Introduction to the
Linux/Unix command line

A workshop by Isabelle Giguère for the BiblioTECH Jumpstart Program 2023

This workshop was built using this: https://librarycarpentry.org/lc-shell/aio.html

Summary and Setup

This Library Carpentry lesson introduces librarians to the Unix Shell. At the conclusion of the lesson, you will: describe the basics of the Unix shell; explain why and how to use the command line; use shell commands to work with directories and files; use shell commands to find and manipulate data.

PREREQUISITES

To complete this lesson, you will need a Unix-like shell environment -see Setup. You will also need to download the file shell_lesson.zip from GitHub to your desktop and extract it there (once you have unzipped/extracted the file, you should end up with a folder called “shell_lesson”).
To participate in this Library Carpentry lesson, you will need a working Unix-like shell environment. Specifically, we will be using Bash (Bourne Again Shell) which is standard on Linux and macOS. macOS Catalina users will have zsh (Z shell) as their default version. Even if you are a Windows user, learning Bash will open a powerful set of tools on your personal machine, in addition to familiarizing you with the standard remote interface used on almost all servers and super computers.

TERMINAL SETUP

Bash is the default shell on most Linux distributions and macOS. Windows users will need to install Git Bash to provide a Unix-like environment.

Linux: The default shell is usually Bash, but if your machine is set up differently you can run it by opening a terminal > and typing bash. There is no need to install anything. Look for Terminal in your applications to start the Bash shell.

macOS: Bash is the default shell in all versions of macOS prior to Catalina, you do not need to install anything. Open Terminal from >/Applications/Utilities or spotlight search to start the Bash shell. zsh is the default in Catalina.

Windows: On Windows, CMD or PowerShell are normally available as the default shell environments. These use a syntax and set of applications unique to Windows systems and are incompatible with the more widely used Unix utilities. However, a Bash shell can be installed on Windows to provide a Unix-like environment. For this lesson we suggest using Git Bash, part of the >Git for Windows package:
Download the latest Git for Windows installer.
Double click the .exe to run the installer (for example, Git-2.41.0.3-64-bit.exe) using the default settings.
Once installed, open the shell by selecting Git Bash from the start menu (in the Git folder).

There are also some more advanced solutions available for running Bash commands on Windows. A Bash shell command-line tool is available for Windows 10, which you can use if you enable the Windows Subsystem for Linux. Additionally, you can run Bash commands on a remote computer or server that already has a Unix Shell, from your Windows machine. This can usually be done through a Secure Shell (SSH) client. One such client available for free for Windows computers is PuTTY. If you encounter issues, the Carpentries maintains a Configuration Problems and Solutions wiki page that may help.

DATA FILES

You need to download some files to follow this lesson:
Download shell_lesson.zip and move the file to your Desktop.
Unzip/extract the file (ask your instructor if you need help with this step). You should end up with a new folder called shell_lesson on your Desktop.
Open the terminal and type ls followed by the enter key.

```
$ ls
```

You should see a list of files and folders in your current directory. Then type:

```
$ pwd
```

This command will show you where you are in your file system, which should now be your home directory. In the lesson, you will find out more about the commands ls, pwd and how to work with the data in shell_lesson folder.

1. What is the shell?

What is the shell, and why should I use it?

If you've ever had to deal with large amounts of data or large numbers of digital files scattered across your computer or a remote server, you know that copying, moving, renaming, counting, searching through, or otherwise processing those files manually can be immensely time-consuming and error prone. Fortunately, there is an extraordinarily powerful and flexible tool designed for just that purpose.

The shell (sometimes referred to as the "Unix shell", for the operating system where it was first developed) is a program that allows you to interact with your computer using typed text commands. It is the primary interface used on Linux and Unix-based systems, such as macOS, and can be installed optionally on other operating systems such as Windows.

It is the definitive example of a "command line interface", where instructions are given to the computer by typing in commands, and the computer responds by performing a task or generating an output. This output is usually displayed on the screen, but can be directed to a file, or even to other commands, creating powerful chains of actions with very little effort.

Using a shell sometimes feels more like programming than like using a mouse. Commands are short (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 you to combine existing tools into powerful pipelines and to handle large volumes of data automatically. This automation not only makes you more productive, but also improves the reproducibility of your workflows by allowing you to save and then repeat them with a few simple commands. Understanding the basics of the shell provides a useful foundation for learning to program, since some of the concepts you will learn here—such as loops, values, and variables—will translate to programming.

The shell is one of the most productive programming environments ever created. Once mastered, you can use it to experiment with different commands interactively, then use what you have learned to automate your work.

In this workshop, we will introduce task automation by looking at how data can be manipulated, counted, and mined using the shell. We will cover a small number of basic commands, which will constitute building blocks upon which more complex commands can be constructed to fit your data or project. Even if you do not do your own programming or your work currently does not involve the command line, knowing some basics about the shell can be useful.

Where is my shell?

The shell is a program that is usually launched on your computer much in the way you would start any other program. However, there are numerous kinds of shells with different names, and they may or may not be already installed. The shell is central to Linux-based computers, and macOS machines ship with Terminal, a shell program. For Windows users, popular shells such as Cygwin or Git Bash provide a Unix-like interface but may need to be installed separately. In Windows 10, the Windows Subsystem for Linux also gives access to a Bash shell command-line tool.

2. Navigating the filesystem

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

Let's start by opening the shell. This likely results in seeing a black or white window with a cursor flashing next to a dollar sign. This is our command line, and the $ is the command prompt to show that the system is ready for our input. The appearance of the prompt will vary from system to system, depending on how the set up has been configured. Other common prompts include the % or # signs, but we will use $ in this lesson to represent the prompt generally.

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 screen.

Let's type pwd and press enter to execute the command (Note that 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.).

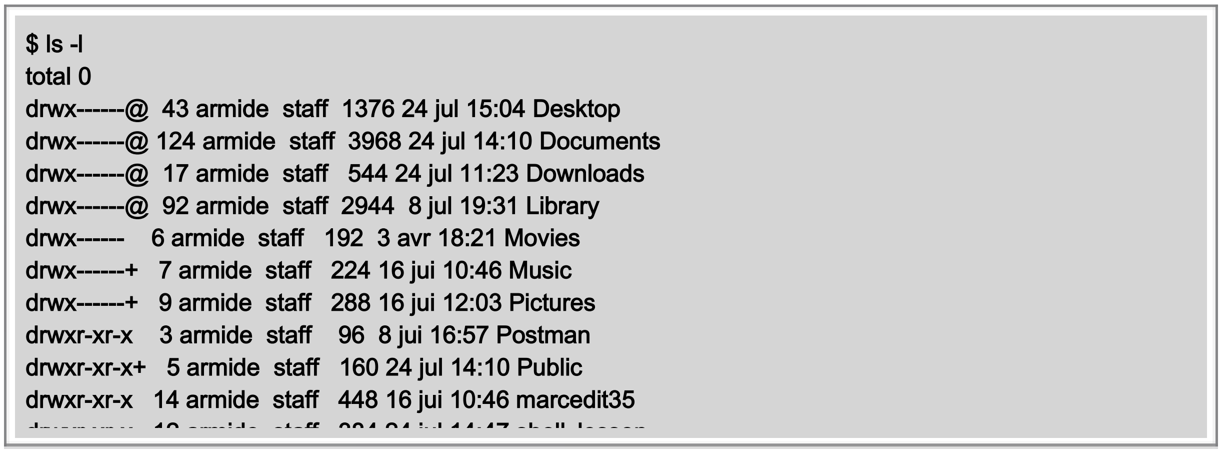
The output will be a path to your home directory. Let's check if we recognize it by looking at the contents of the directory. To do that, we use the ls command. This stands for "list" and the result is a print out of all the contents in the directory (here on a Mac):

```
$ pwd
/Users/armide
$ ls
Desktop Downloads       Movies  Pictures Public
Documents       Library Music            Postmanmarcedit35       shell_lesson
```

