

Command Line Introduction

Devin Thomas¹
Jordan Ramsdell¹

February 8, 2017

¹UNH HCGS, supported by New Hampshire-INBRE through an Institutional Development Award (IDeA), P20GM103506, from the National Institute of General Medical Sciences of the NIH

Contents

1	Introduction	3
1.1	What is a computer?	3
1.2	Before we begin	4
2	Introduction to bash	5
2.1	Anatomy of a Command	5
2.1.1	Basics	5
2.1.2	using cal	6
2.1.3	Discussion Questions	6
2.2	Finding Help	7
2.2.1	Built in Documentation	7
2.2.2	Google-fu	7
2.2.3	Dealing With Errors	8
2.2.4	Discussion Questions	8
2.3	Pathways and Directories	8
2.3.1	Some definitions	9
2.3.2	An Allegory	9
2.3.3	Literally Paths	10
2.3.4	Discussion Questions	11
2.4	Diagnostic	11
3	Navigating the File system	12
3.1	Moving About	12
3.1.1	Discussion Questions	13
3.2	Move, Copy and Delete	13
3.2.1	Files and Directories	13
3.2.2	Copying	13
3.2.3	Moving	14
3.2.4	Removing	14
3.2.5	Discussion Questions	15
3.3	Diagnostic	15

4	Viewing and Manipulating Files	16
4.1	Odds and Ends	16
4.1.1	The control key	16
4.1.2	Special Characters	17
4.1.3	[h]top	17
4.1.4	Discussion Questions	18
4.2	Text Editors	18
4.2.1	nano	18
4.2.2	emacs	19
4.2.3	vi[m]	19
4.2.4	Discussion Questions	20
4.3	Viewing Text	20
4.3.1	Discussion Questions	20
4.4	grep	21
4.4.1	Getting Acquainted with grep	21
4.4.2	Bioinformatics with grep	21
4.4.3	Discussion Questions	22
4.5	Diagnostic	22
5	Programming in Bash	23
5.1	Pipes and Redirecting	23
5.1.1	Discussion Questions	24
5.2	Globbing and Wildcards	24
5.2.1	Discussion Questions	25
5.3	Variables	25
5.3.1	Discussion Questions	25
5.4	Configuration	26
5.4.1	Alias	26
5.4.2	Environmental Variables	26
5.4.3	Discussion Questions	26
5.5	File permissions	26
5.5.1	The (somewhat) Human Readable Method	27
5.5.2	The Less Verbose Way	27
5.5.3	Discussion Questions	28
5.6	Shell Scripts	28
5.6.1	Shebang!	29
5.6.2	Comments	29
5.6.3	Example	30
5.7	Diagnostic	30

Chapter 1

Introduction

The rapid drop in price of DNA sequencing has transformed genetics into a computationally heavy discipline in a very short time period. Computers are tools, that enable entirely new levels of analysis in genetics, but to do so you must understand what a computer is, and how it can be used to answer your questions. The original "computers" were teams of people performing calculations. Now we have much faster computers made out of silicon and copper instead of flesh.

1.1 What is a computer?

A computer is anything that computes. For our purposes lets start by looking at the simplest common setup you would expect from a modern computer. The CPU, or central processing unit is the brain of the computer, it follows instructions, performing operations and calculations. Modern CPUs will have several cores, which in effect means that they have several fully functional individual CPUs. This means that they can do several tasks at once.

The CPU would not be much use on its own though it needs somewhere to store the data as it crunches through it. There are two main types of storage, ram and hard drive storage. Ram, random access memory is so called because it allows quick access to random bits of information. In contrast traditional hard drives perform much worse when they are asked to look for random bits of data, instead of one long continuous chunk. Ram is much faster, and much more expensive than hard drives though. Recently SSDs or solid state drives have begin to take over from hard drives. They are not as fast as ram, but are much faster than hard drives, and like ram they don't lose much speed when randomly accessing small chunks of information.

On the software side there are three main components that make up the operating system. The kernel acts as the go between for the hardware and the rest of the software on the computer. It provides a common set of instructions so that you can create software without having to know the specific hardware on the computer.

1.2 Before we begin

You should have received a diagnostic to complete before you start this course! These diagnostics help us gauge how effective these course materials are, and will help inform the improvements we make to them!

Chapter 2

Introduction to bash

2.1 Anatomy of a Command

Explore the basic UNIX command syntax and see how different options and arguments can modify the output of programs.

2.1.1 Basics

Computers only do what they are told, Our goal is to learn the language of computers. You are welcome to try beeping at it, but we want to understand a more useful language, bash. In bash you will communicate through statements have the format

```
command [arguments]
```

Let's break this down, the required part for every statement is the command. This tells the computer what program you would like to run. Sometimes this is all it needs to know. Try running `cal`, `pwd` or `aquarium`.

