Homework 1

[ 100 points - due by 11:59 pm, Sunday, February 12, 2017 ]

Submit these files to the CS submission system at the usual place by 11:59. You may work on your own or with 1-2 partners on the programming portions of this assignment. (The reading/response is individual only.) Groups larger than 3 should split into smaller groups. Remember that partners need to work in the same physical location, share composition time equally (or each compose on their own machines) and be fully equal owners and producers of their work. *Have fun experimenting!*  [cs35 homepage](https://www.cs.hmc.edu/~dodds/cs35/)

**Downloads**

There's one (zipped) starter file to download -- grab it at the start of class & follow along:

* [The zip file to start all of this week's problems…](https://drive.google.com/open?id=0BwPWh-3AmiLxaFZLSTdQbmRTVUk)

**Setup**

Let me know if your Python and/or submission-site account isn't working...

**Submission**

Again we ask you to submit a zipped archive named hw1.zip with your cs35\_week1.py file of solutions -- plus any other Python files and/or json files you downloaded and saved -- those can make it even easier to run your solutions. As usual, submit the reading response in its own spot at the [usual submission site](http://cicero.cs.hmc.edu/).

This week's hw includes one lab problem and one-choice problem beyond the lab and one open-ended problem that uses your own choice of web data as input…

**Problem 0**: *(When) Is webscraping legal and/or ethical?* [5 pts]

This week's reading includes two short articles: the [first](http://www.storybench.org/to-scrape-or-not-to-scrape-the-technical-and-ethical-challenges-of-collecting-data-off-the-web/) is entitled, "To Scrape Or Not To Scrape: Technical And Ethical Challenges Of Collecting Data Off The Web," by Sophie Chou, a student in the MIT media lab. The [second](https://www.promptcloud.com/blog/is-data-scraping-ethical) is "Is data scraping an ethical practice? We explain." Both articles take a big-picture view of the ethics of different kinds of programmatic access to web-available data. After reading over these (short) articles, reflect on either of these two prompts: (a) Near the end of the second article is the sentence "As long as you follow [websites' rules], you are doing nothing unethical." Do you agree or disagree - or, perhaps, take a middle stance? Or, consider (b) The first article includes a flow chart whose outcomes range from "No" to "Go for it!" In your web-prediction solution (problem 3, below), do you fall in the "Go for it!" category? What is a change to your question that would change that category? As with each week's reading, responses should carefully considered, but need not be very long (4-5 sentences is wonderful).

**[Lab problem] Problem 1: Google Maps Challenge** [35 pts]

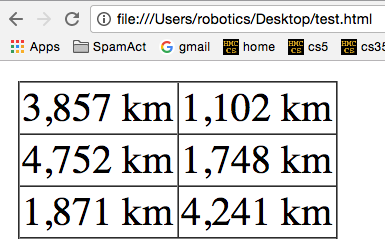
* In class, we ran code to find the distance between Claremont and Seattle. That code is included in the starter file in two parts: a function that grabs the webdata and saves it to a json file locally, and another function that reads from that json file and allows you to access it arbitrarily/programmatically. Take a look at that code and make sure that it runs for you.
* Skim over the [API Documentation](https://developers.google.com/maps/documentation/distance-matrix/intro?hl=en) to get an idea of what else you can do with the Google Maps API - notice the data limits (this is why saving to a file is so useful -- otherwise, every bugfix/test would deduct from your "dataplan" :-)
  + Specifically, check out how *multiple* cities need to be handled (it's an unusual separator character!) It's under "Request parameters" - "Required parameters" - "origins"
* We also mentioned (but didn’t show) that you could use the API to get the pair-wise distance between *multiple* cities. Your challenge is to write two functions:
* **multicity\_distance\_scrape( Origins, Dests, filename\_to\_save="multicity.json" )**  should
  + take in two lists: The first should be a list of “origin” locations, and the second should be a list of “destination” locations.
  + it should uses the Google Maps API to get the distance from each of the “origin” locations to each of the “destination” locations.
  + The Google Maps API takes can take in a single string for multiple origins and another (single) string for multiple destinations. In each case, the component cities need to be separated by the vertical bar "|" -- so, you'll want to write a loop to create those two large strings with each city separated by a vertical bar, e.g., "Pittsburgh,PA|Boston,MA"
  + All of this is in the Google Maps API -- you're encouraged to explore any of the additional options you can pass the API in terms of travel mode, route preferences, etc.
  + Once it gets the resulting JSON data, this function should save that json data to a json file (for consistency, use our sample code's approach of a default filename, "multicity.json"). *Then,*
* **multicity\_distance\_process( filename\_to\_read="multicity.json" )**  should
  + read in the json data from the file (again, use a consistent default name), and
  + Should create and return an HTML table with the distances between each of the locations. *You don't need to label with the cities - that's extra credit (try it, if you have time!)*
  + Here is an example with two pairs of cities and a (plain) resulting table:

