SI 206 – Winter 2017 – Data-Oriented Programming
Project 2

DEADLINE: All files must be submitted on Canvas by Sunday, March 12th at 11:59 PM

WHAT YOU SHOULD DOWNLOAD TO WORK ON:

- The `SI206W17_project2.py` file, which contains some provided code
- The `SAMPLE_206project2_caching.json` file, which is an *example* of what your cache file might look like when you have completed the project

WHAT YOU SHOULD SUBMIT:

- Your edited `SI206W17_project2.py` file
- Your cache file: `206project2_caching.json`
- A text file with anything you need to tell us about the project / your submission, if applicable

Your .py file must run successfully using Python3. We do not grade projects that do not run. Make sure you check that your program runs before submitting it!

TOTAL POSSIBLE POINTS: 1500

The project is in four parts, though a couple of them are quite small, but each part builds on both the homework you have done previously AND on the previous part of the project.

Your homeworks 3-5 are likely to be very helpful for this project.

We have provided starter code at the top of the file for setting up Tweepy, which you'll use later in the project (you should rely on the same **twitter_info.py** file you used in HW5, or create one, if you did not).

We have included starter import statements and comments indicating where you may want to place your code in the file.

The file also includes tests, which are the primary determining factor as to how you get points (but we do look at your code, etc).

Make sure your file runs without syntax errors!

You should start this early! Break it up into pieces, write out a plan, do a little bit at a time, and keep running and testing it, adding and committing and pushing to GitHub.

Part 0

Set up caching for your Project 2 program.

Your cache file should be called **`206project2_caching.json`.** We have provided a sample file called `SAMPLE206project2_caching.json`. It has the same format your file should have.

The points for caching are in 2 parts.

Ultimately, your cache file should contain:

- cached data from the UMSI website (data: a list of HTML-formatted strings)
- cached data from requests to the Twitter REST API (data: a big list or dictionary that you get as a result from invoking a Tweepy method)

(a) Initial setup (100 points) No tests for this: we will see whether or not your program correctly uses a cache/reads from a cache file/generates a file!

You'll need that initial setup of the caching pattern in your file: the try/except statement, the determination of the variables for your cache file and cache dictionary…

(b) Unique identifiers for your cache file (150 points)
This will make more sense once you've read through the whole project instructions.

Your cache file should ultimately contain JSON-formatted data that represents a big Python dictionary (your `CACHE_DICTION`).

That `CACHE_DICTION` dictionary should have least 2 keys by the end, strings that are unique identifiers of specific sets of data – but it could contain more, depending upon what requests you make while you work on your program! The two keys it *must* have are:

- `"umsi_directory_data"`
  - The value associated with this string should be a list of HTML-formatted strings that you get from the UMSI directory in Part 2.
- `"twitter_University of Michigan"`
  - The value associated with this string should be a big dictionary that represents a response from a Twitter API search, performed via Tweepy

All that data should be saved, in a JSON-formatted way, to your **`206project2_caching.json`** file, once you've completed the project.

You will make that happen while you write the code for Parts 2 and 3, as long as you have the cache set up correctly and manage the caching process correctly in your functions that you write later on.

Part 1 – Regular expressions + function definition

- Define an additional function using Python regular expressions called **find_urls**.
- **INPUT:** any string
- **RETURN VALUE:** a list of strings that represents all of the **URLs** in the input string
- See code file for example.
- There are tests for this. **200 points.**

Useful notes for Part 1:

- You can define what a URL is in a simple way in this assignment, though sometimes this can be quite complicated. For us, a URL is:
    - Anything that begins with **http://** or **https://** AND
    - Includes at least one **.** character within it, and each **.** character must be followed by at least 2 characters
        - E.g. **http://bbc.co.uk** is a valid url
        - And **https://www.gmail.com** and **https://gmail.com** are also both valid urls
        - As is **http://nationalparkservice.gov/pictures/badlands**
        - But **gmail.gov** is not a valid url for our purposes
        - And **http://bbc.c** is not a valid url for our purposes either
    - Includes no spaces or any other whitespace characters
    - (OK to be more specific than our requirements above if you want to challenge yourself to get absolutely all URLs even if they're quite unusual. You must at least handle all of all of those listed requirements.)

