Problem Set 2

Due Friday Sep. 15, 10 am

Comments

- This covers material in Units 3 and 4.
- It's due at 10 am (Pacific) on September 15, both submitted as a PDF to Gradescope as well as committed to your GitHub repository.
- Please see PS1 for formatting and attribution requirements.
- Note that using chunks of bash code in Qmd may be troublesome.
  - You will need to add `engine: knitr` to the YAML preface of your qmd document. The `jupyter` engine won't work unless you install the Jupyter bash kernel, and even then you can't mix bash and Python code chunks.
  - For the `knitr` engine, you'll need to have R installed on your computer, including the `knitr` package. Quarto will then process the code chunks through `knitr` (which will use the `reticulate` package to handle Python chunks).
  - If you have trouble on your own computer, you can always render your solution for this problem set on an SCF machine (we won't generally use bash chunks in future problem sets). (In particular I'm not quite sure what will happen if you render on Windows.)
  - We can help troubleshoot and feel free to post on Ed.
- You will probably need to use `sed` in a basic way as we have used it so far in class and in the bash tutorial. You should not need to use more advanced functionality nor should you need to use `awk`, but you may if you want to.

Problems

1. This problem provides practice using shell scripting. We'll use United Nations Food and Agriculture Organization (FAO) data on agricultural production that we saw in class. If you go to http://data.un.org/Explorer.aspx?d=FAO and click on `Crops`, you'll see a bunch of agricultural products with `View data` links. Click on "apricots" as an example and you'll see a `Download` button that allows you to download a CSV of the data. In class we inspected the HTTP requests that the site handles (using the Network information in the browser Developer Tools) and found out that you can download a file directly via a URL that sends a GET request in the following format: `http://data.un.org/Handlers/DownloadHandler.ashx?DataFilter=itemCode:526&DataMartId=FAO&Format=csv&c=2,3,4,5,6,7&s=countryName:asc,elementCode:asc,year:desc`. That downloads the data for Item 526 (apricots). You can see the `itemCode` for other products by hovering over `View data` link for the relevant product.

a. Using the shell to do the following. Download the data for apricots. Extract the data, excluding the metadata and the global total, into another file. Then subset the data to the year 2005. Based on the "area harvested" determine the ten regions/countries using the most land to produce apricots. Now automate your analysis and examine the top five regions/countries for 1965, 1975, 1985, 1995, and 2005. (Do not just copy and paste your code one time for each of the years!). Everything you do should be done using shell commands you save in your solution file. So you can't say "I downloaded the data from such-and-such website" or "I unzipped the file"; you need to provide the bash code that we could run to repeat what you did. This is partly for practice in writing shell code and partly to enforce the idea that your work should be reproducible and documented.

   Warning: You may need to put the URL inside double quotes when using a UNIX shell command to download it. Also make sure there are no carriage returns in the URL (I had to break the URL above to fit on the page).

   Tip: dealing with the comma separation in the file can be a problem as there are commas used inside character strings as well. You'll probably want to convert the file to be delimited by something other than a comma - you can do this from the command line or if you would like to do it via Python, you can run `python` from the command line to run a little bit of Python code to do the conversion.

b. Write a bash function that takes as input a single item code (e.g., 526 for apricots, 572 for avocados), downloads the data, and prints out to the screen (i.e., to `stdout`) the data stored in the CSV file, such that the information could be piped to another UNIX command. Your function should detect if the user provides the wrong number of arguments and return a useful error message. It should also give useful help information if the user invokes the function as: `myfun -h`.

   Warning: the exact syntax for `if` statements in the shell can be finicky about spaces.

2. Add documentation, error-trapping and testing for your code from Problem 4 of PS1. You may use a modified version of your PS1 solution, perhaps because you found errors in what you did or wanted to make changes based on Chris' solutions (to be distributed in class on Monday Sep. 11) or your discussions with other students. These topics will be covered in Lab 2 (Sep. 8) and are in some new material (as of Sep. 6) in Unit 4. Note: given our difficulties with the MLDb server on PS1, you may need to manually download the HTML for various cases that you use in developing and demonstrating your assertions and tests.

   a. Add an informative doc string to your main, user-facing function.
   b. Add a small number of assertions using `assert` as sanity checks of specific conditions that should be true if your code is operating correctly. These are generally intended to help in debugging.
   c. Add exceptions for handling run-time errors. You should try to catch the various incorrect inputs a user could provide and anything else that could go wrong (e.g., what happens if the server refuses the request or if one is not online?). In some cases you will want to raise an error, but in others you may want to catch an error with `try-except` and return `None`.
   d. Use the `pytest` package to set up a small but thoughtful set of tests of your functions. In deciding on your tests, try to think about tricky cases that might cause problems in terms of what the website returns when you search it.