

THE MISSING GUIDE

The **BIOSTAR HANDBOOK**

BIOINFORMATICS DATA ANALYSIS GUIDE

RNA-SEQ

ANALYSIS

GENE

EXPRESSION

NGS

SEQUENCING

BLAST

SEARCH

GENOME

ANALYSIS

Book updated on December 14, 2020

Part III

UNIX COMMAND LINE

Chapter 15

Introduction to Unix

The terms “**Unix**” and “**Command Line**” are used interchangeably both in this Handbook and in the real world. Technically speaking, Unix is a class of operating system that originally used the command line as the primary mode of interaction with the user. As computer technologies have progressed, the two terms have come to refer to the same thing.

15.1 What is the command line?

People interact with computers via so-called **user interfaces**. Clicking an icon, scrolling with the mouse, and entering text into a window are all user interfaces by which we instruct the computer to perform a specific task.

When it comes to data analysis, one of the most “ancient” of interfaces—the text interface—is still the most powerful way of passing instructions to the computer.

In a nutshell, instead of clicking and pressing buttons you will be entering *words* and *symbols* that will act as direct commands and will instruct the computer to perform various processes.

15.2 What does the command line look like?

For the uninitiated command line instructions may look like a magical incantation:

```
cat sraids.txt | parallel fastq-dump -O sra --split-files {}
```

In a way it is modern magic - with just a few words we can unleash complex computations that would be impossible to perform in any other way. With the command line, we can slay the dragons and still be home by six.

As for the above – and don’t worry if you can’t follow this yet – it is a way to download all the published sequencing data that corresponds to the ids stored in the `sraids.txt` file. The command as listed above invokes three other commands: a funny one named `cat`, a more logical sounding one `parallel` and a mysterious `fastq-dump`.

This series of commands will read the content of the `sraids.txt` and will feed that line by line into the `parallel` program that in turn will launch as many `fastq-dump` programs as there are CPU cores on the machine. This parallel execution typically speeds up processing a great deal as most computers can execute more than one process at a time.

15.3 What are the advantages of the command line?

The command line presents a word by word representation of a series of actions that are explicit, shareable, repeatable and automatable. The ability to express actions with words allows us to organize these actions in the most appropriate way to solve every problem.

15.4 What are the disadvantages of the command line?

Running command line tools requires additional training and a more indepth understanding of how computers operate. Do note that this knowledge is quite valuable beyond the domain of bioinformatics.

15.5 Is knowing the command line necessary?

Yes. While it is entirely possible to perform many bioinformatics analyses without running a command line tool most studies will benefit immensely from using a Unix like environment. Most importantly understanding Unix develops a way of thinking that is well suited to solving bioinformatics problems.

15.6 Is the command line hard to learn?

That is not the right way to think about it.

Understand that learning the command line (and programming in general) is not just a matter of learning the concepts. Instead, it is primarily about learning to think a certain way - to decompose a problem into very simple steps each of which can be easily solved. Some people have an affinity to this – they will pick it up faster, while others will need more practice. In our observation, prior preparation/experience does not affect the speed at which people can learn the command line.

At the same time, you have to realize that you can only succeed at changing how you think when you perform these actions. Besides it also means that you can't just learn it all in a day or even a week. So keep at it for a little while longer building your skills. In the end, it is only a matter of time and practice.

Try to actively make use of command line tools by integrating them into your daily work. Making use of command line tools is usually quite easy to do since Unix tools are useful for just about any type of data analysis.

15.7 Do other people also make many mistakes?

When you start out using the command line you will continuously make errors. That can be very frustrating. You could ask yourself, “Do other people make lots of mistakes too?”

Yes, we all do. We correct these errors faster.

Your skill should be measured not just in making fewer mistakes but primarily in how quickly you fix the errors that you do make. So get on with making mistakes, that's how you know you are on the right path.

15.8 How much Unix do I need to know to be able to progress?

A beginner level Unix expertise is sufficient to get started and to perform the majority of analyses that we cover in the book.

You should have a firm grasp of the following:

1. Directory navigation: what the directory tree is, how to navigate and move around with `cd`
2. Absolute and relative paths: how to access files located in directories
3. What simple Unix commands do: `ls`, `mv`, `rm`, `mkdir`, `cat`, `man`
4. Getting help: how to find out more on what a unix command does
5. What are “flags”: how to customize typical unix programs `ls` vs `ls -l`
6. Shell redirection: what is the standard input and output, how to “pipe” or redirect the output of one program into the input of the other

15.9 How do I access the command line?

When using graphical user interfaces the command line shell is accessed via an application called “Terminal”. When the “Terminal” runs it launches a so-called “shell” that by default opens into a folder that is called your “home” directory.

