Software Carpentry Advanced Python Workshop October 8-9 2018

Day 1

https://pad.carpentries.org/2018-10-08-IACS

~~ Set-up check ~~

- Have you downloaded anaconda with Python 3?
https://www.anaconda.com/download/

- Have you downloaded the data-shell directory?
http://swcarpentry.github.io/shell-novice/setup.html

- Open your terminal
    - if on Windows, open the Git Bash terminal
    - Navigate to the data-shell directory

- Unfortunately, some commands do not work in the Git bash emulator, e.g.:
    - 'man command'; instead use '<command> --help'

~~ Introducing the Shell ~~

- Computers do 4 things:

  run programs
  store data
  communicate with each other, and
  interact with us

- Bash = Bourne Again SHell (derived from shell written by Stephen Bourne)
    - most popular Unix shell
    - it's job is to run other programs rather than do calculations itself
    - works by reading command, evaluating command, printing output of command, and looping back to wait for next command (read-evaluate-print loop, REPL)

- Anatomy of the command-line interface:
    - Example: `$ ls -F /`
        - prompt = first line ending in $
        - command = everything after $
        - flags/options/switches = -[letter] or --[word]
        - argument = tell the command what to operate on (e.g., file, directory)
        - separator = spaces [number 1 reason to avoid spaces in your file and directory names!]
    - commands don't always need flags or arguments
    - commands and flags are case sensitive (but depends on core.ignorecase value, default is false)

- Going back to the REPL concept, this is what the computer is doing:

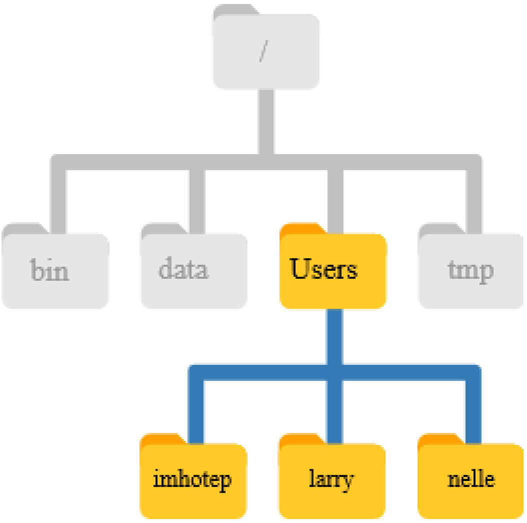
1. Read what was typed (ls -F /). The shell uses the spaces to split the line into the command, flags, and arguments
2. Evaluate:
    a. Find a program called ls
    b. Execute it, passing it the flags and arguments (-F and /) to interpret as the program sees fit
3. Print the output produced by the program
    4. Print prompt and wait for next command

- error messages
    - e.g., 'command not found'
    - check spelling and syntax

- reasons to learn bash
    - automates repetitive tasks
    - write/execute scripts with a bunch of bash commands
        - many useful data manipulation/editing commands
    - write/execute scripts written in other languages, e.g., R or Python
    - interact with remote computers and supercomputers

~~ Navigating Files and Directories ~~

- Where am I?
    - `pwd` = print working directory
- Hierarchical file systems
    - single / = the root directory (top directory)
    - moving down in the hierarchy, there are 4 directories in the root directory
    - slash inside a name /Users/nelle/ = separator; here between directories



- What's in the directory?
    - `ls` = prints names of files and directories in current directory

        - -F: distinguishes between directories/ and files
    - check flags: `man ls` or `ls --help`
    - ‘invalid option’ for unsupported flags
    - **TRY: What do these commands do?**
        - `ls -l`
        - `ls -h`
        - `ls -lh`
        - `ls -R`
        - `ls -t`
        - `ls -R -t`

- Change directory
    - `cd <name of directory>`
        - `cd data-shell/`
        - `pwd`
        - note use of tab-completion
- Going back
    - ‘..’ is a special flag
        - `ls -a` = show all, shows ‘..’
    - ./ is current directory
    - ../ is one directory up
    - `cd ..` = to go back 1 directory up
        - `pwd`
    - `cd ../../..` = to go back 3 directories up
        - `pwd`
- Going home
    - `cd`
    - `$HOME`
    - `cd ~`
        - ~ means current user’s home directory
        - If /Users/nelle is home directory, then /Users/nelle/data = ~/data

- Absolute vs. relative paths
    - An absolute path specifies a location from the root of the file system.
    - A relative path specifies a location starting from the current location.

