101 Bash Introduction

- Offered by the Scientific Computing Facility @ CBG.
- Contact: `scicomp@mpi-cbg.de`

- **Learning objectives**
  - Navigate around the Unix file system
  - Differentiate between full and relative paths
  - List files in a directory
  - Copy, remove and move files
  - Implement tab completion when writing paths
  - Use of the asterisk `*` wildcard to select multiple items
  - List a few shortcuts
  - View the contents of a file
  - Create a new file using the Vim text editor
  - Execute basic shortcuts in the Vim text editor
  - Search for characters or patterns in a text file using the `grep` command
  - Write to and append a file using output redirection
  - Use the pipe (|) character to chain together commands
- **For whom and which prerequisites**
  - CBG/CSBD affiliated with computer
  - Prior Shell knowledge unnecessary
- **Course format**
  - Self-study
    - Students read material and do exercises themselves
    - In case of questions or wish to discuss particular points, write an Email to schedule a Zoom chat with us: `scicomp@mpi-cbg.de`
  - Online or class-room course
    - Offered on demand
    - Minimum number of participants: 5
    - Write us an Email to evince your interest: `scicomp@mpi-cbg.de`
- **Installation before class on your computer**
  - Mac/Linux users: No installation required
  - Windows users: Install Windows Subsystem for Linux Details
    1. open an admin CMD
    2. wsl --install -d ubuntu
    3. wcl --set-version-default 2
- **Origin of material**
  - Taken and partly adapted from the teaching material of the Harvard Chan Bioinformatics Core Training.

Starting with the shell

Let's look at what is inside the data folder and explore further. First instead of clicking on the folder name to open it and look at its contents, we have to change the folder we are in. When working with any programming tools, **folders are**

called directories. We will be using folder and directory interchangeably moving forward.

To look inside the `unix_lesson` directory, we need to change which directory we are *in*. To do this we can use the `cd` command, which stands for "change directory".

```
$ cd unix_lesson
```

Did you notice a change in your command prompt? The "~" symbol from before should have been replaced by the string `unix_lesson`. This means that our `cd` command ran successfully and we are now *in* the new directory. Let's see what is in here by listing the contents:

```
$ ls
```

You should see:

```
genomics_data  other  raw_fastq  README.txt  reference_data
```

Arguments

There are five items listed when you run `ls`, but what types of files are they, or are they directories or files?

We can modify the default behavior of `ls` with one or more **"arguments"** to get more information.

```
$ ls -F

genomics_data/  other/  raw_fastq/  README.txt  reference_data/
```

Anything with a "/" after its name is a directory. Things with an asterisk "*" after them are programs. If there are no "decorations" after the name, it's a normal text file.

You can also use the argument `-l` to show the directory contents in a long-listing format that provides a lot more information:

```
$ ls -l
```

```
total 124
drwxrwsr-x 2 mp298 mp298  78 Sep 30 10:47 genomics_data
drwxrwsr-x 6 mp298 mp298 107 Sep 30 10:47 other
drwxrwsr-x 2 mp298 mp298 228 Sep 30 10:47 raw_fastq
-rw-rw-r-- 1 mp298 mp298 377 Sep 30 10:47 README.txt
drwxrwsr-x 2 mp298 mp298 238 Sep 30 10:47 reference_data
```

Each line of output represents a file or a directory. The directory lines start with `d`. If you want to combine the 2 arguments `-l` and `-F`, you can do so by saying the following:

```
ls -lF
```

Do you see the modification in the output?

▸ *Explanation*

> **Tip** - **All commands are essentially programs** that are able to perform specific, commonly-used tasks.

Most commands will take additional arguments that control their behavior, some of them will take a file or directory name as input. How do we know what the available arguments that go with a particular command are? Most commonly used shell commands have a manual available in the shell. You can access the manual using the `man` command. Let's try this command with `ls`:

```
$ man ls
```

This will open the manual page for `ls` and you will lose the command prompt. It will bring you to a so-called "buffer" page, a page you can navigate with your mouse or if you want to use your keyboard we have listed some basic key strokes:

- 'spacebar' to go forward

- 'b' to go backward
- Up or down arrows to go forward or backward, respectively

To get out of the `man` "buffer" page and to be able to type commands again on the command prompt, press the `q` key!

Exercise

- Open up the manual page for the `find` command. Skim through some of the information.
    - Do you think you might be able to learn this much information about the very many command by heart?
    - Do you think this format of information display is useful for you?
- Quit the `man` buffer and come back to your command prompt.

> **Tip** - Shell commands can get extremely complicated. No one can possibly learn all of these arguments, of course. So you will probably find yourself referring to the manual page frequently.
>
> **Tip** - If the manual page within the Terminal is hard to read and traverse, the manual exists online too. Use your web searching powers to get it! In addition to the arguments, you can also find good examples online; ***Google is your friend.***

The Unix directory file structure (a.k.a. where am I?)

Let's practice moving around a bit. Let's go into the raw_fastq directory and see what is in there.

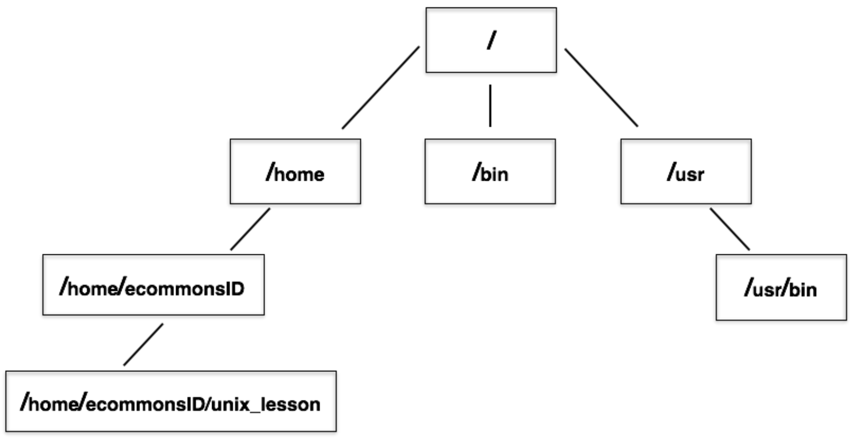
```
$ cd raw_fastq/

$ ls -l
```

Great, we have now traversed some sub-directories, but where are we in the context of our pre-designated "home" directory that contains the `unix_lesson` directory?!

The "root" directory!

Like on any computer you have used before, the file structure within a Unix/Linux system is hierarchical, like an upside down tree with the "/" directory, called "root" as the starting point of this tree-like structure:



> **Tip** - Yes, the root folder's actual name is just `/` (a forward slash).

That `/` or root is the 'top' level.

When you log in to a remote computer you land on one of the branches of that tree, i.e. your pre-designated "home" directory that usually has your login name as its name (e.g. `/home/rsk27`).

> **Tip** - On mac OS, which is a UNIX-based OS, the root level is also "/".
>
> **Tip** - On a windows OS, it is drive specific; "C:" is considered the default root, but it changes to "D:/", if you are on that drive.

Paths

Now let's learn more about the "addresses" of directories, called **"path"** and move around the file system.

Let's check to see what directory we are in. The command prompt tells us which directory we are in, but it doesn't give information about where the `raw_fastq` directory is with respect to our "home" directory or the `/` directory.

The command to check our current location is `pwd`, this command does not take any arguments and it returns the path or address of your **p**resent **w**orking **d**irectory (the folder you are in currently).

```
$ pwd
```

In the output here, each folder is separated from its "parent" or "child" folder by a "/", and the output starts with the root `/` directory. So, you are now able to determine the location of `raw_fastq` directory relative to the root directory!