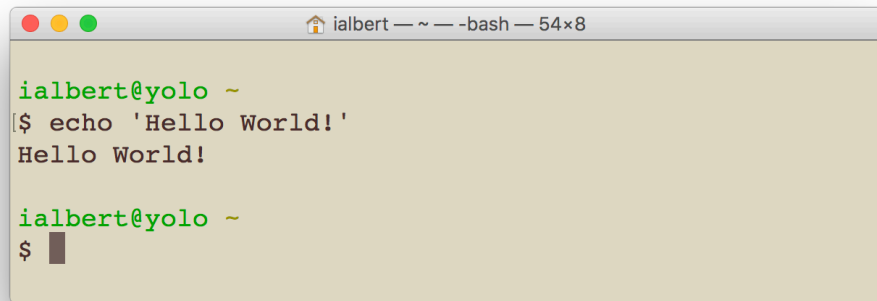
15.10 What is a shell?

The shell is an interface that interprets and executes the commands that entered into it via words. There are different types of shells, the most commonly used shell is called `bash`. Below we run the `echo` command in `bash`

```
echo 'Hello World!'
```

and we see it executed in a Mac OSX Terminal below. The default terminal

that opens up for you will probably look a bit different. There are options to change the fonts, background colors etc. Besides, our so-called command prompt (the line where you enter the commands), is probably simpler than yours. See the Setup profile for details on how to change that.



```
ialbert@yolo ~  
[ $ echo 'Hello World!' ]  
Hello World!  
  
ialbert@yolo ~  
$
```

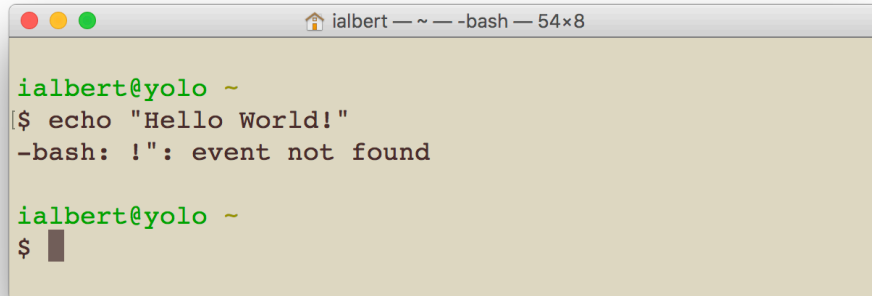
It is essential to recognize that the shell is not just a way to launch commands but a **full programming language** that provides many powerful constructs to streamline its use. Alas, initially these powers tend to get in the way and cause some frustrations. Suppose that instead of

```
echo 'Hello World!'
```

you've enclosed the text with double quotes:

```
echo "Hello World!"
```

now instead of echoing the output, it will produce:

A terminal window titled 'ialbert ~ -bash - 54x8'. The prompt is 'ialbert@yolo ~'. The user enters '\$ echo "Hello World!"'. The terminal outputs '-bash: !": event not found'. The prompt returns to 'ialbert@yolo ~' and the user enters '\$', followed by a cursor.

```
ialbert@yolo ~  
[$ echo "Hello World!"  
-bash: !": event not found  
  
ialbert@yolo ~  
$
```

Maddeningly enough the error occurs only if the string contains the `!` character as the last element. For example, this works just fine.

```
echo "Hello World"
```

Understandably getting mysterious errors when writing seemingly simple text can be quite confusing. The error occurred because in `bash` the `!` character allows you to rerun a previous command. For example

```
!e
```

Will rerun the last command that starts with `e`. In this case the first `echo 'Hello World!'`. The feature was added as help to avoid retyping long commands. Tip: try `!!` see what happens.

Then, as it turns out in `bash` text enclosed within double quotes is interpreted differently than text enclosed in single quotes. Single quoted content is passed down exactly as written, whereas in double-quoted text some letters that have may special meaning like: `!`, `$` (and others) will be interpreted as bash commands. Initially this gets in the way, as you don't know which letter is "special". Later this feature allows you to build more powerful commands.

Note: As a rule, in bash type single quotes if you want the text the stay the same, and type double quotes when the text needs to be modified later on.

Gotcha's like these are numerous. Everyone gets tripped up at some time. Google is your friend. Bash has been around for a

long time (decades) and did not change much. Every problem you run into has most likely been experienced by others as well, and will have solutions written out in great detail.

15.11 Wait there is more

As it turns out sometimes

```
echo "Hello World!"
```

will print

```
Hello World!
```

as expected. If you were to put the command into a file (script):

```
# Check out the mix of single and double quotes here.
echo 'echo "Hello World!"' > script.sh
```

now if you were to run it like so:

```
bash script.sh
```

It works without raising an error. But why?

You see, the “interactive” bash terminal tries to assist you in minimize typing. A feature called “history substitution” is turned on when you type into bash, but it is turned off when you run a script via bash. Doing so allows you to use the event shortcuts when you are typing, but you don’t need to be use these in a script.

You can turn off the history substitution from the command line yourself. And on some systems this is done automatically in the bash configuration file. Typing:

```
set +H
```

will turn off history substitution. It is a bit odd that we need to use + sign to turn a feature off, and a - to turn the feature back on. But we’ll live with that.

15.12 What is the best way to learn Unix?

We offer a number of tutorials right here in this book.