Starting from `/Users/amanda/data/`, which of the following commands could Amanda use to navigate to her home directory, which is `/Users/amanda`?

1. `cd .`
2. `cd /`
3. `cd /home/amanda`
4. `cd ../..`
5. `cd ~`
6. `cd home`

7. `cd ~/data/..`
8. `cd`
9. `cd ..`

Put your answers on the etherpad. If your answer is already there, add a '+1' next to it
https://pad.carpentries.org/2018-10-08-IACS

~~ Working with Files and Directories ~~

- Making new directories
    - `mkdir <name of new directory>`
        - `mkdir thesis`
        - `ls -F`

- Naming files and directories dos and don'ts
    - Stick with letters, numbers, period (.), dash (-), and underscore (_)
    - Don't <u>begin</u> name with a dash (-)
        - names starting with a dash are treated as options
    - Don't use whitespace
        - if you need to refer to file or directory with whitespace or another non-alphanumeric character, surround name in quotes ("")

- Writing text files with nano
    - `cd thesis`
    - `nano mythesis.txt`
        - Editor keys (not case-sensitive):
            - Ctrl-O to save
            - Ctrl-K to cut text
            - Ctrl-U to uncut text
            - Ctrl-X to exit
            - Ctrl-G for help

- `touch mythesis2.txt`
- `ls -lh mythesis2.txt`

- Deleting is FOREVER!

Tools for finding and recovering deleted files do exist, but there's no guarantee they'll work in any particular situation, since the computer may recycle the file's disk space right away

- `rm` = removes file
- `rmdir` = removes empty directory
- `rm -r` = removes all files in a directory
- `rm -i` = safest option. The -i flag will ask you before deleting each file
    - `rm -i mythesis2.txt`

- Moving and copying

- `mv` = moves file; it will overwrite any existing file with same name (thus, the `mv` command is also used to rename files or directories)
    - `mv -i` = will ask for confirmation before overwriting
- Renaming
    - `mv thesis/ thesis2/`
    - `mv mythesis.txt mythesis_final.txt`
    - `ls`

- Moving
- `touch thesis_idea2.txt`
- `mkdir new_thesis_idea`
- `mv thesis_idea2.txt ./new_thesis_idea`
- `ls -lh ./new_thesis_idea`

- Copying
- `cp` = copy file to new location
    - `cp ./new_thesis_idea/thesis_idea2.txt ./thesis`

- Wildcards
    - * = matches zero or more characters
        - `ls molecules/*.pdb`
        - `ls molecules/p*.pdb`
    - ? = matches a single character
        - `ls molecules/propan?.pdb`

- TRY:
Sam has a directory containing calibration data, datasets, and descriptions of the datasets:

```
2015-10-23-calibration.txt
2015-10-23-dataset1.txt
2015-10-23-dataset2.txt
2015-10-23-dataset_overview.txt
2015-10-26-calibration.txt
2015-10-26-dataset1.txt
2015-10-26-dataset2.txt
2015-10-26-dataset_overview.txt
2015-11-23-calibration.txt
2015-11-23-dataset1.txt
2015-11-23-dataset2.txt
2015-11-23-dataset_overview.txt
```

Before heading off to another field trip, she wants to back up her data and send some datasets to her colleague Bob. Sam uses the following commands to get the job done:

```
$ cp *dataset* /backup/datasets
$ cp ____calibration____ /backup/calibration
$ cp 2015-____-____ ~/send_to_bob/all_november_files/
$ cp ____ ~/send_to_bob/all_datasets_created_on_a_23rd/
```

Help Sam by filling in the blanks. Put your answers on the etherpad

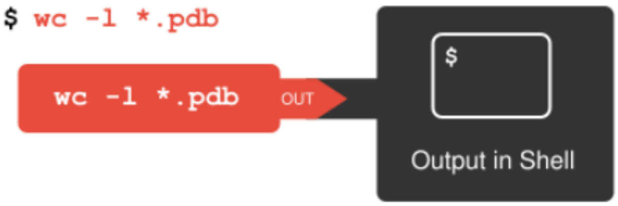
~~ Pipes and Filters ~~

- Pipes allow us to easily combine programs (|)
- Filters transform input (e.g., `wc`, `sort`)
- Redirecting results of command to file (>)

