BIOL*3300 Lab4 F21

Advanced Command Lines

To be sure we are still working in the same place, let's login in Compute Canada Graham account and run the following commands:

```
cd scratch/Biol3300
mkdir Lab4
cd Lab4
pwd
```

Variables

We can think of a variable as a placeholder for a value that will change with every iteration of our loop. To set a variable at the command line, we need to provide the variable name we want, an equals sign, and then the value we want the variable to hold (with no spaces in between any of that). Let’s try it:

```
my_var=100
```

Nothing prints out when a variable is set, but the value "100" has been stored in the variable "my_var". To use what’s stored in a variable, the variable name needs to be preceded by a $ so the shell knows to evaluate what follows, rather than just treat it as generic characters. We’ll use the echo command to check what's being stored in our variable. Note that if we don’t put the $ in front, echo just prints out the text we gave it:

```
echo $my_var
```

Recall that spaces are special characters on the command line. If we wanted to set a variable that contained spaces, we could surround it in quotations to tell Unix it should be considered as one thing:

```
my_new_var="Hello world!"
echo $my_new_var
```

Running programs

If a file has execute permission it means the code inside a file may be read and acted upon. Here is a file with some code:

```
echo 'echo "hello world"' > hello.sh
```

The user does not have execute permission by default, so we need to change the permission:

```
ls -l
chmod u+x hello.sh
ls -l
```

The file can now be executed, but it can't be executed by name alone.

```
hello.sh
```

Making a file executable is not sufficient for executing the contents of the file by writing its name alone. We need to give a path to the file's directory:

```
pwd
```

And include the file name with the path:

```
~/scratch/Biol3300/Lab4/hello.sh
```

Or you can use **.** because it means the path of current working directory:

```
./hello.sh
```

A program is any file that has executable permission. To call a program by name, it must be in your $PATH. Unix will check through a list of predefined directories to see if that program exists in those locations.

You could move files that you want to make programs like this, but it is untidy, and you run the risk of clobbering an existing, important application if you are not careful. A better solution is to add a directory for Unix to look into. The order of directories within the **PATH** variable

determines the order in which Unix will search for programs. This could be relevant if you have executable files with the same names in different directories.

We can modify the $PATH as follows.

```
export PATH=$PATH:~/scratch/Biol3300/Lab4/
```

Let's check with the PATH:

```
echo $PATH
```

In this case, the program hello.sh within the directory Lab4 within the directory Biol3300 within the directory scratch within my home directory contains the executable. Now try:

```
hello.sh
```

For Loops

Loops are useful for quickly telling the shell to perform one operation after another, in series. There are 4 special words in the syntax of a For Loop in Unix languages: for, in, do, and done. For example:

```
for i in {1..10}; do echo $i >> a; done
cat a
```

The list can be a sequence of numbers or letters, or a group of files specified with wildcard characters:

```
for i in {3,2,1,Apple,Banana}; do echo $i; done
```

You can do multiple things within the body of the loop (the lines between the special words do and done). Here we'll add another line that also writes the words into a file we'll call "words.txt":

```
for item in {car,truck,ukulele}; do echo $item; echo $item >> words.txt; done
```

We can use cat to pull the items we want to loop over from the file, instead of us needing to

type them out like we did above. Here is an example with our "words.txt" file we just made:

```
for item in $(cat words.txt); do echo $item; done
```

You can also use for loop to process multiple files, let's create 2 files with:

```
echo "hello" > temp1
echo "there" > temp2
```

We can print out the contents of all the temp files with the variable in the numerical list.

```
for i in {1..2}; do cat temp$i; done
```

You can also do some creative loop with your familar command **ls**

```
for i in $(ls); do cat $i; done
```

Standard error

'stderr' (for Standard error) is another Unix data stream like standard input and standard out. We previously created temp1 and temp2 files:

```
cat temp1 temp2
```

Now we introduce an error on purpose as we didn't create temp3 file:

```
cat temp1 temp3
```

We get the contents of the first file and an error with the second file. However, the two data streams are not the same. The file temp1 contains "hello", but the error message from the second file went to the screen.

We can redirect standard error through **2>**

```
cat temp1 temp3 2>errors
cat errors
```

You can redirect standard error with a for loop:

```
for i in {1..3}; do cat temp$i 2> error_file; done
```

This for loop will also print out content of temp1 and temp2 and the error for temp3 would go into error_file.

Reference

These lab materials are from the following tutorials:

1. https://astrobiomike.github.io/unix/getting-started
2. https://ucdavis-bioinformatics-training.github.io/2020-Intro_Single_Cell_RNA_Seq/prerequisites/cli/advanced-command-line