*nix Command Line Interface - Bootcamp Module

Preface:

1. Download the sample data from here. Move to the Desktop and unzip.
2. Rename the directory to whatever you'd like for easier finding during the workshop.
3. Open up the appropriate Bash Shell for your OS (we'll walk through some options).

0: Introduction to the Shell

This section is largely taken from the Library Carpentry curriculum: Shell Lessons for Librarians.

In this session we will introduce programming by looking at how data can be manipulated, counted, and mined using the **Unix shell,** a command line interface to your computer and the files to which it has access.

The shell is one of the most productive programming environments ever created. Its syntax may be cryptic, but people who have mastered it can experiment with different commands interactively, then use what they have learned to automate their work. Command Line Interfaces like shells and tools were the primary utilities for interaction with computer systems and programs until the introduction of the Graphical User Interfaces. For many tasks shell and the command line can still excel GUI programs, however: particularly for process very large datasets and files.

A Unix shell is a command-line interpreter that provides a user interface for the Linux operating system and for Unix-like systems (such as Mac OS).

For Windows users, popular shells such as Cygwin or Git Bash provide a Unix-like interface. This session will cover a small number of basic commands using Git Bash for Windows users, Terminal for Mac OS. These commands constitute building blocks upon which more complex commands can be constructed to fit your data or project.

What you can quickly learn is how to query lots of data for the information you want super fast. Using Bash or any other shell sometimes feels more like programming than like using a mouse. Commands are terse (often only a couple of characters long), their names are frequently cryptic, and their output is lines of text rather than something visual like a graph. On the other hand, with only a few keystrokes, the shell allows us to combine existing tools into powerful pipelines and handle large volumes of data automatically. This automation not only makes us more productive, but also improves the reproducibility of our workflows by allowing us to repeat them with few simple commands. Also, understanding the basics of the shell is very useful as a foundation before learning to program, since most programming languages necessitate working with the shell.

0.1: Structure of a Shell Command

Let's start by opening the shell. This likely results in seeing a black window with a cursor flashing next to a dollar sign. This is our command line, and the $ i s the command prompt to show the system is ready for our input. The prompt can look somewhat different from system to system, but it usually ends with a $.

The $ sign is used to indicate a command to be typed on the command prompt, but we never type the $ sign itself, just what follows after it.

```
$ command --option(s) argument(s)
# long options
$ ls --all ~/
# short options
$ ls -a ~/
```

0.2: Navigation in the Shell

We will begin with the basics of navigating the Unix shell.

0.2.1 Group Exploration

When working in the shell, you are always *somewhere* in the computer's file system, in some folder (directory). We will therefore start by finding out where we are by using the `pwd` command, which you can use whenever you are unsure about where you are. It stands for "print working directory" and the result of the command is printed to your standard output, which is the terminal.

Let's type `pwd` and hit enter to execute the command:

```
$ pwd
/Users/Christina
```

The output will be a path to your home directory. Let's check if we recognize it by listing the contents of the directory. To do that, we use the `ls` command:

```
$ ls
Applications             Downloads               Music                   Public
__MACOSX                 perl5
Applications (Parallels) Dropbox                 Pictures                Sites
marcedit                 test.json
Desktop                  Library                 Presentations           Tools                   mashcat
Documents                Movies                  Projects                VirtualBox VMs          nifi-0.7.0
```

We may want more information than just a list of files and directories. We can get this by specifying various **flags** (also known as options or switches) to go with our basic commands. These are additions to a command that provide the computer with a bit more guidance of what sort of output or manipulation you want.

If we type `ls -l` and hit enter, the computer returns a list of files that contains information similar to what we would find in our Finder (Mac) or Explorer (Windows): the size of the files in bytes, the date it was created or last modified, and the file name.

```
$ ls -l
total 24
drwxr-xr-x    10 Christina  staff    340 Jan 26 20:32 Applications
drwxr-xr-x@    5 Christina  staff    170 Jan 20 17:37 Applications (Parallels)
drwx------+    4 Christina  staff    136 Feb  3 10:46 Desktop
drwx------+   39 Christina  staff   1326 Feb 14 14:34 Documents
drwx------+   88 Christina  staff   2992 Feb 20 06:02 Downloads
drwx------@    8 Christina  staff    272 Feb 13 09:56 Dropbox
drwx------@   73 Christina  staff   2482 Jan  2 16:21 Library
drwx------+    5 Christina  staff    170 Dec 21  2015 Movies
drwx------+    6 Christina  staff    204 Mar  7  2016 Music
drwx------+ 1620 Christina  staff  55080 Aug 17  2016 Pictures
drwxr-xr-x     7 Christina  staff    238 May 29  2016 Presentations
drwxr-xr-x    44 Christina  staff   1496 Feb  3 10:58 Projects
drwxr-xr-x+    5 Christina  staff    170 Dec 17  2015 Public
drwxr-xr-x    14 Christina  staff    476 Feb 13 11:02 Sites
drwxr-xr-x@   45 Christina  staff   1530 Feb  7 17:06 Tools
drwx------     7 Christina  staff    238 Oct 12 11:21 VirtualBox VMs
drwxr-xr-x     4 Christina  staff    136 Mar  1  2016 __MACOSX
drwxr-xr-x    18 Christina  staff    612 Mar  7  2016 marcedit
drwxr-xr-x     3 Christina  staff    102 Jan 26  2016 mashcat
drwxr-xr-x@   20 Christina  staff    680 Nov  8 11:20 nifi-0.7.0
drwxr-xr-x     6 Christina  staff    204 Jun  6  2016 perl5
-rw-r--r--     2 Christina  staff   8232 Sep 30 12:05 test.json
```