But which is your pre-designated home folder? No matter where you have navigated to in the file system, just typing in `cd` will bring you to your home directory.

```
$ cd
```

What is your present working directory now?

```
$ pwd
```

This should now display a shorter string of directories starting with root. This is the full address to your home directory, also referred to as "**full path**". **The "full" here refers to the fact that the path starts with the root, which means you know which branch of the tree you are on in reference to the root.**

Take a look at your command prompt now, does it show you the name of this directory (your username?)?

No, it doesn't. Instead of the directory name it shows you a `~`.

Why is this so?

This is because `~` = full path to home directory for the user.

Can we just type `~` instead of `/home/username`?

Yes, we can!

Using paths with commands

You can do a lot more with the idea of stringing together *parent/child* directories. Let's say we want to look at the contents of the `raw_fastq` folder, but do it from our current directory (the home directory. We can use the list command and follow it up with the path to the folder we want to list!

```
$ cd

$ ls -l ~/unix_lesson/raw_fastq
```

Now, what if we wanted to change directories from `~` (home) to `raw_fastq` in a single step?

```
$ cd ~/unix_lesson/raw_fastq
```

Voila! You have moved 2 levels of directories in one command.

What if we want to move back up and out of the `raw_fastq` directory? Can we just type `cd unix_lesson`? Try it and see what happens.

Unfortunately, that won't work because when you say `cd unix_lesson`, shell is looking for a folder called `unix_lesson` within your current directory, i.e. `raw_fastq`.

Can you think of an alternative?

You can use the full path to unix_lesson!

```
$ cd ~/unix_lesson
```

Exercises

1. First, move to your home directory.
2. Then, list the contents of the `genomics_data` directory that is within the `unix_lesson` directory.

Tab completion

Typing out full directory names can be time-consuming and error-prone. One way to avoid that is to use **tab completion**. The `tab` key is located on the left side of your keyboard, right above the `caps lock` key. When you start typing out the first few characters of a directory name, then hit the `tab` key, Shell will try to fill in the rest of the directory name.

For example, first type `cd` to get back to your home directly, then type `cd uni`, followed by pressing the `tab` key:

```
$ cd
$ cd uni<tab>
```

The shell will fill in the rest of the directory name for `unix_lesson`.

Now, let's go into `raw_fastq`, then type `ls Mov10_oe_`, followed by pressing the `tab` key once:

```
$ cd raw_fastq/
$ ls Mov10_oe_<tab>
```

Nothing happens!!

The reason is that there are multiple files in the `raw_fastq` directory that start with `Mov10_oe_`. As a result, shell does not know which one to fill in. When you hit `tab` a second time again, the shell will then list all the possible choices.

```
$ ls Mov10_oe_<tab><tab>
```

Now you can select the one you are interested in listed, and enter the number and hit tab again to fill in the complete name of the file.

```
$ ls Mov10_oe_1<tab>
```

> **NOTE:** Tab completion can also fill in the names of commands. For example, enter `e<tab><tab>`. You will see the name of every command that starts with an `e`. One of those is `echo`. If you enter `ech<tab>`, you will see that tab completion works.

Tab completion is your friend! It helps prevent spelling mistakes, and speeds up the process of typing in the full command. We encourage you to use this when working on the command line.

Relative paths

We have talked about **full** paths so far, but there is a way to specify paths to folders and files without having to worry about the root directory. And you have used this before when we were learning about the `cd` command.

Let's change directories back to our home directory, and once more change directories from `~` (home) to `raw_fastq` in a single step. (*Feel free to use your tab-completion to complete your path!*)

```
$ cd
$ cd unix_lesson/raw_fastq
```

This time we are not using the `~/` before `unix_lesson`. In this case we are using a relative path, relative to our current location - wherein we know that `unix_lesson` is a child folder in our home folder, and the `raw_fastq` folder is within `unix_lesson`.

> Previously we had used the following:
>
> ```
> $ cd ~/unix_lesson/raw_fastq
> ```

There is also a handy shortcut for the relative path to a parent directory, 2 periods `..`. Let's say we wanted to move from the `raw_fastq` folder to its parent folder.

```
cd ..
```

You should now be in the `unix_lesson` directory (check command prompt or run `pwd`).

> You will be learning a little more about the `..` shortcut later. Can you think of an example when this shortcut to the parent directory won't work?
>
> ▸ Answer

When using relative paths, you might need to check what the branches are downstream of the folder you are in. There is a really handy command (`tree`) that can help you see the structure of any directory.

```
$ tree
```

If you are aware of the directory structure, you can string together as long a list of directories as you like using either **relative** or **full** paths.

Synopsis of Full versus Relative paths

A full path always starts with a `/`, a relative path does not.

A relative path is like getting directions from someone on the street. They tell you to "go right at the Stop sign, and then turn left on Main Street". That works great if you're standing there together, but not so well if you're trying to tell someone how to get there from another country. A full path is like GPS coordinates. It tells you exactly where something is no matter where you are right now.

You can usually use either a full path or a relative path depending on what is most convenient. If we are in the home directory, it is more convenient to just enter the relative path since it involves less typing.

Over time, it will become easier for you to keep a mental note of the structure of the directories that you are using and how to quickly navigate among them.

Copying, creating, moving and removing data

Now we can move around within the directory structure using the command line. But what if we want to do things like copy files or move them from one directory to another, rename them?

Let's move into the `raw_fastq` directory, this contains some fastq files which are the output of sequencing.

```
cd ~/unix_lesson/raw_fastq
```

> **Tip** - These files are referred to as "raw" data since it has not been changed or analyzed after being generated.

Copying

Let's use the copy (`cp`) command to make a copy of one of the files in this folder, `Mov10_oe_1.subset.fq`, and call the copied file `Mov10_oe_1.subset-copy.fq`. The copy command has the following syntax:

`cp  path/to/item-being-copied  path/to/new-copied-item`

In this case the files are in our current directory, so we just have to specify the name of the file being copied, followed by whatever we want to call the newly copied file.

```
$ cp Mov10_oe_1.subset.fq Mov10_oe_1.subset-copy.fq

$ ls -l
```

The copy command can also be used for copying over whole directories, but the `-r` argument has to be added after the `cp` command. The `-r` stands for recursively copy everything from the directory and its sub-directories". We used it earlier when we copied over the `unix_lesson` directory to our home directories.

Creating

Next, let's create a directory called `fastq_backup` and we can move the copy of the fastq file into that directory.

The `mkdir` command is used to make a directory, syntax: `mkdir  name-of-folder-to-be-created`.

```
$ mkdir fastq_backup
```

> **Tip** - File/directory/program names with spaces in them do not work well in Unix, use characters like hyphens or underscores instead. Using underscores instead of spaces is called "snake_case". Alternatively, some people choose to skip spaces and rather just capitalize the first letter of each new word (i.e. MyNewFile). This alternative technique is called "CamelCase".

Moving

We can now move our copied fastq file in to the new directory. We can move files around using the move command, `mv`, syntax:

`mv  path/to/item-being-moved  path/to/destination`

In this case we can use relative paths and just type the name of the file and folder.

```
$ mv  Mov10_oe_1.subset-copy.fq  fastq_backup
```

Let's check if the move command worked like we wanted:

```
$ ls -l fastq_backup
```

Renaming

The `mv` command has a second functionality, it is what you would use to rename files too. The syntax is identical to when we used `mv` for moving, but this time instead of giving a directory as its destination, we just give a new name as its destination.

Let's try out this functionality!