In addition there are excellent online tutorials, some free others at cost, each with a different take and philosophy on how to teach the concepts. We recommend that you try out a few and stick with the one that seems to work the best for you.

- CodeAcademy: Learn the command line¹
- Software Carpentry: The Unix shell²
- Command line bootcamp³
- Unix and Perl Primer for Biologists⁴
- Learn Enough Command Line to Be Dangerous⁵
- The Command Line Crash Course⁶
- Learn Bash in Y minutes⁷

And there are many other options.

15.13 How do I troubleshoot errors?

If you don't know how to do or fix something try Googling it. Put the relevant section of the error message into your search query. You will find that other people will have had the same problem and someone will have told them what the answer was. Google will likely direct you to one of these sites

- Ask Ubuntu⁸
- StackOverflow⁹

¹<https://www.codecademy.com/learn/learn-the-command-line>

²<http://swcarpentry.github.io/shell-novice/>

³<http://korflab.ucdavis.edu/bootcamp.html>

⁴http://korflab.ucdavis.edu/Unix_and_Perl/current.html

⁵<https://www.learnenough.com/command-line-tutorial>

⁶<http://cli.learncodethehardway.org/book/>

⁷<https://learnxinyminutes.com/docs/bash/>

⁸<http://askubuntu.com/>

⁹<http://stackoverflow.com/>

15.14 Where can I learn more about the shell?

- Command Reference¹⁰
- Explain Shell¹¹

¹⁰<http://files.fosswire.com/2007/08/fwunixref.pdf>

¹¹<http://explainshell.com>

Chapter 16

The Unix bootcamp

The following document has been adapted from the Command-line Bootcamp¹ by Keith Bradnam² licensed via Creative Commons Attribution 4.0 International License. The original content has been substantially reworked, abbreviated and simplified.

16.1 Introduction

This ‘bootcamp’ is intended to provide the reader with a basic overview of essential Unix/Linux commands that will allow them to navigate a file system and move, copy, edit files. It will also introduce a brief overview of some ‘power’ commands in Unix.

16.2 Why Unix?

The Unix operating system³ has been around since 1969. Back then, there was no such thing as a graphical user interface. You typed everything. It may seem archaic to use a keyboard to issue commands today, but it’s much easier to automate keyboard tasks than mouse tasks. There are several variants of

¹<http://korflab.ucdavis.edu/bootcamp.html>

²<http://www.keithbradnam.com/>

³<http://en.wikipedia.org/wiki/Unix>

Unix (including Linux⁴), though the differences do not matter much for most basic functions.

Increasingly, the raw output of biological research exists as *in silico* data, usually in the form of large text files. Unix is particularly suited to working with such files and has several powerful (and flexible) commands that can process your data for you. The real strength of learning Unix is that most of these commands can be combined in an almost unlimited fashion. So if you can learn just five Unix commands, you will be able to do a lot more than just five things.

16.3 Typeset Conventions

Command-line examples that you are meant to type into a terminal window will be shown indented in a constant-width font, e.g.

```
ls -lrh
```

The lessons from this point onwards will assume very little apart from the following:

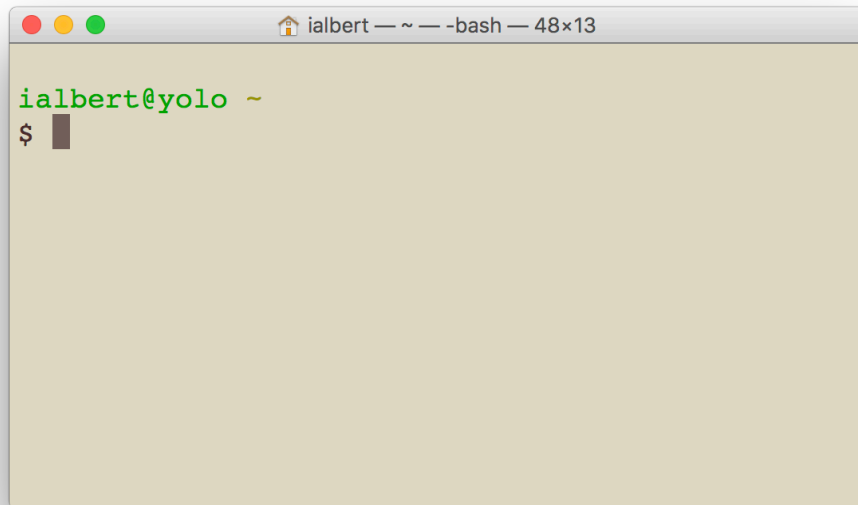
1. You have access to a Unix/Linux system
2. You know how to launch a terminal program on that system
3. You have a home directory where you can create/edit new files

