Homework 3: Linux shell scripting and `git`

Due October 16, 11:59 pm

This homework will give you a short introduction to Linux shell scripting and `git`. These tools will be useful in your projects later in the semester, where it will be crucial that you be able to process and clean your data and collaborate with your teammates.

1 Linux Shell Scripting

There are a number of reasons why you should learn to use bash script effectively. For starters, most large-scale computing resources run Linux, so knowledge of the command line gives us access to those resources. Further, bash script makes it much easier to automate repetitive tasks. Shell scripting for I/O and extracting data from text is usually much easier than doing it in R or another programming language. Anecdotally, 80% of a data analyst's time is spent cleaning up data, so any tools that make it easier to process data files are well worth the investment of time to learn them. There are many data science problems that involve so much data that we can't consider a sophisticated model, but a simple statistic (mean, median) or graph can go a long way toward answering a question of interest. The issue becomes, "Can I even read the data?" For a person who can write a shell script to extract a little information from each of many files, the answer is often "Yes." A few years ago, R's `tidyr` and other packages introduced the idea of data pipelines to R programmers, mimicking what the shell has been doing since the 1970s! Shell scripting ideas can thus improve your use of R and other programming languages: write small tools that do simple things well, using a clean text I/O interface.

1. Run `wget http://pages.stat.wisc.edu/~jgillett/605/linux/Property_Tax_Roll.csv` to download a file of property tax data for the City of Madison from 2018 (the original data is from `http://data-cityofmadison.opendata.arcgis.com/datasets/property-tax-roll`.)
2. Write a script, `school.sh`, that finds the average (possibly rounded down– see below) `TotalAssessedValue` for properties in the `"MADISON SCHOOLS"` district, and echos that average, and nothing else, onto the command line. Your script should operate as a single pipeline. Solutions that require writing partial results, for example, by creating "in-between" or "intermediate" files, will not receive full credit. **Hint:** Write a pipeline with these stages:

- Use `cat` to write `Property_Tax_Roll.csv` to `stdout`. (Or, to work with small input while debugging, use `head` to write only the first few lines.)
- Use `grep` to select only those lines containing `MADISON SCHOOLS`.
- Use `cut` to pick out the `TotalAssessedValue` (7th) column.
- Pipe this as the input to a brace-enclosed "group command" that works in a pipe. This one finds a sum:

  ```
  { my_sum=0; while read line; do
      my_sum=$(( $my_sum + $line ));
  done; echo $my_sum; }
  ```

  The `while read x` pattern creates a while loop which, in each iteration, reads the next line from the while-loop's "stdin" and stores it in a variable `$x`. Revise the group command above so that it finds a mean instead of a sum. Note that the variable being updated inside the while loop must be declared inside of the curly braces.
- Once the while-loop terminates, you'll have a sum and a number of lines. We need to divide one by the other to compute the average. The division operator `/` in bash math evaluation performs *integer division*. That means that, for example, `$(( $my_sum / $num_lines ))`, returns an integer (i.e., throws away the remainder), even if `num_lines` does not divide `my_sum`. Don't worry about that– this is why we said it's okay to return the answer rounded down. In a later lecture, we will see a way to perform floating-point (i.e., decimal) division.

3. Write a script, `digits.sh`, to count how many of the numbers between 1000 and 2000 (inclusive) that contain 2 in one or more of their digits, and print the number to standard out.

   Hint: read the man page for the command `seq` to see a simple way to generate the numbers 1000 through 2000. In a few lectures, we will see another way to do this directly without using a command. Pipe the output of `seq` into `grep`, then to `wc`.

4. Write a script `ten_dirs.sh` that does these tasks:
   - make a directory `ten`
   - make ten subdirectories `ten/dir01`, `ten/dir02`,... through `ten/dir10`. Note the formatting `dir01`, `dir02`, etc., rather than `dir1`, `dir2`, etc. Read the `seq` man page to see how to do this easily.
   - in each subdirectory, make four files, `file1.txt` through `file4.txt`, such that `file1.txt` has one line consisting of the digit `1`, `file2.txt` has two lines, each consisting of the digit `2`, `file3.txt` has two lines, each containing the digit `3`, and `file4.txt` has four lines, each containing the digit `4`.