The name Mov10_oe_1.subset-copy.fq is not very informative, we want to make sure that we have the word "backup" in it so we don't accidentally delete it.

```
$ cd fastq_backup

$ mv  Mov10_oe_1.subset-copy.fq   Mov10_oe_1.subset-backup.fq

$ ls
```

> **Tip** - You can use move to move a file and rename it at the same time!

Important notes about `mv`:

- When using `mv`, shell will **not** ask if you are sure that you want to "replace existing file" or similar unless you use the -i option.
- Once replaced, it is not possible to get the replaced file back!

Removing

We find out that we did not need to create backups of our fastq files manually as backups were generated by our collaborator; in the interest of saving space on the cluster, we want to delete the contents of the `fastq-backup` folder and the folder itself.

```
$ rm  Mov10_oe_1.subset-backup.fq
```

Important notes about `rm`

- `rm` permanently removes/deletes the file/folder.
- There is no concept of "Trash" or "Recycle Bin" on the command-line. When you use `rm` to remove/delete they're really gone.
- **Be careful with this command!**
- You can use the `-i` argument if you want it to ask before removing, `rm -i file-name`.

Let's delete the fastq_backup folder too. First, we'll have to navigate our way to the parent directory (we can't delete the folder we are currently in/using).

```
$ cd ..

$ rm  fastq_backup
```

Did that work? Did you get an error?

▸ *Explanation*

```
$ rm -ri fastq_backup
```

- `-r` : recursive, commonly used as an option when working with directories, e.g. with `cp` .
- `-i` : prompt before every removal.

Exercise

1. Create a new folder in `unix_lesson` called `selected_fastq`
2. Copy over the Irrel_kd_2.subset.fq and Mov10_oe_2.subset.fq from `raw_fastq` to the `~/unix_lesson/selected_fastq` folder
3. Rename the `selected_fastq` folder and call it `exercise1`

Saving time with wildcards and other shortcuts

Wild cards

The "*" wildcard:

Navigate to the `~/unix_lesson/raw_fastq` directory. This directory contains FASTQ files from a next-generation sequencing dataset.

The "*" character is a shortcut for "everything". Thus, if you enter `ls *` , you will see all of the contents of a given directory. Now try this command:

```
$ ls *fq
```

This lists every file that ends with a `fq` . Try this command:

```
$ ls /usr/bin/*.sh
```

This lists every file in `/usr/bin` directory that ends in the characters `.sh` . "*" can be placed anywhere in your pattern. For example:

```
$ ls Mov10*fq
```

This lists only the files that begin with 'Mov10' and end with `fq` .

So how does this actually work? The Shell (bash) considers an asterisk "*" to be a wildcard character that can match one or more occurrences of any character, including no character.

> **Tip** - An asterisk/star is only one of the many wildcards in Unix, but this is the most powerful one and we will be using this one the most for our exercises.

The "?" wildcard:

Another wildcard that is sometimes helpful is `?` . `?` is similar to `*` except that it is a placeholder for exactly one position. Recall that `*` can represent any number of following positions, including no positions. To highlight this distinction lets look at a few examples. First, try this command:

```
ls /bin/d*
```

This will display all files in `/bin/` that start with "d" regardless of length. However, if you only wanted the things in `/bin/` that start with "d" and are two characters long then you can use:

```
ls /bin/d?
```

Lastly, you can chain together multiple "?" marks to help specify a length. In the example below, you would be looking for all things in `/bin/` that start with a "d" and have a name length of three characters.

```
ls /bin/d??
```

Exercise

Do each of the following using a single `ls` command without navigating to a different directory.

1. List all of the files in `/bin` that start with the letter 'c'
2. List all of the files in `/bin` that contain the letter 'a'
3. List all of the files in `/bin` that end with the letter 'o'
4. BONUS: Using one command to list all of the files in `/bin` that contain either 'a' or 'c'. (Hint: you might need to use a different wildcard here. Refer to this post for some ideas.)

   ▸ ***Answers***

Shortcuts

There are some very useful shortcuts that you should also know about.

Home directory or "~"

Dealing with the home directory is very common. In shell, the tilde character, "~", is a shortcut for your home directory. Let's first navigate to the `raw_fastq` directory (try to use tab completion here!):

```
$ cd
$ cd unix_lesson/raw_fastq
```

Then enter the command:

```
$ ls ~
```

This prints the contents of your home directory, without you having to type the full path. This is because the tilde "~" is equivalent to "/home/username", as we had mentioned in the previous lesson.

Parent directory or ".."

Another shortcut you encountered in the previous lesson is "..":

```
$ ls ..
```

The shortcut `..` always refers to the parent directory of whatever directory you are in currently. So, `ls ..` will print the contents of `unix_lesson`. You can also chain these `..` together, separated by `/`:

```
$ ls ../..
```

This prints the contents of `/home/username`, which is two levels above your current directory (your home directory).

Current directory or "."

Finally, the special directory `.` always refers to your current directory. So, `ls` and `ls .` will do the same thing - they print the contents of the current directory. This may seem like a useless shortcut, but recall that we used it earlier when we copied over the data to our home directory.

To summarize, the commands `ls ~`, `ls ~/.`, and `ls /home/username` all do exactly the same thing. These shortcuts can be convenient when you navigate through directories!

Command History

You can easily access previous commands by hitting the up arrow key on your keyboard, this way you can step backwards through your command history. On the other hand, the down arrow key takes you forward in the command history.

Try it out! While on the command prompt hit the up arrow a few times, and then hit the down arrow a few

times until you are back to where you started.

You can also review your recent commands with the `history` command. Just enter:

```
$ history
```

You should see a numbered list of commands, including the `history` command you just ran!

Only a certain number of commands can be stored and displayed with the `history` command by default but you can increase or decrease it to a different number. It is outside the scope of this workshop, but feel free to look it up after class.

> **NOTE:** So far we have only run very short commands that have very few or no arguments. It would be faster to just retype it than to check the history. However, as you start to run analyses on the command-line you will find that the commands are longer and more complex, and the `history` command will be very useful then!

Cancel a command

Sometimes as you enter a command, you realize that you don't want to continue or run the current line. Instead of deleting everything you have entered (which could be very long), you could quickly cancel the current line and start a fresh prompt with Ctrl + C.

```
$ # Run some random words, then hit "Ctrl + C". Observe what happens
```

Other handy command-related shortcuts

- Ctrl + A will bring you to the start of the command you are writing.
- Ctrl + E will bring you to the end of the command.

Exercise

1. Checking the output of the `history` command, how many commands have you typed in so far?
2. Use the up arrow key to check the command you typed before the `history` command. What is it? Does it make sense?
3. Type several random characters on the command prompt. Can you bring the cursor to the start with Ctrl + A? Next, can you bring the cursor to the end with Ctrl + E? Finally, what happens when you use Ctrl + C?

Examining Files

We now know how to move around the file system and look at the contents of directories, but how do we look at the contents of files? On your laptop, viewing a file is as simple as finding it in the file explorer window and double clicking to open it. As you will have noticed so far, the point and click of the mouse is not very useful when working on the command-line. Instead we will need to equip ourseleves with some helpful commands.

`cat` command

The easiest way to examine a file is to just print out all of its contents using the command `cat`. We can test this out by printing the contents of `~/unix_lesson/other/sequences.fa`. Enter the command followed by the filename, including the path when necessary:

```
$ cat ~/unix_lesson/other/sequences.fa
```

The `cat` command prints out the all the contents of `sequences.fa` to the screen.

> `cat` stands for catenate; it has many uses and printing the contents of a files onto the terminal is one of them.