16.4 1. The Terminal

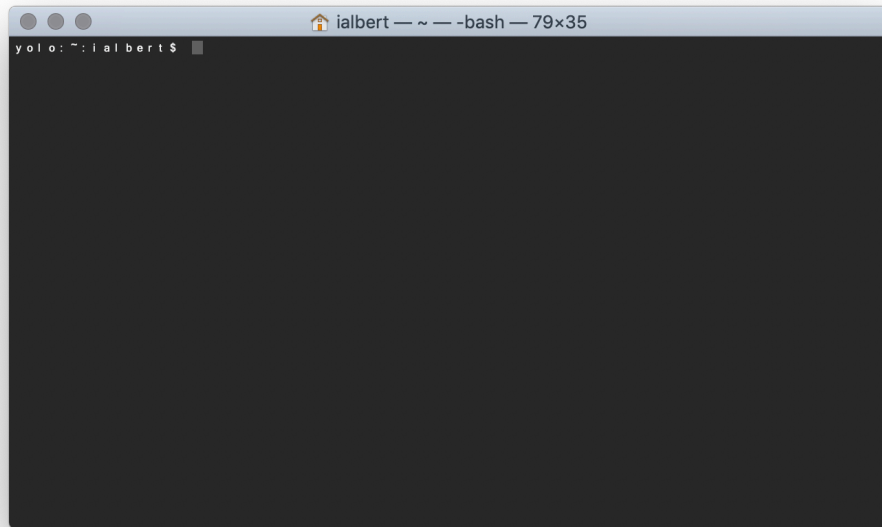
A *terminal* is the common name for the program that does two main things. It allows you to type input to the computer (i.e. run programs, move/view files etc.) and it allows you to see output from those programs. All Unix machines will have a terminal program available.

Open the terminal application. You should now see something that looks like the following:

⁴<http://en.wikipedia.org/wiki/Linux>



Your terminal may not look like the image above. Depending on how your system is set up, it may even look like the image below:



If your terminal is not easily readable customize it. Make the fonts BIGGER, make the background lighter. On a Mac select Preferences, on Windows right click the top bar and set more appropriate colors and font.

Understand that learning Unix is also a “physical” skill that requires a sort of hand-eye coordination. You have to teach your body to cooperate. You have to be comfortable, you have to see the commands clearly and with no effort. The presence of a single character in a multiline pipeline may change the meaning of the entire process. Ensure that you have minimal distractions and that the text in your terminal is eminently legible.

There will be many situations where it will be useful to have multiple terminals open and it will be a matter of preference as to whether you want to have multiple windows, or one window with multiple tabs (there are typically keyboard shortcuts for switching between windows, or moving between tabs).

16.5 2. Your first Unix command

It is important to note that you will always be *inside* a single directory when using the terminal. The default behavior is that when you open a

new terminal you start in your own *home* directory (containing files and directories that only you can modify). To see what files and directories are in your home directory, you need to use the `ls` command. This command lists the contents of a directory. If we run the `ls` command we should see something like:

```
ls
```

prints (this depends on what computer you use):

```
Applications Desktop Documents Downloads
```

There are four things that you should note here:

1. You will probably see different output to what is shown here, it depends on your operating system and computer setup. Don't worry about that for now.
2. In your case, you may see a `$` symbol. This is called the **Unix command prompt**. Note that the command prompt might not look the same on every Unix system. We do not include the command prompt in our examples so that you can copy-paste code. In general you use the command prompt to be able to tell that the computer is waiting on commands to be entered.
3. The output of the `ls` command lists two types of objects: files and directories. We'll learn how to tell them apart later on.
4. After the `ls` command finishes it produces a new command prompt, ready for you to type your next command.

The `ls` command is used to list the contents of *any* directory, not necessarily the one that you are currently in. Try the following:

```
ls /bin
```

it will print a lot of program names, among them,

```
bash pwd mv
```

16.6 3: The Unix tree

Looking at directories from within a Unix terminal can often seem confusing. But bear in mind that these directories are exactly the same type of folders

that you can see if you use any graphical file browser. From the *root* level (*/*) there are usually a dozen or so directories. You can treat the root directory like any other, e.g. you can list its contents:

```
ls /
```

prints:

```
bin    dev    initrd.img    lib64    mnt    root    software    tmp    vmlinuz
boot  etc    initrd.img.old  lost+found  opt    run    srv        usr    vmlinuz.old
data  home   lib           media    proc   sbin   sys        var
```

You might notice some of these names appear in different colors. Many Unix systems will display files and directories differently by default. Other colors may be used for special types of files. When you log in to a computer, you are typically placed in your home directory, which is often inside a directory called ‘Users’ or ‘home’.

16.7 4: Finding out where you are

There may be many hundreds of directories on any Unix machine, so how do you know which one you are in? The command `pwd`⁵ will print the working directory. That’s pretty much all this command does:

```
pwd
```

prints:

```
/Users/ialbert
```

When you log in to a Unix computer, you are typically placed into your *home* directory. In this example, after we log in, we are placed in a directory called ‘ialbert’ which itself is a *subdirectory* of another directory called ‘Users’. Conversely, ‘Users’ is the *parent* directory of ‘ialbert’. The first forward slash that appears in a list of directory names always refers to the top level directory of the file system (known as the *root directory*). The remaining forward slash (between ‘Users’ and ‘ialbert’) delimits the various parts of the directory hierarchy. If you ever get ‘lost’ in Unix, remember the `pwd` command.