   Note that hard-coded solutions (e.g., writing `mkdir dir01; mkdir dir02; ...`), will not receive full credit. You should use a loop of some kind or another.

   Hint: A convenient way to remove the `ten` directory and all its files is `rm -r ten` (search the `rm` manual page for `-r` to see what it does), so a convenient way to rerun the scrip several times as you develop it is `rm -r ten; ten_dirs.sh`

5. Write a script `rm_n.sh` whose usage statement is `usage: rm_n.sh <dir> <n>` that removes all files in directory `dir` larger than `<n>` bytes. **Note:** By convention for usage statements, the "`<...>`" delimiters around "`<dir>`" indicate a required argument. Square brackets, "`[...]`", indicate an optional argument.

   So `rm_n.sh` takes two required arguments, a directory name and a number. Your script should print a usage message to `stderr`. You may assume that the second argument, `n`, is a number, but your script should check that the first argument is a directory, and print an error message to `stderr` if not. Something like "ERROR: input is not a directory." is sufficient. Your script should also print an error message to standard error in the event that more or fewer than two arguments are supplied. In the event of either of these errors, your script should exit with exit status 1 (see `man exit`). You can try out your script on your `ten` directory from the previous problem by running `rm_n.sh ten 3`.

   Hint: use the command `find`. The man page is 1200 lines long–don't read it all. You just need the `size` argument. Search for "`Numeric arguments`" in the man page (in `less`, you can search by typing a slash followed by your search string).

   A couple of other points:

   - Make sure that you use the script-name variable `$0` in your usage statement, rather than hard-coding the name "`rm_n.sh`". This way, the usage statement will be correct even if you change the script name later.
   - Because the usage statement is for humans to read, and not for further programs in a pipeline, we write it to `stderr`. One way to do this is via `echo`. Normally, `echo` writes to `stdout`. Redirect its `stdout` to go to `stderr` via "`1>&2`" as in `echo "hello" 1>&2`.

6. Write a script, `mean.sh`, with usage statement `usage: mean.sh <column> [file.csv]` (see above for more information on usage statements), that reads the columns specified by `<column>` (a positive number) from the comma-separated-values file (with header) specified by `[file.csv]` (or from `stdin` if no `[file.csv]` is specified) and writes its mean (possibly rounded down). You may assume that the selected column contains numerical data. Here are three example runs:

   - `mean.sh` prints the usage statement to standard error and exits with status 1 (because it did not receive the required argument `<column>`).
   - `mean.sh 3 mtcars.csv` finds the mean of the third column of `mtcars.csv`. (To create the test file `mtcars.csv`, run `Rscript -e 'write.csv(mtcars, "mtcars.csv")'`.)
   - `cat mtcars.csv | mean.sh 3` also finds the mean of the third column of `mtcars.csv`. (Here `mean.sh 3`, with no file specified, reads from `stdin`.)
   - `mean.sh 3 mtcars.csv foo.txt` prints the usage statement to standard error and exits with status 1 (because it received too many arguments).

   Hint: One approach processes command-line arguments and then uses a pipeline:

   - Use `cut` to select the required column
   - Use `tail` to start on the second line (to skip the header)

- Use a pattern like that suggested for `school.sh` above to accumulate a sum and line count. Once the loop terminates, use the sum and count to find the mean, and `echo` it. Once again, since bash's division is integer division, you will actually get the mean rounded down, but that is okay.

To handle reading from `file.csv` or from `stdin`, one approach is to set a variable `file` to either the file specified on the command line or to `/dev/stdin` in the case that the user did not provide `file.csv` on the command line. Then you can read from the `file` variable in either case.