What does this file contain?

```

>SRR014849.1 EIXKN4201CFU84 length=93
GGGGGGGGGGGGGGGGCTTTTTTTGTTTGGAACCGAAAGGGTTTTGAATTTCAAACCCTTTTCGGTTTCCAACCTTCCAAAGCAATGCCAATA

>gi|340780744|ref|NC_015850.1| Acidithiobacillus caldus SM-1 chromosome, complete genome
ATGAGTAGTCATTCAGCGCCGACAGCGTTGCAAGATGGAGCCGCGCTGTGGTCCGCCCTATGCGTCCAACTGGAGCTCGTCACGAG
```

```
TCCGCAGCAGTTCAATACCTGGCTGCGGCCCCTGCGTGGCGAATTGCAGGGTCATGAGCTGCGCCTGCTCGCCCCCAATCCCTTCG
TCCGCGACTGGGTGCGTGAACGCATGGCCGAACTCGTCAAGGAACAGCTGCAGCGGATCGCTCCGGGTTTTGAGCTGGTCTTCGCT
CTGGACGAAGAGGCAGCAGCGGCGACATCGGCACCGACCGCGAGCATTGCGCCCGAGCGCAGCAGCGCACCCGGTGGTCACCGCCT
CAACCCAGCCTTCAACTTCCAGTCCTACGTCGAAGGGAAGTCCAATCAGCTCGCCCTGGCGGCAGCCCGCCAGGTTGCCCAGCATC
CAGGCAAATCCTACAACCCACTGTACATTTATGGTGGTGTGGGCCTCGGCAAGACGCACCTCATGCAGGCCGTGGGCAACGATATC
CTGCAGCGGCAACCCGAGGCCAAGGTGCTCTATATCAGCTCCGAAGGCTTCATCATGGATATGGTGCGCTCGCTGCAACACAATAC
CATCAACGACTTCAAACAGCGTTATCGCAAGCTGGACGCCCTGCTCATCGACGACATCCAGTTCTTTGCGGGCAAGGACCGCACCC

>gi|129295|sp|P01013|OVAX_CHICK GENE X PROTEIN (OVALBUMIN-RELATED)
QIKDLLVSSSTDLDTTLVLVNAIYFKGMWKTAFNAEDTREMPFHVTKQESKPVQMMCMNNSFNVATLPAE
```

`less` command

`cat` is a terrific command, but when the file is really big, it can be annoying to use. In practice, when you are running your analyses on the command-line you will most likely be dealing with large files. In our case, we have FASTQ files. Let's take a look at the list of raw_fastq files and add the `-h` modifier to see how big the files are.

```
$ ls -lh ~/unix_lesson/raw_fastq
```

> The `ls` command has a modifier `-h` when paired with `-l`, will list the files and also print sizes of files in human readable format.

In the fourth column you wll see the size of each of these files, and you can see they are quite large, so we probably do not want to use the `cat` command to look at them. Instead, we can use the `less` command.

Move into our `raw_fastq` directory and enter the following command:

```
$ less Mov10_oe_1.subset.fq
```

Rather than printing to screen, the `less` command opens the file in a new buffer allowing you to navigate through it. Does this look familiar? You might remember encountering a similar interface when you used the `man` command. This is because `man` is using the `less` command to open up the documentation files! The keys used to move around the file are identical to the `man` command. Below we have listed some additional shortcut keys for naviagting through your file when using `less`.

Shortcuts for `less`

key	action
`SPACE`	to go forward
`b`	to go backwards
`g`	to go to the beginning
`G`	to go to the end
`q`	to quit

Use the shortcut keys to move through your FASTQ file, we will explore these files in more detail later in the workshop.

Searching files with `less`

`less` also gives you a way of searching through files.

Just type in `/` to begin a search, you will see that the `/` will show up at the bottom of the `less` buffer. Now, enter the name of the string of characters you would like to search for and hit the enter key. The interface will move to show you the location where that string is found, and highlight the string. If you hit `/` then `ENTER`, `less` will just repeat the previous search.

`less` searches from the current location and works its way forward. For instance, let's search for the sequence `CAGAAT` in our file `S1WTgex_S1_L001_R1_001.fastq` in folder `raw_fastq`. You can see that we go right to that sequence and can see what it looks like.

If you start a search when you are at the end of the file, `less` will not find it. You need to go to the beginning of the file and search.

To exit hit `q`. There are other more sophisticated commands to search through your file (and we will cover these later), but this shortcut search is useful for a quick scan through. You can think of it as being analagous to using the `Ctrl-F` keystroke when searching on your laptop.

`head` and `tail` commands

There's another way that we can look at files, and just look at part of them. In particular, if we just want to see the beginning or end of the file to see how it's formatted.

The commands are `head` and `tail` and they just let you look at the beginning and end of a file respectively.

```
$ head S1WTgex_S1_L001_R1_001.fastq
```

```
$ tail S1WTgex_S1_L001_R1_001.fastq
```

By default, the first or last 10 lines will be printed to screen. The `-n` option can be used with either of these commands to specify the number `n` lines of a file to display. For example, let's print the first/last line of the file:

```
$ head -n 1 S1WTgex_S1_L001_R1_001.fastq

$ tail -n 1 S1WTgex_S1_L001_R1_001.fastq
```

Exercise

1. Change directories into `genomics_data`. You can do this using a full or relative path.
2. Use the `less` command to open up the file `mouse_riboprotein_genes.tab`.
3. Search for the string `chr11`; you'll see all instances in the file highlighted.
4. Staying in the `less` buffer, use the shortcut to get to the end of the file. Report the three highlighted lines at the end of the file where you see `chr11` highlighted.
5. Exit the `less` buffer and come back to the command prompt.
6. Print to screen the last 5 lines of the file `mouse_riboprotein_genes.tab`. Report what you see as the output within the Terminal.

Writing files

We've been able to do a lot of work with files that already exist, but what if we want to write and/or create our own files? Obviously, we're not going to type in sequence information for a FASTA file, but you'll see as we go that there are a lot of situations in which we would need to write/create a file or edit an existing file.

In order to create or edit files we will need to use a **text editor**. When we say, "text editor," we really do mean "text": these editors can only work with plain character data, not tables, images, or any other media. The types of text editors available can generally be grouped into two categories: **graphical user interface (GUI) text editors** and **command-line editors**.

GUI text editors

A GUI is an interface that has buttons and menus that you can click on to issue commands to the computer and you can move about the interface just by pointing and clicking. You might be familar with GUI text editors, such as BBEdit, Sublime, and Notepad++, which allow you to write and edit plain text documents. These editors often have features to easily search text, extract text, and highlight syntax from multiple programming languages. They are great tools, but since they are 'point-and-click', we cannot efficiently use them from the command line.

Command-line editors

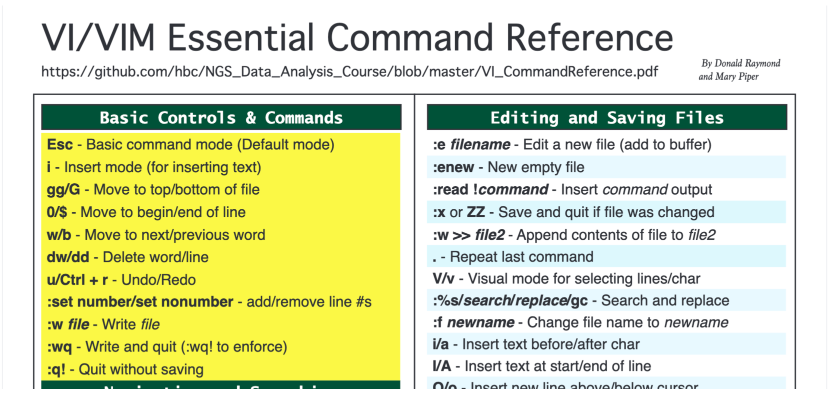
When working remotely, we need a text editor that functions from the command line interface. With command-line editors you must navigate the interface using the arrow keys and shortcuts, since you do not have the option to 'point-and-click'. Some popular editors include Emacs, Vim, or a graphical editor such as Gedit. These are editors which are generally available for use on high-performance compute clusters. There are also simpler editors available for use on the cluster (e.g. nano), but tend to have limited functionality.

