Lab 3: July 24, 2019

Subjects to review this week: exceptions, input/output data, Git/GitHub, Jupyter notebook

Learning outcomes:

1. Write code that properly incorporates the use of exceptions
2. Use Python to input/output data files in a number of forms
3. Confirm understanding of how use of *pip3* and *pydoc3* can streamline installation/acclimation when running new Python packages
4. "Git" your computer linked up with the course's GitHub lab repository.
5. Set up and play around with Jupyter notebook on your local machine.

Big picture: results from last week's lab and the first exam suggest you guys are up and operational in Python. This week, we add some tools to your coding kit. As most coding projects rely on team-based efforts, version control is key. Git is one of the most widely used version control systems available – it's likely that most of you have borrowed (ahem… stolen) code from GitHub for your own projects. We'll get you set up and familiarized with running it from the command line. Similarly, Jupyter notebook is the Pythonic version of *literate programming* – the combination of prose, equations, and code blocks into a single "document" to be read for demonstration, research, or teaching objectives.[1]

Because we're setting up stuff on your personal computers today, everyone please submit this lab separately. It's fine if you work as a team, but everyone needs to submit their own files on GitHub.

Part I: Setting up Git/Github

In what follows, I'll describe how to get started with Git. If you already have Git installed on your system and a GitHub account, skip ahead to gaining access to the course repository so you can work with this week's lab materials.

So why use Git?[2] Two primary reasons: first, it keeps detailed track of every file change we make, noting when and what exactly was modified and who made the modification. Second, it eases collaborative merging of files in a team setting; coders can work on different versions of the same file at the same time, then merge to an updated, synchronized version.

Essentially, Git is a powerful automated version control software. If you make local files changes that are catastrophic, or accidently delete an important file, the software can recover an older version of the repository (equivalent of a project folder) for you. If your computer melts down and you've saved your local repository to the cloud (i.e. GitHub or the like), your files will be safely waiting there for you, with a recorded history of versions to choose from.

---

[1] Analogous to RMarkdown, for those of you taking the ML course or with previous experience.

[2] I draw on some material from Cornelia's research group notes in this section. I also draw on Git material from Grant McDermott's fantastic R data science course at U Oregon: https://github.com/uo-ec607/lectures.

(Vaguely related for R connoisseurs: Ed Rubin (also @ U Oregon) has an online econometrics course that ties in nicely to the above course: https://github.com/edrubin/EC525S19)

Work through these steps to get going:

1. If you don't have a GitHub account, create one here: https://github.com/join. If you do have an account, log in!
    a. **Once you've signed up, please tell me your username so I can invite you to collaborate on the course repository.**
2. If you haven't installed Git yet, download it here: https://git-scm.com/downloads.
    a. Note: MacOS users, you can (1) check for it and (2) download automatically if missing at the command line by typing: `git --version`
3. After download, next step is to initialize Git from command line.
    a. First, set your user name: `git config –global username "YOUR NAME HERE"`
    b. Next, your user email: `git config –global user.email "EMAIL HERE"`
4. You should have by now received an invitation to collaborate on: **aae875_summer2019_labs.** Accept it and enter the repository. Click on the green button that says "Clone or download" and copy the HTTPS address link.
5. From your command line, navigate to a location where you want to store the course repository locally. If you have a folder where you've been keeping materials for this class, that would be a good place. Once you're there, we're going to set up your local repository…
    a. Step 1: we need to clone the GitHub (public) repository and place it into the location you select. From the command line, type: `git clone HTTPS_link_here`
        i. Enter your GitHub username/password (*credentials*) when they are requested.
        ii. Once the computer completes this command, all the files from the GitHub repository should be in a git-maintained local folder on your system. Your local version is identical to the most recent public version of the repository.
    b. Step 2: To ensure everything is functioning, let's make some (for now temporary) changes to your local repository. Create a new folder in the repository that is named after you (as an example, there is already a folder in the repository called *AdamT)*. Once in your named folder, open your favorite text editor and type your name and email-address. Save this file as *myinfo.txt*.
    c. Step 3: Add your repository updates to the repo history. To add ***all*** updates you've made, type in the terminal: `git add –A`
    d. Step 4: Tell Git that you are ready to commit to these proposed changes – this is the equivalent of saving your repository locally. Make sure to briefly describe your changes: `git commit –m "Describe changes here inside the quotes"`
    e. Step 5: Let's suppose you want to send your updated files to the public repository on GitHub. Best practice is to (1) ***pull*** any new changes from the GitHub repository since your last update, and then (2) ***push*** your changes to the public space. In terminal:

```
git pull
git push
```