⁵<http://en.wikipedia.org/wiki/Pwd>

As you learn Unix you will frequently type commands that don't seem to work. Most of the time this will be because you are in the wrong directory, so it's a really good habit to get used to running the `pwd` command a lot.

16.8 5: Making new directories

If we want to make a new directory (e.g. to store some lecture related files), we can use the `mkdir` command:

```
mkdir edu
ls
```

shows:

```
edu
```

16.9 6: Getting from 'A' to 'B'

We are in the home directory on the computer but we want to work in the new `edu` directory. To change directories in Unix, we use the `cd` command:

```
cd edu
pwd
```

will print:

```
/Users/ialbert/edu
```

Let's make two new subdirectories and navigate into them:

```
mkdir lecture
cd lecture
pwd
```

prints:

```
/Users/ialbert/edu/lecture
```

then make a data directory:

```
mkdir data
cd data/
pwd
```

prints:

```
/Users/ialbert/edu/lecture/data
```

We created the two directories in separate steps, but it is possible to use the `mkdir` command in way to do this all in one step.

Like most Unix commands, `mkdir` supports *command-line options* which let you alter its behavior and functionality. Command-like options are – as the name suggests – optional arguments that are placed after the command name. They often take the form of single letters (following a dash). If we had used the `-p` option of the `mkdir` command we could have done this in one step. E.g.

```
mkdir -p ~/edu/lecture/docs
```

Note the spaces either side of the -p!

16.10 7: The root directory

Let's change directory to the root directory:

```
cd /
cd Users
cd ialbert
```

In this case, we may as well have just changed directory in one go:

```
cd /Users/ialbert
```

The leading `/` is incredibly important. The following two commands are very different:

```
cd /step1/step2
cd step1/step2/
```

The first command specifies a so called *absolute path* . It says go the root directory (folder) then the **step1** folder then the **step2** folder that opens from the **step1** directory. Importantly there can only be one **/step1/step2** directory on any Unix system.

The second command specifies a so called *relative path*. It says that from the *current location* go to the **step1** directory then the **step2** directory.

There can potentially be many **step1/step2** directories that open from different directories.

Learn and understand the difference between these two commands. Initially it is very easy to miss the leading / yet it is essential and fundamentally important. With a little time and practice your eyes will be trained and you will quickly notice the difference.

16.11 8: Navigating upwards in the Unix filesystem

Frequently, you will find that you want to go ‘upwards’ one level in the directory hierarchy. Two dots `..` are used in Unix to refer to the *parent* directory of wherever you are. Every directory has a parent except the root level of the computer. Let’s go into the **lecture** directory and then navigate up two levels:

```
cd
cd edu
pwd
```

prints:

```
/Users/ialbert/edu
```

but now:

```
cd ..
pwd
```

prints:

```
/Users/ialbert
```

What if you wanted to navigate up *two* levels in the file system in one go? Use two sets of the `..` operator, separated by a forward slash:

```
cd ../../
```

16.12 9: Absolute and relative paths

Using `cd ..` allows us to change directory *relative* to where we are now. You can also always change to a directory based on its *absolute* location. E.g. if you are working in the `~/edu/lecture` directory and you want to change to the `~/edu/tmp` directory, then you could do either of the following:

```
# Go down one level. Use a relative path.
cd ../tmp
pwd
```

For me it prints:

```
/Users/ialbert/edu/tmp
```

or:

```
# Use an absolute path.
cd ~/edu/tmp
pwd
```

the last command also prints:

```
/Users/ialbert/edu/tmp
```

The `~` is a shorthand notation that gets substituted to my home directory `/Users/ialbert/`. You can think of it as a mixture of relative and absolute paths. It is a full path, but it expands to different locations for different people. If I were to specify the path as:

```
# This is an "even more" absolute path.
cd /Users/ialbert/edu/tmp
pwd
```

Then the example would only work on my computer (or other computers where the home directory is also named `ialbert`). Using the `~` allows us to

get the best of both worlds. It is an absolute path with a little bit of *relativity* in it.

All examples achieve the same thing, but the later examples require that you know about more about the location of your directory of interest. It is important that you understand the slight differences here, as the absolute and relative paths are important concepts that you will have make use of frequently.

Sometimes it is quicker to change directories using the relative path, and other times it will be quicker to use the absolute path.

16.13 10: Finding your way back home

Remember that the command prompt shows you the name of the directory that you are currently in, and that when you are in your home directory it shows you a tilde character (~) instead? This is because Unix uses the tilde character as a short-hand way of [specifying a home directory][home directory].

See what happens when you try the following commands (use the `pwd` command after each one to confirm the results):

```
cd /  
cd ~  
cd
```

Hopefully, you should find that `cd` and `cd ~` do the same thing, i.e. they take you back to your home directory (from wherever you were). You will frequently want to jump straight back to your home directory, and typing `cd` is a very quick way to get there.

