*nix Walkthrough

1. Silence is golden (or frustrating)
2. Capitalization matters
3. Spaces matter
4. Everything is a file

Navigation

Open a BASH terminal window and type:

```
$ whoami
```

The response will be *someusername*. Let's find out where we are:

```
$ whereami
```

While it might seem like this would tell you where you are (within the directory structure of the system, not in the galaxy) it will fail with something like *whereami: command not found*. This is an opportunity to reinforce the bazaar (not "bizarre") nature of Unix (a la Eric S. Raymond).

```
$ pwd
```

This stands for "Print Working Directory" and you'll get something that looks like */Users/someusername*.

Looking Around

Let's look around inside the current directory.

```
$ ls
```

We'll see a list of files inside this directory. Let's call this the "***l***ist directory ***s***tructure" command
Have them try the following to see what happens:

```
$ ls -a

$ ls -l
```

-a will show all files in the directory. *-l* will give a long format with more information.

What follows the **ls** command in the above examples are flags. There are a lot of these. To see what they are for each command you have two options:

1. Use the man pages. Show them how to do this using **q**, **h** (point out how to use **w** and **z** to move up and down).

   $ man ls

2. Search on the Internet.

> ***Question:*** *What will ls -al do?*
> ***Answer:*** *List all the files in a long format.*

Look at the output and point out the files **./** and **../**. Point out that the single dot is a self reference to this directory and that the double dot is a reference to the directory above this one.

We can also look around from where we are:

```
$ ls /someDirectoryListedInThePreviousResults
```

This should give the contents of whatever lower directory that they chose. Have them start to type the command again but this time show them **tab completion**.

> ***Question:*** *What does ls . do?*
> ***Answer:*** *Show the contents of the directory above the current one.*

At this point they are ready to recognize the basic command structure:

```
**command** *space* **flags** *space* **list of files separated by spaces**
```

Have them look in the root directory with:

```
$ ls /
```

> ***Comprehension Test:*** *What do the flags -s, -h, and -r do when combined with the ls command?*
> ***Answer:*** *They can look these up using* ***man ls*** *or just try them out.*

Moving Around

The change directory command (**cd**) can be substituted for the list directory structure (**ls**) command with everything you have learned so far.

> ***Challenge:*** *Go to the root directory*
> ***Solution:*** *Use the* ***cd*** *command with the root directory shortcut /.*

After this challenge a bigger problem faces the students: how to get back the where they were. The long way would be to figure out the the directory structure with tab completion. Better would be to "just know" the directory structure but this takes a long time. Fortunately there is the shortcut:

$ cd ~

The "~" is the "tilde" character and it is usually found with the "backtick" character under the ESC key. It can be accessed by holding SHIFT and pressing "`".

> ***Comprehension Test:*** *What does the command* ***cd*** *without a directory name do?*
>
> 1. *It has no effect.*
> 2. *It changes the working directory to /.*
> 3. *It changes the working directory to the user's home directory.*
> 4. *It produces an error message.*

Answer: *#1.*

> **Comprehension Test:** *For a hypothetical filesystem location of* ***/home/amanda/data/****, select each of the below commands that Amanda could use to navigate to her home directory, which is* ***/home/amanda***
>
> 1. *cd .*
> 2. *cd /*
> 3. *cd /home/amanda*
> 4. *cd ../..*
> 5. *cd ~*
> 6. *cd home*
> 7. *cd ~/data/..*
> 8. *cd*
> 9. *cd ..*
>
> ***Answer:*** *3, 5, 9*

Creation & Destruction

Start this section by having each user navigate to their *Desktop directory*.

Creating Directories

We are going to create a directory to hold the files that we will be working with. We are going to do this in the Desktop directory because it will be very easy to see the consequences of what you do here.

```
& mkdir SoftwareCarpentry
```

Have them minimize their window and go and look at their desktop to see the new folder. Have them actually open the folder with their GUI so that they can watch what happens *and have multiple ways of interacting with the files*. This last part is important since there is more than one way to get things done.