Origins = ['Pittsburgh,PA','Boston,MA','Seattle,WA'] # starts

Dests = ['Claremont,CA','Atlanta,GA'] # goals

multicity\_distance\_scrape( Origins, Dests ) # should save results to file

* This creates the multicity.json file.
* From there, running multicity\_distance\_process()
* *and pasting the results into a plain HTML file*  yields this:



Full disclosure: this uses <table border="1px" style="border-collapse: collapse;">

* Be sure to include a description (in a triple-quoted string) of any choices you made in working on this problem -- and please be sure to let us know any extra things you included into these two functions! Some ideas:
* ***EC Options***: Add column and row labels in appropriate spots! And/or style your table using CSS or other HTML/web-output ideas -- for example, you could highlight in some distinct way the shortest pairwise distance and/or the longest. Or add NFL logos, links to the cities' wikipedia pages, and/or post the resulting table to your GitHub repository.
* ***Ludicrous EC Options***: These web-scraping api calls can be made from *within*  a webpage using Javascript (after all, the JS in JSON stands for Javascript…). Research how to do this and create a live distance-table-generating page in your GitHub repository. **Caution**: I have not done this and am not sure how difficult it is, but suspect it's no picnic! This isn't really EC (though it would be, for sure!) -- but it's to point out that it *can be* done. If combined with some additional graphics/processing, for example, this would be an interesting *final* project in cs35.

**Problem 2a: iTunes Store Challenge**

[30 pts; choose *either* problem2a or problem 2b - the other can be done for +8 pts EC]

* The iTunes store gives API access to searching for artists, albums, movies, etc. This week’s starter file includes code to lookup the iTunes ID for an artist (in our case, The Beatles) and then to use that ID in order to retrieve a list of all of the albums in the iTunes store for that artist.
* Skim over the [API Documentation](https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/) to get an idea of what else you can do with the iTunes Store API.
* Write a pair of functions (in a spirit similar to the starter code):
  + **most\_productive\_scrape(artist1, artist2, fname1="artist1.json", fname2="artist2.json")** that takes the names of two artists as input, converts those names to AppleIDs (notice that this requires an API call!) and then makes *another* API call in order to gather all of the album/work information from iTunes. It should save those results into the filenames **fname1** and **fname2** (with reasonable defaults, as you see above).
  + For example, calling **most\_productive\_scrape("Steve Perry", "Katy Perry")** should get cross-generational "Perry" information into those two files... Then, write
  + **most\_productive\_process(fname1="artist1.json", fname2="artist2.json")** which reads in those two files and then prints out (a) the artists and (b) the number of works they have int he iTunes store. You'll need to look the artists' names from the json files - this is pretty typical, even if it feels backwards!
  + For reference, here is what prints out when these two functions are run in our solutions (feel free to alter what's printed, of course). Also, remember that most of the work is behind the scenes in the json files!

Python> most\_productive\_scrape("Steve Perry", "Katy Perry")

file artist1.json written.

file artist2.json written.

Python> most\_productive\_process()

# of results for Steve Perry == 9

# of results for Katy Perry == 31

* Be sure to include a docstring for each function describing (briefly) what it does...
* *Include at least three test cases comparing interesting pairs of artists of your own choosing… !*
* Be sure to include a description (in a triple-quoted string) of any choices you made in working on this problem -- and please be sure to let us know any extra things you included into these two functions! Some ideas:
* ***EC Options***: Expand your function to use more of the data from the iTunes store in order to ask broader / more interesting questions. More artists, perhaps? Or, you might explore which artist has the most single-word album titles; which has the most album titles with numbers in them. There is lots of additional data returned in the json files, as well… If you do this option, *be sure* to provide a couple of interesting test cases, and to include a (brief) description of what you did in your comments (triple-quoted string).