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, parameters, or, most frequently, arguments) to go with our basic commands. Arguments modify the workings of the command by telling the computer what sort of output or manipulation we want.

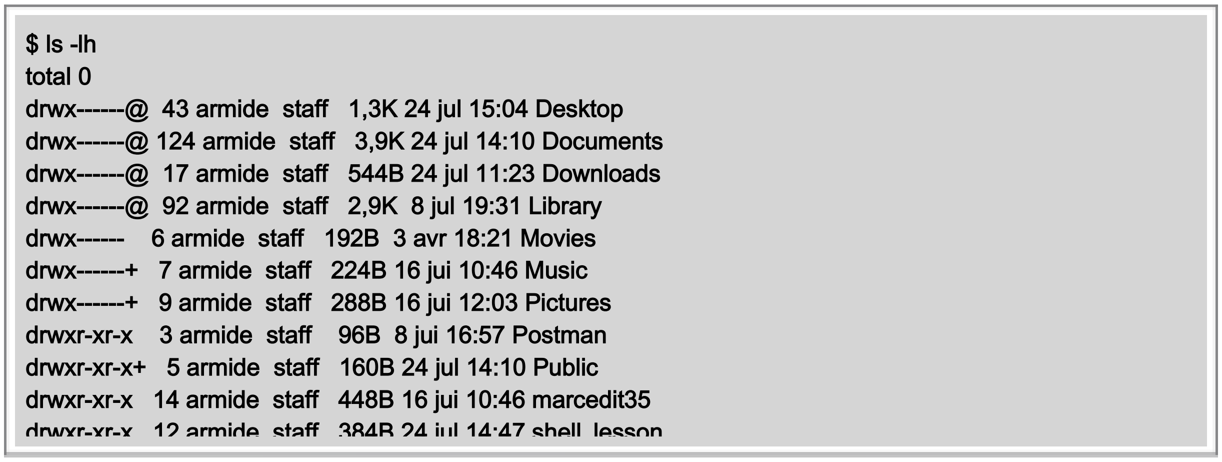
If we type `ls -l` and press enter, the computer returns a list of files that contains information like what we would find in our Finder (Mac) or Explorer (Windows): the size of the files in bytes, the date it was created or last modified, and the file name.

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.



```
$ ls -l
total 0
drwx------@  43 armide  staff  1376 24 jul 15:04 Desktop
drwx------@ 124 armide  staff  3968 24 jul 14:10 Documents
drwx------@  17 armide  staff   544 24 jul 11:23 Downloads
drwx------@  92 armide  staff  2944  8 jul 19:31 Library
drwx------    6 armide  staff   192  3 avr 18:21 Movies
drwx------+   7 armide  staff   224 16 jui 10:46 Music
drwx------+   9 armide  staff   288 16 jui 12:03 Pictures
drwxr-xr-x    3 armide  staff    96  8 jui 16:57 Postman
drwxr-xr-x+   5 armide  staff   160 24 jul 14:10 Public
drwxr-xr-x   14 armide  staff   448 16 jui 10:46 marcedit35
```

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 pressing enter we receive an output in a human-readable format (note: the order here doesn’t matter).



```
$ ls -lh
total 0
drwx------@  43 armide  staff   1,3K 24 jul 15:04 Desktop
drwx------@ 124 armide  staff   3,9K 24 jul 14:10 Documents
drwx------@  17 armide  staff   544B 24 jul 11:23 Downloads
drwx------@  92 armide  staff   2,9K  8 jul 19:31 Library
drwx------    6 armide  staff   192B  3 avr 18:21 Movies
drwx------+   7 armide  staff   224B 16 jui 10:46 Music
drwx------+   9 armide  staff   288B 16 jui 12:03 Pictures
drwxr-xr-x    3 armide  staff    96B  8 jui 16:57 Postman
drwxr-xr-x+   5 armide  staff   160B 24 jul 14:10 Public
drwxr-xr-x   14 armide  staff   448B 16 jui 10:46 marcedit35
drwxr-xr-x   12 armide  staff   384B 24 jul 14:47 shell lesson
```

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.)

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

If something had gone wrong, however, the command would have told you. Let’s test that by trying to move into a non-existent directory:

```
$ cd "my first directory"
```

```
$ cd "my first directory"
bash: cd: my first directory: No such file or directory
```

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 “my first directory”’, not ‘six different things’, is to use (single or double) quotation marks.

We've now seen how we can go ‘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, the place where we started.

PREVIOUS DIRECTORY

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

TRY EXPLORING

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.

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.

GETTING HELP

Use the man command to invoke the manual page (documentation) for a shell command. For example, man ls displays all the arguments 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. Use q at any time to quit.

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. In some systems the command name followed by --help will work, e.g., ls --help.

Also, the manual lists commands individually, e.g., although -h can only be used together with the -l option, you'll find it listed as -h in the manual, not as -lh.

FIND OUT ABOUT ADVANCED ls COMMANDS

Find out, using the manual page, how to list the files in a directory ordered by their filesize. Try it out in different directories. Can you combine it with the -l argument you learned before?

Afterwards, find out how you can order a list of files based on their last modification date. Try ordering files in different directories.

Knowing where you are in your directory structure is key to working with the shell.

3. Working with files and directories

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 that cropping work by using shell commands.

Before getting started, we will use ls to verify where we are. Using ls periodically to view your options is useful to orient oneself.
We will try a few basic ways to interact with files. Let's first move into the shell_lesson directory on your desktop.

