Lab 2: July 17, 2019

Subjects to review this week: functions, string manipulation, classes

Learning outcomes:

1. Define key terms and ideas from Ch. 6-8 of course lectures
2. Gain experience manipulating fairly complex string variables
3. Write mathematical functions to manipulate numeric variables
4. Develop basic skills for scraping data from the online public domain.
5. Comment/manipulate/execute code written by economists to understand results classic urban/public economics model

Big picture: last week most people seemed comfortable coding up simple tasks in Python (i.e. things that require maybe <5 lines of code). Hopefully, you've also figured out what kind of resources are available online if you aren't sure how to tackle a problem. We're going to ramp up the complexity in this lab. You're going to be dealing with bigger and messier data objects. You will have to import modules (I'll provide hints as to useful tools that are worth investigating) and rely on Google/StackExchange to figure out how they work.

You guys are a team! Work together when figuring out how to solve problems. As always, this lab is learning-by-doing!

Part I: Key terms (~15 minutes)

As before: please write out (on paper, in .txt file, etc.) a couple of sentences highlighting the most important or course-relevant aspects of these terms:

- Practical uses of *enumerate()*
- Input vs. output of functions
- Local vs global variable
- Function stub
- Code block
- Global space vs heap space vs call frame
- String methods: find/replace, counting, comparison, splitting, joining
- ***Regular expressions*** (Also known as ***regex***. Super useful tool in python, R, and from the command line as well. See this website for primer)
- Class vs. method vs. function
- Constructor
- Instantiation

Part II: Scraping Wikipedia and string manipulation (~45 minutes?)

As you likely know, it's incredibly straightforward to scrape data from websites' html in Python; analysts/developers/nerds use the language for this all the time.

We're going to scrape data from here: https://en.wikipedia.org/wiki/List_of_counties_in_Wisconsin. Notice the table at the bottom the page. We want to extract several elements of the table into lists of data.

Your tasks are the following:

1. Create a list called *county_names* containing county names. Make sure to drop the word "County" from all strings – i.e. "Adams County" should only be "Adams" in your list.

2. Create a list of state+county FIPS codes called *county_fips*. Wisconsin's state FIPS code is 55. Your list should contain a 5-digit numeric value for each county. For example, the Adams County FIPS code would be 55001. Dane County would be 55025. And so on…
3. Create a list of county populations called *county_pops*. Again, this should be a numeric list.
4. Create a list of county spatial areas called *county_area*. Make sure the area is in kilometers, and a numeric value.
5. Write a simple function to return a county's population per kilometer. Then using your new function, create one final list that contains each county's population per kilometer.

Hints: there are a number of modules that will be helpful here. The standard for scraping is called ***BeautifulSoup***. You're also going to want to use ***requests***. A super basic primer on using these packages while scraping Wikipedia can be found at this website – please give it a read to get started!!! As usual, Google will also be your friend. We will start using the ***pandas*** module quite a bit in coming weeks, but you shouldn't necessarily need it today to complete the tasks above.

Part III: Writing functions to manipulate scraped numeric data (~45 minutes?)

Your task here is to begin scraping historic data about wind patterns at North American airports and shape them into data formats that are more appealing for analysis. The US FAA requires airports to collect and publish rich information (formally called METARs) about wind patterns at high frequency throughout the day so that planes can safely navigate the skies. A private entity has compiled and aggregated this METAR data for the years 2006-2010; the result is the website windhistory.com.

As you can see at the link above, there is information for several thousand airports in the US and beyond. Play around with the site a bit to familiarize yourself with the data available.

