Modified from content HERE, HERE, and HERE with thanks to the earlier contributors Cam Macdonell, Tracy Teal, Greg Wilson, Dennis Tennen, Paul Wilson, Milad Fatenejad, Sasha Wood, and Radhika Khetani.

*nix Walkthrough

This is a script of sorts for running the Bash Shell portion of a Data 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.

Shell Overview

The *shell* is a program that presents a command line interface which allows you to control your computer using commands entered with a keyboard instead of controlling graphical user interfaces (GUIs) with a mouse/keyboard combination. The terms “shell” and “command line” are often used interchangeably. When most people talk about working in a shell or on a command line the operating system that they are using is usually a flavour of GNU-Linux and will often be assumed to be the case.[1]

There is more than one shell and each shell can come in different versions. The shell that we will be using today is the most common. It is known as BASH, which stands for “Bourne Again SHell”.[2]

There are many reasons to learn about the shell.

- For some tools, such as Mallet (a popular topic modeller) you have to use the shell. There is no graphical interface. If you want to collaborate with people from the sciences this will become even more common.
- Most shells were written in the C language to process text context. They are very fast and can give you the power to do your work more efficiently and more quickly. When you need to do things tens to thousands of times, knowing how to use the shell is transformative.
- Using the shell gives you the ability to have repeatable data processing.
- To use remote computers or cloud computing, you need to use the shell.

- We're going to use it in this class, for all of the reasons above.
- If you are going to work with GNU-Linux, the standard OS for scientific and high-performance computing, then you'll end up here sooner or later.

In this portion of the workshop we're going to:

- learn how to use BASH by using BASH
- learn about why shells are the way they are
- automate a repetitive process

Information on the shell

Shell cheat sheets:
* http://fosswire.com/post/2007/08/unixlinux-command-cheat-sheet/ * https://github.com/swcarpentry/boot-camps/blob/master/shell/shell_cheatsheet.md

Explain shell - a web site where you can see what the different components of a shell command are doing. * http://explainshell.com * http://www.commandlinefu.com

How to access the shell

The shell is already available on Mac and Linux. For Windows, you'll have to download a separate program.

Mac

Mac OSX is built on top of a version of BSD (Berkeley Software Distribution), a series of Unix variants. As a consequence it has a shell built in that will do everything we need it to today. To access the shell either:

- From the Finder go through the following folders: Applications -> Utilities -> Terminal.
- Hold down the Command Key and press the Spacebar. In the Spotlight search window that appears start typing "Terminal".

Regardless of the method used you will likely find it useful to drag the Terminal application icon to your Dock for easy access in the future.

Windows

Windows is its own operating system with its own development history and as a result it has something that *looks* like BASH but that isn't, the Command Prompt. So, we'll need to install a program that emulates BASH

on Windows. They're are more than a few options for this but we're going to recommend the home, portable edition of MobaXterm. The portable edition installs everything in one folder, making it easy to find and easy to remove if you decide you don't like it. We recommend installing it on your desktop so it is easy to find.

MobaXterm has the ability to install software that we might need later and we will make use of this feature as the need arises. One tool that we will need to install outside of the shell is Git (this is a version control tool but we will use it to make sure you have a copy of all the workshop content). To install it go to the MobaXterm Plugins Page and download the git plugin. This will put a file named "Git.mxt3" in your downloads folder. Just drag and drop this file into the MobaXterm folder and you're done.

GNU-Linux

GNU-Linux, even versions with Graphical User Interfaces, is built on/around a shell environment so you will have one installed by default. There are many variants of GNU-Linux so describing how to get to the shell is unlikely to be of much help. Fortunately, most GNU-Linux users usually come to the workshop knowing how to access the shell on their system. If this is not the case then just let someone who is part of the workshop know and they'll help you out.

Note that it is possible that your shell isn't BASH. In somecases this won't matter and you can likely follow along since most of the commands we will look at today will be interchangeable across shells. If this proves challenging then you can likely switch to BASH by typing `$ bash` at the command prompt (we'll explain this next). If you are unsure what shell you have you can type `$ echo $0` and you should see `-bash` as the result if you are using BASH.

Entering Commands & 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 "$" in the command examples shared here. Don't type the "$" it's there just to tell you to type what is after it. If you see content in the text block that doesn't have a prompt then it is either wraping over from the line above or output, context will usually make this clear.