In everyday usage we are more used to units of measurement like kilobytes, megabytes, and gigabytes. Luckily, there's another flag `-h` that when used with the -l option, use unit suffixes: Byte, Kilobyte, Megabyte, Gigabyte, Terabyte and Petabyte in order to reduce the number of digits to three or less using base 2 for sizes.

Now `ls -h` won't work on its own. When we want to combine two flags, we can just run them together. So, by typing `ls -lh` and hitting enter we receive an output in a human-readable format (note: the order here doesn't matter).

```
$ ls -lh
total 24
drwxr-xr-x    10 Christina  staff   340B Jan 26 20:32 Applications
drwxr-xr-x@    5 Christina  staff   170B Jan 20 17:37 Applications (Parallels)
drwx------+    4 Christina  staff   136B Feb  3 10:46 Desktop
drwx------+   39 Christina  staff   1.3K Feb 14 14:34 Documents
drwx------+   88 Christina  staff   2.9K Feb 20 06:02 Downloads
drwx------@    8 Christina  staff   272B Feb 13 09:56 Dropbox
drwx------@   73 Christina  staff   2.4K Jan  2 16:21 Library
drwx------+    5 Christina  staff   170B Dec 21  2015 Movies
drwx------+    6 Christina  staff   204B Mar  7  2016 Music
drwx------+ 1620 Christina  staff    54K Aug 17  2016 Pictures
drwxr-xr-x     7 Christina  staff   238B May 29  2016 Presentations
drwxr-xr-x    44 Christina  staff   1.5K Feb  3 10:58 Projects
drwxr-xr-x+    5 Christina  staff   170B Dec 17  2015 Public
drwxr-xr-x    14 Christina  staff   476B Feb 13 11:02 Sites
drwxr-xr-x@   45 Christina  staff   1.5K Feb  7 17:06 Tools
drwx------     7 Christina  staff   238B Oct 12 11:21 VirtualBox VMs
drwxr-xr-x     4 Christina  staff   136B Mar  1  2016 __MACOSX
drwxr-xr-x    18 Christina  staff   612B Mar  7  2016 marcedit
drwxr-xr-x     3 Christina  staff   102B Jan 26  2016 mashcat
drwxr-xr-x@   20 Christina  staff   680B Nov  8 11:20 nifi-0.7.0
drwxr-xr-x     6 Christina  staff   204B Jun  6  2016 perl5
-rw-r--r--     2 Christina  staff   8.0K Sep 30 12:05 test.json
```

We've now spent a great deal of time in our home directory. Let's go somewhere else. We can do that through the `cd` or Change Directory command:

(Note: On Windows and Mac, by default, the case of the file/directory doesn't matter. On Linux it does.)

```
$ cd Desktop
```

Notice that the command didn't output anything. This means that it was carried out successfully. Let's check by using `pwd`:

```
$ pwd
/Users/Christina/Desktop
```

If something had gone wrong, however, the command would have told you. Let's see by trying to move into a (hopefully) non-existing directory:

```
$ cd "Evil plan to destroy the world"
cd: The directory 'Evil plan to destroy the world' does not exist
```

Notice that we surrounded the name by quotation marks. The **arguments** given to any shell command are separated by spaces, so a way to let them know that we mean 'one single thing called "Evil plan to destroy the world"', not 'six different things', is to use (single or double) quotation marks.

We've now seen how we can do 'down' through our directory structure (as in into more nested directories). If we want to go back, we can type `cd ..`. This moves us 'up' one directory, putting us back where we started. If we ever get completely lost, the command `cd` without any arguments will bring us right back to the home directory, right where we started.

To switch back and forth between two directories use: `cd -`.

(Remember that if this cd command is special, there is no output in the terminal).

```
$ cd ..
```

0.2.2: Exercise - Try Exploring on Your Own

Take 5 minutes to move around the computer, get used to moving in and out of directories, see how different file types appear in the Unix shell. Be sure to use the pwd and cd commands, and the different flags for the ls command you learned so far.

If you run Windows, also try typing

```
explorer .
```

to open Explorer for the current directory (the single dot means "current directory"). If you're on Mac or Linux, try

```
open .
```

instead. Being able to navigate the file system is very important for using the Unix shell effectively. As we become more comfortable, we can get very quickly to the directory that we want.