Here, we will create a new directory and move into it:

```
$ cd
$ cd shell_lesson
$ pwd
/Users/armide/shell_lesson
$ mkdir 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 press the Tab key. We

```
$ cd firstdir
$ pwd
/Users/armide/shell_lesson/firstdir
```

notice that the shell completes the line to `cd firstdir/`.

TAB FOR AUTO-COMPLETE

Pressing 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!).

Reading files

If you are in firstdir, use `cd ..` to get back to the shell_lesson directory.

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

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 `cat` 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), leaving us with a new prompt and the last few lines of the book above this prompt.

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
```

If ten lines is not enough (or too much), we would check `man head`(or `head --help` when using Windows) 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).

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.

RE-USING COMMANDS

On a blank command prompt, press 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 labour-saving function and something we'll use a lot.

Press 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
```

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 command in a more compact way: `head *.txt`.

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.

Moving, copying and deleting 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 called gulliver.txt:

```
$ ls
2014-01-31_JA-africa.tsv   2014-01-31_JA-america.tsv           2014-01_JA.tsv              2014-02-02_JA-
britain.tsv           201403160_01_text.json    33504-0.txt                              diary.html
          firstdir                          gulliver.txt                          loans_2022_2023
```

- 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. 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!

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

- 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 backup folder.

- Use the history command to see a list of all the commands you've entered during the current.

You can also use Ctrl + r to do a reverse lookup. Press Ctrl + r, then start typing any part of the command you're looking for. The past command will autocomplete. Press enter to run the command again, or press the arrow keys to start editing the command. If multiple past commands contain the text you input, you can Ctrl + r repeatedly to cycle through them. If you can't find what you're looking for in the reverse lookup, use Ctrl + c to return to the prompt. If you want to save your history, maybe to extract some commands from which to build a script later on, you can do that with history > history.txt. This will output all history to a text file called history.txt that you can later edit. To recall a command from history, enter history. Note the command number, e.g., 2045. Recall the command by entering !2045. This will execute the command.

USING THE echo COMMAND

The echo command simply prints out a text you specify. Try it out: echo 'Library Carpentry is awesome!'. Interesting, isn't it?

You can also specify a variable. First type `NAME=` followed by your name, and press enter. Then type `echo "$NAME is a fantastic library carpentry student"` and press enter. What happens?

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

Why do you think the echo command is quite important in the shell environment?

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`.

KEYPOINTS

The shell can be used to copy, move, and combine multiple files

4. Automating the tedious with loops

Writing a Loop

Loops are key to productivity improvements through automation as they allow us to execute commands repetitively. Similar to wildcards and tab completion, using loops also reduces the amount of typing (and typing mistakes). Suppose we have several hundred document files named project_1825.txt, project_1863.txt, XML_project.txt and so on. We would like to change these files, but also save a version of the original files, naming the copies backup_project_1825.txt and so on.

We can use a loop to do that. Here's a simple example that creates a backup copy of four text files in turn.

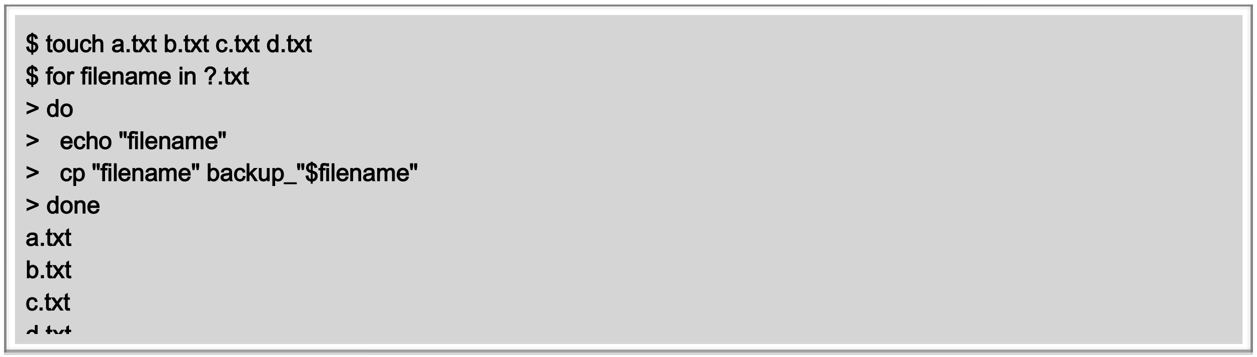
Let's first create those files:

```
$ touch a.txt b.txt c.txt d.txt
```

This will create four empty files with those names.

Now we will use a loop to create a backup version of those files. First let's look at the general form of a loop:

```
for thing in list_of_things
do
   operation_using $thing
done
```



```
$ touch a.txt b.txt c.txt d.txt
$ for filename in ?.txt
> do
>   echo "filename"
>   cp "filename" backup_"$filename"
> done
a.txt
b.txt
c.txt
d.txt
```

When the shell sees the keyword for, it knows to repeat a command (or group of commands) once for each thing in a list. For each iteration, the name of each thing is sequentially assigned to the loop variable and the commands inside the loop are executed before moving on to the next thing in the list. Inside the loop, we call for the variable's value by putting $ in front of it. The $ tells the shell interpreter to treat the variable as a variable name and substitute its value in its place, rather than treat it as text or an external command.

DOUBLE-QUOTING VARIABLE SUBSTITUTIONS

Because real-world filenames often contain white spaces, we wrap $filename in double quotes ("). If we didn't, the shell would treat the white space within a filename as a separator between two different filenames, which usually results in errors. Therefore, it's best and generally safer to use "$..." unless you are sure that no elements with white-space can ever enter your loop variable (such as in episode 5).

In this example, the list is four filenames: 'a.txt', 'b.txt', 'c.txt', and 'd.txt 'Each time the loop iterates, it will assign a file name to the variable filename and run the cp command. The first time through the loop, $filename is a.txt. The interpreter prints the filename to the screen and then runs the command cp on a.txt, (because we asked it to echo each filename as it works its way through the loop). For the second iteration, $filename becomes b.txt. This time, the shell prints the filename b.txt to the screen, then runs cp on b.txt. The loop performs the same operations for c.txt and then for d.txt and then, since the list only included these four items, the shell exits the for loop at that point.

FOLLOW THE PROMPT

The shell prompt changes from $ to > and back again as we were typing in our loop. The second prompt, >, is different to remind us that we haven't finished typing a complete command yet. A semicolon (;) can be used to separate two commands written on a single line.

SAME SYMBOLS, DIFFERENT MEANINGS

Here we see > being used as a shell prompt, but > can also be used to redirect output from a command (i.e., send it somewhere else, such as to a file, instead of displaying the output in the terminal) — we'll use redirection in episode 5. Similarly, $ is used as a shell prompt, but, as we saw earlier, it is also used to ask the shell to get the value of a variable.

If the shell prints > or $ then it expects you to type something, and the symbol is a prompt.

If you type > in the shell, it is an instruction from you to the shell to redirect output.

If you type $ in the shell, it is an instruction from you to the shell to get the value of a variable.

We have called the variable in this loop filename in order to make its purpose clearer to human readers. The shell itself doesn’t care what the variable is called.

- Complete the blanks in the for loop below to print the name, first line, and last