Point out that they can swap between windows with CMD-TAB (MAC) / ALT-TAB (WINDOWS / LINUX).

Return to the Terminal and enter the newly created directory:

```
$ cd SoftwareCarpentry
```

See what's here:

```
$ ls
```

It's empty, as it should be (compare to GUI).

Let's make another new directory and then look at the contents of the current directory:

```
$ mkdir Terminal Testing
$ ls
```

The **ls** command will show that we have made a mistake: there are *two* directories---one called "Terminal" and the other called "Testing"---rather than one "Terminal Testing". This highlights two important things for the class to remember:

1. The computer does what you tell it, not what you wanted.
2. Spaces matter.

We can fix the first by being patient and careful. We can fix the second by:

1. Not using spaces via:
    1. Camel Case
    2. Dashes
    3. Underscores
2. Escaping the space by preceding it with a "\".

Let's make the proper directory using underscores and then move into that directory:

```
$ mkdir Terminal_Testing
$ cd Terminal_Testing
```

Creating a File

We are now going to make a file. It is important for the participants to note that what matters here is that *they can create a text file* not what tool they use to make that text file. We will use **nano** because it is simple, has the instructions listed on the bottom of the screen, and keeps us

in the terminal window (which is convenient, that's all), *beyond these reasons there is nothing special about nano*. What matters is that the tool they choose is a **TEXT EDITOR** and the that they can use it. If they want to use another terminal program (like vim or emacs) or a GUI tool (like TextWrangler, Sublime, or Notepad++) that's just fine. They need to know that they cannot use tools like Word or LibreOffice.

```
$ nano Important_Ideas
```

This will open the nano program and allow people to type into it. It is text-only, there is no fancy formatting. The available commands are on the bottom of the screen. The "^" means "hold the control key and then press the key to the right".

Have the students make a list with the top three things that they have learned in this workshop so far. They will then save it by:

```
^x      (hold control and press 'x')
y       (to respond "yes" when they are asked if they want to save the file)
```

Check that the file is in the folder:

```
$ ls
```

To view the content of the file we could run nano again (If not already covered this is a great time to cover using the up arrow to scroll through past commands):

```
$ nano Important_Ideas
```

This can get annoying though and of course there is a command to show the contents of a file without having to go through the burden of opening a full program.

```
$ cat Important_Ideas
```

cat is the concatenate files command. Concatenation is a join operator. The command also prints out the consequences of the concatenation to the screen. If you only give it one file then it just spits back that one file. This is a great example of tools being very useful but not quite in the way that their name or historical purpose might suggest.

> ***Question:*** *What will the -n flag do when run with* ***cat****?*
> ***Answer:*** *Prints the line numbers for each line in each file.*

Copying Files

We only have one file in this directory so keeping it around is kind of a waste. Let's move this file up one directory and then delete the directory.

> ***Comprehension Test:*** *If* ***cp*** *is the copy command then how do we make a copy of the file Important_Ideas up one directory?*
> ***Answer:***

```
$ cp Important_Ideas ../
```

Note the structure of the command.

Deleting Files

Now we need to get rid of the old folder. We will do this as follows:

```
$ rm Terminal_Testing
```

This will produce an error because we are in the directory and so it can't find a directory with that name:

```
rm: TerminalTesting: No such file or directory
```

We can try to remove the current directory with a **.**:

```
$ rm .
```

But this too produces an error:

```
rm: "." and ".." may not be removed
```

The problem is that we are in the directory and we are not allowed to cut off the branch on which we are sitting. So, we must go up one level and try again:

```
$ cd ..
$ rm Terminal_Testing
```

We will once again be presented with an error:

```
rm: Terminal_Testing/: is a directory
```

This indicates that we must remove this with the "recursive" flag for directories:

```
$ rm -r Terminal_Testing
```

[Alternatively the **rmdir** command could be used but why introduce a new command when this one will do the trick?]