You can also use the `~` as a quick way of navigating into subdirectories of your home directory when your current directory is somewhere else. I.e. the quickest way of navigating from anywhere on the filesystem to your `edu/lecture` directory is as follows:

```
cd ~/edu/lecture
```

16.14 11: Making the `ls` command more useful

The `..` operator that we saw earlier can also be used with the `ls` command, e.g. you can list directories that are ‘above’ you:

```
cd ~/edu/lecture/  
ls ../../
```

Time to learn another useful command-line option. If you add the letter ‘l’ to the `ls` command it will give you a longer output compared to the default:

```
ls -l ~
```

prints (system-dependent):

```
drwx-----  5 ialbert  staff   170B May 13 15:24 Applications  
drwx-----+  7 ialbert  staff   238B Aug 20 05:51 Desktop  
drwx-----+ 30 ialbert  staff   1.0K Aug 12 13:12 Documents  
drwx-----+ 14 ialbert  staff   476B Aug 24 10:43 Downloads
```

For each file or directory we now see more information (including file ownership and modification times). The ‘d’ at the start of each line indicates that these are directories. There are many, many different options for the `ls` command. Try out the following (against any directory of your choice) to see how the output changes.

```
ls -l  
ls -R  
ls -l -t -r  
ls -lh
```

Note that the last example combines multiple options but only uses one dash. This is a very common way of specifying multiple command-line options. You

may be wondering what some of these options are doing. It is time to learn about Unix documentation....

16.15 12: Man pages

If every Unix command has so many options, you might be wondering how you find out what they are and what they do. Well, thankfully every Unix command has an associated ‘manual’ that you can access by using the `man` command. E.g.

```
man ls
man cd
man man # Yes, even the man command has a manual page
```

When you are using the `man` command, press **space** to scroll down a page, **b** to go back a page, or **q** to quit. You can also use the up and down arrows to scroll a line at a time. The `man` command is actually using another Unix program, a text viewer called `less`, which we’ll come to later on.

16.16 13: Removing directories

We now have a few (empty) directories that we should remove. To do this use the `rmdir` command. This will only remove empty directories, so it is quite safe to use. If you want to know more about this command (or any Unix command), then remember that you can just look at its man page.

```
cd ~/edu/lecture/
rmdir data
cd ..
rmdir lecture
ls
```

Note, you have to be outside a directory before you can remove it with `rmdir`

16.17 14: Using tab completion

Saving keystrokes may not seem important now, but the longer that you spend typing in a terminal window, the happier you will be if you can reduce the time you spend at the keyboard. Especially as prolonged typing is not good for your body. So the best Unix tip to learn early on is that you can [tab complete] the names of files and programs on most Unix systems. Type enough letters to uniquely identify the name of a file, directory, or program and press tab – Unix will do the rest. E.g. if you type ‘tou’ and then press tab, Unix should autocomplete the word to ‘touch’ (this is a command which we will learn more about in a minute). In this case, tab completion will occur because there are no other Unix commands that start with ‘tou’. If pressing tab doesn’t do anything, then you have not have typed enough unique characters. In this case pressing tab *twice* will show you all possible completions. This trick can save you a LOT of typing!

Navigate to your home directory, and then use the `cd` command to change to the `edu` directory. Use tab completion to complete directory name. If there are no other directories starting with ‘e’ in your home directory, then you should only need to type ‘cd’ + ‘e’ + ‘tab’.

Tab completion will make your life easier and make you more productive!

Another great time-saver is that Unix stores a list of all the commands that you have typed in each login session. You can access this list by using the `history` command or more simply by using the up and down arrows to access anything from your history. So if you type a long command but make a mistake, press the up arrow and then you can use the left and right arrows to move the cursor in order to make a change.

16.18 15: Creating empty files with the touch command

The following sections will deal with Unix commands that help us to work with files, i.e. copy files to/from places, move files, rename files, remove files, and most importantly, look at files. First, we need to have some files to play

with. The Unix command `touch` will let us create a new, empty file. The `touch` command does other things too, but for now we just want a couple of files to work with.

```
cd edu
touch heaven.txt
touch earth.txt
ls
```

prints:

```
earth.txt  heaven.txt
```

16.19 16: Moving files

Now, let's assume that we want to move these files to a new directory ('temp'). We will do this using the Unix `[mv]` (move) command. Remember to use tab completion:

```
mkdir temp
mv heaven.txt temp
mv earth.txt temp
ls temp
```

For the `mv` command, we always have to specify a source file (or directory) that we want to move, and then specify a target location. If we had wanted to, we could have moved both files in one go by typing any of the following commands:

```
mv *.txt temp
mv *t temp
mv *ea* temp
```

The asterisk `*` acts as a *wild-card character*, essentially meaning 'match anything'. The second example works because there are no other files or directories in the directory that end with the letter 't' (if there were, then they

would be moved too). Likewise, the third example works because only those two files contain the letters ‘ea’ in their names. Using wild-card characters can save you a lot of typing.

The `?` character is also a wild-card but with a slightly different meaning. See if you can work out what it does.