```
___ file in *.txt
__
        echo "_file"
        head -n 1 ______
        ____ __ _ ______
____
```

  line of each text file in the current directory.

This is our first look at loops. We will run another loop in the Counting and Mining with the Shell episode.

RUNNING THE LOOP FROM A BASH SCRIPT

Alternatively, rather than running the loop above on the command line, you can save it in a script file and run it from the command line without having to rewrite the loop again. This is what is called a Bash script which is a plain text file that contains a series of commands like the loop you created above. In the example script below, the first line of the file contains what is called a Shebang (#!) followed by the path to the interpreter (or program) that will run the rest of the lines in the file (/bin/bash). The second line demonstrates how comments are made in scripts. This provides you with more information about what the script does. The remaining lines contain the loop you created above. You can create this file in the same directory you’ve been using for the lesson and by using the text editor of your choice (e.g., vi) but when you save the file, make sure it has the extension .sh (e.g. my_first_bash_script.sh). When you've done this, you can run the Bash script by typing the command bash and the file name via the command line (e.g., bash my_first_bash_script.sh).

```
#!/bin/bash
# This script loops through .txt files, returns the file name, first line, and last line of the file
```

```
for file in *.txt
do
        echo $file
        head -n 1 $file
        tail -n 1 $file
done
```

- Copy my_first_bash_script.sh, you can work on it, later.

For more on Bash scripts, see Bash Scripting Tutorial - Ryans Tutorials.

KEYPOINTS

Looping is the foundation for working smarter with the command line
Loops help us to do the same (or similar) things to a bunch of items

Before continuing we'll delete the empty files, we've just created.

```
$ ls -l
total 618696
-rw-r--r--@ 1 armide  staff    3773660  7 oct  2021 2014-01-31_JA-africa.tsv
-rw-r--r--@ 1 armide  staff    7731914  7 oct  2021 2014-01-31_JA-america.tsv
-rw-r--r--@ 1 armide  staff  131122144 10 jui  2015 2014-01_JA.tsv
-rw-r--r--@ 1 armide  staff    1453418  7 oct  2021 2014-02-02_JA-britain.tsv
-rw-r--r--@ 1 armide  staff     392407  7 oct  2021 201403160_01_text.json
-rw-r--r--@ 1 armide  staff     596315  7 oct  2021 33504-0.txt
-rw-r--r--  1 armide  staff          0 24 jul 16:03 a.txt
-rw-r--r--  1 armide  staff          0 24 jul 16:03 b.txt
drwxr-xr-x  2 armide  staff         64 24 jul 14:47 backup
-rw-r--r--  1 armide  staff          0 24 jul 16:04 backup_a.txt
-rw-r--r--  1 armide  staff          0 24 jul 16:04 backup_b.txt
-rw-r--r--  1 armide  staff          0 24 jul 16:04 backup_c.txt
-rw-r--r--  1 armide  staff          0 24 jul 16:04 backup_d.txt
-rw-r--r--  1 armide  staff          0 24 jul 16:03 c.txt
-rw-r--r--  1 armide  staff          0 24 jul 16:03 d.txt
-rw-r--r--@ 1 armide  staff      18540  7 oct  2021 diary.html
-rw-r--r--@ 1 armide  staff     602147 19 jul 12:50 gulliver.txt
-rw-r--r--  1 armide  staff   26953920 24 jul 14:27 loans_2022_2023
$ ls ?.txt
a.txt	b.txt	c.txt	d.txt
```

5. Counting and mining with the shell

Counting and mining data

Now that you know how to navigate 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. The commands also replicate the sorts of uses library users might make of library data.

Counting and sorting

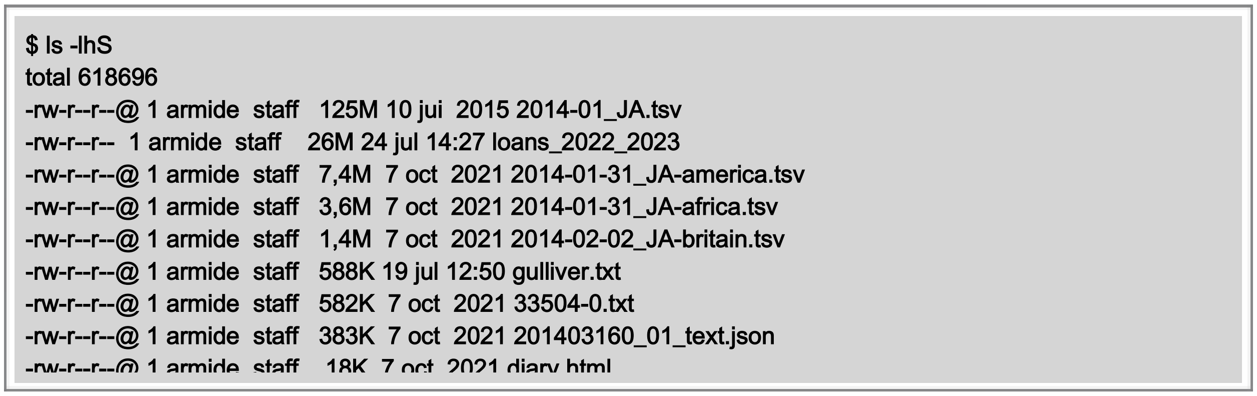
We will begin by counting the contents of files using the Unix shell. We can use the Unix shell to quickly generate counts from across files, something that is tricky to achieve using the graphical user interfaces of standard office suites.

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

```
$ cd shell_lesson
```

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/armide/Desktop/shell_lesson
```



```
$ ls -lhS
total 618696
-rw-r--r--@ 1 armide  staff   125M 10 jui  2015 2014-01_JA.tsv
-rw-r--r--  1 armide  staff    26M 24 jul 14:27 loans_2022_2023
-rw-r--r--@ 1 armide  staff   7,4M  7 oct  2021 2014-01-31_JA-america.tsv
-rw-r--r--@ 1 armide  staff   3,6M  7 oct  2021 2014-01-31_JA-africa.tsv
-rw-r--r--@ 1 armide  staff   1,4M  7 oct  2021 2014-02-02_JA-britain.tsv
-rw-r--r--@ 1 armide  staff   588K 19 jul 12:50 gulliver.txt
-rw-r--r--@ 1 armide  staff   582K  7 oct  2021 33504-0.txt
-rw-r--r--@ 1 armide  staff   383K  7 oct  2021 201403160_01_text.json
-rw-r--r--@ 1 armide  staff    18K  7 oct  2021 diary.html
```

And let's just check what files are in the directory and how large they are with:
In this section, we'll focus on the dataset 2014-01_JA.tsv, that contains journal article metadata, and the three .tsv files derived from the original dataset. Each of these three .tsv files includes all data where a keyword such as africa or america appears in the 'Title 'field of 2014-01_JA.tsv.