Introduction to Vim

To write and edit files, we're going to use a text editor called 'Vim'. Vim is a very powerful text editor, and it offers extensive text editing options. However, in this introduction we are going to focus on **exploring some of the more basic functions**.

How do I keep track of all these shortcuts in Vim?

To help you remember some of the keyboard shortcuts that are introduced below and to allow you to explore additional functionality on your own, we have compiled a cheatsheet linked here. Download it to your computer, it is a useful resource to have open while using Vim.



VI/VIM Essential Command Reference

https://github.com/hbc/NGS_Data_Analysis_Course/blob/master/VI_CommandReference.pdf

By Donald Raymond and Mary Piper

Basic Controls & Commands	Editing and Saving Files
Esc - Basic command mode (Default mode)	**:e** ***filename*** - Edit a new file (add to buffer)
i - Insert mode (for inserting text)	**:enew** - New empty file
gg/G - Move to top/bottom of file	**:read !*command*** - Insert *command* output
0/$ - Move to begin/end of line	**:x** or **ZZ** - Save and quit if file was changed
w/b - Move to next/previous word	**:w >> *file2*** - Append contents of file to *file2*
dw/dd - Delete word/line	**.** - Repeat last command
u/Ctrl + r - Undo/Redo	**V/v** - Visual mode for selecting lines/char
:set number/set nonumber - add/remove line #s	**:%s/*search*/*replace*/gc** - Search and replace
:w *file* - Write *file*	**:f *newname*** - Change file name to *newname*
:wq - Write and quit (:wq! to enforce)	**i/a** - Insert text before/after char
:q! - Quit without saving	**I/A** - Insert text at start/end of line
	O/o - Insert new line above/below cursor

Vim Interface

You can create a document by calling a text editor (in our case `vim`) and providing the name of the document you wish to create.

Change directories to the `~/unix_lesson` folder and create a new folder `vim`. Change into that folder `vim` and create a test file called `draft.txt` using the `vim` command:

```
$ cd ~/unix_lesson
$ mkdir vim
$ cd vim

$ vim draft.txt
```

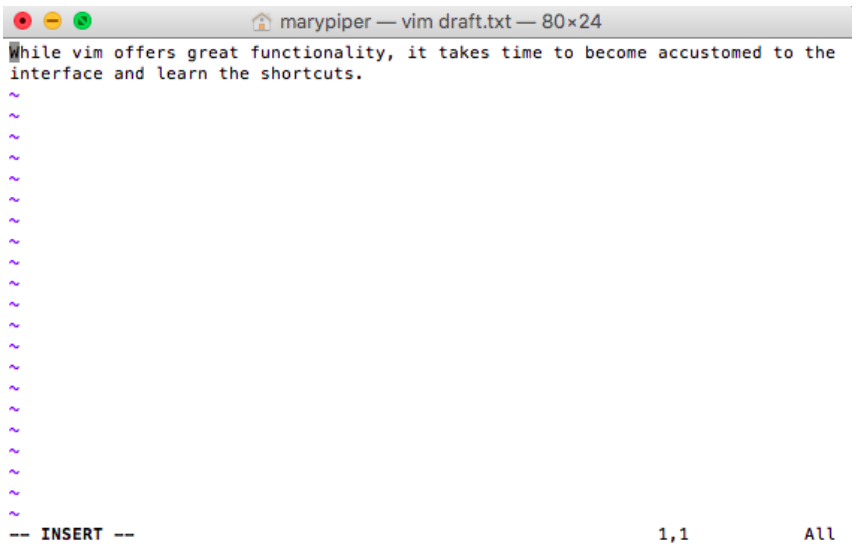
Note the `"draft.txt" [New File]` typed at the bottom left-hand section of the screen. This tells you that you just created a new file in vim.

Vim Modes

Vim has ***two basic modes*** that will allow you to create documents and edit your text:

- ***command mode (default mode):*** will allow you to save and quit the program (and execute other more advanced commands).
- ***insert (or edit) mode:*** will allow you to write and edit text

Upon creation of a file, vim is automatically in command mode. Let's *change to insert mode* by typing `i`. **Note the `--INSERT--` at the bottom left hand of the screen.** Now type in a few lines of text:



marypiper — vim draft.txt — 80×24

```
While vim offers great functionality, it takes time to become accustomed to the
interface and learn the shortcuts.
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
-- INSERT --                                              1,1           All
```

After you have finished typing, **press `esc` to enter command mode.**

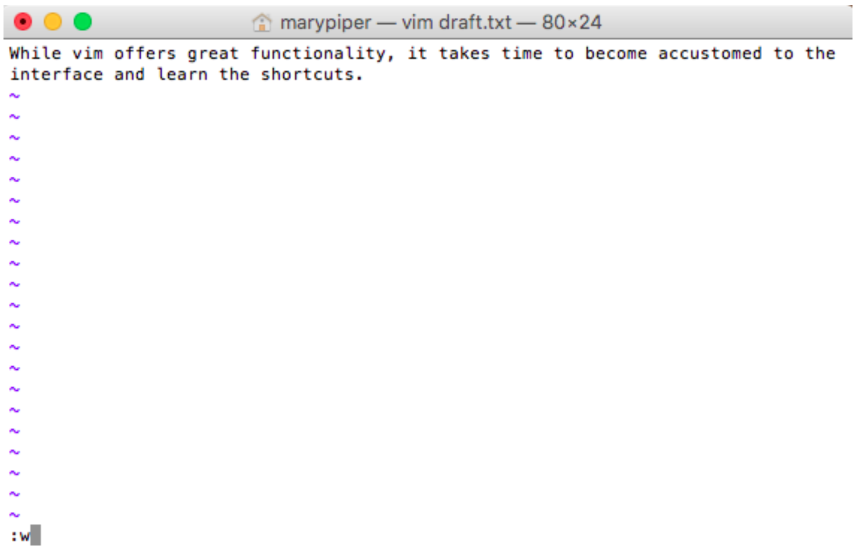
Note the `--INSERT--` has now disappeared from the bottom of the screen.

