To add...

1. Build slides for the challenge/comprehension questions
2. Add aside about existence and use of `find`

*nix Walkthrough

This is a script of sorts for running the Bash shell portion of a Software Carpentry Workshop with Digital Humanities researchers. It is based on content from the Software Carpentry shell-novice lessons.

Four things to make clear before heading into this workshop:

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

Yes, these will be brought up throughout the workshop as they arise but there are enough pitfalls for participants without having these snags waiting for them too.

Navigation

Open a BASH terminal window and you'll see:

```
"some information about the user/system" $
```

We'll ignore the "some information about the user/system" and just abbreviate this to "$". If you type the command: `PS1='$ '` into your shell, your window should look like all the examples in this workshop. This isn’t necessary to follow along (in fact, your prompt may have helpful information you want to know about). This is up to you!

Type:

```
$ whoami
```

Then press the enter key. The response will be *someusername*. More specifically, when we type `whoami` the shell:

1. finds a program called `whoami`,
2. runs that program,
3. displays that program's output, then
4. displays a new prompt to tell us that it's ready for more commands.

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*.

So, the command we actually need is:

```
$ pwd
```

This stands for "Print Working Directory" and you'll get something that looks like */Users/someusername*. But why? Enter slides #1 and #2

SLIDE 1 & 2: Cathedral vs Bazaar / Linux Flavours

Take a moment here to share a brief history of GNU-Linux. Core points:

1. How GNU-Linux Started (Richard Stallman)
2. How it developed (Eric S. Raymond)
3. Where it is now (Linux Flavours)

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` 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 by page rather than by line).

   ```
   $ 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 /
```

Slide 3: Discovering Flag Properties

> **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.

Remember `man -k search_word` for finding `man` commands. You may need to be creative with the terms you use.

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:** `$ cd /`

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 "`".

Slide 4: Going Home

> **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

Slide 5: Translation 1

> **QUESTION:** If someone is in /Users/nelle/ then how do they get to /Users/larry/ using cd and a relative address?
>
> **ANSWER:** `cd ../larry`

Creation & Destruction

Start this section by having each user navigate to their equivalent of `$ cd ~/Desktop/`.

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 Carpentry
```

Have them minimize their window and go and look at their desktop to see the new folders. 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).

Since we are doing command line now and python later we should create a command line folder. Let's move into the Carpentry directory, make a command line folder and then check that it exists:

```
$ cd Carpentry
$ mkdir Command Line Code
$ ls
```

Wait... Folder*S*!?! What happened?

The **ls** command will show that we have made a mistake: there are *three* directories---one called "Command", another called "Line", and another called "Code"---rather than one "Command Line" directory. This highlights two important things for the class to remember:

1. The computer does what you tell it, not necessarily what you wanted.
2. Spaces matter on the command line, they are punctuation.

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 "".
3. Wrapping content with spaces in double quotes.

Let's make the proper directory using underscores, move into that directory, create a sub directory and then move in there (We'll come back to clean up the extra folders later):

```
$ cd Command_Line
$ mkdir Command_Line
```

Creating and Viewing Files

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 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 because they hide other content even though they look like plain text.

> If people are using MobaXterm or some other shell that doesn't have nano installed by default then they can likely get it with one of the following commands:
>
> `$ apt-get install nano`
>
> or
>
> `$ yum install nano`
>
> apt-get and yum are package management tools that are likely installed already that can be used to install other software, in this case the text editor Nano.

We will run nano by issuing the command followed by the name of the file we would like to create/edit.

```
$ nano Important_Ideas1
```

This opens the nano program and allows the declared file to be edited/created. Nano 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_Ideas1
```

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_Ideas1
```

`cat` is the command to concatenate files. 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.
>
> Note that it appears that the -n flag does not funciton with MobaXterm.

Have participants make a new file called "Important_Ideas2" and add some new ideas to it.

> **QUESTION:** How can we print the content of Important*Ideas1 and Important*Ideas2 on the screen *as if* they were a single file?
>
> **ANSWER:** `$ cat Important_Ideas1 Important_Ideas2`

Slide 6: Describing command line format

> **QUESTION:** If you were asked to describe the format of a command on the command line what would you say?
>
> **ANSWER:** They should answer this by talking to someone beside them, actually describing the structure, until they both agree.

Redirecting Standard Out

This is useful, but what if we wanted actually make these into a single file? To do this we need to redirect the output of `cat` away from what is known as "standard out" (aka "the screen") to another place. In this particular case that other place is a file. We do this using the redirect operator:

```
$ cat Important_Ideas1 Important_Ideas2 > Important_Ideas
```

If we run `ls` we'll see a new file in the directory and if we run `cat` on this file we'll see that the contents of both original files are there. Nice.

What has happened here is that the output of `cat` has been *redirected* from "standard out" to a file with a name 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.

Note that using a `>` will *overwrite* content already in a file. If you want to add content to a file then we can append it to the bottom with the `>>` redirect. Try adding a new line to the bottom of the important ideas file by using the `echo` command and redirecting that to our file:

```
$ echo "here is another idea" >> Important_Ideas
```

Use `ls` and `cat` to verify that the contents have changed appropriately.

History and Piping Commands

You've done a lot so far and it might be helpful to see what you've done all at once, rather than the one-line-at-a-time view that using the up and down arrow keys gives you. There is a command for showing your past commands and you should try it now:

```
$ history
```

> **Question:** How can this be saved to a file?

> **Answer:** `history > history.txt`
>
> **Question:** When would we use `>>` in a case like this?

Why would we want to save this to a file? The number of lines held in the history is limited---500 on many systems---so eventually you'll lose what you typed before as it gets bumped. Further, we will often use commands we wrote to write scripts and so having these commands stored in a file can be really useful.

Once we have `history.txt` we can search it using a command called `grep`, or the **G**NU **R**egular **E**xpression **P**arser, for content. **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 word what we are really doing is passing it a set of characters in the form of a "regular expression" (more on this later). For now, let's suppose we know that we used a new command awhile ago but can't remember what it was called but we can remember that we looked its man page:

```
grep man history.txt
```

This will give us a list of all the lines in the history file. The history file that we made isn't current anymore though and there is a slicker way to do this with one final redirect symbol: `|`. This is known as the "pipe" character and it is found with the `\` character just above the enter key on most Western keyboards. With it we can pass the input of one command to another rather than chaining it to a file.

```
$ history | grep man
```

Whether you are trying to remember what you did or diagnosing what someone else did the `history` command is an important one to remember.

Lastly, you can run any line listed by the `history` command by typing an exclamation point and immediately following it with the appropriate line number. For example, if on running `history` we were told that line number 301 was an a particular command that we wanted to run again then we could run that command again with

```
$ !301
```

> **ASIDE:** `grep` is useful for searching *inside* files but it is often helpful to be able to *find* files in the first place. For this we use `find`.

We're now done in this directory so let's move back to ~/Desktop/Software_Carpentry/.

> **Question:** What is the most efficient way to move to ~/Desktop/Carpentry/ from ~/Desktop/Carpentry/Command_Line/:

> - `cd ~`
> - `cd ~/Desktop/Carpentry/`
> - `cd .`
> - `cd ..`
>
> **Answer:** `cd ..`

Cleaning Up

ACHTUNG: We are now at the point where you will learn commands that can seriously damage your system. *Be very sure that you understand what you are doing before you do it.*

We are done with the basics of the command line and about to move on to some "real world" examples and writing our own script. Before we get to this though we should do some quick clean-up. Specifically, we have some junk folders that we made "by accident", some files that it would be nice put together in a "trash" folder, and some files that would benefit from a ".txt" extension.

Let's start by getting rid of the folders in this directory that we don't need. Let's see what is here:

```
$ ls ..
```

Which should show us something like:

```
...
Code
Command
Line
Command_Line
...
```

We want to remove the three accidental folders. We can do this using the `rm` command, which *removes* files from the system. Let's start by testing this with a new junk file:

```
$ touch junkFile
```

`touch` is the command used to change the modification and access times associated with a file. Like `cat` it is overloaded. In this case the overloading creates a file with no content if the file does not already exist and no other parameters are given. Use `cat` to look inside and `ls -l` to confirm that this is the case.

Once you have confirmed that the file is there let's remove it and then check that it is gone:

```
$ rm junkFile
$ ls
```

There is *no* recycle bin on the command line. Once you "rm" something it is truly gone (except for the possibility of some very advanced forensics).

Let's try and remove `Line/` (We'll save `Command` and `Code` for a little trick):

```
$ rm Line
```

Running this command gives us a warning though and will not complete. We are told that these are directories. Directories are just files at their core but they are special files that hold/point to other files and so we cannot simply delete them without deleting their contents.

> **Question:** How do we see what files are in these directories that we cannot delete?
>
> **Answer:** `ls -a`

What is holding us back are the . and .. files. If you try to delete them you will be told: `"." and ".." may not be removed` (Think about sitting at the end of a tree branch and cutting it off). What we need is a special flag to use with the `rm` command. That flag is `-r` which is the "recursive" flag, telling the command to enter a directory and remove everything inside it, all the way to the bottom. Let's try it

```
$ rm -r Line
$ ls
```

And it is gone.

Note that like `cat` we can pass multiple files at the same time to `rm` and it will delete each one. If you have a lot of files to delete the this can still be tedious. Fortunately there is a wild card character. Let's test it with some junk files:

```
$ touch junk1 junk2 junk3
$ ls
$ rm junk*
$ ls
```

Nice. Now let's remove `Command/` and `Code/`:

```
$ rm Co*
```

Only *think* about the next question. Do not figure it out by running it.

Slide7: A serious mistake

> **Question:** *Without running this command*, 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. For some "real-world" consequences see this unfortunate forum post. It is possible that you will have some permission or other security mechanisms in place to prevent this but if you are a full administrator it can be done.

Slide 8: Make me a sandwich

> Often you will be prevented from doing dangerous things---like deleting crucially important files---because you are just a regular user 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)
```

> You can try sudo with *any* command to see how it works. Let's try it with `ls`

```
$ sudo ls
Password:
<list of directory contents>
```

> XKCD has a nice little web comic to bring home how `sudo` is used.

Let's finish up by looking in the Command_Line folder using ls:

```
$ ls Command_Line
```

Which will give output that looks like:

```
Important_Ideas
Important_Ideas1
Important_Ideas2
history.txt
```

We really don't need Important*Ideas1 or Important*Ideas2. We could delete them but maybe we want to keep them around a while, just not in this directory. So, let's create a new directory called "recycle" inside the ~/Desktop/Carpentry directory and move the files in. You already know how to create a directory. The move command is `mv` and you know how to figure out how commands work so go to it. =)

```
$ mkdir ../recycle/
$ mv I*1 I*2 ../recycle/
```

Note the use of the `*` as a wild card character. Also note that here the syntax with the list of files is that the *last* file is where all those before it are put.

The directory structure of Command_Line should now look like:

```
$ ls
Important_Ideas
history.txt
```

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).

"Real World" Examples

Getting Files from the Internet

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

1. Use `curl`. (Installed on OSX, MobaXterm, and many Linux flavours by default)

2. Use `wget`. (Not installed on OSX by default)
3. Use your browser and save it to a directory

We will use option #1 but it really doesn't matter as long as each program is installed. Here's how to do it with curl:

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

The "-L" flag tells curl to follow redirects and the "-O" (a capital letter o and not a zero, think of it as the "Output" flag) tells curl to write the content of the url to a file.

Let's see what this file is:

```
$ cat MDxHM
```

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

```
$ head MDxHM
```

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 MDxHM
```

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 MDxHM
```

"1TTicb6" 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 MDxHM MobyDick.txt
>
> $ cp MobyDick.txt MobyDick-BackUp.txt

Regular Expressions

Now that we know what file we have let's set ourselves a challenge: find all the instances of the word 'whale' in

Moby Dick.

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

Which will give us each line with the character string whale on it. We can have `grep` 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.

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

> **QUESTION:** How can you figure out what the numbers mean?
>
> **ANSWER:** `man wc` or Google or some testing and the output is in the order "word, line, character".

So we can count the number of lines that the character string whale appears on but...

> **QUESTION:** What is wrong with the returned information? What is it catching or overlooking that it shouldn't be? Think about how you've seen `grep` work.
>
> **ANSWER:** There are two problems:
>
> 1. Could be more than one instance on a line
> 2. We're grabbing things that aren't really the word "whale", such as "whalers"
>
> We'll deal with the second case first by writing a better regular expression, one that captures only "whale" or "whales".