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.

First, let's have a look at the largest data file, using the tools we learned in Reading files:

$ cat 2014-01_JA.tsv

Like 829-0.txt before, the whole dataset cascades by and can't really make any sense of that amount of text. To cancel this on-going concatenation, or indeed any process in the Unix shell, press Ctrl+c.

In most data files a quick glimpse of the first few lines already tells us a lot about the structure of the dataset, for example the table/column headers:

```
$ head -n 3 2014-01_JA.tsv
File        Creator  Issue     Volume  Journal  ISSN      ID          Citation  Title       Place Labe        Language
            Publisher          Date
History_1a-rdf.tsv Doolittle, W. E.    1        59        KIVA -ARIZONA- 0023-1940       (Uk)RN001571862
            KIVA -ARIZONA- 59(1), 7-26. (1993)          A Method for Distinguishing between Prehistoric and Recent
Water and Soil Control Features     xxu      eng        ARIZONA ARCHAEOLOGICAL AND HISTORICAL
SOCIETY           1993
History_1a-rdf.tsv Nelson, M. C.       1        59        KIVA -ARIZONA- 0023-1940       (Uk)RN001571874
            KIVA -ARIZONA- 59(1), 27-48. (1993)         Classic Mimbres Land Use in the Eastern Mimbres Region,
Southwestern New Mexico xxu      eng        ARIZONA ARCHAEOLOGICAL AND HISTORICAL SOCIETY  1993
```

Next, let's learn about a basic data analysis tool: wc is the "word count" command: it counts the number of lines, words, and bytes. 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 *.tsv
   13712  511261 3773660 2014-01-31_JA-africa.tsv
   27392 1049601 7731914 2014-01-31_JA-america.tsv
  507732 17606310 131122144 2014-01_JA.tsv
    5375  196999 1453418 2014-02-02_JA-britain.tsv
  554211 19364171 144081136 total
```

The first three columns contain the number of lines, words and bytes.

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 favourite 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

```
$ wc -l *.tsv
   13712 2014-01-31_JA-africa.tsv
   27392 2014-01-31_JA-america.tsv
  507732 2014-01_JA.tsv
    5375 2014-02-02_JA-britain.tsv
  554211 total
```

lines, not the number of words and bytes:

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 *.tsv` 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 *.tsv > 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

```
$ cat lengths.txt
   13712 2014-01-31_JA-africa.tsv
   27392 2014-01-31_JA-america.tsv
  507732 2014-01_JA.tsv
    5375 2014-02-02_JA-britain.tsv
  554211 total
```

any text editor).

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,

```
$ sort -n lengths.txt > sorted-lengths.txt
bash-3.2$ cat sorted-lengths.txt
    5375 2014-02-02_JA-britain.tsv
   13712 2014-01-31_JA-africa.tsv
   27392 2014-01-31_JA-america.tsv
  507732 2014-01  JA.tsv
```

and we use cat to check the results:

Finally we have our old friend head, that we can use to get the first line of the sorted-lengths.txt:

```
$ head -n 1 sorted-lengths.txt
    5375 2014-02-02_JA-britain.tsv
```

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 *.tsv`) 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 |.

```
$ wc -l *.tsv | sort -n
    5375 2014-02-02_JA-britain.tsv
   13712 2014-01-31_JA-africa.tsv
   27392 2014-01-31_JA-america.tsv
  507732 2014-01_JA.tsv
  554211 total
```

Let’s try with one pipe first:
Notice that this is the same output that ended up in our sorted-lengths.txt earlier.
Let’s add another pipe:

```
$ wc -l *.tsv | sort -n | head -n 1
    5375 2014-02-02_JA-britain.tsv
```

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.

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.

ADDING ANOTHER PIPE

We have our wc -l *.tsv | sort -n | head -n 1 pipeline.

- What would happen if you piped this into cat? Try it!

COUNT THE NUMBER OF WORDS, SORT AND PRINT (FADED EXAMPLE)

To count the total lines in every tsv file, sort the results and then print the first line of the file we use the following:

```
wc -l *.tsv | sort -n | head -n 1
```

Now let's change the scenario. We want to know the 10 files that contain the most words. 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). Fill in the blanks below to count the words for each file, put them into order, and then make an output of the 10 files with the most words (Hint: The sort command sorts in ascending order by default).

```
wc __ *.tsv | sort __ | ____
```

COUNTING NUMBER OF FILES

- 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.

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.

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?

COUNTING THE NUMBER OF WORDS

- We learned about the -w flag above, so now try using it with the .tsv files.
- 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.

Mining or searching

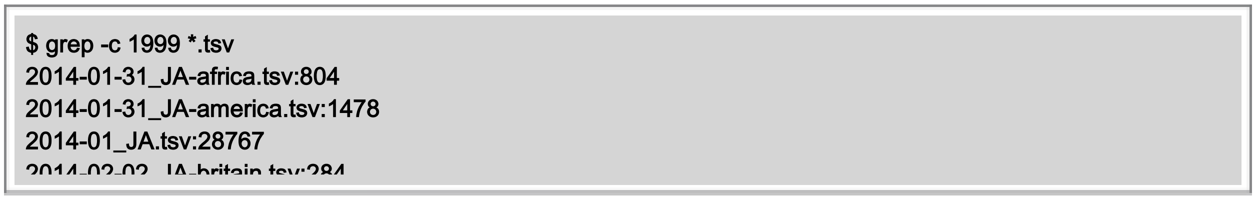
Searching for something in one or more files is something we'll often need to do, so let's introduce a command for doing that: grep (short for global regular expression print). As the name suggests, it supports regular expressions and is therefore only limited by your imagination, the shape of your data,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

To begin using grep, first navigate to the shell_lesson directory if not already there. Then create a new directory "result":

```
$ mkdir results
```

Now let's try our first search:

```
$ grep 1999 *.tsv
```



```
$ grep -c 1999 *.tsv
2014-01-31_JA-africa.tsv:804
2014-01-31_JA-america.tsv:1478
2014-01_JA.tsv:28767
2014-02-02_JA-britain.tsv:284
```

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

STRINGS

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

Press the up arrow once in order to cycle back to your most recent action. Amend grep 1999 *.tsv to grep -c 1999 *.tsv and press enter.

The shell now prints the number of times the string 1999 appeared in each file. If you look at the output from the previous command, this tends to refer to the date field for each journal article.

We will try another search:

```
$ grep -c revolution *.tsv
2014-01-31_JA-africa.tsv:20
2014-01-31_JA-america.tsv:34
2014-01_JA.tsv:867
2014-02-02_JA-britain.tsv:9
```

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

```
$ grep -ci revolution *.tsv
2014-01-31_JA-africa.tsv:118
2014-01-31_JA-america.tsv:1018
2014-01_JA.tsv:9327
2014-02-02_JA-britain.tsv:122
```

This repeats the query, but prints a case insensitive count (including instances of both revolution and Revolution and other variants). Note how the count has increased nearly 30 times for those journal article titles that contain the keyword ‘america’. As before, cycling back and adding > results/, followed by a filename (ideally in .txt format), will save the results to a data file.

So far we have counted strings in files 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 revolution *.tsv
```

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

We let the shell add today’s date to the filename:

```
$ grep -i revolution *.tsv > results/$(date "+%Y-%m-%d")_JAi-revolution.tsv
```

This saves the subsetted data to a new file.