Review of Vim modes

key	action
i	insert mode - to write and edit text
esc	command mode - to issue commands / shortcuts

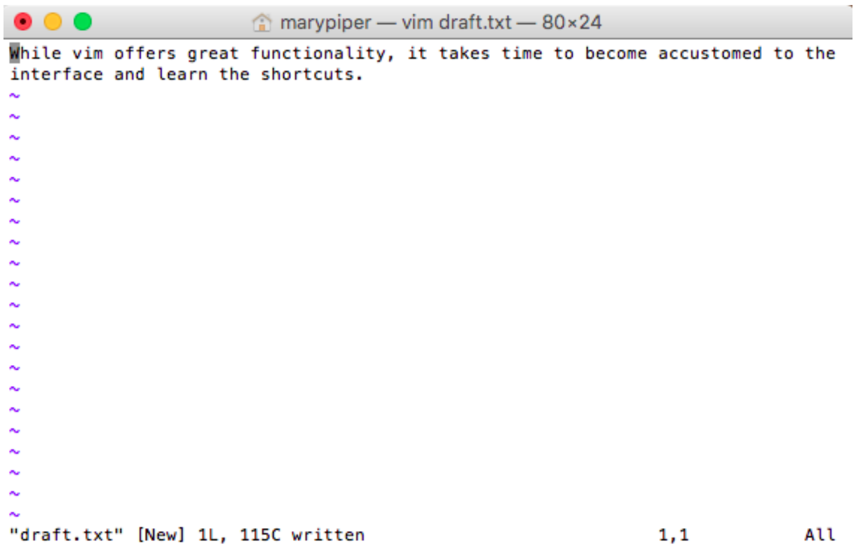
Saving and Quitting

To **"write to file"** or save the modifications made to the file, **type** `:w` when in command mode. You can see the commands you type in the bottom left-hand corner of the screen.



```
While vim offers great functionality, it takes time to become accustomed to the
interface and learn the shortcuts.
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
:w
```

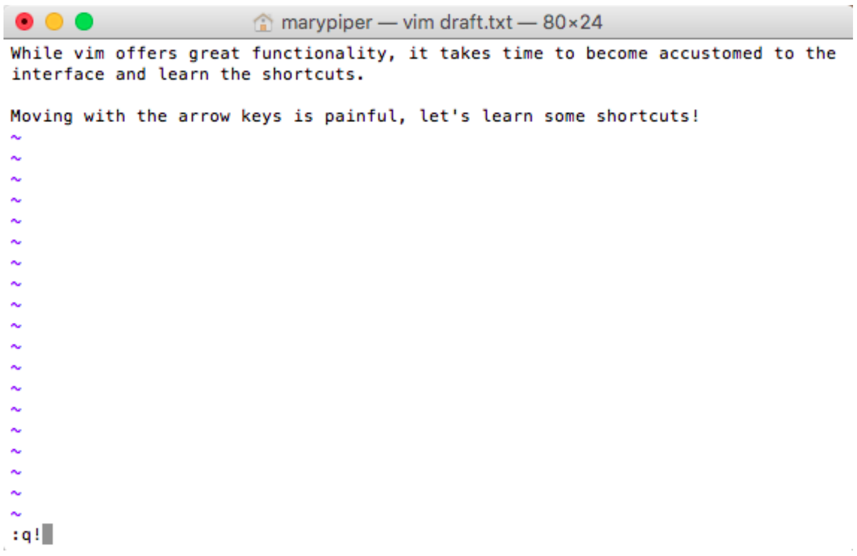
After you have saved the file, the total number of lines and characters in the file will print out at the bottom left-hand section of the screen.



```
While vim offers great functionality, it takes time to become accustomed to the
interface and learn the shortcuts.
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"draft.txt" [New] 1L, 115C written                           1,1           All
```

Alternatively, we can **write to file (save changes) and quit** all at once by **typing** `:wq`. Now, you should have exited vim and returned back to your command prompt.

To edit the newly created `draft.txt` file, you can open it again with vim: `vim draft.txt`. First, change to *insert mode* and type a few additional lines (you can move around the lines using the arrows on the keyboard). This time we decide to **quit without saving** by going into command mode by pressing the esc key, and then **typing** `:q!`.



```
While vim offers great functionality, it takes time to become accustomed to the
interface and learn the shortcuts.

Moving with the arrow keys is painful, let's learn some shortcuts!
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
:q!
```

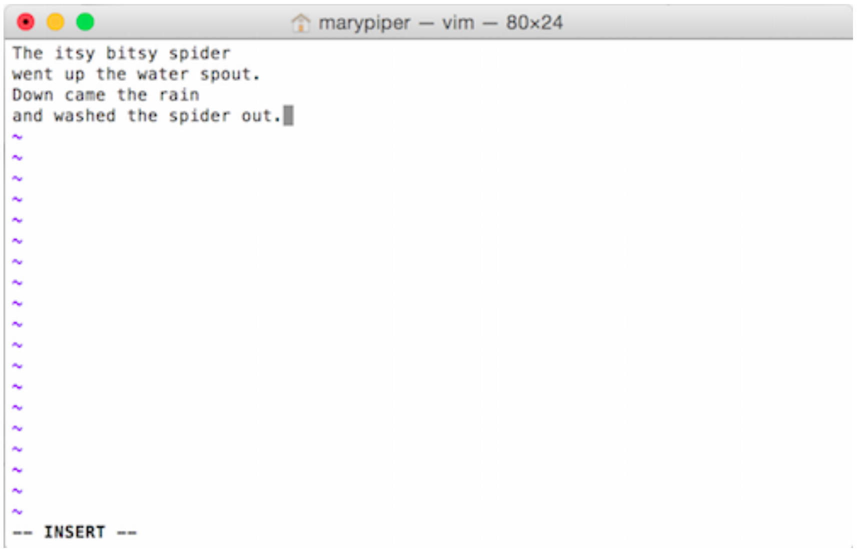
Review of saving and quitting

key (in command mode)	action
:w	to write to file (save)
:wq	to write to file and quit
:q!	to quit without saving

Shortcuts in Vim

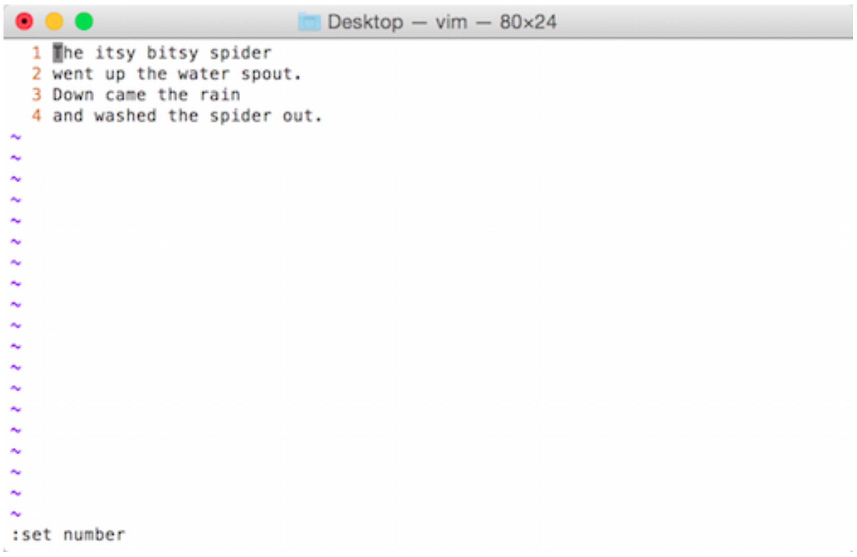
While we cannot point and click to navigate the document, we can use the arrow keys to move around. However, navigating with arrow keys can be very slow, so Vim has shortcuts (which are completely unintuitive, but very useful as you get used to them over time).

Create a new file called `spider.txt` using `vim`. Go into *insert mode* and enter the text as shown below in the screenshot:



```
The itsy bitsy spider
went up the water spout.
Down came the rain
and washed the spider out.
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
-- INSERT --
```

Once you have finished typing, you can display line numbers by changing to *command mode* and then typing the `:set number` command. Later, if you choose to remove the line numbers you can reset it with `:set nonumber`.



```
The itsy bitsy spider
went up the water spout.
Down came the rain
and washed the spider out.
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
:set number
```

key (in command mode)	action
:set number	to number lines
:set nonumber	to remove line numbers

Save the document. Check to see what mode you are currently in. **While in command mode**, try moving around the file `spider.txt` and familarizing yourself with some of these shortcuts!