        i. Let me know if these return errors (they normally shouldn't, but we can troubleshoot as needed).
        ii. If no errors, check the GitHub repository online. Are your files there? Are your classmates' files there? Welcome to the world of Git.

So as a review, generally, the best practice while using git from team projects:

(1) Make local changes → (2) Stage (**add**) these changes → (3) Commit (with comments) → (4) Pull from GitHub repo in case others made changes → (5) Push your changes to GitHub

You can up the flexibility with Git/GitHub: branches, forks, etc. We'll leave those for you to learn as you go, and as needed for your various projects. Do know that you can create your own private (locally/or on GitHub) repositories fairly easily from the command line: variations on `git init`

Additionally, we only worked with Git from the command line – this is often the most flexible way to set up and use repositories. It is also fairly straightforward to set these up directly within your IDE of choice. See some Eclipse directions here: https://www.vogella.com/tutorials/EclipseGit/article.html. Some directions for RStudio here: https://support.rstudio.com/hc/en-us/articles/200532077-Version-Control-with-Git-and-SVN.

Part II: Setting up Jupyter notebook

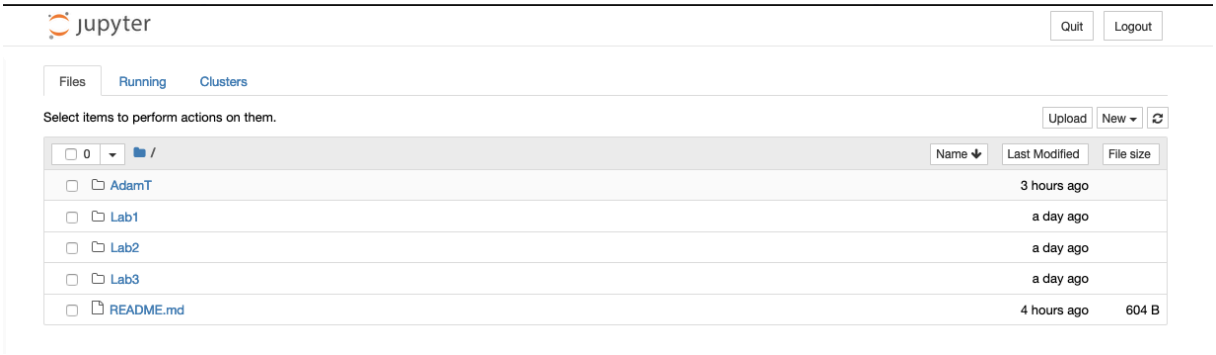
Next up today is setting up Jupyter notebook. A couple of you have asked about this since you are familiar with RMarkdown. This is the Python equivalent (though Jupyter also can read code in R, Julia, Ruby, etc. – quite flexible!), and is a really nice tool when you're working on a project collaboratively and want to organize code, comments, math models, and output into a single human-readable file. Another nice feature of this app is that you can write the output to many useful formats: .txt, .py, .html, .tex, .pdf, etc. (Side comment/FYI: Notebook will be fully replaced by JupyterLabs eventually. The learning curve to migrate doesn't look too high, fortunately…)

If you're curious about what notebooks can look like, take a couple minutes to explore this website and see what is possible: https://github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks#machine-learning-statistics-and-probability.