ALTERNATIVE DATE COMMANDS

This way of writing dates is so common that on some platforms, you can get the same result by typing $(date -I) instead of $(date "+%Y-%m-%d").

However, if we look at this file, it contains every instance of the string ‘revolution ’ including as a single word and as part of other words such as ‘revolutionary’. This perhaps isn’t as useful as we thought… Thankfully, the -w flag instructs grep to look for whole words only, giving us greater precision in our search.

```
$ grep -iw revolution *.tsv > results/$(date "+%Y-%m-%d")_JAiw-revolution.tsv
```

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 .tsv file.

We can show the difference between the files we created.

```
$ wc -l results/*.tsv
   10585 results/2023-07-24_JAi-revolution.tsv
    7779 results/2023-07-24_JAiw-revolution.tsv
   18364 total
```

AUTOMATICALLY ADDING A DATE PREFIX

Notice how we didn't type today’s date ourselves but let the date command do that mindless task for us.

- Find out about the "+%Y-%m-%d" option and alternative options we could have used.

BASIC, EXTENDED, AND PERL-COMPATIBLE REGULAR EXPRESSIONS

There are, unfortunately, different ways of writing regular expressions. Across its various versions, grep supports “basic”, at least two types of “extended”, and “PERL-compatible” regular expressions. This is a common cause of confusion, since most tutorials, including ours, teach regular expressions compatible with the PERL programming language, but grep uses basic by default. Unless you want to remember the details, make your life easy by always using the most advanced regular expressions your version of grep supports (-E flag on macOS X, -P on most other platforms) or 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]' *.tsv
```

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]' *.tsv
```

CASE SENSITIVE SEARCH

- Search for all case sensitive instances of a whole word you choose in all four derived .tsv files in this directory. Print your results to the shell.
- Search for all case sensitive instances of a word you choose in the ‘America ’and ‘Africa ’.tsv files in this directory. Print your results to the shell.
- Count all case sensitive instances of a word you choose in the ‘America ’and ‘Africa ’.tsv files in this directory. Print your results to the shell.
- Count all case insensitive instances of that word in the ‘America ’and ‘Africa ’.tsv files in this directory. Print your results to the shell.
- Search for all case insensitive instances of that word in the ‘America ’and ‘Africa ’ .tsv files in this directory. Print your results to a file results/hero.tsv.
- Search for all case insensitive instances of that whole word in the ‘America ’and ‘Africa ’.tsv files in this directory. Print your results to a file results/hero-i.tsv.
- Use regular expressions to find all ISSN numbers (four digits followed by hyphen followed by four digits) in 2014-01_JA.tsv and print the results to a file results/issns.tsv. Note that you might have to use the -E flag (or -P with some versions of grep, e.g. with Git Bash on Windows).

FINDING UNIQUE VALUES

If you pipe something to the uniq command, it will filter out adjacent duplicate lines. In order for the uniq command to only return unique values though, it needs to be used with the sort command.

- Try piping the output from the command in the last exercise to sort and then piping these results to uniq and then wc -l to count the number of unique ISSN values.

Using a Loop to Count Words

We will now use a loop to automate the counting of certain words within a document. For this, we will be using the Little Women e-book from Project Gutenberg. The file is inside the shell_lesson folder and named pg514.txt. Let's rename the file to littlewomen.txt.

```
$ mv pg514.txt littlewomen.txt
```

This renames the file to something easier to type.

Now let's create our loop. In the loop, we will ask the computer to go through the text, looking for each girl's name, and count the number of times it appears. The results will print to the screen.

```
$ for name in "Jo" "Beth" "Meg" "Amy"
> do
>   echo "$name"
>   grep -wo "$name" littlewomen.txt |wc -l
> done
Jo
   1355
Beth
    459
Meg
    683
Amy
```

What is happening in the loop?

`echo "$name"` is printing the current value of $name
`grep "$name" littlewomen.txt` finds each line that contains the value stored in $name. The -w flag finds only the whole word that is the value stored in $name and the -o

flag pulls this value out from the line it is in to give you the actual words to count as lines in themselves.
The output from the grep command is redirected with the pipe, | (without the pipe and the rest of the line, the output from grep would print directly to the screen)
wc -l counts the number of lines (because we used the -l flag) sent from grep.
Because grep only returned lines that contained the value stored in $name, wc -l corresponds to the number of occurrences of each girl's name.

WHY ARE THE VARIABLES DOUBLE-QUOTED HERE?

In the "Automating…" section, we learned to use "$..." as a safeguard against white space being misinterpreted. Why could we omit the "-quotes in the above example?

What happens if you add "Louisa May Alcott" to the first line of the loop and remove the " from $name in the loop's code?

SELECTING COLUMNS FROM OUR ARTICLE DATASET

When you receive data it will often contain more columns or variables than you need for your work. If you want to select only the columns you need for your analysis, you can use the cut command to do so. cut is a tool for extracting sections from a file. For instance, say we want to retain only the Creator, Volume, Journal, and Citation columns from our article data. With cut we'd:

```
$ cut -f 2,4,5,8 2014-01_JA.tsv | head -n 5
Creator  Volume  Journal  Citation
Doolittle, W. E.    59        KIVA -ARIZONA- KIVA -ARIZONA- 59(1), 7-26. (1993)
Nelson, M. C.       59        KIVA -ARIZONA- KIVA -ARIZONA- 59(1), 27-48. (1993)
Deegan, A. C.       59        KIVA -ARIZONA- KIVA -ARIZONA- 59(1), 49-64. (1993)
Stone, T.           59        KIVA -ARIZONA- KIVA -ARIZONA- 59(1), 65-82. (1993)
```

Above we used cut and the -f flag to indicate which columns we want to retain. cut works on tab delimited files by default. We can use the flag -d to change this to a comma, or semicolon or another delimiter. If you are unsure of your column position and the file has headers on the first line, we can use head -n 1 <filename> to print those out.

- Select the columns Issue, Volume, Language, Publisher and direct the output into a new file. You can name it something like 2014-01_JA_ivlp.tsv.