16.20 17: Renaming files

In the earlier example, the destination for the `mv` command was a directory name (`temp`). So we moved a file from its source location to a target location, but note that the target could have also been a (different) file name, rather than a directory. E.g. let’s make a new file and move it whilst renaming it at the same time:

```
touch rags
ls
mv rags temp/riches
ls temp/
```

prints:

```
earth.txt  heaven.txt  riches
```

In this example we create a new file (`‘rags’`) and move it to a new location and in the process change the name (to `‘riches’`). So `mv` can rename a file as well as move it. The logical extension of this is using `mv` to rename a file without moving it. You may also have access to a tool called `rename`, type `man rename` for more information.

```
mv temp/riches temp/rags
```

16.21 18: Moving directories

It is important to understand that as long as you have specified a ‘source’ and a ‘target’ location when you are moving a file, then it doesn’t matter

what your *current* directory is. You can move or copy things within the same directory or between different directories regardless of whether you are in any of those directories. Moving directories is just like moving files:

```
mv temp temp2
ls temp2
```

16.22 19: Removing files

You've seen how to remove a directory with the `rmdir` command, but `rmdir` won't remove directories if they contain any files. So how can we remove the files we have created (inside `temp`)? In order to do this, we will have to use the `rm` (remove) command.

Please read the next section VERY carefully. Misuse of the `rm` command can lead to needless death & destruction

Potentially, `rm` is a very dangerous command; if you delete something with `rm`, you will not get it back! It is possible to delete everything in your home directory (all directories and subdirectories) with `rm`. That is why it is such a dangerous command.

Let me repeat that last part again. It is possible to delete EVERY file you have ever created with the `rm` command. Are you scared yet? You should be. Luckily there is a way of making `rm` a little bit safer. We can use it with the `-i` command-line option which will ask for confirmation before deleting anything (remember to use tab-completion):

```
cd temp
ls
rm -i earth.txt heaven.txt rags
```

will ask permission for each step:

```
rm: remove regular empty file 'earth.txt'? y
rm: remove regular empty file 'heaven.txt'? y
rm: remove regular empty file 'rags'? y
```

We could have simplified this step by using a wild-card (e.g. `rm -i *.txt`) or we could have made things more complex by removing each file with a separate `rm` command.

16.23 20: Copying files

Copying files with the `cp` (copy) command has a similar syntax as `mv`, but the file will remain at the source and be copied to the target location. Remember to always specify a source and a target location. Let's create a new file and make a copy of it:

```
touch file1
cp file1 file2
ls
```

What if we wanted to copy files from a different directory to our current directory? Let's put a file in our home directory (specified by `~`, remember) and copy it to the lecture directory (`~/edu/lecture`):

```
$ touch ~/edu/file3
$ cp ~/edu/file3 ~/edu/lecture/
```

In Unix, the current directory can be represented by a `.` (dot) character. You will often use for copying files to the directory that you are in. Compare the following:

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

In this case, using the dot is somewhat pointless because `ls` will already list the contents of the current directory by default. Also note how the trailing slash is optional.

16.24 21: Copying directories

The `cp` command also allows us (with the use of a command-line option) to copy entire directories. Use `man cp` to see how the `-R` or `-r` options let you copy a directory *recursively*.

16.25 22: Viewing files with less (or more)

So far we have covered listing the contents of directories and moving/copying/deleting either files or directories. Now we will quickly cover how you can look at files. The `more` or `less` commands let you view (but not edit) text files. We will use the `echo` command to put some text in a file and then view it:

```
echo "Call me Ishmael."  
Call me Ishmael.  
echo "Call me Ishmael." > opening_lines.txt  
ls
```

prints:

```
opening_lines.txt
```

we can view the content of the file with:

```
more opening_lines.txt
```

On its own, `echo` isn't a very exciting Unix command. It just echoes text back to the screen. But we can redirect that text into an output file by using the `>` symbol. This allows for something called file *redirection*.

Careful when using file redirection (`>`), it will overwrite any existing file of the same name

When you are using `more` or `less`, you can bring up a page of help commands by pressing `h`, scroll forward a page by pressing `space`, or go forward or backwards one line at a time by pressing `j` or `k`. To exit `more` or `less`, press `q` (for quit). The `more` and `less` programs also do about a million other useful things (including text searching).

16.26 23: Viewing files with cat

Let's add another line to the file:

```
echo "The primroses were over." >> opening_lines.txt
cat opening_lines.txt
```

prints:

```
Call me Ishmael.
The primroses were over.
```

Notice that we use `>>` and not just `>`. This operator will **append** to a file. If we only used `>`, we would end up overwriting the file. The `cat` command displays the contents of the file (or files) and then returns you to the command line. Unlike `less` you have no control on how you view that text (or what you do with it). It is a very simple, but sometimes useful, command. You can use `cat` to quickly combine multiple files or, if you wanted to, make a copy of an existing file:

```
cat opening_lines.txt > file_copy.txt
```

