.org/robots.txt
   a. It allows everything except accessing the API and exporting games, neither of which we are doing, so we are good to go!
   b. Note that lichess.org has an API, so if we were actually interested in getting data from their site we'd use the API. However, this is a learning exercise, so we'll scrape the front-end.
4. Ok so our approach will be roughly structured as follows:
   a. Get the list of top 200 bullet players from https://lichess.org/player/top/200/bullet
   b. Go to each of their accounts and get the information we need
      i. Account pages have the following structure: **"https://lichess.org/@/" + user_ID**
      ii. So we're going to need usernames if we want to write code to scrape all top 200 players' statistics
   c. Compile the information into a pandas DataFrame
   d. Save the dataframe to a .csv
5. **Step 1: getting top 200 usernames**
6. ***NOTE FOR PRESENTER:***
   a. ***This is the section you should probably skip***
7. First, let's look at the top 200 users, who are listed at https://lichess.org/player/top/200/bullet
8. the first thing we can notice is that it looks like the top users are stored in a table! We can try to use pd.read_html() to get this table, since that would be easiest.
   9. **players_df_list = pd.read_html("https://lichess.org/player/top/200/bullet")**
      # let's see the output

players_df_list

a. Examining the output, we see that there's only one dataframe in this list of dataframes (which makes sense, since there's only one table at that URL. So, we take this dataframe and save it as a new variable, **players_df.**

b. Examining **players_df**, we have two columns we aren't interested in (the user's recent rating change and the user's ranking), so we'll drop those columns from the dataframe.

c. Let's also rename the remaining columns, so they have more useful labels.

d. **players_df.rename(columns={1:'User', 2:'Rating'}**, then

e. **players_df.rename(columns={1:'User', 2:'Rating'}, inplace=True)**

f. Now that we have the players' usernames, let's place them in the "URL formula" to see if they work:

   i. **"https://lichess.org/@/" + players_df['User'].iloc[0]**

g. Okay there's some kind of weird "\xa0" thing in here… turns out this "xa0" is not a kind of dumpling, it's a "non-breaking space". It's not important what exactly that is, but we do need to get rid of it. One way is to define a function to replace it in our username strings.

h. Now that our usernames don't have this "/xa0" anymore, let's retry printing the URL using our "formula".

i. **"https://lichess.org/@/" + players_df['User'].iloc[0]**

j. if we copy and paste the first user's username into the URL for their account page, we see that the link doesn't work! For one, there's a space in this URL… we can also see that their actual URL doesn't have their title in it.

k. So, we now have to create a new column in our dataframe which just contains the username of each player, without the title. We're going to do this by writing a function to get the username out of each username string.

l. There are multiple ways to do this, but the simplest way is probably to write two functions – one to get the user's title, and the other to get their username.

m. Okay, now that we finally have their usernames in order, we can use this dataframe to scrape their individual account pages.

10. **Step 2: visit each user's account page and take the info we want**

    a. We're going to use the first-ranked player for our example – we'll make sure that our functions work on his page, and then we'll write a big function to apply all of these functions to any lichess.org player page. This is all working toward the final goal of applying this big function to each of the top 200 players' pages.

    b. Get name

    c. Get # followers

    d. Get # games played