Here we will spend a little bit of time focusing on getting only the word whale and not other words that don't count' like "whaler". We'll visit https://regex101.com/ and work out the regular expression that should do the trick: `\b[Ww]hales?\b` [or `(\bWhale\b)|(\bWhales\b)|(\bwhale\b)|(\bwhales\b)].

When we come back to the command line and try it though it will be prettry clear that it doesn't work since it returns nothing. The problem is that we are running into some confusion around interpreting the special characters. There are at least four remedies:

1. `$ grep -n "\b[Ww]hales\?\b" MobyDick.txt`
2. `$ grep -nw "[Ww]hales\?" MobyDick.txt`

3. `$ grep -n \\b[Ww]hales\\?\\b MobyDick.txt`
4. `$ grep -nw [Ww]hales\\? MobyDick.txt`

The quotes in #1 and #2 ensure the entire expression is evaluated by grep instead of by the user's shell. We could use single quotes as well, we just need to do so consistently. We'll use #4 since it requires the least typing

> **QUESTION:** How do we combine grep and wc to count the number of lines that "whale" appears on?
>
> **ANSWER:**
>
> grep -nw [Ww]hales\? MobyDick.txt | wc
>
> 1435 16792 105936

At this point we've solved the problem of not grabbing exactly the right content (Unless someone really wants to be picky about hyphenated words).

We're still not done yet though since the first problem remains: "whale" may appear twice in a line and we're ignoring those instances if they exist. To overcome this problem we'll need to trick the `wc` command into counting words as lines.

> **QUESTION:** What would we need to do to the MobyDick.txt file to use `wc` to count words (note that "what" is different from "how", we'll worry about the "how" next)?
>
> **ANSWER:** Make it the case that every word gets its own line.

Now that we know *what* we need to do we can worry about *how*, which is answered as follows:

```
$ cat MobyDick.txt | tr ' ' '\n'
```

> **ASIDE:** The above can also be accomplished by:
>
> $ tr ' ' '\n' < MobyDick.txt
>
> Whether this is noted or used is a matter of assessing the current state of the class

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

> **Question:** How can **tr** be used with **wc** to output the number of lines? **HINT:** Multiple pipes.
>
> **Answer:**
>
> $ cat MobyDick.txt | tr ' ' '\n' | grep -nw [Ww]hales\? | wc -l

Four commands chained together to process your file is pretty sophisticated. It's also pretty simple once you understand how the redirects work and know the commands. It gets even better when you can turn it into a script and generalize it such that you can pass it *any* regular expression and *any* file...

Scripting

Everything that we have done so far can be wrapped up into a simple program called a script. The term "script" is usually reserved for short programs that tell other programs or tools what to do. It might be useful to think of them as recipes.

Note that there are different languages that scripts can be written in. In this lesson we're working with Bash so we'll be doing *Bash Scripting*. Other languages, such as Python, R, Ruby, Java, etcetra, have their own syntax. Regardless the general principles remain the same.

We'll see how Bash Scripts work by generalizing the solution given at the end of the previous section.

Building and running a simple program

We'll begin by using nano to create a new file:

```
$ nano corpusWordCounter.sh
```

Note the use of the ".sh" at the end of the file name. This extension is typically used to designate **sh**ell scripts. It is not necessary but is a useful courtesy to anyone trying to figure out what a file is or does.

Once inside nano type the following (note that you could copy and paste from the terminal in advance) and then exit and save the file:

```
#! /bin/sh
# counts the character string

cat MobyDick.txt | tr ' ' '\n' | grep -nw [Ww]hales\\? | wc -l
```

If we look in the directory now we'll see a new file:

```
$ ls
[Bunch of other files]
corpusWordCounter.sh
```

We can check the content of this new file with `cat` :

```
#! /bin/sh
# counts the character string

cat MobyDick.txt | tr ' ' '\n' | grep -nw [Ww]hales\\? | wc -l
```

We can run it by doing the following:

```
$ sh corpusWordCounter.sh

1504
```

Typing `bash` tells Bash (our shell/terminal environment) to interpret the contents of "corpusWordCounter.sh" as instructions given on the command line and to run each line as such. We can stack any number of lines in this file and then execute them sequentially.