Then we have arguments, which are put in square braces. The square braces signify that these are optional, not that you actually type square braces. For instance if my command is `cal`, and my argument is `-y` the statement you enter is `cal -y` not `cal [-y]` Sometimes there will be other arguments, which come after the command, these can tell the computer what to perform the action on, where to put the results or anything else the command might want or need to know.

Take a look at the classic first program, hello world! Which tells your computer to say hello world.

```
echo hello world!
```

The command we use is `echo`, which "echos back" whatever arguments it receives. If the computer understood English we could rephrase this as: you - "computer repeat after me, hello world!" computer - "hello world!"

How many arguments are there in this command? What separates them?? What does the command `echo hello world!` return?

What if we wanted `echo` to literally echo back hello world!? By default `echo` adds a newline to the end of whatever it echos. Let's see what happens if we tell `echo` not to do this. Options are a special type of argument, they modify the behavior of the command. Options come in two flavors, for `echo` we will use one of it's short single letter options "-n". "All" single letter options will be preceded by a single dash, in this case -n tells the `echo` command to not put on that trailing newline.

Take a look at how the -n option changes hello world.

```
echo -n hello world!
```

So what is a newline? Computers store files as one dimensional streams of data, so we need some way to signify how the text is meant to be displayed on a two dimensional screen. A newline is a character in this stream of data that tells your programs (`less`, `nano`, `cat` etc...) where to start a new line.

2.1.2 using cal

Try out the `cal` command.

```
cal
```

```
cal -y
```

```
cal -3
```

```
cal -j
```

Notice that multiple single character options can be combined.

```
cal -3j
```

2.1.3 Discussion Questions

- I In the command `cal -j 1999`:
- II What is the command?
- III What is the option?
- IV What is the argument?
- V What happens when conflicting options are given?

- VI Try asking for 3 (3 months) at the same time as y (one year).
- VII There is something strange in the calendar of 1752.
- VIII When is it and what is it? (use `cal` to look at the year first then use Wikipedia to investigate further)

2.2 Finding Help

There are two main ways to find help for working with bash. When you know the command you want to use, and just need a refresher on how it works the fastest way to find that out is usually the built in documentation `man`, `whatis`, `--help`. Any other situation you should strongly consider looking on the internet `google`, `stack exchange`, `forums`.... In all likelihood you are not the first person to want to something, and a quick search will usually provide an easily adaptable example to most situations.

2.2.1 Built in Documentation

The two commands we will look at are `whatis` and `man`. You can use the `whatis` command to find out what the `whatis` command does.

```
whatis whatis
```

Let's try using `whatis` to find out what the `ls` command does

```
whatis "ls"
```

Lets compare that with the `whatis` for the `sl` command, included to punish those prone to dyslexia.

That `whatis` is rather vague, so let's look at the `man` page instead.

```
man "sl"
```

Interesting, based off this `man` page make a small locomotive fly across your screen. Now that that is done use the tools at your disposal to figure out how to use `ls` to display verbose output of the files in your current directory, with human readable file sizes.

2.2.2 Google-fu

With programming our googling is probably going to be on the Zui Quan side of google-fu. The most efficient method is almost always just google a somewhat generic version of your question. For instance if I wanted to learn how to get bash to tell me

the time I would google "bash time", I would not google, "In bash I want to know the time, but I can't look at my phone so how could I do that?" or "bash clock". "bash clock" is not a great search because the instructions we get are how to get a clock, as in something that updates every second. This is way more complicated than just finding the time once.

In general just try and keep it simple, and if a wording does not give you an answer you can use try to reword your query. Look for answers on sites like stack exchange, and it is always a good idea to throw in the word Linux if you are getting swamped with windows specific answers.

2.2.3 Dealing With Errors

It should seem obvious, but if you get an error please actually read it often the error will state very clearly what the issue is. If that is not enough for you to diagnose it, the best solution is often to copy and paste a big chunk of the error into google. You should always try at least these few steps before asking for help, (please we really appreciate it).

2.2.4 Discussion Questions

- I Find the command to display your current directory.
- II Run the command `firefox`. What should this command do? What does the error mean?

2.3 Pathways and Directories

Modern file systems are hierarchical structures, of directories which can contain other directories or files. This is often referenced as a tree, because it expands from a single root directory. Paths are the most common arguments you will encounter, any time you need to specify a file or folder as input or output to a command you will do so by passing the command a path. The next set of commands we will learn about cover the basics of how to move yourself, and files and directories around. These are some of the most common tasks you will want to do and all of them take paths.

Let's explore this with an example path.

```
"/usr/bin/python" ## the python executable
```

Firstly the character "/" delimits directories. So where does this path take us? From the root "/" we go into the "usr" directory. The "usr" directory is where Linux systems store programs and data intended to be used by the user, (as opposed to the system). From there we move into the "/usr/bin" directory, which contains most

of the executable programs for the system. The full path points us to the python executable within the `"/usr/bin"` directory.

2.3.1 Some definitions

There are some terms related to paths which are very useful when referencing them. Root `"/"` this is the base folder of the file system, everything else is inside of it. (notice that root is just signified with the delimiter. On windows it is usually `"C:\\"`)

Working directory `"."` this is where you are right now in the file system.