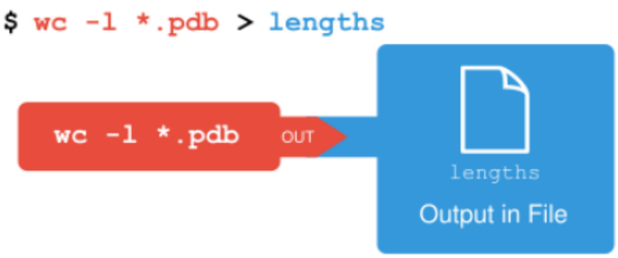
- To illustrate, go to molecules directory:
    - `wc` = word count
        - `wc -l` = line count
        - `wc -c` = character count
        - `wc -w` = word count
    - `wc -l *.pdb > lengths.txt`
        - if ‘lengths.txt’ already existed, this command will overwrite it
    - Viewing new file
        - `cat lengths.txt` prints to screen
        - `less lengths.txt` shows part of the file, particularly useful for very long text files
            - up and down arrows to scroll, q to quit
    - we can sort the results
        - `sort lengths.txt | less` sorted by 1st number
        - `sort -n lengths.txt | less` sorted numerically

    - `sort -n lengths.txt | head -n 2`
    - note use of the pipe to send sorted result to less or head without saving the result
        - reads left to right

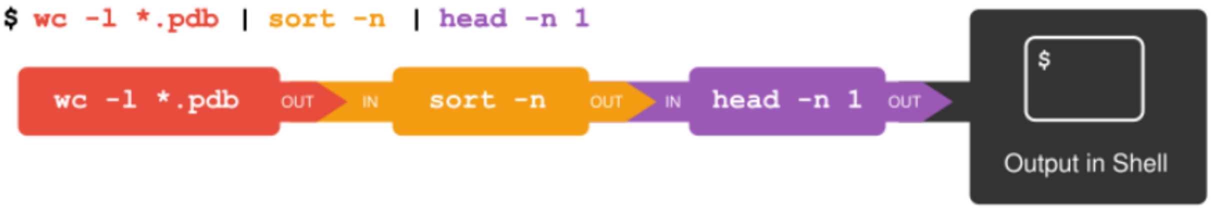
    - `sort -n lengths.txt > sorted-lengths.txt`
    - `tail -n 2 sorted-lengths.txt`
- This entire process can be written as:
- `wc -l *.pdb | sort -n > sorted-lengths.txt`



```
$ wc -l *.pdb
```



```
$ wc -l *.pdb > lengths
```



```
$ wc -l *.pdb | sort -n | head -n 1
```

TRY:
We have seen the use of >, but there is a similar operator >> which works slightly differently. By using the `echo` command to print strings, test the commands below to reveal the difference between the two operators:

```
$ echo hello > testfile01.txt
```

and:

```
$ echo hello >> testfile02.txt
```

Hint: Try executing each command twice in a row and then examining the output files.

- Other useful filters
    - `uniq` = gets rid of adjacent duplicates. To remove all duplicates, sort file first.
    - `cut` = method for selecting pieces of file using delimiters

TRY:

1) A file called `animals.txt` (in the `data-shell/data` folder) contains the following data:

```
2012-11-05,deer
2012-11-05,rabbit
2012-11-05,raccoon
2012-11-06,rabbit
2012-11-06,deer
2012-11-06,fox
2012-11-07,rabbit
2012-11-07,bear
```

What text passes through each of the pipes and the final redirect in the pipeline below?

```
$ cat animals.txt | head -n 5 | tail -n 3 | sort -r > final.txt
```

Hint: build the pipeline up one command at a time to test your understanding

2) For the file `animals.txt` from the previous exercise, the command:

```
$ cut -d , -f 2 animals.txt
```

uses the `-d` flag to separate each line by comma, and the `-f` flag to print the second field in each line, to give the following output:

```
deer
rabbit
raccoon
rabbit
deer
fox
rabbit
bear
```

What other command(s) could be added to this in a pipeline to find out what animals the file contains (without any duplicates in their names)?