> If you want to give yourself some extra space by removing the "some information about the user/system" for the duration of this BASH session then 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! [3]

Let's get started. Type:

```
$ whoami
```

Then press the enter key. The response will be `someusername` (Hopefully yours!).

What is happening here when we type `whoami`? A process along the lines of the following takes place:

1. the computer looks for a program called `whoami` in what is known as the PATH, a set of directories it expects to find programs in.
2. it runs that program
3. it displays that program's output, then
4. it 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 doesn't `$ whereami` work? 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)

Basics of Looking, Moving, and Creating

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.

We can also look around from where we are. It is really likely that when you ran `ls` that it told you that there was a folder called "Desktop" in your current directory. Let's see what is inside there:

```
$ ls Desktop
```

What you see when you run this command should be exactly what is on your Desktop. To confirm this you are encouraged to use your Graphical User Interface to change your desktop contents and then run the command again.

Rather than just look at the Desktop folder let us make it the active directory:

```
$ cd Desktop
```

`cd` is the *change directory* command and it allows us to change what our active directory is within the shell. To see what this means run the `ls` command again and note how the result is now the same as when `ls Desktop` was run when your home directory was the active one.

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 Data Carpentry
```

> 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).

Double check that the folder is ther by using `ls` too. Wait… Folder*s*!?! What happened?

The **ls** command will show that we have made a mistake: there are *two* directories—one called "Data" and another called "Carpentry"—rather than one "Data Carpentry" 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 and move into that directory (We’ll come back to clean up the extra folders later):

```
$ mkdir Data_Carpentry
$ cd Data_Carpentry
```

> If **scrolling through the history** with the arrow keys has not come up yet then this is a good time to prompt it.

Now we’ll get the data that we need for the rest of this portion of the workshop by issuing the following command:

```
$ git clone https://github.com/ComputeCanada/dhsi-coding-fundamentals-2018
```

(For this to work you need both a tool called git installed and a working internet connection on the computer you are using. If it fails just let someone helping with the workshop know and we’ll get you helped out.)

Git is a tool designed for managing software development by providing version control. One of the nice bonuses tha comes along with using it is that you are able to make copies of other people’s work as long as they have been shared within an online space for this purpose (these spaces are usually known as “repositories”). Git is covered in Software Carpentry so this is all we’re going to do with it today.

Beyond the Basics of Looking, Moving, Creating, and now *Destroying*

If successful the `git clone` command will have made new directory on your desktop called “DC-shell_automation”. Let’s move into it:

```
$cd DC-shell_automation
```

> If **tab completion** has not come up by now then this is the time to introduce it.

Once you are inside this directory have a look around:

```
$ ls
```

Once you've seen what's here try the following variants of the `ls` command to see what happens:

```
$ ls -a

$ ls -l

$ ls -F
```

These new options are called "flags" and adding them to the command turns on (or sometimes off) various features. `-a` is the "show all" flag and it will show all files in the directory. `-l` is the "long format" flag and it will give more information about each file. `-F` is the "disambiguate folders" flag and will put a `/` after folders and a `*` after exectuables.

There are often a lot of flags available for each command. To see what they are for each command you have two options:

1. Use the "man" pages if they are available.

   ```
   $ man ls
   ```

2. Search on the Internet. This will be the case for those using MobaXterm (sorry these are not installed by default and installing them is beyond the scope of this workshop).

> **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.
>
> **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 “`”.

Actually, there is even a shortcut for the shortcut! `cd` is the equivalent of `cd ~`.

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, 8, 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`

> **EXERCISE:** A treasure hunt. Find the file named “youfoundit.txt” located somewhere within the hidden folder in the content that we pulled down from GitHub using the GitClone command

Working with Data

Going to Use a modification of Cam’s setup here. Have the EPL Hold data and want to know which books have been in the top ten the longest. Will first do this by branch and then do it for all branches to show off the power of * and the terminal in general. Finally, will make it a script so that it can be run any time an update to the dataset is pulled in.

Consider the following data as an example of a spreadsheet that a librarian may find themself working with.

Our task is to figure out the top ten titles for 2016 at a given library branch in Edmonton. We could do this with a spreadsheet program but only if we only had to do it a few times. After a point being able to write a script to automate a process becomes really valuable.

We could also use a programming language to do this but we’re going to wait until tomorrow to see what we can do there. Besides, there are things that can be done on the command line much faster than the

alternatives once the commands are known. The point here is not an expectation that you'll be a shell scripting expert after this but rather that you'll have a strong sense about what is possible.

Viewing files

We'll start by moving into the LibraryData folder and having a look at the content of the files.