**Problem 2b: USGS Earthquake Challenge**

[30 pts; choose *either* problem2a or problem 2b - the other can be done for +8 pts EC]

* The USGS gives API access to their record of earthquakes that have occurred during a given time range. This week’s starter file includes code to print out the location of the first earthquake returned by a query of the API.
* Skim over the [API Documentation](http://earthquake.usgs.gov/fdsnws/event/1/) to get an idea of what else you can do with the USGS API.
* In this case, the starter code has provided a few lines of examples -- to try from the file or from the command-prompt, but we *don't* provide stand-alone functions. Your task is to design, create, test, and debug those! (Feel free to name them, as well!)
* Write a function that prints out the number of earthquakes for each of the previous 7 days -- in addition, it should determine which day (of the previous 7) had the most earthquakes.
  + Since the API includes *very* tiny earthquakes, your function should take as input an argument named *threshold,* and it should ignore any earthquakes returned whose magnitude is less than *threshold*. (We use 2.42 as a reasonable threshold.)
  + **Hint**: Take a look at the [datetime](https://docs.python.org/3/library/datetime.html) Python library and the examples for help with getting the current date -- and the previous 6 days as well!
* As always, be sure to include a description (in a triple-quoted string) of any choices you made in working on this problem -- and please be sure to let us know any extra things you included into these two functions! Some ideas:
* ***EC Options***: Introduce some other sort of filtering on the earthquake results. For example, you might restrict your results to only earthquakes that happened in California. Whatever extensions you create, please be sure to highlight what you've done, in a triple-quoted string near by your quaky functions!

**Problem 3: The Wisdom of the Web?**

[30 pts; with EC certainly available!]

* The truth is out there… *but where?*  It's the distinctive badge of our era that the web has answers to all questions -- but we're left with sifting through those answers for the truth… This problem asks you to pose - and answer - a question via at least two web sites of your own choosing…
* This problem lets you choose your own Web datasource to scrape and then analyze. It can be raw webfiles (to practice parsing with BeautifulSoup) *or* it can be JSON files of the sorts we used above. (And, if you're excited about yet-another-sort of web data, ask about it -- it's very likely it's totally ok to use that instead.)
* First, take a look at our annotated Scavenger Hunt example (already in the starting hw2 archive in the file named ) It asks the question *Who will win the Superbowl this year?* But it doesn't take the customary approach of simply looking up the score! Instead -- it *predicts* which team will win by scraping and analyzes two sources of webdata… .
* Admittedly, predicting this year's winner is not overly impressive, but it can also be used to predict last year's! (Or, in fact, the outcome of any NFL game -- and a number of other sports', as well.)
* Your scavenger hunt can be as surreal or serious as you'd like - but it should
  + include at least two functions, each of which grabs and at least one webpage/website using **requests**. From the request, either BeautifulSoup objects or JSON filenames, which can then be read for the contents, should be returned. It's also OK to use a standard filename, if you'd like.
  + Then, include at least two functions that ***use*** those BeautifulSoup objects or JSON files to extract data that's important to your question. (More than two functions are certainly OK!)
  + The reason that these two parts are separate is that *it's much more efficient not to download the same page again and again --*  as with the prior problems, you can download once and use the resulting object or file many times, e.g., for debugging
  + include a short write-up of the (1) question you asked, (2) the approach you used to construct an answer, and (3) the answer itself. Include a series of cells that the graders can run to see your web-wisdom in action!
  + This write-up does not have to be extensive - one paragraph of explanation certainly suffices -- and more is ok, too.
* Possible APIs you might consider using (and that people ask about):
  + Note that these *won't* require you to use requests (since they have their own interface!)
  + <https://www.npmjs.com/package/google-trends-api> [for Google Trends]
  + <https://github.com/lukaszbanasiak/yahoo-finance> [Yahoo stock quotes]
  + <https://pypi.python.org/pypi/googlefinance> [Google stock quotes]
  + and, many, many more…
* ***EC Options***: Wide open here!

**Problem 4: GPS Coordinates**

[??? pts]

If you don't have the files from last week, [here](https://github.com/ScriptingBeyondCS/CS-35/tree/master/week_0_to_2) they are!

Last week, you began exploring the laptop of a missing grutor. You might have noticed a single file *Untitled.txt* right on the surface, not hidden in any directories. It certainly looked suspicious. Navigate to it and open it now. The list of numbers of initially baffles you. Then you realize these numbers look a lot like GPS coordinates. They could be the key to finding the grutor! You decide to build a script to discover where each of these coordinates are located.

Your task is to use both the Google [Geocoding API](https://developers.google.com/maps/documentation/geocoding/start) and Beautiful Soup to build a function in Python that takes in a set of latitude/longitude coordinates and prints the address to which they refer.

* As a hint, the following example url requests the address corresponding to the latitude/longitude 40.714224,-73.961452. It specifies that the output must be in XML format. The corresponding address is 277 Bedford Ave, Brooklyn, NY 11211, USA

http://maps.googleapis.com/maps/api/geocode/xml?latlng=40.714224,-73.961452&sensor=true

* Be sure to print the address, city, state, zip code, and country for each set of coordinates as shown above

Your script has revealed five locations. It feels like the grutor must be at one of them, but which one?