Installation is simple – as usual. From the command line: `pip3 install jupyter`

Make sure your current directory is still the Git repository that we set up in Part I above. Calling the notebook (also from the terminal) is the moment of truth – it should open the application without error, but we can troubleshoot issues if they arise (had some on Mac, myself, while reinstalling the latest version): `jupyter notebook`

If successful, a local server will be instantiated by Jupyter, and a window should pop-up in your default browser. Since you opened the notebook while in your local course lab repository, your interface should look roughly like the following:



Click on your named folder that you previously, and then create a new Python 3 notebook inside. The interface is pretty straightforward so hopefully you can figure out how to do this. **This notebook will be your lab document for the day – so name it accordingly**! (No Eclipse for what follows. All should be done in this notebook – which you will save to local repository and upload to GitHub at the end of Lab for grading).

You'll quickly learn how to the notebook works as you progress through the rest of this lab (it's super intuitive), but if you want help to start, here are some basic references:

- Really simple demo notebook is available in the lab repository – check in the folder *AdamT* and open the file *SampleNotebook.ipynb*
- Keyboard shortcuts: http://maxmelnick.com/2016/04/19/python-beginner-tips-and-tricks.html

Part III: Key terms

This week, the key terms are (data analysis/econometrics/optimization)-related packages I think you should be aware of.

In your new .ipynb file on Jupyter, provide Markdown descriptions of the following packages under the heading "Python packages of interest". For each package, write a ~2 sentence summary highlighting the package's main uses. Prioritize the coding exercise below, but if you have time, google around and include hyperlink(s) to examples of the package in use so that you can reference them later.

1. NumPy
2. SciPy
3. PANDAS
4. Matplotlib
5. Plotly
6. Seaborn
7. Statsmodels
8. Scikit-learn
9. PyStan
10. PySpark

Part IV: Coding exercises

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. When using Jupyter notebook to write up your code, be liberal in commenting about your work process in Markdown:

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 an object called *raw_index_data*. What are the object's keys?
        i. When writing this simple script, make use of an exception (try…) so that if the website hosting this file were ever down, your code would still run without error.
        ii. 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. (Hint: if you're comfortable "learning on the fly" here, use pandas to set up ***airport_info*** as a *pandas* DataFrame!)

2. Your next step is to write a function that calculates the birds-flight 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 (i.e. an array) called ***airport_dist_matr*** that contains the distance between every airport in ***airport_info***. Essentially, you are creating a distance matrix using only lists; if you reshaped your lists into a square here, the diagonal of your new distance matrix would 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. In the GitHub repository, there is a file called *latlon2fips.py*. The function in this file takes a lat/long combination as inputs and returns the FIPS code of the county where the location resides. (For those interested, it does this by calling the FCC's census block location API…) Putting this function into your own code and then calling it, create a list of FIPS values for each airport, and then append it to ***airport_info***.
   a. You might quickly notice the API call starts to take a while for 2000+ airports. It may be wise to output ***airport_info*** to a .csv file as soon as the FIPS addresses have been appended; this way you can avoid rerunning this slower code over and over.

5. Last week in lab, you scraped data from Wikipedia on Wisconsin counties. The result is the file *wi_counties.csv*. Input this file as a list/DataFrame object ***wi_counties*** using a Python method of your choice.

6. Using the objects ***airport_info*** and ***wi_counties***, create a new "merged" list/DataFrame object ***wi_airport_info*** that:

   a. Contains a sub-list for each available variable (*airport_id, airport_lat, airport_long, airport_cs_dist, airport_fips, county_name, county_pop, county_area, county_pop_per_km)*
   b. Only contains airports that are located in Wisconsin.

   Save/output this new object as *wi_airport_info.csv*.

7. **If time remains, write up a summary of how you would tackle this problem -- 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 left 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.

When you're done with these tasks and have written the code up nicely in your Jupyter notebook (export it as a .pdf file, please!), save all work in the folder named after you in your local repository, and then add/commit/pull/push changes to GitHub so your work can be graded.