Home `"~"` The directory you start a new terminal in. It's sub directories will usually contain all of your user files (`"Documents"`, `"Pictures"` etc. on a personal computer. Servers will usually be quite sparse by default.

`".."` Every directory has a sub directory `".."` which is a link to the directory above it. ie `"/usr/bin/.."` is the same as `"/usr/"` There are two main types of paths.

Absolute paths are constructed in reference to the root, and as such they start with `"/"`.

Relative paths are constructed in reference to your current working directory, they start with a sub directory or file of your current working directory.

2.3.2 An Allegory

Path can be a tricky concept, if you are having trouble visualizing them directly try out this example of a similar system.

You are an instructor for this course.

Your life consists solely of Gregg Hall.

There is nothing outside of Gregg.

You are currently in room 420.

Every morning you materialize in cubicle 9 in Gregg 430.

Let's represent Gregg as a file system. Directories are appropriate subsets of Gregg or other directories.

For example the home directory `"~"` would be `"/4th_floor/room_430/cube_9/"`

What is your current working directory?

What is the absolute path to the third floor bathroom? (just call it bath or something don't worry about the room number)

What is the relative path to the second floor bathroom?

What is the relative path to the pencil `in` the bottom of your backpack?

You are `in` the second floor hallway (just consider this the second floor directory)

What is your current working directory?

What is the relative path of Dr. Thomas's office (room 409)

2.3.3 Literally Paths

Lets look at some actual paths now.
Log in to the server.

What is the path to your home?

What is the path to your current directory?

For each of the following absolute file paths give the relative file path. And a brief description of what it contains. (google, ask your class mates, take a contextual guess)

`"/bin/"`

`"/boot/"`

`"/etc/"`

`"/mnt/"`

`"/etc/hosts"`

`"/var/cache"`

`"~/ .bashrc"`

2.3.4 Discussion Questions

- I You are in a folder called data which is in your home directory.
- II Give three examples of how to list the contents of your home directory.
- III Use one absolute path, one relative path, and one shorthand. (The argument for the shorthand should be a single character, while the simplest relative path is two characters.)

2.4 Diagnostic

After completing this section, please complete our short diagnostic (select "bash" when prompted "Which bioinformatics section did you just complete?"), it helps inform the further development of these course materials. Thank you!

Chapter 3

Navigating the File system

Congratulations! You've learned the basic structure of bash, now that that is covered it would be nice to start doing something. When working in the terminal you have a location, called your current working directory. Files in the same location as you have relative paths which are just their filenames, which is pretty handy. In this chapter we are going to learn how you can move yourself, as well as files and directories around.

3.1 Moving About

The `cd` command changes your current working directory.

Look at the [whatis](#) for `cd`.

`cd` by itself with no arguments will change you to your home directory `"~"`.

`cd` - acts like the back arrow on your web browser, returning you to your previous working directory.

Otherwise use `cd path` to change your current working directory to the directory specified by the path.

Try out these examples, if you want help figuring out where you are remember to use the `pwd` command.

```
".."
```

```
"."
```

```
"~"
```

```
".././"
```

```
"/."
```

The last example shows why you need to be careful with the difference between `"folder"` and `"/folder"`

3.1.1 Discussion Questions

Show two examples of how, from anywhere in your file system you can change directories to the "example_reads" folder.

- I The first should be one step using the absolute path.
- II The second should be two steps, first changing to the home directory and then second changing to "example_reads".

3.2 Move, Copy and Delete

Moving yourself around is all fine and dandy, but you probably also want to be able to manipulate files and directories.

There are three basic operations we want to cover. Moving `mv` changes the path of the file or directory. Copying `cp` duplicates the file or directory tree at a new path. Deleting `rm` removes the file or directory tree at the path.

3.2.1 Files and Directories

A common dogma in Linux is "everything is a file". If you plug in an external hard drive Linux mounts it as a file. All the sensors, microprocessors and devices that make up the server have their inputs and outputs mounted as files. This provides a generic way to communicate with the computer hardware, without you having to know what it actually is (mostly). You can create a file using the `touch` command, try it out! Touch is fairly useless, most of the time you create a file you will want to put something in it, but it is there if you ever need an empty file for some reason.

Directories can hold files and other directories, unlike files you will often find yourself wanting to create an empty directory, do do that use the `mkdir` command.

Some early file systems, such as early floppy disk file systems and the original Macintosh file system were flat, and could not hold directories within other directories. This is no longer the case, and one of the most useful commands to know is how to create a new directory.

Look at the documentation for `mkdir`, generally the default behavior is ideal, however the `-p` option can be useful. Try making an empty directory, and then moving into it and making an empty file.

3.2.2 Copying

The `cp` command can be used to copy files and directories. It takes paths as its arguments. The most common option to use is the `-r` flag, which controls recursion. `cp -r` lets you copy directories and their contents recursively, as in it copies the directory, and all of its contents, and all of its contents contents...