> **ASIDE: If someone wants to make the script an executable file (i.e. a file that can be run like a program rather than passed to a program to be run) then this can be done with: `$ chmod u+x corpusWordCounter.sh`. Chances are that the current directory is not in their PATH so the script will need to be run with `$ ./corpusWordCounter.sh`. If they want this to behave like any other command then they can either add this directory to the PATH (not usually recommended) or move the file to a directory like `/usr/bin` or `/usr/local/bin/` which is where most of the standard linux commands can be found (use `$ echo $PATH` to see a list of possible locations separated by `:`).]

Telling people what you are doing

Writing scripts is a really excellent way to economize on time. A little bit of effort put in upfront will have a lot of returns in the future by handling repetitive tasks quickly. This benefit is lost if looking at the script in six months requires puzzling through what it does and why or figuring out how exactly it fits into the workflow again. Adding comments to the script is a way to overcome this.

It is easy to either forget to add comments or to make them useful only in the moment. Keep in mind that the person most likely to read them is a future self. Help yourself out and err on the side of too much information, particularly when you are learning.

Every program you write should have as a minimum:

1. a comment at the top that serves as a description of what the script as a whole does plus who wrote in and when it was last updated.
2. comments at the start of each set of commands that are meant to run as a whole to complete a task.

We add comments to Bash scripts by beginning the line with a `#` (accessible on most keyboards by holding SHIFT and pressing 3), which is commonly known as a hash or pound sign.

Using nano we can change the content of corpusWordCounter.sh to read:

```
# Description: Takes all the txt files in a directory and then counts how often a spe
cified word appears.
# Author: Your Name
# Last modified: February 16, 2016

cat MobyDick.txt | tr ' ' '\n' | grep -nw [Ww]hales\\? | wc -l
```

If we run corpusWordCounter.sh again using `bash` we see no difference in the output. This is as it should be since comments are ignored by the execution engine.

```
$ bash corpusWordCounter.sh

1604
```

Generalizing the script

The power of a program or script is further expanded if it can be generalized such that it can be used for a range of similar tasks. Programs are easily generalized by introducing variables, little bits of code that can take on a range of different values.

By allowing someone to specify the word they want to count within the file on the command line and then passing that to the script as a variable we can build a tool that won't just count how often "whale" shows up in a file but how often *any* given word shows up.

In Bash, variables are passed from the commandline on the same input line as the call to run the script by listing these variables after name of the script.

Within the script we access these variables in the order that they were presented by placing a `$` immediately in front of the numeral representing the order that the variable appearded in the command. So, a `$1` (no spaces!) would access the first variable that was passed, a `$2` the second variable, and so on.

It is often adviseable to wrap these variable names in quotes so that if what has been passed to them has spaces Bash understands that the space is not being passed a string of variables separated by spaces but rather one variable that contains spaces. File names are a case in point. "My\ File.txt" on the the commandline would become two separate instructions---"My" and "File.txt"---as a variable if the variable is not wrapped in quotes. Of course there are exceptions to this and you might want to play around to see if you can find the exceptions to this advice.

In our script we will use nano to replace "whale" with `"$1"`:

```
# Description: Takes all the txt files in a directory and then counts how often a spe
cified word appears.
# Author: Your Name
# Last modified: February 16, 2016

cat MobyDick.txt | tr ' ' '\n' | grep -w "$1" | wc -l
```

Now, if we wanted to run the script to search for the word "horse" we would run:

```
$ bash corpusWordCounter.sh horse
```

If we wanted to specify exactly which text to search the word for we could do that as well by adding a second variable to the script in place of MobyDick.txt:

```
# Description: Takes a specified text file and then counts how often a specified word
 appears within that file.  To run the script type: bash <script name> <word to searc
h for> <file to search in>
# Author: Your Name
# Last modified: February 16, 2016

 cat "$2" | tr ' ' '\n' | grep -w "$1" | wc -l
```

Note that the description was also updated to make it clear what the syntax is for running the script. Making sure that your instructions remain accurate and useful is important for saving your future self from frustration.

> Comprehension Test: Modify the program so that the file name is the first variable passed on the commandline and the word to search for the second.