Navigating around the file

key (in command mode)	action
gg	to move to top of file
G	to move to bottom of file
$	to move to end of line
0	to move to beginning of line
w	to move to next word
b	to move to previous word

Practice some of the editing shortcuts, then **quit the document without saving any changes**.

Editing the file

key (in command mode)	action
dw	to delete word
dd	to delete line
u	to undo
Ctrl + r	to redo
/*pattern*	to search for a pattern (*n*/*N* to move to next/previous match)
:%s/*search*/*replace*/g	to search for a pattern and replace for all occurences

---

Exercise

We have covered some basic commands in vim, but practice is key for getting comfortable with the program. Let's practice what we just learned in a brief challenge.

1. Open `spider.txt`, and delete the word "water" from line #2.

2. Quit without saving.
3. Open `spider.txt` again, and replace every occurrence of "spider" with "unicorn".
4. Delete: "Down came the rain."
5. Save the file.
6. Undo your previous deletion.
7. Redo your previous deletion.
8. Delete the first and last words from each of the lines.
9. Save the file.
10. Open up the file and copy and paste the contents to a text editor on your local laptop to submit as homework.

Searching files with `grep` command

We went over how to search within a file using `less`. We can also search within files without even opening them, using `grep`. Simply put `grep` is a command-line utility for searching plain-text data sets for lines matching a pattern or **reg**ular **ex**pression (regex).

> Why the word "grep"? It is a shortened form of **g**lobally search for a **r**egular **e**xpression and **p**rint matching lines (g/re/p).

The syntax for `grep` is as follows: `grep search_term filename`. The pattern that we want to search is specified in `search_term` slot, and the file we want to search within is specified in the `filename` slot. Let's give it a try by searching the FASTQ files in the `raw_fastq` directory.

FASTQ files contain the sequencing reads (nucleotide sequences) output from a sequencing facility. Each sequencing read in a FASTQ file is associated with four lines, with the first line (header line) always starting with an `@` symbol. A whole FASTQ record for a single read should appear similar to the following:

```
@HWI-ST330:304:H045HADXX:1:1101:1111:61397
CACTTGTAAGGGCAGGCCCCCTTCACCCTCCCGCTCCTGGGGGANNNNNNNNNNANNNCGAGGCCCTGGGGTAGAGGGNNNNNNNNNNNNNNGATCTTGG
+
B?@DDDDDDHHH?GH:?FCBGGB@C?DBEGIIIIAEF;FCGGI#########################################################
```

> **More information about the FASTQ file format**
>
> | Line | Description |
> |---|---|
> | 1 | Read name preceded by '@' |
> | 2 | The actual DNA sequence |
> | 3 | Read name (same as line 1) preceded by a '+' or just a '+' sign |
> | 4 | String of characters which represent the quality score of each nucleotide in line 2; must have same number of characters as line 2 |
>
> You can find more information about FASTQ files in this lesson from our RNA-seq workshop.

Suppose we want to see how many reads in our file `S1WTgex_S1_L001_R1_001.fastq` contain the subsequence CAGAAT

```
$ cd ~/unix_lesson/raw_fastq

$ grep CAGAAT S1WTgex_S1_L001_R1_001.fastq
```

We get back a lot of reads or lines of text!

What if we wanted to see the whole FASTQ record for each of these reads? We would need to modify the default behavior of `grep` and specify some argument/options. To look for all available options for the `grep` command, we can type `grep --help` (or `man grep`).

Looks like the `-B` and `-A` arguments for grep will be useful to return the matched line plus one before (`-B 1`) and two lines after (`-A 2`). Since each record is four lines, using these arguments will return the whole record. Within the whole record, the second line will be the actual sequence that has the pattern we searched for.

```
$ grep -B 1 -A 2 CAGAAT S1WTgex_S1_L001_R1_001.fastq
```

```
@A01182:88:HCWMWDSX3:1:1101:9607:1000 1:N:0:ATGCGAATGG+NGACACTTGT
GNCTATGTCATAACCAAAGGCGGTCAAAAGGATAATCGGTATGCGTAGTTGGTTGTTCTTTTTATAATATATTTTTTAGAATTTTTAAGATACCTCAGAAT
+
F#FFFFFFFFFFFFFFFFFFFFFFFFFFF:,,:,,::,,,,,,F,,,,,F:,,,F:F:FFFFFFF,F:FFF:,:::FFFF:F::F:::,F,F:FFF,,FFF:
--
@A01182:88:HCWMWDSX3:1:1101:30156:1016 1:N:0:ATGCGAATGG+NGACACTTGT
CNATCACTCTCACTATCAATTACGGCTGGGCGCCCCAAGCAGAATAATTGGGGGGTTTTTTTTTGCTTCCCTGACCATCTACAGAAAATATTCTAACAATC
+
F#:FFFFFFFFFFFFFFFFFFFFFFFFF,,:F,F,,,,,,,,::,,,,,::,FFF,,,F,F,F,:,:,,,,:F,:F,FF,,,,F:,,,FF,:,,F::F,FF
```

Exercises

1. Search for the sequence CTCAAT in `S1WTgex_S1_L001_R1_001.fastq`. How many sequences do you find?
2. In addition to finding the sequence, how can you modify the command so that your search also returns the name of the sequence?
3. If you want to search for that sequence in **all** Mov10 replicate fastq files, what command would you use?

   ▸ ***Answers***

More about searching text files with `grep`

Group separators (--), and how to remove them

You will notice that when we use the `-B` and/or `-A` arguments with the `grep` command, the output has some additional lines with dashes (`--`), these dashes work to separate your returned "groups" of lines and are referred to as "group separators". This might be problematic if you are trying to maintain the FASTQ file structure or if you simply do not want them in your output. Using the argument `--no-group-separator` with `grep` will disable this behavior:

```
$ grep -B 1 -A 2 --no-group-separator CAGAAT S1WTgex_S1_L001_R1_001.fastq
```

Now your output should be returned as:

```
@A01182:88:HCWMWDSX3:1:1101:9607:1000 1:N:0:ATGCGAATGG+NGACACTTGT
GNCTATGTCATAACCAAAGGCGGTCAAAAGGATAATCGGTATGCGTAGTTGGTTGTTCTTTTTATAATATATTTTTTAGAATTTTTAAGATACCTCAGAAT
+
F#FFFFFFFFFFFFFFFFFFFFFFFFFFF:,,:,,::,,,,,,F,,,,,F:,,,F:F:FFFFFFF,F:FFF:,:::FFFF:F::F:::,F,F:FFF,,FFF:
@A01182:88:HCWMWDSX3:1:1101:30156:1016 1:N:0:ATGCGAATGG+NGACACTTGT
CNATCACTCTCACTATCAATTACGGCTGGGCGCCCCAAGCAGAATAATTGGGGGGTTTTTTTTTGCTTCCCTGACCATCTACAGAAAATATTCTAACAATC
+
F#:FFFFFFFFFFFFFFFFFFFFFFFFF,,:F,F,,,,,,,,::,,,,,::,FFF,,,F,F,F,:,:,,,,:F,:F,FF,,,,F:,,,FF,:,,F::F,FF
```

Which line number has a match?