3) The file `animals.txt` contains 586 lines of data formatted as follows:

```
2012-11-05,deer
2012-11-05,rabbit
2012-11-05,raccoon
```

```
2012-11-06,rabbit
...
```

Assuming your current directory is `data-shell/data/`, what command would you use to produce a table that shows the total count of each type of animal in the file?

1. `grep {deer, rabbit, raccoon, deer, fox, bear} animals.txt | wc -l`
2. `sort animals.txt | uniq -c`
3. `sort -t, -k2,2 animals.txt | uniq -c`
4. `cut -d, -f 2 animals.txt | uniq -c`
5. `cut -d, -f 2 animals.txt | sort | uniq -c`
6. `cut -d, -f 2 animals.txt | sort | uniq -c | wc -l`

~~ Loops ~~

Pipes streamline filters, but what if you need to filter many different sets of files? You need a loop!

Use data-shell/creatures directory:

The basic structure of a for loop in bash is:

```
     for datafile in *.dat [enter]
          do [enter]
               head -n 3 $datafile [enter]
          done [enter]
```

OR use ‘;’ to indicate the indentations

```
     for datafile in*.dat; do head -n 3 $datafile; done
```

- the $ tells shell interpreter to treat the variable as a variable name substitute its value in its place, rather than treat it as text or an external command

- Testing your loop
    - use `echo` to print out the commands in the loop before you actually run
    - `for file in *.pdb; do echo “analyze $file > analyzed-$file”; done`

TRY:

Run each of these commands to familiarize yourself with the syntax. Try making your own loops using the filters and commands we learned earlier (ls, cut, sort, uniq).

- We can put pipes in loops

```
     for filename in *.dat; do echo $filename; head -n 3 $filename | tail -n 20; done
```

- We can rename files in loops

```
     for filename in basilisk.dat unicorn.dat; do cp $filename original-$filename; done
```

- We can process data and save new results with loops

- go to data-shell/north-pacific-gyre

    - `for datafile in NENE*[AB].txt; do echo $datafile; bash goostats $datafile stats-$datafile; done`

- We can save files in loops

    - `for datafile in *.pdb; do cat $datafile >> all.pdb; done`

- Run a past command
    - `history | tail -n 5` shows last 5 commands you ran
    - `!<line number>` will re-execute the command
    - `!!` and up arrow = both will retrieve your command
    - `!$` retrieves last word of last command
    - Ctrl-R allows you to search history

~~ Shell Scripts ~~

- reasons to use shell scripts
    - execute bash commands in a script instead of typing out over and over
    - execute other scripts
- # indicates a comment, not run
- run the script with `bash <script name>`
- bash script file extension is .sh

- We can write a bash script to take user-defined arguments
    - `cd molecules`
    - `nano middle.sh` #version 1
        - `head -n 15 octane.pdb | tail -n 5`
    - `bash middle.sh`

$1, $2, etc. refer to 1st command-line argument, 2nd command-line argument, etc.

- `nano middle.sh` #version 2
    - `head -n $1 $2 | tail -n $3`
- `bash middle.sh 15 pentane.pdb 5`

Add usage comment for your script

- `nano middle.sh` #version 3
    - `#Usage: bash middle.sh end_line filename num_lines`
    - `head -n $1 $2 | tail -n $3`
- `bash middle.sh 4 pentane.pdb`

Place variables in quotes if the values might have spaces in them

- `nano middle.sh` #version 4
    - `#Usage: bash middle.sh end_line filename num_lines`
    - `head -n “$1” “$2” | tail -n “$3”`
- `bash middle.sh 4 pentane.pdb`

If we want to be able to get a sorted list of other kinds of files, we need a way to get all those names into the script. We can't use `$1`, `$2`, and so on because we don't know how many files there are.