Move to your `"example_reads"` folder.

We are going to be fiddling with the contents of this folder, so it would be a good idea to have a backup. create this using the command:

```
cp -r ../example_reads ../example_reads_backup
```

We will go through `cp` in more detail in a moment.

Look at the contents. It makes it awfully difficult having two files instead of just one.

make two new folders, `"forward"` and `"reverse"`.

use `ls` to see what you have created.

Next we want to move the two fastq files into these shiny new folders.

start by creating a copy the forward file (R1) in the `"forward"` directory.

If you don't remember the syntax or order of arguments for `cp` look at the `man` page.

3.2.3 Moving

You can use the `mv` command to move a file. `mv` is mechanically very similar to `cp`, you give it a source path and destination path. The main difference is that `mv` is just changing the path to access the file or directory, it does not actually make a new copy of the files on the hard drive. This means that you can move giant files very quickly, provided you are not moving them between different physical disks. This also means that `mv` does not need a recursive flag.

3.2.4 Removing

It is nice not to delete things, but your file system will be a mess if you only use `cp`. So let's learn how to clean up the mess. The first tool is `rm`. `rm` can easily wipe out huge chunks of your file system if you mess up using it, so always be careful using it. Look at the `man` page for `rm` and then use it to remove the forward reads file that is left in the `"example_reads"` folder. (But leave the copy in the `"forward"` folder!).

It would be a bit silly to have to use two commands every time you wanted to move a file around. The `mv` command effectively combines the `cp` and `rm` steps we just performed. Look at the documentation then use it to move the reverse reads into the reverse folder.

Now with a bit of thought it should be apparent that giving each of these files there own folder was pointless. It hopefully has helped you practice these commands. Practice them some more by returning the files to the way you found them. You can do this several ways. Either move the files back that you moved earlier, then remove the folders. Or use the backup you made to restore the original file structure.

When you are done remove the backup.

3.2.5 Discussion Questions

I Which of these commands should you never ever execute? (don't test them please, use the documentation to figure out what each does) (These are commented out because they are all terrible to run, focus on thinking about what they would do)

```
# rm -rf /*
```

```
# rm -rf --no-preserve-root /
```

```
# mv -r ~ /dev/null
```

II Why is there not a `rename` command?

III How can you restore the backup you created earlier?

3.3 Diagnostic

After completing this section, please complete our short diagnostic (select "Navigating the File System" when prompted "Which bioinformatics section did you just complete?"), it helps inform the further development of these course materials. Thank you!

Chapter 4

Viewing and Manipulating Files

Now that we know how to move around different parts of our file system let's learn how to work with text files in bash. There are two general categories of files, text files and binary files. Text files have information encoded in a familiar text encoding, like ASCII or UTF-8 on a modern system. This means you can open these files up, read what's in them and edit them with your keyboard with no issues. Our Linux dogma from before could just as easily have been "everything is a text file" because unlike windows pretty much all of the configuration is done through editing plain text files. Binary files are any other files, they exist because they are more efficient, in space and speed when human readability is not a concern (or not more important than speed and size!).

We are going to concentrate on text files.

4.1 Odds and Ends

But first lets take a 90° turn and talk about some random stuff you may find useful.

4.1.1 The control key

There are lots of handy little hotkeys, check out [this article on howtogeek](#) for a longer list of keyboard shortcuts but a few crucial ones are:

1. ctrl-c kill currently running process
2. ctrl-d log out of shell
3. ctrl-z send your current process to the background (make sure you look up how to bring it back up front!)
4. tab makes you good at typing (auto complete)

4.1.2 Special Characters

Special characters in `bash` are characters with a meta-meaning. For instance a filename beginning with `'.'` is treated as a hidden file by `ls`. There is however a way to tell `bash` to ignore special characters.

To have `bash` ignore all special characters, wrap your statement in single quotes. 'like this' double quotes work similarly, but do not ignore a few special characters. "In most cases they are interchangeable."

Also note that `'` is different from `'`, sometimes a program will not run, or be throwing up a strange error and occasionally making sure that the correct single quote is used can fix the issue.

Now let's take a look at some hidden files, and why they are useful.

Go to your home directory and run an `ls -la` (or just `-a`).

You should see several hidden files, with names beginning with `."`

The first few you should be already familiar with `."` and `.."`

`bash` is the name of the interpreter we are using, you will see several files named `."bash**"`

`."bash_history"` stores the history of the commands that you have entered previously.

It is the source for the `history` command. `."bashrc"` is the configuration file for `bash`, we will dig through this later.

4.1.3 [h]top

Let's take a look at `top`, run `top` with the `-u "$USER"` argument to see just your own processes. It should hopefully be pretty boring, likely `top` will be the most intensive program you are running. For all of the calls to `top` in this section feel free to substitute `htop`, which is a prettier more visual version of `top`. If you are on another server or Linux computer remember that `htop` is probably not installed by default so you may have to revert to `top`.