Answer:

```
cat "$1" | tr ' ' '\n' | grep -w "$2" | wc -l
```

Repeating the script

When scripting/programming being able to repeat an action is on par with being able to use variables for generalizability in terms of importance. Repeatability can be obtained by copying and pasting the same lines over and over again in the script but it is typically more efficient to use the control structure known as a loop.

There are two types of loops but for the purposes of this demonstration we will only make use the *for loop*, leaving the others to be covered in the Python portion of this workshop.

The setup for a for loop in Bash is:

```
for <variable name you choose> in <some list values to pass to the variable>
do
    <command #1>
    <command #2 with $<variable name you choose> >
    ...
done
```

The list of values to pass to the variable and iterate over is all the files in the current directory that end in .txt and we can pick all of those out with *.txt. We already have the command that we want to run and just need to change the `$2` to some variable that we choose and initialize at the start of the loop. So, we modify *corpusWordCounter.sh* as follows:

```
# Description: Takes all the txt files in a directory and then displays how often a s
pecified word appears for each file.  To run the script type: bash <script name> <wor
d to search for>
# Author: <Your Name>
# Last modified: <Date YOU created/modified this file>

for file in *.txt
do
    echo $file
    tr ' ' '\n' < "$file" | grep -w "$1" | wc -l
done
```

Note the addition of the `echo $file` command. This command ensures that we know *which* file the word counts are being reported for. Note as well the modification of the description to accurately reflect what the script does with these modifications.

A Challenge: Cutting Heads and Tails

Now, if we want to do any processing of the MobyDick.txt file or any other such 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. It is important to note that Project Gutenberg books end their 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 and find their line numebers.

```
$ 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
```

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:** Turn the cleaning method into a script that can be run on a collection of texts.
>
> **Answer:** This is another script building challenge and the answer is more advanced than the previous one, but only a little. If we focus on *planning* the script first then the rest becomes a matter of finding the right syntax and that can often be done online or by asking a friend.

Planning the script

To plan the script we need to deconstruct the process used to make this line of code run:

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

To run the above line of code successfully as a script we need:

1. The name of the input file
2. The line number for cutting the head
3. The line number for cutting the tail (produced via subtraction)
4. The name of the output file as a derivative of the input file

Recognizing this we will plan a script and then use this plan to write the script.

Open a new text file using the nano editor:

```
$ nano GutenbergCleaner.sh
```

Within this file we will add the following comments:

```
# Description: Removes the preamble and postscript from a specified Project
# Gutenberg file creating a clean version in the same directory.
# Author: <Your Name>
# Last modified: <Date YOU created/modified this file>

# Take a file name as input

# Find the start and end lines of the file with line numbers grep with -n flag

# Use the start and end lines to determine where to make the second cut

# Determine the name of the output file

# Make the cuts and redirect to the output file

# repeat
```

Filling in the Plan

With the plan laid out we will start to fill in content comment by comment.

The first comment asks us to take a file name as input, which we can do using out `$` notation:

```
# Take a file name as input
file = $1
```

Next we find the starte and end line numbers:

```
# Find the start and end lines of the file with line numbers grep with -n flag
startNum=$(grep -n "START OF THIS PROJECT GUTENBERG" $file | cut -d: -f1)
endNum=$(grep -n "END OF THIS PROJECT GUTENBERG" $file | cut -d: -f1)
```

Here a command that has not been shown before is having the output of grep with the -n flag piped to it: `cut` . This command splits up a line at the character that immediately follows the -d (divide) flag and from the list that results returns the list item numbered with the -f (find) flag. In this case we are cutting at all colons and returning the very first string of characters from the resulting list, which is the line number that START/END OF THIS... was found on.

So, if `grep -n "START OF THIS PROJECT GUTENBERG" $file` returned `108: *** START OF THIS PROJECT GUTENBERG...` then piping this output to `cut -d: -f1` would returne `108`.