There are still two left over folders in the current directory though, "Terminal" and "Testing". We could delete these one by one but there is a faster way: using wildcards.

```
$ rm -r Te*
```

> ***Question:*** *What would happen if the command issued was* ***rm -r **** *?*
> ***Answer:*** *Everything in the directory that we have permission to delete would be gone.*

It is important to note that you will often be prevented from deleting crucially important files because you are just a regular user on the account and not logged in as the *super user*. It is possible to become the super user on most systems by entering the command **sudo** in front of any other command. For those working as system administrators on UNIX-like systems a

common workflow is the following:

```
$ tell the computer to do X
computer says "No, you don't have permission"
$ sudo tell the computer to do X
the computer does it (after the right password is entered)
```

> ***Question:*** *What would happen if the command issued was* ***rm -rf /****?*
> ***Answer:*** *Everything goes since you are telling the computer to recursively remove everything in the root folder.*

Renaming Files

It is good practice to have a file extension on the end of file names to indicate the file type. *Important_Ideas* is a text file and by convention such files end with the extension *.txt*. Let's add this:

```
$ mv Important_Ideas Important_Ideas.txt
```

This is another example of how a tool with one function (moving files) can be used to do something that on the outside seems to be different from what we want but turns out to be exactly what is wanted (renaming = *moving* a file from its current name to another name).

Plumbing & Searching

Getting Files from the Internet

We need a file to do some manipulation with. There are three ways to do this:

1. Use **curl**.
2. Use **wget**.
3. Use your browser and save it to a directory

We will use option #1 but it really doesn't matter:

```
$ curl -L -O http://bit.ly/1ylW533
```

The "-L" flag tells curl to follow redirects and the "-O" (a capital letter o and not a zero/0) tells curl to keep the file name rather than making up its own.

Let's see what this file is:

```
$ cat 1ylW533
```

Whoa! That's a lot of text. Fortunately there is a command just looking at the top of a document:

```
$ head 1ylW533
```

So, we can now see that this is Moby Dick. We can control how much text we see by passing a number as a flag to head and we'll then only see that many lines.

```
$ head -3 1ylW533
```

If there is a command called "head" that will show the top of the file then perhaps there is command called "tail" that will show the bottom of the file and indeed there is:

```
$ tail 1ylW533
```

"1ylW533" is really a terrible file name. Let's do something about it.

> ***Comprehension Test:*** *Rename the file MobyDick.txt and make a copy of the file as a backup?*
> ***Answer:***

```
$ mv 1ylW533 MobyDick.txt
$ cp MobyDick.txt MobyDick-BackUp.txt
```

Regular Expressions

Let's find all the instances of 'whale' in MobyDick.txt.

```
$ grep whale MobyDick.txt
```

grep is the command to use a regular expression search tool on a specified set of files. While it looks like we are passing it a work what we are really doing is passing it a set of characters in the form of a "regular expression". While what we will be doing here will mostly consist of the word "whale" you need to see what they can do and for that we need to do two things:

1. Go to the course website and copy all the text under the **General Information** section.
2. Go to **http://regexpal.com** and paste that text into the box that says "Write your data here..."

[At this point spend a little bit of time showing them how RegexPal works so that they can just what is happening and get an idea for what they can do in the future. Make sure to point out the the quick reference menu and that they can overburden the tool by dumping too much text in the data panel.]

We can have it provide line numbers (like **cat**) with the -n flag

```
$ grep -n whale MobyDick.txt
```

While this tells us where the word "whale" is it does not tell us how many instances of "whale" there are. For this we need to introduce the word count command.

Redirects

There is a handy little tool that helps us count the words in a file:

```
$ wc MobyDick.txt
22108  215135 1257296 MobyDick.txt
```

The output is in the order "word, line, character". It is the sort of output that we might want to save for future use so we'll place the output in a file:

```
$ wc MobyDick.txt > MobyWordCount.txt
```

If we look in the directory (**ls**) we'll see a new file called "MobyWordCount.txt". If we look inside it (**cat**) we'll see the output of the word count command.