```
$ cd LibraryData
$ ls
EPLABB.csv      EPLIDY.csv      EPLMLW.csv
EPLCAL.csv      EPLJPL.csv      EPLMNA.csv
EPLCLV.csv      EPLLCP.csv      EPLRIV.csv
EPLCPL.csv      EPLLHL.csv      EPLSPW.csv
EPLCSD.csv      EPLLON.csv      EPLSTR.csv
EPLGMU.csv      EPLMCN.csv      EPLWHP.csv
EPLHIG.csv      EPLMEA.csv      EPLWMC.csv
```

Each of these files is a listing of the top ten books with the highest number of holds at Edmonton Public Library, each week from 9 AM Monday to 9 AM Monday, organized by customer home library branch. These are comma separated value files.

We can look into them individually by using the concatenate command on the first CSV file:

```
$ cat EPLABB.csv
Row ID,Branch ID,Branch Name,Number of Holds,Title,Author,As of Date,Web URL
EPLABB20150316Fifty shades of Grey / E.L. James,EPLABB,Abbottsfield - Penny McKee Bra
nch,5,Fifty shades of Grey / E.L. James,James E L,03/16/2015 12:00:00 AM,http://epl.b
ibliocommons.com/search?t=smart&q=fifty%20shades%20of
EPLABB20150316The back of the turtle : a novel / Thomas King,EPLABB,Abbottsfield - Pe
nny McKee Branch,6,The back of the turtle : a novel / Thomas King,King Thomas,03/16/2
015 12:00:00 AM,http://epl.bibliocommons.com/search?t=smart&q=the%20back%20of
...
```

`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.

chances are that this looks terrible on your screen, making it a real challenge to see what's there. Instead of `cat` we can use a special command just for looking at the top of files:

```
$ head -n 1 EPLABB.csv
Row ID,Branch ID,Branch Name,Number of Holds,Title,Author,As of Date,Web URL
```

`head` allows us to just look at the head of a file. The optional `-n` flag followed by a number tells it how many lines to return. Increasing the flag value to 2 will give us both the header *and* the first line, allowing us to see what we are working with without being overwhelmed.

```
$ head -n 2 EPLABB.csv
Row ID,Branch ID,Branch Name,Number of Holds,Title,Author,As of Date,Web URL
EPLABB20150316Fifty shades of Grey / E.L. James,EPLABB,Abbottsfield - Penny McKee Bra
nch,5,Fifty shades of Grey / E.L. James,James E L,03/16/2015 12:00:00 AM,http://epl.b
ibliocommons.com/search?t=smart&q=fifty%20shades%20of
```

Our process from here is going to roughly be the following:

- get rid of the header
- extract the title from each line
- count the number of times each appears
- sort the resulting list

While we are about to go through the details on how to do this note that typically the details matter much less than being able to imagine how a file might be processed in the first place. The details are just syntax. Here goes!

We can cut the header by using the sister tool to the `head` command with a special input to the number flag:

```
$ tail -n +2 EPLABB.csv
```

Usually when `tail` is passed the `-n` flag it is given a number *without* the `+` and `tail` then counts from the bottom of the file up the number of lines specified and returns those lines. We don't know how many lines from the bottom we are though. By adding the `+` we are telling tail to count from the top and to take everything from that line on down.

So, we can now have the file print on the screen without the header row. The next step in our plan is to extract the titles and to do this we'll need to pass what is currently being printed on the screen to another tool and to do this you will need to learn about the redirect character: `|`. 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.

Let's start with an example. Suppose that you wanted to check that using `+` with `tail` was inclusive and you wanted to avoid scrolling all the way to the top to check that the line just below the header was all that was returned. You could do this by "piping" the tail command to the head command, as follows:

```
$ tail -n +1 EPLABB.csv | head -n 1
Row ID,Branch ID,Branch Name,Number of Holds,Title,Author,As of Date,Web URL
```

What we need to do is take the output of our `tail` command and pipe it to the `cut` tool. Cut allows for lines in files to be sliced at a specified delimiting character into fields and then for the specified fields to be returned. We know that we have a *comma* separated values file and by looking at our header row we see that the title is the fifth item so we can issue the following command:

```
$ tail -n +2 EPLABB.csv | cut -d , -f 5
Fifty shades of Grey / E.L. James
The back of the turtle : a novel / Thomas King
Minecraft annual / [edited by Jane Riordan ; written by Jane Riordan with 3 others]
Yes please / Amy Poehler
Crash & burn : a novel / Lisa Gardner
...
```

That's pretty neat. There is a problem though, some of the lines are not returning the title / author combo that we expect?

> **Question:** What is happening here and why?
>
> **Answer:** `cut` isn't clever enough to know that it is dealing with a CSV file and so it treats commas as commas no matter where they are. The lines that are returning EPLABB actually have commas in the title of the book and this shifts the field that is returned on those lines.

This is potentially a serious problem for us because we cannot count on a particular number of commas. This is also good reason *not* to choose a delimiter that appears elsewhere in the file. There are two fairly straight forward ways around this:

1. Use `csvtool` to recognize what commas are delimiters and which are part of the content of a field.
2. Use the setup of the file to our advantage.

While #1 would be nice we can't really pursue it here because of all the MacOS users in the room. Why is this a problem? As mentioned earlier, MacOS is based on a variant of BSD, a variant that is locked down such that it is not at all straightforward to install new GNU-Linux tools on the system. Since we're going for cross-platform as much as possible we're going to need to find another way.

> **Question:** Where else does the title appear and how might we use `cut` to get it?

> **Answer** In the Row ID and we can cut at the `/` if we are willing to lose the author and keep some junk for the time being.

Let's do this:

```
$ tail -n +2 EPLABB.csv | cut -d / -f 1
EPLABB20150316Fifty shades of Grey
EPLABB20150316The back of the turtle : a novel
EPLABB20150316Minecraft annual
```

So, we're moving along, we just really need to get rid of the coding information before the title. To do this we need to pass our current output to another tool, one that is able to read lines and make replacements/removals of text based on patterns. The tool that can do this is `sed`, the **S**tream **ED**itor.

For our purposes `sed` has a special format that looks like this:

```
sed 's/what_to_find/what_to_replace_with/'
```

What we want to do is remove the "EPL…" content completely so we can just leave the second item empty by removing it completely:

```
sed 's/what_to_find//'
```

This leaves us with the problem of what to find. Here `sed` is expecting a regular expression, which is a way of declaring patterns of text. Let's start with a simple one "EPLABB" and add `sed` to our pipeline:

```
$ tail -n +2 EPLABB.csv | cut -d / -f 1 | sed 's/EPLABB//'
```

Now all the "EPLABB" content is gone. If we could just add the double quotation marks and the numbers to this removal we'd have what we want. Let's deal with these in order. Try just adding a double quotation before the "E":

```
$ tail -n +2 EPLABB.csv | cut -d / -f 1 | sed 's/"EPLABB//'
```

The problem with the double quotation marks is that they do not appear on every line and currently our pattern of text must match exactly. We can fix this by introducing a quantifier, a character that tells the regular expression engine to match a varying number of a character. In this case we want to say "zero or more" and we do this with an `*` like so

```
$ tail -n +2 EPLABB.csv | cut -d / -f 1 | sed 's/"*EPLABB//'
```

Here the `*` says match the preceeding character (a `"`) zero or more times. Nice.

Next we need to get rid of the date information. Here we will make use of ranges and a different quantifier. The regular expression engine understands `[0-9]` to be a stand-in for the numerals 0, 1, 2, 3, up to 9. We can add it to our regular expression and see that it will indeed allow us to remove the first digit.

```
$ tail -n +2 EPLABB.csv | cut -d / -f 1 | sed 's/"*EPLABB[0-9]//'
```

There are always eight numerals so we could just repeat `[0-9]` eight times but there is a more concise way. Instead of writing it out eight times we use a quantifier to tell the regular expression engine exactly how many matches to make. This looks like `{n}` where `n` is the number of matches to make. As with the `*` we add this *after* the character set we wish to quantify:

```
$ tail -n +2 EPLABB.csv | cut -d / -f 1 | sed 's/"*EPLABB[0-9]{8}//'
EPLABB20150316Fifty shades of Grey
EPLABB20150316The back of the turtle : a novel
```

Wait! How'd we lose our progress? The problem here is that in the shell curly braces are special characters and we need to tell the shell to just treat them like any other character. We do this by placing a single `\` in front of each.

