Chem 160 Module 19 Linux
Segment 1 Overview of the Linux Shell

Topics:
1. The Linux shell

Online references:
1. Bash shell
   a. http://www.linuxtopia.org/online_books/bash_guide_for_beginners/index.html
   b. List of Bash tutorials: http://wiki.bash-hackers.org/scripting/tutoriallist
   c. Interactive Shell tutorial: http://www.learnshell.org
   d. More tutorials on the shell: http://swcarpentry.github.io/shell-novice/
   e. http://www.ibm.com/developerworks/linux/library/l-bash/index.html

I. The Shell

A. What is a shell?
   a. Program that interprets what you type
   b. Environment that keeps track of your previous commands and env. variables
   c. Programming language that lets you customize your environment and toolkit

B. How many shells are there?
   a. Many, but most people use one of a small group (most at least 30 years old):
      i. **bash**—this is what we will learn, which is by far the most widely used
      ii. **tcsh**—a more advanced descendent of the even older **csh**
      ```
      tcsh
      set file="/usr/local/prog.py"
      echo $file
      echo $file:t
      echo $file:e
      echo $file:h
      echo $file:t:r
      exit
      ```
      iii. **zsh**—expanded version of Bourne shell with great file completion
      ```
      zsh
      ls /u/lo/in[TAB]
      cd [TAB]
      ls –[TAB]
      exit
      ```
      iv. **fish**—a shell designed to be user friendly (and have a sense of humor)
      ```
      fish
      ls /usr/local/bin
      ls /var/**.log
      exit
      ```
   b. For more info see: **https://en.wikipedia.org/wiki/Unix_shell**

C. You can switch between shells quite easily, just type the shell name to start a new shell and then **exit** to quit from a shell.

D. Look at how bash remembers what you've done
   a. **history**
   b. Recalling old commands
      i. Up arrow
      ii. Recovering commands by number !29
      iii. Recovering commands without executing them !29:p
      iv. Searching for old commands Ctrl-r text
      v. Try typing: Ctrl-r **nano**
      Note that the search is *incremental*, updating with each new letter typed.

E. Let's look as some other features--download the GitHub repository for this module:
`git clone https://github.com/mcolvinphd/chem160module19.git`

F. Wildcards (aka "globbing")
   a. A wildcard is a pattern that can match multiple files (or directories)
   b. Many commands can take wildcards
      i. "*" match 0 or more characters
      ii. "?" match 1 character—can use several
      iii. [ABC] match any character in the list
      iv. [a-e] match in range
      v. {a?,b??} match either a? or b??
   c. Let's use wildcards to sort college abbreviations:
```
cd colleges
ls ???
ls *U??
ls UC?
UC?[A-C]
ls UC{S?,?}
```
   d. Let's try it to collect output files
```
cd ../dna
ls DNA*.pdb
ls DNA*1.pdb
ls DNA?1.pdb
ls DNA?[135]?.pdb
ls DNA[0-1][3-5][13579].pdb
ls DNA[0-1][3-5][13579].pdb > dna.out
```
   e. You can use basic wildcards in the grep command (cd back to **`~/class2a`**)
```
grep GL[YN] PK.pdb | grep CA
grep [A-Z]E[A-Z] PK.pdb | grep CA
grep 3[02468][13579][02468][7-9] PK.pdb | grep CA
grep 3[02468][13579][02468][7-9] PK.pdb | grep CA > CA.out
```
   f. A more sophisticated form or wildcards are "regular expressions" (Regex's) which are much more flexible (and complex) than wild cards
   g. Regex's are used in many languages and Linux commands, including Perl, Python, awk, egrep, and sed. If you've ever filling in an electronic form, you've probably used regex's.
   h. Regex's can be used in Bash scripts, as we'll see when we cover conditionals in Bash
   i. Example of using regular expression in Bash—match only sets of 8 digits [0-9]. Enter this at the Bash prompt:
```
date=11182019
[[ $date =~ ^[0-9]{8}$ ]] && echo "Date" || echo "Not date"
date=111819
```
(use up arrow to get this command back)
```
[[ $date =~ ^[0-9]{8}$ ]] && echo "Date" || echo "Not date"
```
   j. A more complicated date regex matcher is in the script: **`datetest.bash`** which will match dates of the format 11/18/19, 11-18-19, 11/18/2019, etc. and will check whether the ranges are correct for the days (01-31) and months (01-12) but not whether the dates actually exist in a given month (e.g. 02/31/2019 is okay)
```
datetest.bash 11/19/2019
datetest.bash 13/19/2019
datetest.bash 11/39/2019
```
   k. If you search for "date regex expressions" you'll see more sophisticated regex's that check if a date actually exists in a given year.

Chem 160 Module 19 Linux
Segment 2 First Bash Script Example

Topics:

1. Examples of Shell scripting

I. A first immersion into shell scripts

A. Shell script is a program written in the language of the shell
   i. Bundles a set of commands in a useful way
   ii. Can includes loops over sets of files or input values
   iii. Can include conditionals (if statements) to do different things under different conditions

B. cd into the **`chem160module19`** directory

C. Edit a file called **`files.bash`** (using **`nano`**)
   i. Go into insert mode and start the script with line "**`#!/bin/bash`**"
   ii. With the exception of the first line, we use "**`#`**" to comment out a line (needed for debugging scripts or adding explanatory comments)
   iii. Add the following 3 lines to the script (note quotes in echo optional)

```
echo "Directory contains this many files and dirs:"
pwd
ls | wc | colrm 10
```

D. Make the script runnable by giving it "execute" permission:

```
chmod +x files.bash
```

E. Run the script:

```
files.bash
```

II. Miniproject: Script to count how many of a particular amino acid is in a pdb file.

A. This script will require 2 inputs, the amino acid abbreviation and the file name

B. The Bash shell provide a special way to pass command line arguments in a script.

C. If we have a script called **`count_aa.bash`** and run it as follows:

```
count_aa.bash  GLY  PK.pdb
```

Then inside the script:
   i. **`$1`** will contain "GLY"
   ii. **`$2`** will contain "PK.pdb"

D. Writing the script:
   i. Use nano to edit a file called **`count_aa.bash`**
   ii. First line: **`#!/bin/bash`**
   iii. Now echo what the result means:

```
echo -n "The number of $1 amino acids in $2 is:"
```

   iv. Next use the grep, wc and colrm commands to get this information

```
grep $1 $2 | grep CA | wc | colrm 10
```

   v. Now save the script and make it executable
   vi. Try running it:

```
count_aa.bash  GLY  PK.pdb
count_aa.bash  ALA  PK.pdb
```

   vii. What if you forget an argument or misspell the pdb file name?

```
count_aa.bash  PK.pdb
count_aa.bash  GLY PL.pdb
```

viii. Let’s improve the script by adding some *conditionals* to test for these issues

ix. Edit the script again and add the following right after the first line

```
if [ $# -ne 2 ]; then
    echo "Script takes 2 arguments, an AA and a pdb file"
    exit 1
fi
```

x. Test running the script with the wrong number of arguments

xi. Edit the file again and add the following lines after the first conditional

```
if [ ! -f $2 ]; then
     echo "There is no file $2"
     exit 1
fi
```

xii. Test running the script with a misspelled pdb file name

xiii. Finally let’s put your script into a loop (hint: this will help in Homework 2):

```
for i in GLU GLN GLY ALA PHE
do
count_aa.bash  $i  dynein.pdb
done
```