What has happened here is that the output of the word count command has been *redirected*

from what is called "standard out" (aka the screen) to a file that we specified. The single greater-than symbol is the most basic redirect symbol, taking the output from the command on the left and passing it to the file on the right.

> ***Question:*** *The single greater-than symbol is not the only redirection tool. There is also the double greater-than symbol ">>". What does it do?*
> ***Answer:*** *The double greater-than symbol appends output to the bottom of the file rather than over-writing it.*

Back to Regular Expressions

Coming back to the problem at hand, we want to count the number of times "whale" shows up in the story. To do this we'll need to combine both **grep** and **wc** together, taking the output of one and passing it to the other

```
$ grep whale MobyDick.txt | wc
1274   15119   87694
```

The "|" is the "pipe" character and on most keyboards it can be found with the backslash character ("\") just above the ENTER key. It can be accessed by holding SHIFT and pressing "\". What it does is take the output of the command on the left and pass it as an input to the command on the right. So, there are 1274 lines containing "whale".

As it stands though it is finding every instance of *the character string* "whale" rather than every instance of *the word* "whale". We can change ask it to search for words only by using the -w flag which will set the search to find one the specified string when it is wrapped in word boundaries (like spaces).

```
$ grep -w whale MobyDick.txt | wc
888   10591   60984
```

So, it should be clear that the number of times the word "whale" is used is less that the number of times the character string "whale" is used.

We're still not done yet though. Some lines might have "whale" appear in them more than once and we'd be missing those instances if they existed. To overcome this problem we'll need to trick the **wc** command into counting words as lines and to do this we need to make it the case that every word gets its own line. To do this we need to use the translate command, **tr**. See

what it does by trying the following

```
$ tr ' ' '\n' < MobyDick.txt
```

It has replaced every space with a newline character ("\n").

> ***Question:*** *How can* ***tr*** *be used with* ***wc*** *to output the number of lines?*
> ***Answer:***

```
$ tr ' ' '\n' < MobyDick.txt | grep -w whale | wc -l
```

Pipes

Now, if we want to do any processing of the MobyDick.txt file for research purposes we're likely going to need to cut off the extra text at the top and the bottom of the file. Important to note that Gutenberg ends its preamble with "*** START OF THIS PROJECT GUTENBERG EBOOK..." and begins its endnotes with "*** END OF THIS PROJECT GUTENBERG EBOOK..." Knowing this we can **grep** with the -n flag to search for these strings.

```
$ grep -n 'START OF THIS PROJECT GUTENBERG' MobyDick.txt
20  *** START OF THIS PROJECT GUTENBERG EBOOK MOBY DICK; OR THE WHALE ***
$ grep 'END OF THIS PROJECT GUTENBERG' MobyDick.txt
21750:*** END OF THIS PROJECT GUTENBERG EBOOK MOBY DICK; OR THE WHALE ***
```

[We are dropping the *'s from the **grep** command since it produces an error that isn't worth trying to sort out.]

> ***Challenge:*** *Using only* ***cat***, ***grep***, ***head***, ***tail***, *pipes and redirects create a file called MobyDick-Clean.txt that has the extra text removed from the top and the bottom?*
> ***Answer:***

```
$ head -21749 MobyDick.txt | tail -21729 > MobyDick-Clean.txt
```

> ***Challenge:*** *How many times does the word "whale" appear in MobyDick-Clean.txt?*
> ***Answer:***

```
$ tr ' ' '\n' < MobyDick-Clean.txt | grep -w whale | wc -l
907
```

Point out that there is *still* some additional text that could be culled (just run a **tail** on MobyDick-Clean.txt to see) but what matters is the process. You could refine this.

> ***Challenge:*** *Write a script that would do this for you again and again?*
> ***Answer:*** *What you are really being asked to do at this point is write a program and while we could use BASH to do this what we're going to do instead is leave BASH behind and move to Python because it is easier.*