`top` exists to let you see what the computer is working on, and how hard it is working. My most common usage for it is keeping an eye on what everyone else is running on `ron`, but your most common use will probably be running `top -u $USER` to look at what processes you are running.

This can be handy when you are running something and want to see if it is done yet. Or let's say some program is taking a weirdly long time to run, you can use `top` to see if it is behaving weirdly (using almost no `cpu` when you would expect it to be churning away) or maybe you see that 15 of your classmates are running assemblies and you're having to share resources with all of them!

You can also use `top` to kill a process, maybe one that has frozen or that you put in the background. Let's look at an example:

`top` is taking an awfully lot of your resources, and is very dangerous so we must deal with it. (Don't worry `top` is not really dangerous, if you are doing this exercise it is

definitely running though which makes this easier to plan for me). This can be useful when you ran something in the background, and need to get rid of it.

While in `top` press 'k' to open up the kill interface. Find the `top` command in the list, and read off its PID (process ID). Enter this ID into the kill prompt. Then press enter twice, which should kill `top` and return you to your prompt.

4.1.4 Discussion Questions

Much like this entire section, the discussion will have little relation to everything around it. The root of the file system, "/" is the trunk of the proverbial tree, what happens if you try to `cd` into the roots? Predict what will happen if you ran the following commands:

```
ls "/"
ls "../"
```

I What do you think will happen? Why?

II Now run it, what actually happened? Make a short argument for why this is or is not a good behavior.

4.2 Text Editors

One of the most common tasks you will have on a server is editing text files. To do this effectively over a remote connection you want a command line editor. There are several options, and we will take a look at a few of them here.

4.2.1 nano

Lets start by creating a new file called "`nano_test.txt`".

Do this, while opening `nano` with:

```
nano "nano_test.txt"
```

Enter some text and see how it responds, enter and tab do what you would expect.

If you use programming languages like `python`, you may want to edit your "`.nanorc`" to replace tabs with 4 spaces. Let's do this.

hit ctrl-o to save the file, hit enter when it asks you about the name.

now hit ctrl-x to leave `nano`

Let's use `nano` to create the "`.nanorc`" file, don't forget the period!

in "`.nanorc`" enter two lines,

```
set tabsize 4
set tabstospaces
```

The first makes tabs 4 spaces wide, and the second makes tabs out of spaces instead of tabs.

Save this then open `nano` again on your test file. Check out how tabs behave now. Feel free to change the number of spaces to your preferences. Different software, and software projects use different numbers of spaces for tabs. `python` usually should be, 4 space tabs, while the Linux kernel is coded with 8 space tabs!

You can also use the `".nanorc"` to add syntax highlighting and other tweaks to `nano`.

4.2.2 emacs

`emacs` is a gnu (gnus not unix) text editor, it has command line and graphical versions and is very powerful. For now you should probably stick with `nano`, until you have time and the desire to sit down and learn `emacs` or `vim`. They will enable you to preform more powerful tasks, but at the cost of a steeper learning curve and less universality.

Open `emacs` the same way you ran `nano`,

```
emacs "emacs_text.txt"
```

You should see something somewhat similar to what you saw with `nano`, notice that unlike `nano` the control options are not displayed on the bottom of your screen, which makes it slightly less beginner friendly. For now lets concentrate on how you can exit `emacs`. You may have noticed that `q` and `ctrl-c` don't work for that.

To save your file press `ctrl-x`, then `ctrl-s`. to exit press `ctrl-x` then `ctrl-c`.

4.2.3 vi[m]

`vi`, or the improved version `vim` have a very different set of controls than what you are likely used to. Like `emacs`, `vim` is very powerful. Unfortunately unlike `emacs` it tends to be the default editor for a lot of configuration programs, it is not uncommon for you to find yourself in a `vi` interface needing to know how to navigate and leave. Just like above run

```
vim "vim_test.txt"
```

Play around a bit, you should be able to type text and navigate around with your arrow keys as you would expect. When you want to quit enter `:q` then hit enter. If you have entered any text, you will probably need need to change out of insert mode, do this by hitting `esc` then `:q`. If you are having issues, make sure you read the instructions `vim` gives you!

4.2.4 Discussion Questions

Someday you will probably run into something which returns an error and a line number. For example compiling this document returns a warning on line 42. For these it is handy to be able to know what line you are on in `nano`.

- I What is the command line option for this? (you want to know your cursor position)
- II If you forget or choose not to run `nano` with this option there is a hotkey to display your current cursor position, what is it?
- III Another handy thing to know is how to undo, what is the key combination for this?

(alternatively feel free to answer these questions in `emacs` or `vim`.)

4.3 Viewing Text

Editors like `nano` can be very useful, but sometimes you just want to quickly read through a file, or the output of a program. For that tools like `less` can be more useful than using a full editor like `nano`.

Let's begin at the beginning, the `head` command displays the first lines of a file. Look at the `head` of one of the example files you made with the text editors.