1: Getting Help with Commands

Use the man command to invoke the manual page (documentation) for a shell command. For example,

```
$ man ls
LS(1)                     BSD General Commands Manual                    LS(1)
NAME
     ls -- list directory contents
SYNOPSIS
     ls [-ABCFGHLOPRSTUW@abcdefghiklmnopqrstuwx1] [file ...]
DESCRIPTION
     For each operand that names a file of a type other than directory, ls displays its name as well as any
requested, associated
     information.  For each operand that names a file of type directory, ls displays the names of files
contained within that direc-
     tory, as well as any requested, associated information.
     If no operands are given, the contents of the current directory are displayed.  If more than one operand
is given, non-directory
     operands are displayed first; directory and non-directory operands are sorted separately and in
lexicographical order.
     The following options are available:
     -@      Display extended attribute keys and sizes in long (-l) output.
     -1      (The numeric digit ``one''.)  Force output to be one entry per line.  This is the default when
output is not to a termi-
             nal.
     -A      List all entries except for . and ...  Always set for the super-user.
# more text
```

displays all the flags/options available to you - which saves you remembering them all! Try this for each command you've learned so far. **Use the spacebar to navigate the manual pages, and q to quit this man output.**

> Note: this command is for Mac and Linux users only. It does not work directly for Windows users. If you use windows, you can search for the Shell command on http://man.he.net/, and view the associated manual page. You can also use the help switch --help after a command to display the help documentation. e.g. ls --help

1.1.1: Exercise - Find out about advanced ls commands

Try to find out, using the man command:

1. How to list the files in a directory ordered by their filesize.
2. Try it out in different directories.
3. Can you combine it with the -l flag you learned before?
4. Afterwards, find out how you can order a list of files based on their last modification date.
5. Try ordering files in different directories.

To order files in a directory by their filesize, in combination with the -l flag:

ls -lS

Note that the S is case-sensitive!

To order files in a directory by their last modification date, in combination with the -l flag:

ls -lt

2: Working with files and folders

As well as navigating directories, we can interact with files on the command line: we can read them, open them, run them, and even edit them. In fact, there's really no limit to what we *can* do in the shell, but even experienced shell users still switch to graphical user interfaces (GUIs) for many tasks, such as editing formatted text documents (Word or OpenOffice), browsing the web, editing images, etc. But if we wanted to make the same crop on hundreds of images, say, the pages of a scanned book, then we could automate the cropping using shell commands.

We will try a few basic ways to interact with files. Let's first move into the `CUL-MWG-Workshop-master` directory on your desktop (if you don't have this directory, please ask for help).

```
$ cd
$ cd Desktop/CUL-MWG-Workshop-master
$ pwd
/Users/Christina/Desktop/CUL-MWG-Workshop-master
```

Here, we will create a new directory and move into it. Make the directory name 'firstdir':

```
$ mkdir firstdir
$ cd firstdir
```

Here we used the mkdir command (meaning 'make directories') to create a directory named 'firstdir'. Then we moved into that directory using the cd command.

But wait! There's a trick to make things a bit quicker. Let's go up one directory.

```
$ cd ..
```

Instead of typing cd firstdir, let's try to type cd f and then hit the Tab key. We notice that the shell completes the line to cd firstdir/.

2.0 Tab for Auto-complete

Hitting tab at any time within the shell will prompt it to attempt to auto-complete the line based on the files or sub-directories in the current directory. Where two or more files have the same characters, the auto-complete will only fill up to the first point of difference, after which we can add more characters, and try using tab again. We would encourage using this method throughout today to see how it behaves (as it saves loads of time and effort!).

2.1 Reading files

If you are in firstdir, use `cd ..` to get back to the `CUL-MWG-Workshop-master` directory.

Here there are copies of two public domain books downloaded from Project Gutenberg along with other files we will cover later.

```
$ ls -lh
total 65408
-rw-r--r--@ 1 Christina  staff   3.6M Feb 20 06:13 2014-01-31_JA-africa.tsv
-rw-r--r--@ 1 Christina  staff   7.4M Feb 20 06:13 2014-01-31_JA-america.tsv
-rw-r--r--@ 1 Christina  staff   1.4M Feb 20 06:13 2014-02-02_JA-britain.tsv
-rw-r--r--@ 1 Christina  staff    13M Feb 20 06:13 2017-ecommons-CU-etds.csv
-rw-r--r--@ 1 Christina  staff   5.0M Feb 20 06:13 2017-ecommons-CUL-community.csv
-rw-r--r--@ 1 Christina  staff   582K Feb 20 06:13 33504-0.txt
-rw-r--r--@ 1 Christina  staff   598K Feb 20 06:13 829-0.txt
-rw-r--r--@ 1 Christina  staff   144K Feb 20 06:13 CUlecturetapes_metadata_ingest-ready.csv
-rw-r--r--@ 1 Christina  staff    13B Feb 20 06:13 gallic.txt
```

The files `829-0.txt` and `33504-0.txt` holds the content of book #829 and #33504 on Project Gutenberg. But we've forgot *which* books, so we try the `ca` command to read the text of the first file:

```
$ cat 829-0.txt
```

The terminal window erupts and the whole book cascades by (it is printed to your terminal). Great, but we can't really make any sense of that amount of text.