Instead, we use the special variable `$@`, which means, "All of the command-line arguments to the shell script." We also should put `$@` inside double-quotes to handle the case of arguments containing spaces

```
    - nano sorted.sh
        - #Usage: bash sorted.sh one_or_more_filenames
        - wc -l “$@” | sort -n
    - bash sorted.sh *.pdb ../creatures/*.dat
```

- We can write bash scripts to execute other scripts
- First, we will write a python script

```
    - mkdir bikedata
    - nano get_bike_data.py
        from urllib.request import urlretrieve
        URL='https://data.seattle.gov/api/views/65db-xm6k/rows.csv?accessType=DOWNLOAD'
        urlretrieve(URL, ‘Fremont.csv’)
    - nano bash_wrapper.sh
        python get_bike_data.py
    - bash bash_wrapper.sh
    - ls
```

# On Windows:
# Write the python script with nano in Git bash, but must execute python in Anaconda prompt
# Git bash does not recognize python command
- Seattle bike count dataset:
URL='https://data.seattle.gov/api/views/65db-xm6k/rows.csv?accessType=DOWNLOAD'

- debugging a script

```
    - bash -x <script name>
```

TRY:
Write a shell script called `longest.sh` that takes the name of a directory and a filename extension as its arguments, and prints out the name of the file with the most lines in that directory with that extension. For example:

```
$ bash longest.sh /tmp/data pdb
```

would print the name of the `.pdb` file in `/tmp/data` that has the most lines.

~~ Finding Things ~~

- `grep` = global/regular expression/print; finds and prints lines in files that match a pattern
- `find` = finds files themselves

- grep examples
- We’ll use haikus from 1988 magazine - more here! http://wiki.c2.com/?ComputerErrorHaiku
    - `cd data-shell/writing`
    - `cat haiku.txt`

    - Search pattern = words or parts of words
        - `grep not haiku.txt`
        - `grep The haiku.txt`
        - `grep -n The haiku.txt` tells you the line number of matches
        - `grep -w The haiku.txt` restricts match to a word boundary
        - `grep -nwi ‘the’ haiku.txt` makes search case-insensitive
        - `grep -nwv ‘the’ haiku.txt` returns matches that do not include ‘the’

    - Search pattern = phrase
        - `grep -w ‘is not’ haiku.txt`

    - There are many grep options
        - grep --help

    - Search pattern = regular expressions
        - Regular expression are both complex and powerful - more information here: https://v4.software-carpentry.org/regexp/index.html
        - `grep -E ‘^.o’ haiku.txt`

            -E = tells grep the search pattern is a regular expression
            ^ = anchors match to start of line
            . = matches single character
            o = matches the letter ‘o’

TRY:
You and your friend, having just finished reading *Little Women* by Louisa May Alcott, are in an argument. Of the four sisters in the book, Jo, Meg, Beth, and Amy, your friend thinks that Jo was the most mentioned. You, however, are certain it was Amy. Luckily, you have a file `LittleWomen.txt` containing the full text of the novel (`data-shell/writing/data/LittleWomen.txt`).

Using a `for` loop, how would you tabulate the number of times each of the four sisters is mentioned?

Hint: one solution might employ the commands `grep` and `wc` and a `|`, while another might utilize `grep` options.

There is often more than one way to solve a programming task, so a particular solution is usually chosen based on a combination of yielding the correct result, elegance, readability, and speed.

Put your answer on the etherpad. Add ‘+1’ if you agree with an answer on the etherpad.

- find examples
    - in data-shell/writing
    - `find .` will show all the files and directories within writing/
    - `find . -type f` will show all the files within writing/
    - `find . -type d` will show all the directories within writing/

TRY: Why do you get different results from these commands?

- `find . -name *.txt` will show all the files that end in ‘.txt’ in current directory only
- `find . -name ‘*.txt’` will show all the files that end in ‘.txt’ within writing/

- combining commands - another way other besides pipes: $()
    - `wc -l $(find . -name ‘*.txt’)`
    - same as `wc -l *.txt` but does line count for all the .txt files in sub-directories

    - `grep ‘FE’ $(find .. -name ‘*.pdb’)`

TRY:

The `-v` flag to `grep` inverts pattern matching, so that only lines which do *not* match the pattern are printed. Given that, which of the following commands will find all files in `/data` whose names end in `s.txt` (e.g., `animals.txt` or `planets.txt`), but do *not* contain the word `net`? Once you have thought about your answer, you can test the commands in the `data-shell` directory.

1. `find data -name '*s.txt' | grep -v net`
2. `find data -name *s.txt | grep -v net`
3. `grep -v "temp" $(find data -name '*s.txt')`
4. None of the above.