2 Tracking files in git

This brief exercise will give you a chance to get familiar with `git`. `git` is a wildly popular program for collaborating on software projects.

1. Let's begin by creating a git repository.
   (a) Create a directory in which to practice via `mkdir ~/Desktop/gitExample`, then change to the new directory via `cd ~/Desktop/gitExample`.
   (b) Configure git as follows. In the first line, use your name, and in the second, use your email address. You may have already done some of this if you were following along with the lecture video on `git`. That is fine.

   ```
   git config --global user.name "Your Name"
   git config --global user.email "Your email address"
   git config --global core.editor emacs
   git config --list
   ```

   (Vim users should replace `emacs` with `vim`.)
   (c) Create an empty git repository in your directory: `git init`
   (d) Create a file `names.txt` by revising the following command to use your NetID, last name, and first name and then running the command (`names.txt` will be a sort of roster, containing people's names, contact information, and roles): `echo "boss,NetID,FamilyName,GivenName" > names.txt`
   (e) Stage the new file for tracking: `git add names.txt`
   (f) Check the status of your repository: `git status`. You should see `names.txt` in the output.
   (g) Save a snapshot of your file; that is, commit your changes with a descriptive message: `git commit -m "Create names.txt with a boss line."`
2. Now, let's put your repository on GitHub.
   (a) Access a GitHub account (whose account name we'll refer to as ID):

    - If you have a GitHub account, you may use it.
    - If you do not have a GitHub account (or you want a new one), create one:
        i. Visit `https://github.com`
        ii. Click “Sign up”
        iii. Follow the instructions (I chose a free account. I rejected receiving emails.).

(b) On GitHub, create a repository called `gitExample`. The new account dialog gives an opportunity, or you can use the “+ > New repository” menu in the upper right corner.

   i. Choose “Public”, not “Private”.
   ii. Leave “Add a README” unchecked.
   iii. Leave “Add .gitignore” unchecked.
   iv. Leave “Choose a license” unchecked.
   v. Click “Create repository.”

(c) In your `gitExample` directory, to tie your local repository to GitHub, run
`git remote add origin git@github.com:GitID/gitExample.git`
(after changing `GitID` to your GitHub ID).
Note: To make changes to `origin` after setting it, run `git remote set-url origin <URL>`, where `<URL>` is the new URL you want to use.)

(d) Run `git remote -v` to confirm that your remote has been set up correctly.

(e) **Optional.** Set up SSH (Secure Shell) if you want to avoid typing your username and password every time you push or pull.

   i. Generate new SSH keys:
   `ssh-keygen`
   (just hit `Enter` a few times to accept the default answers to the three questions)
   ii. Write the public key to the clipboard:
      i. Install xclip: `sudo apt install xclip`
      ii. Add an alias: `alias clip='xclip -sel clip'`
      iii. Save the alias for future shell sessions:
      `echo "alias clip='xclip -sel clip'" >> ~/.bash_aliases`
      iv. Copy the key to the clipboard: `cat ~/.ssh/id_rsa.pub | clip`
   iii. At GitHub, click the top right (profile picture) menu and choose “Settings,” click “SSH and GPG keys” in the left-hand menu. Click “New SSH key,” paste your key, and click “Add SSH key.” You may be prompted for your Git password, in which case you should enter it.

(f) Finally, push your local repository to GitHub: `git push origin master` You may get an error message about the authenticity of the host `github.com`, asking if you want to continue connecting. Say “yes”.

3. Now, we’re going to start collaborating through `git`. To do that, we need some collaborators. The instructors’ and the TA’s GitHub IDs are:

- John Gillett's GitHub ID: jgillett-605
- Keith Levin's GitHub ID: kdlevin-uwstat
- TA's GitHub ID: bwu62

In addition to these three collaborators, you will interact with two students from the class.