2.2: Canceling Commands

To cancel this print of 829-0.txt, or indeed any ongoing processes in the Unix shell, hit **Ctrl+C**

2.3 Moving around a Text File

Often we just want a quick glimpse of the first or the last part of a file to get an idea about what the file is about. To let us do that, the Unix shell provides us with the commands `head` and `tail`.

```
$ head 829-0.txt
The Project Gutenberg eBook, Gulliver's Travels, by Jonathan Swift

This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever.  You may copy it, give it away or
re-use it under the terms of the Project Gutenberg License included
with this eBook or online at www.gutenberg.org
```

This provides a view of the first ten lines, whereas `tail 829-0.txt` provides a perspective on the last ten lines:

```
$ tail 829-0.txt

Most people start at our Web site which has the main PG search facility:
     http://www.gutenberg.org
This Web site includes information about Project Gutenberg-tm,
including how to make donations to the Project Gutenberg Literary
Archive Foundation, how to help produce our new eBooks, and how to
subscribe to our email newsletter to hear about new eBooks.
```

If ten lines is not enough (or too much), we would check `man head` to see if there exists an option to specify the number of lines to get (there is: `head -n 20` will print 20 lines).

Another way to navigate files is to view the contents one screen at a time. Type `less 829-0.txt` to see the first screen, `spacebar` to see the next screen and so on, then `q` to quit (return to the command prompt).

```
$ less 829-0.txt
```

Like many other shell commands, the commands cat, head, tail and less can take any number of arguments (they can work with any number of files). We will see how we can get the first lines of several files at once. To save some typing, we introduce a very useful trick first.

3. History & Re-using commands

3.1: Command Reuse

On a blank command prompt, hit the up arrow key and notice that the previous command you typed appears before your cursor. We can continue pressing the up arrow to cycle through your previous commands. The down arrow cycles back toward your most recent command. This is another important time-saving function and something we'll use a lot.

Hit the up arrow until you get to the `head 829-0.txt` command. Add a space and then `33504-0.txt` (Remember your friend Tab? Type 3 followed by Tab to get `33504-0.txt`), to produce the following command:

```
$ head 829-0.txt 33504-0.txt
==> 829-0.txt <==
The Project Gutenberg eBook, Gulliver's Travels, by Jonathan Swift

This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever.  You may copy it, give it away or
re-use it under the terms of the Project Gutenberg License included
with this eBook or online at www.gutenberg.org

==> 33504-0.txt <==
The Project Gutenberg EBook of Opticks, by Isaac Newton
This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever.  You may copy it, give it away or
re-use it under the terms of the Project Gutenberg License included
with this eBook or online at www.gutenberg.org

Title: Opticks
       or, a Treatise of the Reflections, Refractions, Inflections,
```

3.2 History Command

Use the history command to see a list of all the commands you've entered during the current session. Use the space bar to navigate through pages, and q to quit.

4: Wildcards

All good so far, but if we had *lots* of books, it would be tedious to enter all the filenames. Luckily the shell supports **wildcards**! The ? (matches exactly one character) and * (matches zero or more characters) are probably familiar from library search systems. We can use the * wildcard to write the above `head` c ommand in a more compact way:

```
$ head *.txt
```

4.1: More on wildcards

Wildcards are a feature of the shell and will therefore work with *any* command. The shell will expand wildcards to a list of files and/or directories before the command is executed, and the command will never see the wildcards. As an exception, if a wildcard expression does not match any file, Bash will pass the expression as a parameter to the command as it is. For example typing `ls *.pdf` results in an error message that there is no file called *.pdf.

4.2: The wildcards and regular expressions

The ? wildcard matches one character. The * wildcard matches zero or more characters. If you attended the lesson on regular expressions, do you remember how you would express that as regular expressions?

(Regular expressions are not a feature of the shell, but some commands support them, we'll get back to that.)

- The ? wildcard matches the regular expression . (a dot)
- The * wildcard matches the regular expression .*

5: Moving, copying and deleting files

5.1: Moving (Renaming) Files

We may also want to change the file name to something more descriptive. We can move it to a new name by using the `mv` or move command, giving it the old name as the first argument and the new name as the second argument:

```
$ mv 829-0.txt gulliver.txt
```

This is equivalent to the 'rename file' function.

Afterwards, when we perform a `ls` command, we will see that it is now `gulliver.txt`:

```
$ ls
2014-01-31_JA-africa.tsv 2014-02-02_JA-britain.tsv gulliver.txt 2014-01-31_JA-america.tsv 33504-0.txt 2014-
01_JA.tsv
```

5.2: Copying a file

Instead of *moving* a file, you might want to *copy* a file (make a duplicate), for instance to make a backup before modifying a file using some script you're not quite sure how works. Just like the `mv` command, the `cp` command takes two arguments: the old name and the new name.