Another useful option when using `grep` is the `-n` option, which will print out the line number from the file for the match. Adding this option to our previous command would work like this:

```
$ grep -B 1 -A 2 --no-group-separator -n CAGAAT S1WTgex_S1_L001_R1_001.fastq
```

This would return the output:

```
37-@A01182:88:HCWMWDSX3:1:1101:9607:1000 1:N:0:ATGCGAATGG+NGACACTTGT
38:GNCTATGTCATAACCAAAGGCGGTCAAAAGGATAATCGGTATGCGTAGTTGGTTGTTCTTTTTATAATATATTTTTTAGAATTTTTAAGATACCTCAGAAT
39-+
40-F#FFFFFFFFFFFFFFFFFFFFFFFFFFF:,,:,,::,,,,,,F,,,,,F:,,,F:F:FFFFFFF,F:FFF:,:::FFFF:F::F:::,F,F:FFF,,FFF:
225-@A01182:88:HCWMWDSX3:1:1101:30156:1016 1:N:0:ATGCGAATGG+NGACACTTGT
226:CNATCACTCTCACTATCAATTACGGCTGGGCGCCCCAAGCAGAATAATTGGGGGGTTTTTTTTTGCTTCCCTGACCATCTACAGAAAATATTCTAACAATC
227-+
228-F#:FFFFFFFFFFFFFFFFFFFFFFFFF,,:F,F,,,,,,,,::,,,,,::,FFF,,,F,F,F,:,:,,,,:F,:F,FF,,,,F:,,,FF,:,,F::F,FF
```

A small thing you should note is that when using the `-n` option, lines that have a `:` after the line number correspond to the lines with the match (e.g `38:GNCTATGTCATAACCAAAGGCGG...`), while lines with a `-` after the line number are the surrounding lines retrieved when using the `-A` and/or `-B` options (e.g. `37-@A01182:88:HCWMWDSX3:1:1101:9607:1000...`).

Only returning lines that DO NOT match

One last `grep` option you might find quite useful is the `-v` option, which does an inverted match. This will return everything that does ***not*** match the pattern. In order to demonstrate this let's first view a smaller file that you have.

```
$ cat ~/unix_lesson/other/experimental_design.tab
```

This is the metadata file for some FASTQ data. This should return:

```
condition       replicate
g1      gfpn_g1_blue
g2m     gfpn_g2m_blue
g2m     gfpp_g2m_blue
g1      gfpp_g1_blue
g1      gfpn_g1_red
g2m     gfpn_g2m_red
g2m     gfpp_g2m_red
g1      gfpp_g1_red
g1      gfpn_g1_1
g2m     gfpn_g2m_1
g2m     gfpp_g2m_1
g1      gfpp_g1_1
g1      gfpn_g1_2
g2m     gfpn_g2m_2
g2m     gfpp_g2m_2
g1      gfpp_g1_2
```

Now, let's consider the case that we didn't want to output the "g1" cell type. We can use the `-v` option in `grep` like this:

```
$ grep -v g1 ~/unix_lesson/other/Mov10_rnaseq_metadata.txt
```

This will return all of lines that don't have "normal" in the line.

```
condition       replicate
g2m     gfpn_g2m_blue
g2m     gfpp_g2m_blue
g2m     gfpn_g2m_red
g2m     gfpp_g2m_red
g2m     gfpn_g2m_1
g2m     gfpp_g2m_1
g2m     gfpn_g2m_2
g2m     gfpp_g2m_2
```

Redirection

When we use `grep`, the matching lines print to the Terminal (also called Standard Output or "stdout"). If the result of the `grep` search is a few lines, we can view them easily, but if the output is very long, the lines will just keep printing and we won't be able to see anything except the last few lines. You have experienced this when you searched for the pattern `CAGAAT`. How can we capture them instead?

We can do that with something called "redirection". The idea is that we're redirecting the output from the Terminal (all the stuff that went whizzing by) to something else. In this case, we want to save it to a file, so that we can look at it later.

Redirecting with ">"

The redirection command for writing something to file is `>`.

Let's try it out and put all the sequences that contain 'CAGAAT' from the `S1WTgex_S1_L001_R1_001.fastq` into another file called `S1WTgex_S1_L001_R1_001_selected.fastq`.

```
$ cd ~/unix_lesson/raw_fastq

$ grep -B 1 -A 2 CAGAAT S1WTgex_S1_L001_R1_001.fastq > S1WTgex_S1_L001_R1_001_selected.fastq
```

The prompt will go away for a little bit and then you will get it back, but nothing will be printed on the Terminal. But, you should have a new file called `S1WTgex_S1_L001_R1_001_selected.fastq`!

```
$ ls -l
```

Take a look at the file and see if it contains what you think it should.

> NOTE: If we already had a file named `S1WTgex_S1_L001_R1_001_selected.fastq` in our directory, it would have overwritten it without any warning!

Redirecting (and appending) with ">>"

The redirection command for appending something to an existing file is `>>`.

If we use `>>`, it will append to the existing content in a file, rather than overwrite it. This can be useful for saving more than one search. For example, the following command will append the bad reads from Mov10_oe_2 to the bad_reads.txt file that we just generated.

```
$ grep -B 1 -A 2 CTCAAT S1WTgex_S1_L001_R1_001.fastq >> S1WTgex_S1_L001_R1_001_selected.fastq

$ ls -l
```

Did the size of the `bad_reads.txt` file change?

Since our `S1WTgex_S1_L001_R1_001_selected.fastq` file isn't a raw_fastq file, we should move it to a different location within our directory. Let's move it to the `other` folder using the command `mv`.

```
$ mv S1WTgex_S1_L001_R1_001_selected.fastq ../other/
```

Redirecting with pipes "|" (or piping)

The redirection command for using the output of a command as input for a different command is `|`.

The pipe key (`|`) is very likely not something you use very often (it is on the same key as the back slash (`\`), right above the Enter/Return key).

What `|` does is take the output from one command, e.g. the output from `grep` that went whizzing by and runs it through the command specified after it. When it was all whizzing by before, we wished we could just take a look at it! Maybe we could use `less` instead of the rapid scroll. Well, it turns out that we can! We can **pipe the output `grep` command** to `less` to slowly scroll through, or to `head` to just see the first few lines.

```
$ cd unix_lesson/raw_fastq

$ grep -B 1 -A 2 CAG S1WTgex_S1_L001_R1_001.fastq | less
```

Now we can use the arrows to scroll up and down and use `q` to get out.

Or we could just take a glance to see what the output looks like.

```
$ grep -B 1 -A 2 CAG S1WTgex_S1_L001_R1_001.fastq | head -n 5
```

Another thing we can also do is count the number of lines output by `grep`.

The `wc` command stands for ***w**ord **c**ount*. This command counts the number of lines, words and characters in the text input given to it. The `-l` argument will only count the number of lines instead of counting everything.

```
$ grep CAG S1WTgex_S1_L001_R1_001.fastq | wc -l
```

Try it out without the `-l` to see the full output.

> **Tip** - Similar to `grep`, you can type `wc --help` or `man wc` to see all options.

About Pipes:

- The pipe is a very important/powerful concept in Shell
- You can string along as many commands together as you like

The philosophy behind the three redirection operators (`>`, `>>`, `|`) you have learned so far is that none of them by themselves do a lot. BUT when you start chaining them together, you can do some really powerful things really efficiently.

To be able to use the shell effectively, becoming proficient in the use of the pipe and redirection operators

is essential.

Commands

```
cd          # change directory
ls          # list contents
man         # manual for a command
pwd         # check present working directory
tree        # prints a tree of the file structure
cp          # copy
mkdir       # make new directory
mv          # move or rename
rm          # remove/delete
ctrl + c    # cancel current command
ctrl + a    # start of line
ctrl + e    # end of line
history     # list previously used commands
cat         # print content of text file to screen
less        # maneuver through content of text file on screen
head        # see first lines of text file
tail        # see last lines of text file
grep        # search word in text files
>           # redirect standard output of command
>>          # redirect standard output of command
|           # pipe: chain several commands together
```

Shortcuts

```
~           # home directory
.           # current directory
..          # parent directory
*           # wildcard of any length
?           # wildcard of length 1
```