- On Canvas, you have been added to a group. Click on "People" in the left-hand menu, and click on the "Groups" tab. You will see groups listed with names of the form "HW3 X" where X is a number. Each of these groups consists of three students. Find your group using the search functionality.
- Ordering the three names in your group alphabetically (first by family name, then by given name), your "employee" is the student after your in the ordering (the first student alphabetically if you are last alphabetically), and your "boss" is the student before you in the ordering (the last student in the list if you are first alphabetically).
- Use the "HW3: git usernames" discussion board to exchange github usernames with the other two members of your group by posting under the thread corresponding to your group (email the instructors if no such group exists).

4. Now, let's grant access to your collaborators.

   (a) On GitHub, click on your `gitExample` repository, then "Settings" on the menu under the repository name, and "Manage Access" in the left margin.
      - Scroll down and click the green "Invite a collaborator" button.
      - In the "Search by username, full name, or email" box, enter your four collaborators' GitHub IDs (two professors, TA, "employee").
      - Click "Add collaborator".

   (b) If you do not have a GitHub username from your "employee" yet, just add the professors and TA. Don't forget to come back and add your "employee" later.

5. Once you have accepted access to the GitHub repo belonging to your "boss", it's time to act as an "employee" and contribute to the repository of your "boss" by making a one-line addition:

   (a) Copy your boss's repository from GitHub into a directory called `gitBoss`:
   ```
   git clone git@github.com:bossID/gitExample.git ~/Desktop/gitBoss
   ```
   (where `bossID` is your *boss*'s GitHub ID)

   (b) Change to your "boss" repository directory: `cd ~/Desktop/gitBoss`

   (c) Run `git status` to verify that `names.txt` is in the repository.

   (d) Run
   ```
   echo "employee,NetID,FamilyName,GivenName" >> names.txt
   ```
   (after replacing `NetID` with your NetID) to *append* a line of information to the roster information. Note that we have *two* greater-than symbols instead of the one that we have seen for redirects before. This means to *append*, i.e., add a line, to the file `names.txt`, instead of overwriting the file.

(e) Add the changed file to the (boss) stage: `git add names.txt`

(f) Commit the change to the local (boss) repository:
`git commit -m "Add a line to boss's file."`

(g) Pull the GitHub repository of your "boss": `git pull origin master`, and fix conflicts if necessary (there should not be any conflicts).

(h) Push your changes to the repository: `git push origin master`

6. Okay, back to acting like a "boss".

   (a) Return to your first repository: `cd ~/Desktop/gitExample`

   (b) Update your local repository to receive your employee's change: `git pull origin master`

   (c) Check your `names.txt` file to see that is has two lines, your `boss` line and your employee's `employee` line. If it isn't there, that's okay, check back again soon.

   (d) Add a header line to `names.txt` by adding a line at the beginning of the file of the form `role,NetID,FamilyName,GivenName`. This time, `NetID` should be written literally. Do *not* substitute your NetID.

   (e) Add your changes, commit them with a descriptive message and push them to your repository.

What to turn in

Create a directory whose name is `NetID_hw3`, where `NetID` is your NetID. Copy your shell script files, `school.sh`, `digits.sh`, `ten_dirs.sh`, `rm_n.sh`, and `mean.sh` into `NetID_hw3` (but use your NetID, not `NetID` literally). In `NetID_hw3`, create a file called `README` with a single line line of the form `NetID,FamilyName,GivenName,GitID`, where `NetID` is your NetID, `GitID` is your GitHub username, and `FamilyName` and `GivenName` are your family ("last", in most Western languages) name and given ("first", in most Western languages) name. From the parent directory of `NetID_hw3`, run `tar cvf NetID_hw3.tar NetID_hw3` (again, use your NetID twice, not `NetID` literally). Turn in `NetID_hw3.tar` as Canvas's HW3 assignment.

To verify that you did everything correctly, try downloading your submission file from Canvas, and then

1. Extract it into a new directory (`tar xvf NetID_hw3.tar test_HW3`)
2. List the contents to make sure all your files are there: `ls test_HW3`.