Now look at the end, using `tail`. In the man page for `tail` and `head` you can find out how to select how many lines are displayed.

To read through an entire file use `less`. Open one of the fastq files from earlier with `less`. You will notice that the text is wrapping because it is wider than your terminal. We don't want this so use `q` to exit `less`.

Look in the `man` page for `less` to find the option to disable line wrapping. As you are scrolling by notice all the options for scrolling through text. The main two are `f` to move down the file, and `b` to move up.

Notice that the fastq files are actually compressed. If you try and open them with `head` or `tail` you will get gibberish. A benefit of `less` is that it automatically decompresses compressed files. `head` and `tail` do not. On longer files `less` can be helpful because you can navigate through the text by percent. To go 44 percent through the document type `44%`. Give this a try to navigate around the fastq file.

4.3.1 Discussion Questions

The `man` page for `less` makes several arguments for why it should be used over its predecessor `more` and `vi` when viewing text files.

- I What is the main advantage claimed by `less` over each of these?

- II Navigate to your example assembly and view the contigs.fasta file.
- III navigate to 44.7% through the document. What is the length of the contig you are in the middle of? (if you have your terminal taller than normal or full screen keep in mind that I am referring to the location of the first line you are now viewing.)

4.4 grep

`grep` has a wide range of uses, almost any time you want to know if and how to find _____ in _____ the answer is yes you can with `grep`.

4.4.1 Getting Acquainted with grep

Let's start by grepping through your command history. From your home directory run:

```
grep "ls" ".bash_history"
```

`grep` works by returning any line that contains the expression you give it to search for.

Try grepping through your history for other commands like `echo`.

What if you wanted to know how many times you had used `ls`?

Use the `-c` option to have `grep` count how many lines contain `ls`.

Another handy option is to ignore case, i.e. have cheese match with CHeeSe.

`grep` has a big `man` page, so lets use `grep` to `grep` through `grep`'s `man` page!

We are going to use a pipe to have the output of `man` be the input to `grep`.

```
man "grep" | grep "case"
```

The `|` is the pipe, we will go through these in detail later a similar usage is grepping through `history` using the `history` command.

```
history | grep "sl"
```

4.4.2 Bioinformatics with grep

`grep`, like `head` and `tail` earlier does not work by default with compressed files. To `grep` through a compressed file instead use `zgrep`.

Let's use `grep` to check out our example `"contigs.fasta"`. The first thing to do is pull out the header for each contig. Use `head` (or something else) too look at the contigs file and find a pattern unique to headers, but common to them all. (Hint it is a special character, don't forget to wrap it in single quotes)!

Once you have done that you should get a huge list of contigs. Take those and count how many there are.

How many are length 433? Make sure your command is unambiguous! You should not need ">" anymore, but you do need to make sure the length is 433, just `grep` for 433 will pull out a bunch of stuff in nodes 4330 2433 etc...

Lets look at the annotation, go into the prokka report and take a look at the .faa file. This has the amino acid sequences of the CDSs prokka found, and their annotations. When prokka finds a cds it can not annotate it calls it hypothetical, how many hypothetical proteins are there?

How can you get all of the headers so you can see just the annotations without all the sequence?

How could you take that and get rid of all the hypothetical proteins?

4.4.3 Discussion Questions

Locate and change into your example assembly directory. There should be a "prokka_report" directory located within it. Change into this directory.

I `grep "PROKKA_07182016.gff"` for oxidase

II Write down one of the products that `grep` returns.

III How could you count the number of lines that `grep` returns?

IV What would the count be for `grep`ing "PROKKA_07182016.gff" for oxidase?

V In your own words write a short description of what the `grep` options `-i`, `-o` and `-A` do.

VI Would the `-i` option be useful when looking for oxidase in our gene annotations? Why or why not?

4.5 Diagnostic

After completing this section, please complete our short diagnostic (select "Viewing and Manipulating Files" when prompted "Which bioinformatics section did you just complete?"), it helps inform the further development of these course materials. Thank you!

Chapter 5

Programming in Bash

5.1 Pipes and Redirecting

A central philosophy of Unix like environments, like we are working with is simple modular tools which do one thing well. The enabling development for this was piping. Pipes take the output from one command and pipe it to the input of another. For example let's take the output of one of the `grep`s we did in the last section, and pipe it to `less` to make it easier to read.

```
grep ">" "contigs.fasta" | less ## Don't forget to quote >
```

The `|` is the pipe, and the program flows from left to right.

Let's do a more involved example, we want to make a histogram of the length of our contigs. We are going to build up a pipe line by line to process the data required to make this.

First start by pulling out the rows with the length information. These are the contig headers we have already done.

Next we need to pull out just the length column. To do this let's use a tool called `awk`. Look at the `awk man` page, particularly the `-F` option.

What separates the columns? We want to print the contents of the 4th column, do this with:

```
awk -F "_" '{print $4}'
```

Let's pause and think about what each part of this command does.