When you're ready, tackle the following tasks and questions:

1. If you were to work through the site's html code, you would likely come across its airport index file. This file holds identifying info about each airport in the data sample. I've done some leg work for you and provide it here: http://windhistory.com/data/stations-md.json.

    a. What data type would this .json file represent in Python? Notice its formatting… what does it look similar to?

    b. Write a simple script bring this file into Python as a object called *raw_index_data*. What are the object's keys?
        i. Hint: this .json file can be scraped into an object easily using the following modules: ***json, usrlib.request***. Useful methods will be *json.loads* and *usrlib.request.urlopen()*…

    c. From this index file, we want to extract three pieces of information: each airport's ID (i.e. the four letter combination that identifies an airport), its latitude, and its longitude. Create a list of three lists for each of these variable called ***airport_info***. These sub-lists should have a length of 2535.

2. Your next step is to write a function that calculates the distance (in km) from each airport to the Computer Science building in which we are currently working!

    Our current latitude/longitude is approximately: (43.07152, -89.40652). Return a list of distances, called ***airport_cs_dist***, and then append it *airport_info*.

3. Now generalize your efforts from task 2. Use your function to create a list of lists called ***airport_dist_matr*** that contains the distance between every airport in *airport_info*. Essentially,

you are creating a distance matrix/array using only lists; if you reshaped your lists into a square here, the diagonal of your new distance matrix should always be zero because the distance between an airport and itself is zero.

In other words, for the first airport in your list ("KDDC"), you should return a sub-list of distances to itself and all other airports in *airport_info*. Same thing for the second airport in the list, and so on…

4. For this final task, **only pseudocode is required**. Feel free to attempt coding it in your free time, but I'm almost certain the effort would take longer than we have in lab today.

   Using the list of airport IDs you created in task 1, we can access monthly average wind direction counts and wind speeds for each 10-degree arc of the so-called "wind rose" using the website's data architecture. For instance, we can access this monthly, 10-degree arc data for "KDDC" at the link: http://windhistory.com/data/KDDC.json. A symmetric naming scheme follows for all other airports.

   Your job is to aggregate the website's data into annual measures of wind frequency (% of time the wind blows from that direction) and average wind speed for each airport in the sample. Start by creating these measures for each 10-degree arc, but then aggregate into 45-degree arcs representing: N, NW, W, SW, S, SE, E, NE. **Again, only pseudocode this**. Think carefully through the structure of website's data and how it affects your approach.

Part IV: Schelling's model: Python simulation using classes, functions, and modules (~45 minutes)

In this question, you will execute a simple *agent-based simulation* that captures the main thrust of ideas written about by Thomas Schelling in the late 1960s and early 1970s.

Schelling's model argues that if groups of people have even a small taste/preference for living in proximity to their group, segregation is likely to occur if you model a spatial equilibrium. Since this is a general result, you can apply it to your grouping characteristic of choice: race, age, education level, job type, income, etc. This is an important idea for a lot of economic and sociological research – indeed, we see these sorting behaviors all the time in the real world. These sorting patterns overlap in various ways that create headaches for empirical researchers seeking causal effects… but that's another subject.

In any case, this model one of the standard initial pedagogical examples for agent-based simulation, so there is code available online in many different languages. You should have a file called ***schelling.py***. Open it in Eclipse. For this code to run, you must import the module ***matplotlib*****,** so add it your library path from online if needed – we will use it plenty more for this course in the next couple weeks.

Your tasks are the following (read before starting as some can be done simultaneously):

1. In a .txt file, write out a list of the code's functions. Describe what each function's purpose is in a single sentence. Which belong to a class and which don't?

2. Go through the code and write comments that detail carefully what the code is doing – especially loops, conditionals, and counts. No need to comment every line, but it should be possible to read only your comments and know **exactly** what the code is doing.

3. Run the code. In a paragraph or less, explain the results using economic notions.

4. Does anything interesting happen if you double the count of agents of type 0? What about if you increase the number of agents regarded as neighbors? What if agents require a different number of neighbors to be of the same type? In a paragraph or less, report back on any general findings you notice while tinkering with the model's key parameters.

5. Finally, ***if you have time remaining***, do your best to extend the model to a case with three groups rather than two. In other words, add another group type to the model. Create a new file ***schelling3.py***, copy and paste in the original two-group code, and edit it as needed.