Problem Set 2

Due Friday Sep. 25, 10 am

Comments

- This covers material in Units 3 and 4 (up through Section 9).
- It's due at 10 am (Pacific) on September 25, both submitted as a PDF to Pensive as well as committed to your GitHub repository.

Formatting requirements

1. Your electronic solution should be in the form of an Quarto file named `ps2.qmd`, with Python code chunks. If you want to initially work in a Jupyter notebook, that is fine, but you'll need to run `quarto convert file.ipynb` to generate a qmd file before rendering to PDF and submitting.
2. Your solution should not just be Python 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.
3. 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. This problem uses the ideas and tools in Unit 2, Section 5 and Unit 4 Section 2 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. Then 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 polars as pl
import os
x = x.round(decimals = 12)
df = pl.DataFrame(x)
df.write_csv('x.csv', include_header = False)
print(f"{str(os.path.getsize('x.csv')/1e6)} MB")
```

```
30.777383 MB
```

```
df.write_parquet("x.parquet", compression = "uncompressed")

print(f"{str(os.path.getsize('x.parquet')/1e6)} MB")
```

```
16.006827 MB
```

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

b. Read the CSV file into Python using `polars.read_csv`. Compare the speed of reading the CSV to reading the Parquet file using Polars. As a side note, the speed of reading the data with files you just created will likely be a lot faster than reading a file that had been sitting untouched on the filesystem for a while. This has to do with the operating system caching the file in memory (we'll discuss this further in Unit 5 when we talk about databases).

2. Let's investigate the structure of the `pandas` package to get some experience with the structure of a large Python package and with how `import` and the `__init__.py` file(s) are used. You'll need to go into the Pandas source code (see Unit 4). Note that the main `__init__.py` and the `__init__.py` files in the subpackages/submodules are complicated, and I'm not expecting you to understand everything about them. Also note that the following cases involve functions, classes, and class methods. Be sure to be clear to say which of those it is and for the class methods cases, make sure you're clear on what class the method is part of and any class inheritance structure. Run `import pandas` and then consider the following questions:

   a. Consider `pandas.core.config_init.is_terminal`. What namespace is it (or its class) in in your Python session? What file/module is `is_terminal` in on disk? Is it a function, class, or a class method (and if a class method what is the class)? Describe how it is imported by discussing the relevant statement(s) in the relevant `__init__.py` file(s).
   b. Consider `pandas.read_csv`. Answer the same questions as for (a).
   c. Consider `pandas.arrays.BooleanArray`. Answer the same questions as for (a).
   d. Consider `pandas.DataFrame.to_csv`. Answer the same questions as for (a).

   Hints: (1) `grep -R <pattern> <directory>` will search all files within a directory recursively. (2) As you work on this, you may want to be able to modify one or more of the `__init__.py` files to better understand what is happening (e.g., by commenting out a line of code or adding a print statement). A good way to do this is to create a Conda environment in which pandas is installed, so you isolate any changes you make, e.g., `conda create -n test_env python=3.13 pandas`. Then you can edit code files in the environment and when you start Python and import pandas, you should see the effects of your changes. Alternatively, you could use the debugger to set breakpoint(s) in an `__init__.py` file. (3) Or you might create your own small toy package

to experiment and see how things work with nested `__init__.py` files and various ways to use `import`.

3. Data Science for Love, Python+AI style

   The overall goal is to redo your work from problem 2 in PS1 in Python, adding good error trapping and testing.

   Use an AI-assisted coding tool to help solve the problem. You can use a CLI tool, Desktop app, or VS Code extension (many of the tools provide all of these as options). GitHub Copilot in VS Code is pretty easy to set up and can make use of GitHub for Education (see here to apply) to use GitHub Copilot Pro, providing more free queries and use of premium models. But you're welcome to use whatever tool you want, such as Claude Code, Codex, Positron Assistant, OpenCode, and Gemini Code Assist. You should be able to do the assignment using the free tier of whatever tool you choose.

   Approach: I frame the problem as leaning heavily into AI and assessing/improving/iterating/checking/timing what the AI provides, with the main points being (1) to get experience with AI assistance and (2) to get more experience with writing, organizing, and testing good Python code, critically assessing what AI produces. However, if you feel that what you get from the AI assistance is not that helpful or if you prefer to simply practice writing Python code without AI assistance (or with more limited assistance at a syntax level or line-by-line level) you can do that instead.

