Problem Set 1

Due Wednesday Sep. 10, 10 am

Comments

- This covers material in Units 2 and 4 as well as practice with Quarto.
- It's due at 10 am (Pacific) on September 10, both submitted as a PDF to Gradescope as well as committed to your GitHub repository.

Formatting requirements

1. Your electronic solution should be in the form of an Quarto file named `ps1.qmd`, with Python code chunks. Please see Lab 1 and the dynamic documents tutorial for more information on how to do this.
2. Your PDF submission to Gradescope should be the PDF produced from your qmd. Your GitHub submission should include the qmd file, your Python module file for Problem 4, your environment file (see problem 4d) and the final PDF, all named according to the submission guidelines.
3. Your solution should not just be code - you should have text describing how you approached the problem and what the various steps were. Your code should have comments indicating what each function or block of code does, and for any lines of code or code constructs that may be hard to understand, a comment indicating what that code does.
4. You do not need to (and should not) show exhaustive output, but in general you should show short examples of what your code does to demonstrate its functionality. Please see the grading rubric, and note that the output should be produced as a result of the code chunks being run during the rendering process, not by copy-pasting of output from running the code separately (and definitely not as screenshots).

Problems

1. Please read these lecture notes about how computers work, used in a class on statistical computing at CMU. Briefly (a few sentences) describe the difference between disk and memory based on that reference and/or other resources you find. If you're doing an data analysis, you should have an understanding of what situations might lead to running out of disk space and what situations might lead to running out of memory.
2. This problem uses the ideas and tools in Unit 2, Sections 1-3 to explore approaches to reading and writing data from files and to consider file sizes in ASCII plain text vs. binary formats in

light of the fact that numbers are (generally) stored as 8 bytes per number in binary formats.

a. Generate a numpy array (named `x`) of random numbers from a standard normal distribution with 20 columns and as many rows as needed so that the data take up about 16 MB in size.

b. Explain the sizes of the two files created below. In discussing the CSV text file, how many characters do you expect to be in the file (i.e., you should be able to estimate this reasonably accurately from first principles without using `wc` or any explicit program that counts characters). Hint: what do we know about numbers drawn from a standard normal distribution?

```
import os
import pandas as pd
x = x.round(decimals = 12)

pd.DataFrame(x).to_csv('x.csv', header = False, index = False)
print(f"{str(os.path.getsize('x.csv')/1e6)} MB")

pd.DataFrame(x).to_pickle('x.pkl', compression = None)
print(f"{str(os.path.getsize('x.pkl')/1e6)} MB")
```

```
30.778114 MB
16.000573 MB
```

Suppose we had rounded each number to four decimal places. Would using CSV have saved disk space relative to the pickle file?

c. Now consider saving out the numbers one number per row in a CSV file. Given we no longer have to save all the commas, why is the file size unchanged?

d. Read the CSV file into Python using `pandas.read_csv`. Compare the speed of reading the CSV to reading the pickle file with `pandas.read_pickle`. Note that in some cases you might find that the first time you read a file is slower; if so this has to do with the operating system caching the file in memory (we'll discuss this further in Unit 7 when we talk about databases).

e. Finally, in the next parts of the question, we'll consider reading the CSV file in chunks as discussed in Unit 2. First, time how long it takes to read the first 10,000 rows in a single chunk using `nrows`.

f. Now experiment with the `skiprows` to see if you can read in a large chunk of data from the middle of the file as quickly as the same size chunk from the start of the file. What does this indicate regarding whether Pandas/Python has to read in all the data up to the point where the chunk in the middle starts or can skip over it in some fashion? Is there any savings relative to reading all the initial rows and the chunk in the middle all at once?

3. Please read the Code syntax and style section of Unit 4 on good programming/project practices and incorporate what you've learned from that reading into your solution for Problem 4. (You can skip the section on Assertions and Testing, as we'll cover that in Lab.) In particular, lint your code (e.g., using `ruff` or another tool of your choice as discussed in the Unit 4. (This is most straightforward for code in a .py file as set up in Problem 4d. I don't know how to lint

code in a qmd file, but if anyone figures out a good way to directly lint the code chunks, please post on Ed!)

As your response to this question, very briefly (a few sentences) note what you did in your code for Problem 4 that reflects what you read. Please also note anything in what you read in Unit 4 that you disagree with, if you have a different stylistic perspective.

4. We'll experiment with using a web API, in this case the GitHub API. As discussed in part (d) you'll create a module for your solution. For showing your code in your solution, please use `inspect.getsource()` to show us the code within the answers to subparts (a) and (c) rather than having your full Python module as an appendix.

   a. Consider the GitHub repository for the `numpy` Python package (https://github.com/numpy/numpy). Write a function to use the GitHub "commits" API to get the commits made in the repository (by constructing and submitting http GET requests) and return the results as a Pandas (or Polars) dataframe with a reasonable set of fields. Your function should work on any public repository, but you'll run it on the numpy repository. Note that the API limits the number of returned values – make sure you are getting the maximum of 100 results. That's fine for what we're doing here, but see the optional part (f) for how to get more than that.

      Notes:

      - GitHub limits you to 60 requests per hour. So don't make requests too frequently, which may be easy to do as you debug your code. A good thing to do is to get the commits information initially (perhaps with a request made during an initial interactive Python session) and save the resulting object in a pickle file that you can then load into Python for use in developing the rest of the code. Ultimately, though, you'll want to test the full workflow, including the initial request to make sure it all works together (e.g., on some other repository). You don't need to write unit tests at this point (we'll add those in PS2), but do check that it works for a second case.
      - The result for each commit is a complicated object, and there is some heterogeneity in what information is provided for the different commits, so it will take a bit of work to figure out what to extract and put in the data frame.
      - You can extract the information just based on manipulating Python dictionaries, but you might want to explore using JSONPath, which is similar to the XPath language mentioned in Unit 2. For example, using the `jsonpath-ng` package, one can do queries like this: `expr = parse("$.catalog.book[*].author"); authors = [match.value for match in expr.find(data)]` to find the values associated with the `author` key within all the books within the overall catalog if you had a book catalog in JSON format (i.e., information like the book catalog XML example in the notes, but in JSON format instead).

   b. Make a plot showing a histogram of the number of commits per user. You can use `matplotlib` or whatever Python plotting package you want. The plot should be clearly formatted/labelled.

   c. Write a function that finds the committer who has made the most commits and gets user information about that user by using the appropriate GitHub API call. It should return the information. It should also print out a short message with some key info about the user,

but there should be an argument to the function that can turn the message off (i.e., control the "verbosity").

d. Now make sure that you have all your code in a module and that you can import the module and run functions to find the most active developer for any GitHub repository one might be interested in. This should be simple if you've set everything up in parts (a)-(c).

e. Create a requirements file (based on either pip or Conda) that has the necessary information (in particular Python package versions) to reproduce the environment in which you ran your code. Include this file in your GitHub repository directory for this problem set.

f. (extra credit) If you'd like extra practice, modify your function so that you get *all* of the commits, or at least thousands of results. Make a time series plot showing how the number of commits has varied over time (either over years or over the months of the year).