How would you make a copy of the file `gulliver.txt` called `gulliver-backup.txt`? Try it!

```
$ cp gulliver.txt gulliver-backup.txt
```

5.3: Renaming a directory

Renaming a directory works in the same way as renaming a file. Try using the `mv` command to rename the `firstdir` directory to `backup`.

```
$ mv firstdir backup
```

5.4: Moving a file into a directory

If the last argument you give to the mv command is a directory, not a file, the file given in the first argument will be moved to that directory. Try using the mv command to move the file gulliver-backup.txt into the backupfolder.

```
mv gulliver-backup.txt backup
```

This would also work:

```
mv gulliver-backup.txt backup/gulliver-backup.txt
```

6: Using the echo command

6.1: Testing the echo Command

The echo command simply prints out a text you specify. Try it out: echo "Metadata scripting for the win!". Interesting, isn't it?

```
$ echo "Metadata scripting for the win!"
Metadata scripting for the win!
```

You can also specify a variable, for instance NAME= followed by your name. Then type echo "$NAME works at Cornell University Library". What happens?

```
$ NAME=christina
# no response if this is successful
$ echo "$NAME works at Cornell University Library"
christina works at Cornell University Library
```

You can combine both text and normal shell commands using echo, for example the pwd command you have learned earlier today. You do this by enclosing a shell command in $(and), for instance $(pwd). Now, try out the following:

```
$ echo "Finally, it is nice and sunny on" $(date).
Finally, it is nice and sunny on Mon Feb 15 07:04:43 EST 2017.
```

Note that the output of the datecommand is printed together with the text you specified. You can try the same with some of the other commands you have learned so far.

6.2 Why Care about the echo Command?

You may think there is not much value in such a basic command like echo. However, from the moment you start writing automated shell scripts, it becomes very useful. For instance, you often need to output text to the screen, such as the current status of a script.

Moreover, you just used a shell variable for the first time, which can be used to temporarily store information, that you can reuse later on. It will give many opportunities from the moment you start writing automated scripts.

7. Deleting Files & Directories

Finally, onto deleting. **We won't use it now, but if you do want to delete a file, for whatever reason, the command is `rm`, or remove.**

Using wildcards, we can even delete lots of files. And adding the `-r` flag we can delete folders with all their content.

Unlike deleting from within our graphical user interface, **there is *no* warning, *no* recycling bin from which you can get the files back and no other undo options**! For that reason, please be very careful with `rm` and extremely careful with `rm -r`.

1: Counting and mining data

Now that you know a little bit about navigating the shell, we will move onto learning how to count and mine data using a few of the standard shell commands. While these commands are unlikely to revolutionize your work by themselves, they're very versatile and will add to your foundation for working in the shell and for learning to code.

1.1: Counting and sorting

We will begin by counting the contents of files using the Unix shell. We can use the Unix shell to quickly generate counts from across files, something that is tricky to achieve using the graphical user interfaces of standard office suites.

Let's start by navigating to the directory that contains our data using the `cd` command:

```
$ cd CUL-MWG-Workshop-master
```

Remember, if at any time you are not sure where you are in your directory structure, use the pwd command to find out:

```
$ pwd
/Users/Christina/Desktop/CUL-MWG-Workshop-master
```

And let's just check what files are in the directory and how large they are with `ls -lh`.

In this episode we'll focus on the tsv and csv datasets, that contains eCommons, workflow, and article metadata.

1.2: CSV and TSV Files

CSV (Comma-separated values) is a common plain text format for storing tabular data, where each record occupies one line and the values are separated by commas. TSV (Tab-separated values) is just the same except that values are separated by tabs rather than commas. Confusingly, CSV is sometimes used to refer to both CSV, TSV and variations of them. The simplicity of the formats make them great for exchange and archival. They are not bound to a specific program (unlike Excel files, say, there is no CSV program, just lots and lots of programs that support the format, including Excel by the way.), and you wouldn't have any problems opening a 40 year old file today if you came across one.

1.2.1: wc, sort & >

`wc` is the "word count" command: it counts the number of lines, words, bytes and characters in files. Since we love the wildcard operator, let's run the command `wc *.tsv` to get counts for all the `.tsv` files in the current directory (it takes a little time to complete):

```
$ wc *.csv
   12012 1804072 13923125 2017-ecommons-CU-etds.csv
   13172  452320 5233591 2017-ecommons-CUL-community.csv
     238   15220  147314 CUlecturetapes_metadata_ingest-ready.csv
   25422 2271612 19304030 total
$ wc *.*sv
   13712  511261 3773660 2014-01-31_JA-africa.tsv
   27392 1049601 7731914 2014-01-31_JA-america.tsv
    5375  196999 1453418 2014-02-02_JA-britain.tsv
   12012 1804072 13923125 2017-ecommons-CU-etds.csv
   13172  452320 5233591 2017-ecommons-CUL-community.csv
     238   15220  147314 CUlecturetapes_metadata_ingest-ready.csv
   71901 4029473 32263022 total
```

The first three columns contains the number of lines, words and bytes (to show number characters you have to use a flag).

