## (Bash) Shell Scripts: Exercises

### Background: Why learn shell scripting?

- It gives access to large-scale computing on many platforms.
- It makes automating repetetive tasks easy.
- 80% of a data analyst's time is spent cleaning up data. Shell scripting for I/O and extracting data from text can be much easier than doing it in R.
- There are many data science problems with so much data that we can't consider a sophisticated model, but a simple statistic (mean, median) or graph can answer the question. 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 pipeline to R programmers, mimicking what the shell has been doing since the 1970s! Shell scripting ideas can improve your use of R: write small tools that do simple things well, using a clean text I/O interface.

### Exercises

- Run `wget http://pages.stat.wisc.edu/~jgillett/605/linux/Property_Tax_Roll.csv` to download that file of 2018 City of Madison property tax data. (I got it from `http://data-cityofmadison.opendata.arcgis.com/datasets/property-tax-roll`.)

  Write a script, `school.sh`, that finds the average `TotalAssessedValue` for properties in the `"MADISON SCHOOLS"` district.

  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 `awk` to sum the `TotalAssessedValue` (7th) column while also counting the number of terms in the sum; report the sum over the number of terms. Note that the required field separator is a comma (`,`); see `man awk` for how to set this option.

- Write a script, `digits.sh`, to find the sum of the numbers between 1000 and 2000 (inclusive) having digits only from the set {0, 1}.

  Hint: Use a brace expansion to generate the range of numbers, a loop to check each one, and a conditional statement including a regular expression to check whether the four digits are in {0, 1}).

  Hint: In emacs, run `M-x sh-mode` to get help with code formatting including indenting.

- Write a script `five_dirs.sh` that does these tasks:
  - make a directory `five`
  - make five subdirectories `five/dir1` through `five/dir5`
  - in each subdirectory, make four files, `file1` through `file4`, such that `file1` has one line containing the digit `1`, `file2` has two lines, each containing the digit `2`, ..., and `file4` has four lines, each containing the digit `4`

  Hint: A convenient way to remove the `five` directory and all its files is `rm -r five` (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 five; five_dirs.sh`

- 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. Try it on your `five` directory via `rm_n.sh five 3`.

  Hint: use `find`. In emacs, do `M-x man Enter find Enter` to check its man page. The page is 1200 lines long–don't read it all. Just read about its `size` argument and search within it for the text "`Numeric arguments`."

  Note:
  - "`rm_n.sh`" in this usage statement should be is specified in your script as `$0`, so that the usage statement will be correct even if you change the script name later.
  - Write the usage statement, which is for humans to read (not for further programs in a pipeline), to `stderr`. One way to do this is via `echo`. Normally it writes to `stdout`. Redirect `stdout` to go to `stderr` via "`1>&2`" as in `echo "hello" 1>&2`.
  - By convention for usage statements, the "`<...>`" delimiters in "`<dir>`" indicate a required argument, and "`[...]`" delimiters indicate an optional argument.

- Write a script, `mean.sh`, with usage statement `usage: mean.sh <column> [file.csv]`, that reads the column specified by `<column>` (a 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. Here are three example runs:
  - `mean.sh` prints the usage statement
  - `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`.)

  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 compound expression in braces (`{}`) to initialize a sum and line count, run a `while read` loop to accumulate that sum and line count, find the mean, and `echo` it

To handle reading from `file.csv` or from `stdin`, I 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 I could read from my `file` variable in either case.

**What to turn in (once per group):**

Write a plain-text file called `README` that includes information on your group members (1 to several students) in the line format `NetID,LastName,FirstName`. For example, if Wilma Flintstone (NetID: wflint3) and Charlie Brown (NetID: cbrown71) worked together, their `REAMDME` file would be:

```
wflint3,Flinstone,Wilma
cbrown71,Brown,Charlie
```

Make a directory whose name is your `NetID`. Copy your files, `README`, `school.sh`, `digits.sh`, `five_dirs.sh`, `rm_n.sh`, and `mean.sh` into `NetID` (but use your NetID, not `NetID` literally). From the parent directory of `NetID`, run `tar cvf NetID.tar NetID` (but use your NetID twice, not `NetID` literally). Turn in `NetID.tar` as Canvas's Group1linux assignment.