What would you change if the columns were delimited by dashes instead?

How about if we wanted the second column?

What if we wanted columns 4 and 3, in that order?

There is another type of pipe, sometimes instead of outputting to `stdout`, you want your programs to output to a file. To do this use one of `>>` or `>`. `>` works just like `|` except you pipe to a file, where the output is saved. `>` overwrites the file, while `>>`

appends to the file.

Lets put this all together now, pipe together the commands to save a file "lengths.hist" with the lengths of all of your contigs.

This file can now be used with `python` or your favorite histogramming tool to see how the lengths of your contigs are distributed.

5.1.1 Discussion Questions

Locate your "example_assembly" directory.

- I What command can be used to print the first ten lines of a file?
- II What about the first 20?
- III How could we combine two commands to print the first ten lines of a file, reversed?
- IV How can we save the output of that pipe we just made to a file instead of having it output to `stdout`?
- V How many contigs are in "contigs.fasta"?
- VI What `grep` command can you use to find that?
- VII What command can you use by combining a `grep` and a `wc` to find the same thing?

5.2 Globbing and Wildcards

Quite often you will want to use multiple files at the same time. You can do this by 'globbing'.

Let's start simple, most of the commands we have talked about will take globbed paths as well as the single files we have been giving them.

For instance lets use `ls`, globbing and wildcards to look at just our fasta file. Go to the example assembly folder and run:

```
ls -lh *.fasta
```

Lets look at the logs instead

There are two, lets concatenate them together and take a look.

Pipe `cat` into `less` and take a look.

How can you modify the last command to look at the two log files instead?

The `*` is an example of a wildcard, you are globbing the pattern you construct out of characters and wildcards. `*` matches to 0 or more of any characters, and is the most commonly used wildcard.

You can google 'bash wildcards' to see a full list, a lot of the special characters we have been avoiding are actually wildcards.

5.2.1 Discussion Questions

Locate and change into the "example_assembly" directory. Within this directory navigate into the "quast_report" directory.

- I Use `ls`, or `echo` and some globbing to return the files with the extension `.txt`, What command did you use?
- II How would you print all the files with the `.html` or `.txt` extensions?
- III What pattern would we use to glob everything in this directory with the word `report` in its filename?

5.3 Variables

Variables in `bash` are untyped, there is no distinction between numbers, strings etc. We have already encountered a few variables. `$USER` is your user name, `$HOME` is your home directory. The `$` indicates that we want to reference the value of the variable, instead of the name. `echo HOME` and `echo $HOME` should produce different results. In general `bash` variables are all capitalized, though this is not a hard requirement.

Let's get a bit of practice with variables.

Start by declaring two variables, cantaloupes and melons. There are 20 cantaloupes and 5 melons.

```
CANTALOUPE=20
MELONS=5
```

Someone comes by and reminds you that cantaloupes are actually a type of melon, so you need to correct the melon count!

```
MELONS=$(echo "$MELONS + $CANTALOUPE" | bc) # Correct the melon count!
```

Check that `MELONS` is now 25.

Let's walk through that last command. We used `bc`, which is an arbitrary precision calculator to add the number of cantaloupes to the number of melons. We wrapped the math in parenthesis and a `$` so that `MELONS` would be set to the value of that expression.

5.3.1 Discussion Questions

- I What does `MYVAR="hello"` do?

- II Once you've done that what does `echo $MYVAR` return?
- III What is the `$` used for? (What does its inclusion in the last command do?)

5.4 Configuration

Now that you know what a variable is we can look into configuring bash. The easiest configuration to do is setting things like font, color palette, background color etc for your terminal. To do that look in the menus of the terminal emulator you are using on your own computer, either PuTTY or terminal. For bash itself our main tools for that are adding commands to the `.bashrc` or `.profile` configuration files. These files are run each when you log in, so settings you put in them will be set every time you use bash. We have two main tools, you can use the `alias` command to make a shorthand for an other command, and you can set environmental variables.

5.4.1 Alias

The `alias` command lets you define shorthand versions of other commands. for instance look in your `.bashrc` for where we have defined `mytop`. We want `mytop` to run `htop` on your own user, so we used the command:

```
alias mytop='htop -u $USER'
```

5.4.2 Environmental Variables

`PATH` is the list of locations where bash will look for commands you ask it to execute. Take a look at what your path is right now. If you want to add a new location where you can put files, you can use `export PATH="new_folder:$PATH"`

5.4.3 Discussion Questions

- I What are some neat and/or useful aliases you can think of?
- II Why are there many different folders for programs?

5.5 File permissions

Let's take a look at file permissions. We will be using the `chmod` command. There are two ways of specifying the permissions you want to change let's explore both.