If we only have a handful of files to compare, it might be faster or more convenient to just check with Microsoft Excel, OpenRefine or your preferred text editor, but when we have tens, hundreds or thousands of documents, the Unix shell has a clear speed advantage. The real power of the shell comes from being able to combine commands and automate tasks, though. We will touch upon this slightly.

For now, we'll see how we can build a simple **pipeline** to find the shortest file in terms of number of lines. We start by adding the `-l` flag to get only the number of lines, not the number of words and bytes:

```
$ wc -l *sv
   13712 2014-01-31_JA-africa.tsv
   27392 2014-01-31_JA-america.tsv
    5375 2014-02-02_JA-britain.tsv
   12012 2017-ecommons-CU-etds.csv
   13172 2017-ecommons-CUL-community.csv
     238 CUlecturetapes_metadata_ingest-ready.csv
   71901 total
```

The `wc` command itself doesn't have a flag to sort the output, but as we'll see, we can combine three different shell commands to get what we want.

First, we have the `wc -l *sv` command. We will save the output from this command in a new file. To do that, we **redirect** the output from the command to a file using the 'greater than' sign (>), like so:

```
$ wc -l *sv > lengths.txt
```

There's no output now since the output went into the file `lengths.txt`, but we can check that the output indeed ended up in the file using `cat` or `less` (or Notepad or any text editor).

```
$ cat lengths.txt
   13712 2014-01-31_JA-africa.tsv
   27392 2014-01-31_JA-america.tsv
    5375 2014-02-02_JA-britain.tsv
   12012 2017-ecommons-CU-etds.csv
   13172 2017-ecommons-CUL-community.csv
     238 CUlecturetapes_metadata_ingest-ready.csv
   71901 total
```

Next, there is the `sort` command. We'll use the `-n` flag to specify that we want numerical sorting, not lexical sorting, we output the results into yet another file, and we use `cat` to check the results:

```
$ sort -n lengths.txt > sorted-lengths.txt
$ cat sorted-lengths.txt
     238 CUlecturetapes_metadata_ingest-ready.csv
    5375 2014-02-02_JA-britain.tsv
   12012 2017-ecommons-CU-etds.csv
   13172 2017-ecommons-CUL-community.csv
   13712 2014-01-31_JA-africa.tsv
   27392 2014-01-31_JA-america.tsv
   71901 total
```

1.2.2: Pipes

But we're really just interested in the end result, not the intermediate results now stored in `lengths.txt` and `sorted-lengths.txt`.

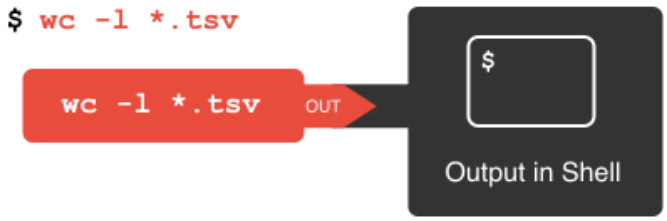
What if we could send the results from the first command (`wc -l *sv`) directly to the next command (`sort -n`) and then the output from that command to `head -n 1`? Luckily we can, using a concept called **pipes**. On the command line, you make a pipe with the vertical bar character |. Let's try with one pipe first:

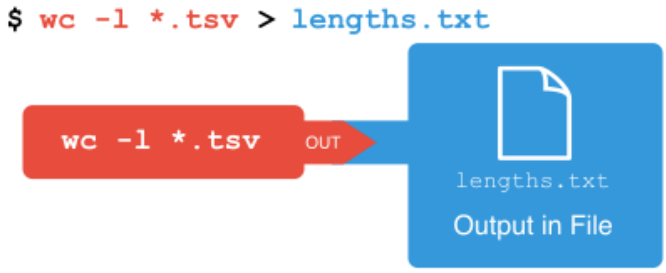
```
$ wc -l *sv | sort -n
     238 CUlecturetapes_metadata_ingest-ready.csv
    5375 2014-02-02_JA-britain.tsv
   12012 2017-ecommons-CU-etds.csv
   13172 2017-ecommons-CUL-community.csv
   13712 2014-01-31_JA-africa.tsv
   27392 2014-01-31_JA-america.tsv
   71901 total
```

