To add/modify:

1. recall previous commands from history with ! . Just ! for the last command.
2. set up scripting piece to focus on producing a clean file first and then finding words in the clean file. This represents the workflow better and will be a better set up for Dennis because I can just stop when the file is clean and he can start "hunting the whale."

*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 could 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 other 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*. 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 necessarily 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. To see some "real-world" consequences of this see this unfortunate forum post.

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 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/1TTicb6
```

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 1TTicb6
```

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

```
$ head 1TTicb6
```

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 1TTicb6
```

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 1TTicb6
```

"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 1TTicb6 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.

Pipes

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
```

or

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

> **Question:** How can this be modified to search for any word on all the files in a directory?
>
> **Answer:** What you are really being asked to do at this point is write a program because doing it over and over by hand would cost a lot of time with a lot of potential for human error. Let's look at how to do this now.

Scripting

[Goal here is to have them build a program by hand to answer the previous challenge. We'll do this line by line, no black boxes.]

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. Think of them as recipes of instructions.

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.

Let's see how Bash Scripts work by solving the problem 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:

```
tr ' ' '\n' < MobyDick.txt | grep -w whale | 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`:

```
$ cat corpusWordCounter.sh

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

We can run it by doing the following:

```
$ bash corpusWordCounter.sh

907
```

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.

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

tr ' ' '\n' < MobyDick.txt | grep -w whale | 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

907
```

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

tr ' ' '\n' < MobyDick.txt | 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

tr ' ' '\n' < "$2" | 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:

```
tr ' ' '\n' < "$1" | 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
```

> **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:** 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 program 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 esidaraP nosrednA eiruaL --

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

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

telnet towel.blinkenlights.nl

One last thing

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.