5.5.1 The (somewhat) Human Readable Method

This is the method Jordan mentioned but did not demonstrate. Look at `chmod`'s `man` page to see its basic usage, we need to give it a mode, and the file to alter. This first syntax is called the symbolic mode, and it has three parts, first we need to specify the users or groups we are changing the permissions for. Next we need to say if we are adding or taking away permissions. Then thirdly which of the permissions we are changing.

so it will look like this:

```
chmod [ugoa][+-][rwx] file
```

If you looked in the `man` page you may have noticed there are more modes, these are the basic ones you will want most of the time.

u corresponds to the user that owns it (the first tuple of permissions)

g corresponds to the file's group (the second tuple)

o corresponds to users not in the file's group (the third group)

a corresponds to everyone

The plus adds permissions, and minus takes them away

r is read

w is write

x is execute

These three can be combined, for instance `+rw` would add both read and write

So what would this look like in practice?

A common task is to make a script you have made executable by anyone trying to run it. To do this you would use the command

```
chmod a+x "script.sh"
```

What if you only want yourself to be able to execute it?

What if you already made it executable for everyone but you want to take away execution rights from everyone but yourself?

What if you realize that you accidental made a super dangerous script and want to stop everyone from being able to read write or execute it?

5.5.2 The Less Verbose Way

You may have noticed that each setting is 3 sets of three binary switches, as Jordan mentioned in the video this can be represented in a shorthand of sets of three numbers from 0-8.5.1

Where a 0 is no permissions, and a 1 is permitted. So for read and write we would use a 6, while for read write and execute we would want a 7. This notation can be more useful when using a script to change a large number of files.

Table 5.1: 3 bit binary table

Number	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

For example say we have a bunch of files that got transferred from your mac, somehow the permissions are all messed up and you need to change everything back to rw for you and r for everyone else. They are all in the same folder so you would use:

```
chmod 744 *
```

What would you use to make everything rwx? (If you have directories that have messed up file permissions this is sometimes the fix you need to do.)

5.5.3 Discussion Questions

Run `ls -l ~`

- I What does the first column refer to?
- II What does the third column refer to?
- III What does the fourth column refer to?
- IV What command would we use to change the permissions of a file?
- V Do you need to be the owner of a file to change its permissions? (on the server)

5.6 Shell Scripts

Scripting is a crucial step in ensuring repeatability in your results. You should make a habit of running everything you do when analyzing your samples in scripts, this enables to look back and see exactly what you did, or if done well swap out new data and perform exactly the same analysis.

5.6.1 Shebang!

Every script should start with a shebang for a `bash` script there are a few options, one of which is:

```
#!/bin/bash
```

This tells the shell to run this script using the `bash` interpreter. This may seem redundant when your shell is a `bash` interpreter, however if you instead used `#!/bin/python` it would be run with python, or `#!/bin/emacs` would run it with emacs.

5.6.2 Comments

Comments are the most important part of your script, they tell future you, and anyone else reading your script what you were trying to accomplish, and hopefully how you achieved that.

To make a comment simply include a `#`, anything after it in a line will be ignored by `bash`.

There are many different schemes for commenting, eventually you will work out your own, or be working with a group with a laid out comment style.

In general it is always a good idea to start with a line explaining what your script is for. Lets make an example script that says hi!

```
#!/bin/bash
## hi.sh, says hi!
echo "hi!"
```

This is probably good for this case, but what about if we have inputs or outputs? It is good to know what to expect.

Lets alter our script to take a `contigs.fasta` and make it into a 'friendly' `contigs.fasta` by appending hi to the start.

When doing this we want to symbolize the inputs and outputs in a comment. A good scheme for this is to just list the inputs then the outputs.

```
## input1 input2... -> output1 output2...
```

```
#!/bin/bash
## happylittlecontig.sh, appends hi! to the beginning of a contigs file
## contigs.fasta -> friendly_contigs.fasta
INPUT=$1
echo "hi!" | cat - "$INPUT" # append hi! to contigs and return to stdout
```

`1` is a special variable in scripts, it represents the first argument given to the script, so in this case we want the name of the contigs file.

Notice that I also explained what the now more complicated line does. This is always a good idea when it is not clear at a glance what a line does (`echo hi!` is pretty clear)

5.6.3 Example

Let's try making a script that can be run in the `quast` report folder of an assembly (use `"example_assembly"`)

We want this script to sort the files by extension into two folders, one of extensions you know the meaning of and one of extensions you don't.

Don't forget to comment and shebang!

First you need to make two folders, one for understood one for not understood.

Next use the `mv` command and `*.extension` to move the files by their extension

Once you have created this script let's try running it!

There are a few options, firstly we can run it by calling `bash` on it:

```
bash "myscript.sh"
```

but the better way to do it is to change the file permissions to allow execution, then just run it

```
./myscript.sh
```

Note that you need the `./` because `bash` doesn't know to look in your current directory for executable files because it is not in your path.

Good job! Now make a second script that undoes what your first script did, we may want to use this `quast` report again later!

5.7 Diagnostic

After completing this section, please complete our short diagnostic (select "Programming in Bash" when prompted "Which bioinformatics section did you just complete?"), it helps inform the further development of these course materials. Thank you!