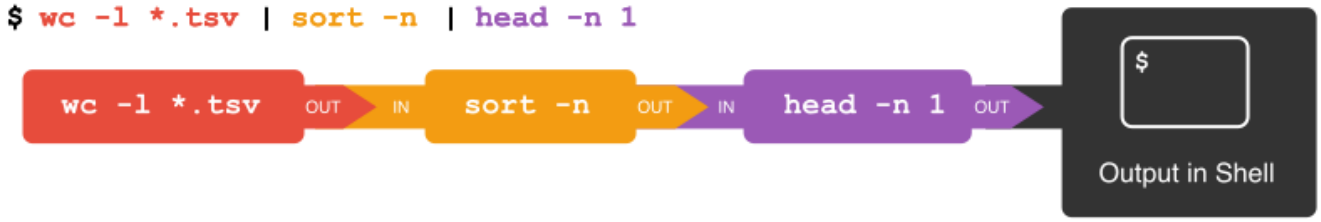
Notice that this is exactly the same output that ended up in our `sorted-lengths.txt` earlier. Let's add another pipe:

```
$ wc -l *sv | sort -n | head -n 1
238 CUlecturetapes_metadata_ingest-ready.csv
```

It can take some time to fully grasp pipes and use them efficiently, but it's a very powerful concept that you will find not only in the shell, but also in most programming languages.







Images taken from Library Carpentry: Counting and mining with the Shell. Also see http://jorol.de/2016-ELAG-Bootcamp/slides/#6 for more abbreviated and technical introduction to working with pipes.

1.3: Pipes and Filters

This simple idea is why Unix has been so successful. Instead of creating enormous programs that try to do many different things, Unix programmers focus on creating lots of simple tools that each do one job well, and that work well with each other.

This programming model is called “pipes and filters”. We’ve already seen **pipes**; a **filter** is a program like wc or sort that transforms a stream of input into a stream of output. Almost all of the standard Unix tools can work this way: unless told to do otherwise, they read from standard input, do something with what they’ve read, and write to standard output.

The key is that any program that reads lines of text from standard input and writes lines of text to standard output can be combined with every other program that behaves this way as well. You can and should write your programs this way so that you and other people can put those programs into pipes to multiply their power.

1.3.1: Adding another pipe

We have our wc -l *sv | sort -n | head -n 1 pipeline. What would happen if you piped this into cat? Try it!

The cat command just outputs whatever it gets as input, so you get exactly the same output from

```
$ wc -l *sv | sort -n | head -n 1
```

and

```
$ wc -l *sv | sort -n | head -n 1 | cat
```

1.4: Counting number of files, part I

Let's make a different pipeline. You want to find out how many files and directories there are in the current directory. Try to see if you can pipe the output from `ls` into wc to find the answer, or something close to the answer.

You get close with

```
$ ls -l | wc -l
```

but the count will be one too high, since the "total" line from `ls` is included in the count. We'll get back to a way to fix that later when we've learned about the `grep` command.

1.5: Writing to files

The `date` command outputs the current date and time. Can you write the current date and time to a new file called `logfile.txt`? Then check the contents of the file.

```
$ date > logfile.txt
$ cat logfile.txt
```

To check the contents, you could also use less or many other commands.

Beware that > will happily overwrite an existing file without warning you, so please be careful.

1.5.1: Appending to a file

While > writes to a file, >> appends something to a file. Try to append the current date and time to the file logfile.txt?

```
$ date >> logfile.txt
$ cat logfile.txt
```

1.6: Counting the number of words

1.6.1 Exercise

1. Check the manual for the wc command (either using man wc or wc --help) to see if you can find out what flag to use to print out the number of words (but not the number of lines and bytes).
2. Try it with the .csv files.
3. If you have time, you can also try to sort the results by piping it to sort. And/or explore the other flags of wc.

From man wc, you will see that there is a -w flag to print the number of words:

```
     -w      The number of words in each input file is written to the standard
             output.
```

So to print the word counts of the .csv files:

```
$ wc -w *.csv
```

```
 1804072 2017-ecommons-CU-etds.csv
  452320 2017-ecommons-CUL-community.csv
   15220 CUlecturetapes_metadata_ingest-ready.csv
 2271612 total
```

And to sort the lines numerically:

```
$ wc -w *.csv | sort -n
```

```
   15220 CUlecturetapes_metadata_ingest-ready.csv
  452320 2017-ecommons-CUL-community.csv
 1804072 2017-ecommons-CU-etds.csv
 2271612 total
```

1.7: Mining or searching

Searching for something in one or more files is something we'll often need to do, so let's introduce a command for doing that: `grep` (short for global regular expression print). As the name suggests, it supports regular expressions and is therefore only limited by your imagination, the shape of your data, and - when working with thousands or millions of files - the processing power at your disposal.

To begin using `grep`, first navigate to the `CUL-MWG-Workshop-master` directory if not already there. Then create a new directory "results":

```
$ mkdir results
```

Now let's try our first search:

```
$ grep metadata *.csv
```

Remember that the shell will expand `*.csv` to a list of all the .csv files in the directory. `grep` will then search these for instances of the string "1999" and print the matching rows.

1.8: Strings

A **string** is a sequence of characters, or "a piece of text".