```
$ tail -n +2 EPLABB.csv | cut -d / -f 1 | sed 's/"*EPLABB[0-9]\{8\}//'
```

We now have all the titles. Nice.

Our next task is to count them. To do this we must first sort them. It should be little surprise that `sort` is a command that we can easily add to our pipeline:

```
$ tail -n +2 EPLABB.csv | cut -d / -f 1 | sed 's/"*EPLABB[0-9]\{8\}//' | sort
```

We'll now lean on a command called `uniq` (short for "unique") to remove all the duplicate entries. More than this though we'll pass it the `-c` flag so that it will *count* the number of entries.

```
$ tail -n +2 EPLABB.csv | cut -d / -f 1 | sed 's/"*EPLABB[0-9]\{8\}//' | sort | uniq
-c
```

One last sort, this one with the `-r` flag to reverse the results so the larger number is at the top, and we're done.

```
$ tail -n +2 EPLABB.csv | cut -d / -f 1 | sed 's/"*EPLABB[0-9]\{8\}//' | sort | uniq
-c | sort -r
```

Of course, if you'd like to specify how many lines to print then you can pipe the result to `head` with the `-n` flag on.

```
$ tail -n +2 EPLABB.csv | cut -d / -f 1 | sed 's/"*EPLABB[0-9]\{8\}//' | sort | uniq -c | sort -r | head -n 10
```

Automating

So, we've built a pipeline for EPLABB but what about the other branches? How can we see the top 10 most popular books in each of those branches? We could just press the up arrow, scroll to the left, and change the name but there ought to be a better way. Actually, there is and we're going to do it now. We're going to turn our pipeline into a script.

Creating 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.

Before we move on copy our current pipeline to your clipboard to save you typing it out from memory.

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

```
$ nano top10.sh
```

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".

Into this file we are going to paste the pipeline. Above this add a line that starts with a `#` and follow it with an explanation about what this pipeline does.

Exit and save your work 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 or just use `cat`. When you are satisfied that it is there you can run this script by issuing the `bash` command and following it with your file name

```
$ bash top10.sh
```

The point with making this script though was to have it work for *any* of the EPL branch files. Now that we hawe it working for the first one let's generalize it. Open it back up:

```
$ nano top10.sh
```

Once inside change the pipeline to the following:

```
tail -n +2 "$1" | cut -d / -f 1 | sed 's/"*EPLABB[0-9]\{8\}//' | sort | uniq -c | sor
t -r | head -n 10
```

When this script is run the `"$1"` tells the shell to grab the first item following the name of the script on the command line and substitute that value for the `"$1"`. Once you save and exit nana you can now repeat what we've already done via:

```
$ bash top10.sh EPLABB.csv
```

And substitue any of the other branch CSVs as well.

> **Optional:** Use nano to add `#!\bin\bash` to the top of the file and then use `chmod +x filename.sh` to make the file executable. Will needt to explain the `./` syntax to run

> this as a “full” script.
>
> **To Be Added:** Using `*` and `cat` to pull together all the EPL branch files into one big file for future processing. This will look like `cat *.csv > EPLALL.csv`.

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 on the Desktop 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.

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 I 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.

1. We use the term “GNU-Linux” her rather than the more common “Linux” to pay respect to its full history rather than just one event. When Richard Stallman started the GNU (GNU’s Not Unix) project he was able to assemble a wide range of important and useable tools, he just wasn’t able to get a kernel, roughly the program that handles all the inputs and outputs from the hardware associated with a computer. That kernel was eventually provided by Linus Torvalds. Both the tools and the kernel are essential to the project and the name “GNU-Linux” appropriately acknowledges this. ↩

2. The shell’s name is an acronym for Bourne-again shell, punning on the name of the Bourne shell that it replaces[11] and on the term “born again” that denotes spiritual rebirth in contemporary American Christianity Wikipedia. This is programmer humour. ↩

3. For a helpful explanation *all* the PS (prompt statement) control commands see: https://www.thegeekstuff.com/2008/09/bash-shell-take-control-of-ps1-ps2-ps3-ps4-and-prompt_command/ ↩