With cut you can't extract the columns out of order, but you can use the awk command.

```
$ awk -F '\t' '{ print $5 "\t" $2 "\t" $4 "\t" $8 }' 2014-01_JA.tsv| head -n 5
```

or

```
$ awk 'BEGIN {FS = "\t" ; OFS = "\t"} {print $5 $2 $8 $4}' 2014-01_JA.tsv |head -n 5
```

In the second line, you define the output field separator (OFS) once. You can learn more about awk here:

https://www.gnu.org/software/gawk/manual/html_node/index.html

---

KEYPOINTS

The shell can be used to count elements of documents
The shell can be used to search for patterns within files
Commands can be used to count and mine any number of files
Commands and flags can be combined to build complex queries specific to your work

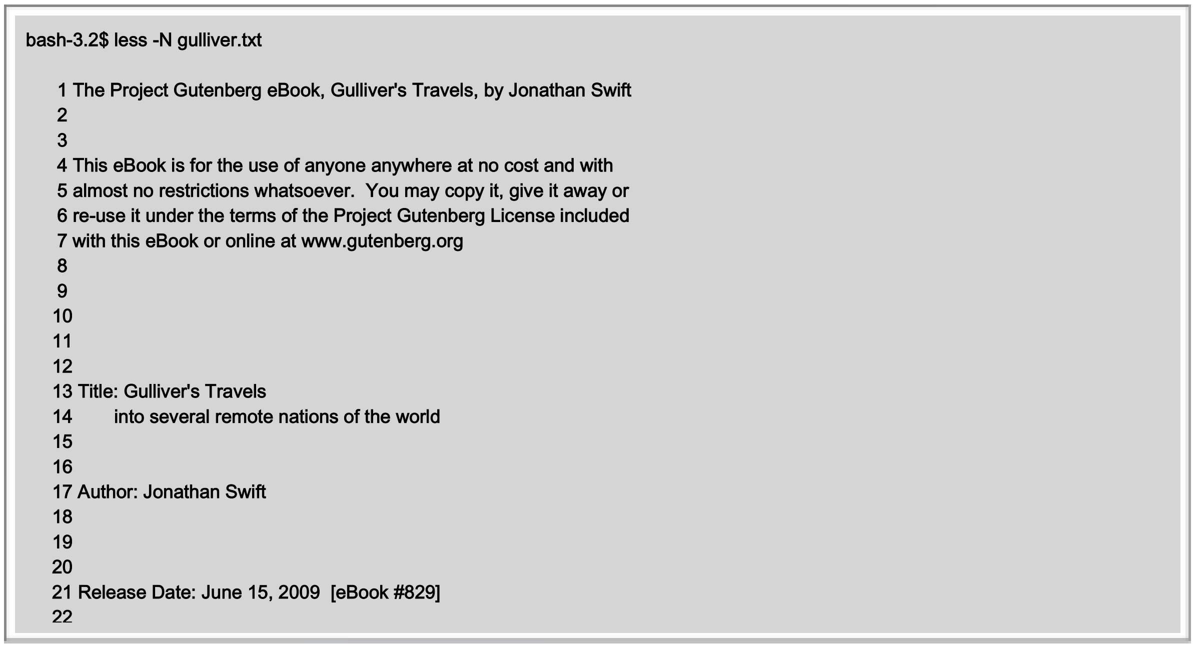
6. Working with free text

So far we have looked at how to use the Unix shell to manipulate, count, and mine tabulated data. Some library data, especially digitized documents, is much messier than tabular metadata. Nonetheless, many of the same techniques can be applied to non-tabulated data such as free text. We need to think carefully about what it is we are counting and how we can get the best out of the Unix shell.

Thankfully there are plenty of folks out there doing this sort of work and we can borrow what they do as an introduction to working with these more complex files. So for this final exercise we're going to leap forward a little in terms of difficulty to a scenario where we won't learn about everything that is happening in detail or discuss at length each command. We're going to prepare and pull apart texts to demonstrate some of the potential applications of the Unix shell. And where commands we've learnt about are used, I've left some of the figuring out to do to you - so please refer to your notes if you get stuck!

Option 1: Hand transcribed text

Grabbing a text, cleaning it up
We're going to work with the gulliver.txt file, which we made in section 3, 'Working with files and directories'. You should (still) be working in the shell_lesson directory. Let's look at the file.



```
bash-3.2$ less -N gulliver.txt

      1 The Project Gutenberg eBook, Gulliver's Travels, by Jonathan Swift
      2
      3
      4 This eBook is for the use of anyone anywhere at no cost and with
      5 almost no restrictions whatsoever.  You may copy it, give it away or
      6 re-use it under the terms of the Project Gutenberg License included
      7 with this eBook or online at www.gutenberg.org
      8
      9
     10
     11
     12
     13 Title: Gulliver's Travels
     14        into several remote nations of the world
     15
     16
     17 Author: Jonathan Swift
     18
     19
     20
     21 Release Date: June 15, 2009  [eBook #829]
     22
```

We're going to start by using the sed command. The command allows you to edit files directly.

```
$ sed '9352,9714d' gulliver.txt > gulliver-nofoot.txt
```

The command sed in combination with the d value will look at gulliver.txt and delete all values between the rows specified. The > action then prompts the script to this edited text to the new file specified.

```
$ sed '1,37d' gulliver-nofoot.txt > gulliver-noheadfoot.txt
```

This does the same as before, but for the header.

You now have a cleaner text. The next step is to prepare it even further for rigorous analysis.

We now use the tr command, used for translating or deleting characters. Type and run:

```
$ tr -d '[:punct:]\r' < gulliver-noheadfoot.txt > gulliver-noheadfootpunct.txt
```

This uses the translate command and a special syntax to remove all punctuation ([:punct:]) and carriage returns (\r). It also requires the use of both the output redirect > we have seen and the input redirect < we haven't seen.

Finally regularize the text by removing all the uppercase lettering.

```
$ tr '[:upper:]' '[:lower:]' < gulliver-noheadfootpunct.txt > gulliver-clean.txt
```

Open the gulliver-clean.txt in a text editor. Note how the text has been transformed ready for analysis.

PULLING A TEXT APART, COUNTING WORD FREQUENCIES

We are now ready to pull the text apart.

```
$ tr ' ' '\n' < gulliver-clean.txt | sort | uniq -c | sort -nr > gulliver-final.txt
```

Here we've made extended use of the pipes we saw in Counting and mining with the shell. The first part of this script uses the translate command again, this time to translate every blank space into \n which renders as a new line. Every word in the file will at this stage have its own line.

The second part uses the sort command to rearrange the text from its original order into an alphabetical configuration.

The third part uses uniq, another new command, in combination with the -c flag to remove duplicate lines and to produce a word count of those duplicates.