1.8.1: Introduction to grep

Press the up arrow once in order to cycle back to your most recent action. Amend `grep metadata *.csv` to `grep -c metadata *.csv` and hit enter.

```
$ grep -c metadata *.csv
2017-ecommons-CU-etds.csv:3
2017-ecommons-CUL-community.csv:139
CUlecturetapes_metadata_ingest-ready.csv:0
```

The shell now prints the number of times the string metadata appeared in each file.

We will try another search:

```
$ grep -c 'Digital Archive' *.csv
2017-ecommons-CU-etds.csv:0
2017-ecommons-CUL-community.csv:3
CUlecturetapes_metadata_ingest-ready.csv:0
```

We got back the counts of the instances of the string `'application/pdf'` within the files. Now, amend the above command to the below and observe how the output of each is different:

```
$ grep -ci 'Digital Archive' *.csv
2017-ecommons-CU-etds.csv:0
2017-ecommons-CUL-community.csv:7
CUlecturetapes_metadata_ingest-ready.csv:0
```

This repeats the query, but prints a **case insensitive** count (including instances of both `digital archive` and `Digital Archive and` other variants).

As before, cycling back and adding `> results/`, followed by a filename (ideally in .txt format), will save the results to a data file. Go ahead and do this on your own.

1.8.2: CSV results from grep

So far we have counted strings in file and printed to the shell or to file those counts. But the real power of `grep` comes in that you can also use it to create subsets of tabulated data (or indeed any data) from one or multiple files.

```
$ grep -i metadata *.csv
```

This script looks in the defined files and prints any lines containing `revolution` (without regard to case) to the shell.

```
$ grep -i metadata *.csv > results/2017-02-15_metadata-ecommons.csv
```

This saves the subsetted data to file.

1.8.3: Whole words (-w) grep

Sometimes you need to capture the whole word only in that form (so revolution, but not revolutionary). The `-w` flag instructs `grep` to look for whole words only, giving us greater precision in our search.

```
$ grep -iw revolution *.csv > results/DATE_JAiw-revolution.csv
```

This script looks in both of the defined files and exports any lines containing the whole word revolution (without regard to case) to the specified .csv file.

We can show the difference between the files we created.

```
$ wc -l results/*.csv
     186 results/2017-02-15_metadata-ecommons.csv
     162 results/DATE_JAiw-revolution.csv
     348 total
```

Finally, we'll use the regular expression syntax covered earlier to search for similar words.

1.9: Basic and extended regular expressions

There is unfortunately both "basic" and "extended" regular expressions. This is a common cause of confusion. Unless you want to remember the details, **just always use extended regular expressions** (-E flag) when doing something more complex than searching for a plain string.

The regular expression 'fr[ae]nc[eh]' will match "france", "french", but also "frence" and "franch". It's generally a good idea to enclose the expression in single quotation marks, since that ensures the shell sends it directly to grep without any processing (such as trying to expand the wildcard operator *).

```
$ grep -iwE 'fr[ae]nc[eh]' *.csv
```

The shell will print out each matching line.

We include the `-o` flag to print only the matching part of the lines e.g. (handy for isolating/checking results):

```
$ grep -iwEo 'fr[ae]nc[eh]' *.csv
```

Pair up with your neighbor and work on these exercies:

1.10: Case sensitive search

Search for all case sensitive instances of a word you choose in all four derived csv files in this directory. Print your results to the shell.

```
$ grep DCAPS *.csv
```

1.11: Exercises

Run some of the now-possible searches on the sample metadata. How can you combine it with the Regex stuff as well as the output options above? Such as:

1. How do you run a search only on files that are from 2017 and a csv?
2. How do you count all case sensitive instances of a word you choose in select csv files?
3. How do you count all case insensitive instances of a word you choose in select csv files?
4. Search for all case insensitive instances of a select word in the csv files in this directory. Print your results to a new csv file.
5. Search for all case insensitive instances of a whole word in the csv files in this directory. Print your results to a new csv file.
6. Use regular expressions to find all ISSN numbers, unis, or other type identifiers with a regular structure in the csv files.

1.12: Finding unique values

If you pipe something to the uniq command, it will filter out duplicate lines and only return unique ones. Try piping the output from the command in the last exercise to uniq and then to wc -l to count the number of unique ISSN values.

```
$ grep -Eo '\d{4}-\d{4}' 2014-01_JA.tsv | uniq | wc -l
```

1.13: Counting number of files, part II

In the earlier counting exercise in this lesson, we tried counting the number of files and directories in the current directory.

- Recall that the command ls -l | wc -l took us quite far, but the result was one too high because it included the "total" line in the line count.
- With the knowledge of grep, can you figure out how to exclude the "total" line from the ls -l output?
- Hint: You want to exclude any line starting with the text "total". The hat character (^) is used in regular expressions to indicate the start of a line.

Solution

To find any lines starting with "total", we would use:

```
$ ls -l | grep -E '^total'
```

To *exclude those lines, we add the -v flag:

```
$ ls -l | grep -v -E '^total'
```

The grand finale is to pipe this into wc -l:

```
$ ls -l | grep -v -E '^total' | wc -l
```