   As a general approach, please build up your solution in stages. Start simple, check results, add tests and then add complexity. AI could probably do fine with this fairly simple problem all at once, but for you to make sure you understand the Python code being produced and do a good job of checking correctness, you should do it in stages.

   To start, write up a very careful, specific, detailed prompt saying what you want (and possibly some things you don't want). Tell it what to do now and what not to do yet (e.g., wait on tests and plotting). You can choose to either work out the scaffolding of the solution (namely what the functions are and how they work together) or have AI come up with that and then iterate to improve it as needed.

   Requirements:

   - Don't use any shell commands directly or by calling the shell from Python. Instead use Python calls, including for any filesystem operations. (I think this should be possible, but if you run into something you can't do in Python and need to make a shell call from Python (e.g., using `subprocess.run`), please post on Ed.)
   - Have your Python code be callable from the shell as `./get_weather.py` with arguments handled using `argparse`.
   - When called from the shell, your Python function should produce an output file (name given by user) and/or a boxplot as requested by the user.
   - In addition, you should be able to import your .py file as a module and call the individual functions to do the work from within Python, with the data stored in a Polars dataframe (please use Polars rather than Pandas to get experience with that newer package that many people like). To do this, the code that is run when running as a script should be inside an `if __name__ == "__main__":` code block.

- Include error trapping of various sorts. Think broadly about what could go wrong when a user actually tries to run the code, working with the AI tool to figure out unexpected/corner cases that should ideally be dealt with.
- Set up tests of the Python functions using `pytest`, including testing the error trapping.
- As in Problem Set 1, avoid repeatedly downloading the same data files during your development process. One way to help with this is to have an argument that indicates whether the input (yearly) data files are already stored (cached) in a particular directory.
- Make sure your code follows best practices noted in Unit 3.
- Your code will likely read individual files into memory in Python before filtering to the data of interest. This is fine though it has the disadvantages we've discussed when doing things line by line in the shell.
- In your final submitted code, there should not be any AI-produced code that you don't understand **in detail**. This is both to avoid errors and as a way for you to learn from the AI-generated code.

a. Provide your complete solution, including example results, tests and results of running tests.
b. Discuss your experience with using AI: briefly summarize your workflow in developing your solution and then briefly describe what you thought worked well or not so well and any areas where the AI tool didn't do a good job (or did a particularly good job, perhaps suggesting something that didn't occur to you). Describe a few things that you learned about Python or useful Python packages from looking through the code that the tool generated.

4. *Memoization* of a function involves storing (caching) the values produced by a given input and then returning the cached result instead of rerunning the function when the same input is provided in the future. It can be a good approach to improve efficiency when a calculation is expensive (and presuming that sometimes/often the function will be run with inputs on which it has already been used). It's particularly useful for for recursive calculations that involve solving a subproblem in order to solve a given (larger) problem. If you've already solved the subproblem you don't need to solve it again. For example if one is working with graphs or networks, one often ends up writing recursive algorithms. (Suppose, e.g., you have a geneology giving relationships of parents and children. If you wanted to find all the descendants of a person, and you had already found all the descendants of one of the person's children, you could make use of that without finding the descendants of the child again.)

   a. Write a decorator that implements memoization. It should handle functions with either one or two arguments (write one decorator, not two!). You can assume these arguments are simple objects like numbers or strings. As part of your solution, explain whether you need to use `nonlocal` or not in this case.

   b. Try your code on basic cases, such as applying the log-gamma function to a number and multiplying together two numbers. These may not be useful cases for memoization, because the lookup process could well take more time than the actual calculation. To assess that, time the memoization approach compared to directly doing the calculation. Be careful that you are getting an accurate timing of such quick calculations (see Unit 4 notes, looking ahead to Section 11).