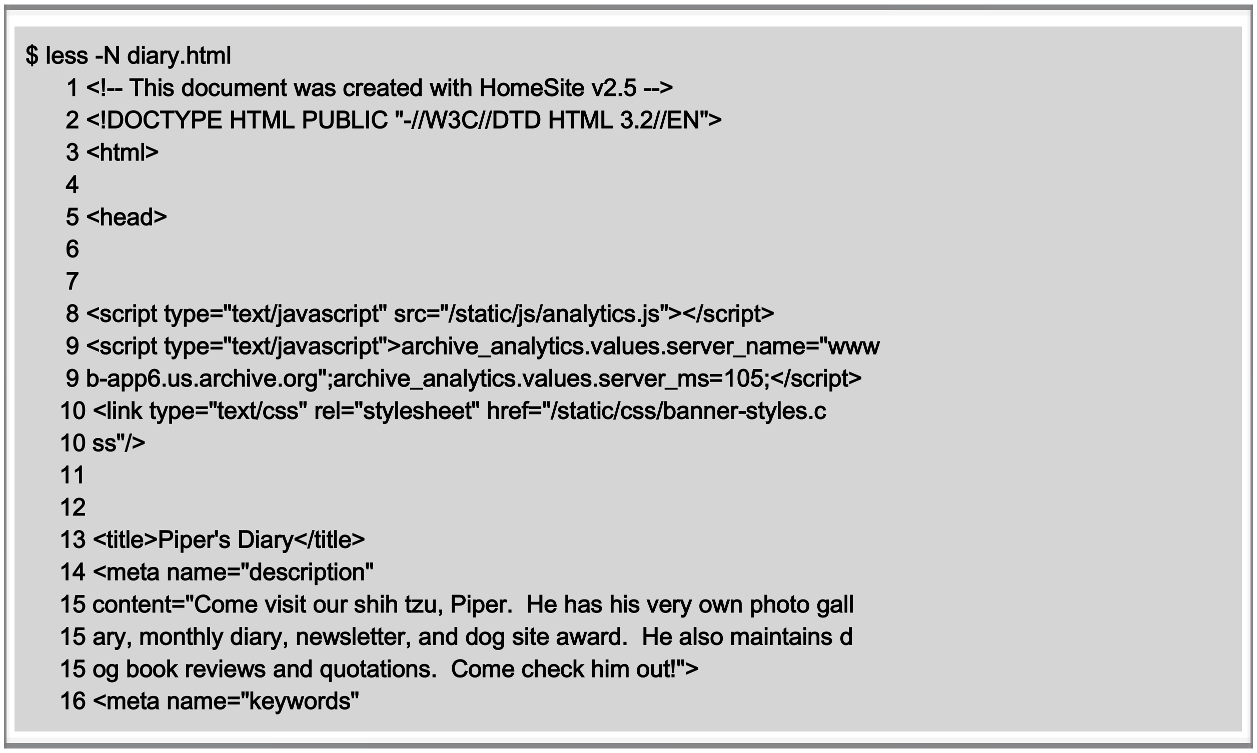
The fourth and final part sorts the text again by the counts of duplicates generated in step three.

We have now taken the text apart and produced a count for each word in it. This is data we can prod and poke and visualize, that can form the basis of our investigations, and can compare with other texts processed in the same way. And if we need to run a different set of transformation for a different analysis, we can return to gulliver-clean.txt to start that work.

And all this using a few commands on an otherwise unassuming but very powerful command line.

Option 2: Grabbing text of a webpage and cleaning it up

We’re going to work with diary.html.



```
$ less -N diary.html
      1 <!-- This document was created with HomeSite v2.5 -->
      2 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
      3 <html>
      4
      5 <head>
      6
      7
      8 <script type="text/javascript" src="/static/js/analytics.js"></script>
      9 <script type="text/javascript">archive_analytics.values.server_name="www
      9 b-app6.us.archive.org";archive_analytics.values.server_ms=105;</script>
     10 <link type="text/css" rel="stylesheet" href="/static/css/banner-styles.c
     10 ss"/>
     11
     12
     13 <title>Piper's Diary</title>
     14 <meta name="description"
     15 content="Come visit our shih tzu, Piper.  He has his very own photo gall
     15 ary, monthly diary, newsletter, and dog site award.  He also maintains d
     15 og book reviews and quotations.  Come check him out!">
     16 <meta name="keywords"
```

Let’s look at the file.

We’re going to start by using the sed command. The command allows you to edit files directly.

```
$ sed '265,330d' diary.html > diary-nofoot.txt
```

The command sed in combination with the d value will look at diary.html and delete all values between the rows specified. The > action then prompts the script to this edited text to the new file specified.

```
$ sed '1,221d' diary-nofoot.txt > diary-noheadfoot.txt
```
This does the same as before, but for the header.

You now have a cleaner text. The next step is to prepare it even further for rigorous analysis.

First we wil remove all the html tags. Type and run:

```
$ sed -e 's/<[^>]*>//g' diary-noheadfoot.txt > diary-notags.txt
```
Here we are using a regular expression (see the Library Carpentry regular expression lesson to find all valid html tags (anything within angle brackets) and delete them). This is a complex regular expression, so don't worry too much about how it works! The script also requires the use of both the output redirect > we have seen and the input redirect < we haven't seen.

We're going to start by using the tr command, used for translating or deleting characters. Type and run:

```
$ tr -d '[:punct:]\r' < diary-notags.txt > diary-notagspunct.txt
```
This uses the translate command and a special syntax to remove all punctuation ([:punct:]) and carriage returns (\r).

Finally regularize the text by removing all the uppercase lettering.

```
$ tr '[:upper:]' '[:lower:]' < diary-notagspunct.txt > diary-clean.txt
```
Open the diary-clean.txt in a text editor. Note how the text has been transformed ready for analysis.

We are now ready to pull the text apart.

```
$ tr ' ' '\n' < diary-clean.txt | sort | uniq -c | sort -nr > diary-final.txt
```
Here we've made extended use of the pipes we saw in Counting and mining with the shell. The first part of this script uses the translate command again, this time to translate every blank space into \n which renders as a new line. Every word in the file will at this stage have its own line.

The second part uses the sort command to rearrange the text from its original order into an alphabetical configuration.

The third part uses uniq, another new command, in combination with the -c flag to remove duplicate lines and to produce a word count of those duplicates.

The fourth and final part sorts the text again by the counts of duplicates generated in step three.

We have now taken the text apart and produced a count for each word in it. This is data we can prod and poke and visualize, that can form the basis of our investigations, and can compare with other texts processed in the same way. And if we need to run a different set of transformation for a different analysis, we can return to diary-final.txt to start that work.

And, again, all this using a few commands on an otherwise unassuming but very powerful command line.

Additional information

The best AWK one-liners
https://www.shebanglinux.com/best-awk-one-liners/

Learning AWK
https://www.tutorialspoint.com/awk/awk_workflow.htm

Regular expressions info
https://www.regular-expressions.info/posixbrackets.html

Information about SED and AWK (not recent but a lot of details)
https://tldp.org/LDP/abs/html/sedawk.html

Regular expressions info
https://www.regular-expressions.info/posixbrackets.html

A fun way of learning regular expressions
https://regex101.com/