- **HINT:** Playing with the websites that we looked at in lecture & discussion and that are posted/discussed on Piazza, as well as checking against the example slides, will be a great way of testing things out to solve this problem.

Part 2 – BeautifulSoup and complex Python mechanics

(a) Now, you'll use the requests module and BeautifulSoup to accumulate a list of the HTML data that represents the ENTIRE UMSI directory.

Write a function called `get_umsi_data` .
INPUT: No input.
BEHAVIOR: The function should check if there is any cached data for the UMSI directory in your cached file – if so, return it, and if not, the function should access each page of the directory, get the HTML associated with it, append that HTML string to a list. The function should cache (save) that list when it is accumulated.

RETURN VALUE: A list of HTML strings representing each page of the UMSI directory.
There are tests for this whole function. **350 points.**

Useful notes for Part 2(a):
You should get ALL the HTML data in the directory, *starting* with the page at this URL:
https://www.si.umich.edu/directory?field_person_firstname_value=&field_person_lastname_value=&rid=All
And ending at the page at this URL:
https://www.si.umich.edu/directory?field_person_firstname_value=&field_person_lastname_value=&rid=All&page=11
and including each of the pages in between in between, each of which can be represented by a big HTML string (just like the one big HTML string you dealt with when doing Part 3 in HW4).

- **HINT:** What do those two page links have in common? What's different? That's the first and the 11th page, respectively. What do you think the URL of the 7th page of the directory looks like?

Remember that to get data from the UMSI site, you must have the headers parameter like so in your request, so that you will not get blocked by the site:
```
requests.get(base_url, headers={'User-Agent': 'SI_CLASS'})
```

(b)

Write code to build a **`umsi_titles`** dictionary whose keys are the names of each person in the UMSI directory, each of which's associated value is that person's title, e.g. "PhD Student" or "Associate Dean of Research and Arthur F. Thurnau Professor of Information, School of Information" … .

e.g. **"Lindsay Blackwell":"PhD Student"** should be one of many key-value pairs inside this dictionary.
You should NOT worry about cases where there is no title / the title is an empty string. Include the empty strings in your keys or values of the dictionary.
There are tests for this. **250 points.**

Useful notes for Part 2(b):
This dictionary should sound familiar… check out your HW4 and the Piazza discussions about building a dictionary like this! Lecture notes may help, too.

Part 3 – REST API requests, Tweepy, and nested data

(a)
Define a function **get_five_tweets.**
INPUT: Any string.
BEHAVIOR: Your function should search for that input string on Twitter, using Tweepy. It should use cached data when possible and cache data to your program's

cache file when it gets new data. (Check out Part 0 for what your unique identifier for each request in the cache should be.)
RETURN VALUE: A list of just the text of 5 different tweets that result from the search.
There are tests for this (though we can't test everything about it in code, you should also run your code and check to see that you get the output you want using print, as always). **200 points.**

You should NOT use the Python input function in this code, you should simply write a function that can take any phrase as input – though you can and should invoke the function with different input values to see what happens.

(b)
Write code to invoke the **get_five_tweets** function with the phrase "University of Michigan"). Save the result in a variable called **five_tweets**. There are tests for this. **100 points**.

(c)
Iterate over the **five_tweets** list, invoke the **find_urls** function that you defined in Part 1 on each element of the list, and accumulate a new list (one list of strings, *not* a list of lists!) of each of the total URLs in all five of those tweets in a variable called **tweet_urls_found**. There are tests for this. **150 points**.

We will use our own version of the `206project2_caching.json` file, like the `SAMPLE206project2_caching.json` file provided, to grade your projects.