Now that we know the start and end numbers we can calculate where to make the second cut:

```
# Use the start and end lines to determine where to make the second cut
(( headCutNum = $endNum - $startNum ))
```

Note the use of double parentheses. This is to alert the Bash Shell that we are performing math and not issuing commands on the shell

We determine the name of the output file using a trick from a website and so we acknowledge that in the comments:

```
# Remove the .txt from the filename using a trick from
# http://www.thegeekstuff.com/2010/07/bash-string-manipulation/
outFile=${file%.*}-clean.txt
```

Finally we perform the action and place the output in a file:

```
# Make the cuts and redirect to the output file
head -$endNum $file | tail -$headCutNum > $outFile
```

Making the entire script:

```
# Description: Removes the preamble and postscript from a specified Project
# Gutenberg file creating a clean version in the same directory.
# Author: <Your Name>
# Last modified: <Date YOU created/modified this file>

# Take a file name as input
file=$1

# Find the start and end lines of the file with line numbers grep with -n flag
startNum=$(grep -n "START OF THIS PROJECT GUTENBERG" $file | cut -d: -f1)
endNum=$(grep -n "END OF THIS PROJECT GUTENBERG" $file | cut -d: -f1)

# Use the start and end lines to determine where to make the second cut
(( headCutNum = $endNum - $startNum ))

# Remove the .txt from the filename using a trick from
# http://www.thegeekstuff.com/2010/07/bash-string-manipulation/
outFile=${file%.*}-clean.txt

# Make the cuts and redirect to the output file
head -$endNum $file | tail -$headCutNum > $outFile
```

> **Challenge:** You will note that the cuts are off by just a few lines. Modify the script to correct this.

Generalizing and Error Trapping

Of course, the real value of a script like this is if it can process a large number of files in a single run. Here's what such a script might look like. Note the changes from the previous version.

```
# Removes the preamble and postscript from any specified Project Gutenberg
# file in a directory.

for file in *.txt
do
    # Find the start and end lines of the file with line numbers
    # Using grep with the -n flag to get the lines with the line number out front
    # Pipe this to the cut tool which will divide the line at the colon and take
    # the first field
    startNum=$(grep -n "START OF THIS PROJECT GUTENBERG" $file | cut -f1 -d:)
    endNum=$(grep -n "END OF THIS PROJECT GUTENBERG" $file | cut -f1 -d:)
    # echo $startNum
    # echo $endNum

    if [ -n "$startNum" ]
    then
        if [ -n "$endNum" ]
        then
            # Do the endNum minus startNum math
            (( headCutNum = $endNum - $startNum ))

            # Remove the .txt from the filename
            # trick from http://www.thegeekstuff.com/2010/07/bash-string-manipulation
/
            outFile=${file%.*}
            # echo $outFile

            # Output the clean version using head to cut the tail and
            # tail to cut the head.
            head -$endNum $file | tail -$headCutNum > "$outFile"-clean2.txt


        fi
    fi
done
```

> **Challenge:** Go back to the script that finds a word in all the .txt files within a directory and modify it so that at then end it prints the *total* count of the word across all the processed files.

Some Fun

On any Debian based system (Ubuntu), try "apt-get moo". If you have aptitude installed try, in sequence:

```
aptitude moo
aptitude -v moo
aptitude -vv moo
aptitude -vvv moo
aptitude -vvvv moo
```

play NetHack on line at https://alt.org/nethack/

The command `rev` which is part of the util-linux package reverses any text you feed to it:

```
fortune | rev
.retteb hcum ,hcum ylno ... won thgir era uoy erehw ekil yltcaxe si esidaraPnosrednA
eiruaL --
```

Install `links` or `lynx` to browse the web

Visit:

```
ssh sshtron.zachlatta.com
```

Then compete! Use WASD or HJKL (vim) to move the specific direction.

```
telnet towel.blinkenlights.nl
```

One last thing to add

How to get your terminal to have fancy colours and other formatting when looking at files *and* how to remove this should you want to not have people get all worked up because their screen doesn't look like yours and you don't want to go down that path because it is really a rabbit hole for another time... =)

One last thing to remember

Before we move to looking at version control and a programming language you'll likely find it useful to save a list of all the commands that you have used so far. You can do this by navigating to a directory where you would like to save them and then running the *history* command with a redirect to a file.

```
$ history > history_file.txt
```

Two other things to note about the history command:

1. it can be combined with **grep** to find commands you have forgotten.

   ```
   $ history | grep cat
   ```

2. it is the history file that you are scrolling through when you press the up arrow on the commandline to avoid typing the same commands over and over.