16.27 24: Counting characters in a file

```
$ ls
opening_lines.txt

$ ls -l
total 4
-rw-rw-r-- 1 ubuntu ubuntu 42 Jun 15 04:13 opening_lines.txt

$ wc opening_lines.txt
  2  7 42 opening_lines.txt

$ wc -l opening_lines.txt
2 opening_lines.txt
```

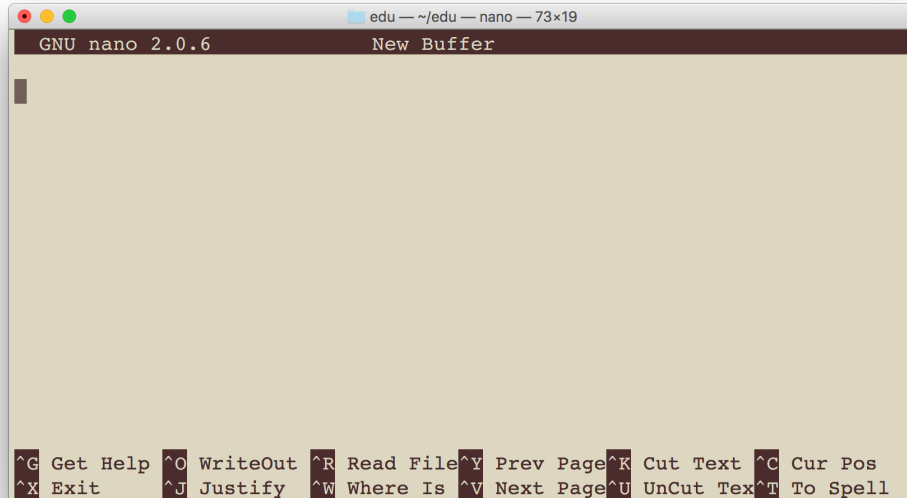
The `ls -l` option shows us a long listing, which includes the size of the file in bytes (in this case ‘42’). Another way of finding this out is by using Unix’s `wc` command (word count). By default this tells you many lines, words, and characters are in a specified file (or files), but you can use command-line options to give you just one of those statistics (in this case we count lines with `wc -l`).

16.28 25: Editing small text files with nano

Nano is a lightweight editor installed on most Unix systems. There are many more powerful editors (such as ‘emacs’ and ‘vi’), but these have steep learning curves. Nano is very simple. You can edit (or create) files by typing:

```
nano opening_lines.txt
```

You should see the following appear in your terminal:



The bottom of the nano window shows you a list of simple commands which are all accessible by typing ‘Control’ plus a letter. E.g. Control + X exits the program.

16.29 26: The \$PATH environment variable

One other use of the `echo` command is for displaying the contents of something known as *environment variables*. These contain user-specific or system-wide values that either reflect simple pieces of information (your username), or lists of useful locations on the file system. Some examples:

```
$ echo $USER
ialbert
$ echo $HOME
/Users/ialbert
echo $PATH
/usr/local/sbin:/usr/local/bin:/Users/ialbert/bin
```

The last one shows the content of the `$PATH` environment variable, which displays a — colon separated — list of directories that are expected to contain programs that you can run. This includes all of the Unix commands that you have seen so far. These are files that live in directories which are run like programs (e.g. `ls` is just a special type of file in the `/bin` directory).

Knowing how to change your `$PATH` to include custom directories can be necessary sometimes (e.g. if you install some new bioinformatics software in a non-standard location).

16.30 27: Matching lines in files with grep

Use `nano` to add the following lines to `opening_lines.txt`:

```
Now is the winter of our discontent.
All children, except one, grow up.
The Galactic Empire was dying.
In a hole in the ground there lived a hobbit.
It was a pleasure to burn.
It was a bright, cold day in April, and the clocks were striking thirteen.
It was love at first sight.
I am an invisible man.
It was the day my grandmother exploded.
When he was nearly thirteen, my brother Jem got his arm badly broken at the el
```

Marley was dead, to begin with.

You will often want to search files to find lines that match a certain pattern. The Unix command **grep** does this (and much more). The following examples show how you can use **grep**'s command-line options to:

- show lines that match a specified pattern
- ignore case when matching (**-i**)
- only match whole words (**-w**)
- show lines that don't match a pattern (**-v**)
- Use wildcard characters and other patterns to allow for alternatives (*****, **.**, and **[]**)

Show lines that match the word **was**:

```
$ grep was opening_lines.txt
The Galactic Empire was dying.
It was a pleasure to burn.
It was a bright, cold day in April, and the clocks were striking thirteen.
It was love at first sight.
It was the day my grandmother exploded.
When he was nearly thirteen, my brother Jem got his arm badly broken at the elbow.
Marley was dead, to begin with.
```

Use:

```
grep --color=AUTO was opening_lines.txt
```

to highlight the match.

Show lines that do not match the word **was** :

```
$ grep -v was opening_lines.txt
Now is the winter of our discontent.
All children, except one, grow up.
In a hole in the ground there lived a hobbit.
I am an invisible man.
```

`grep` has a great many options and applications.

16.31 28: Combining Unix commands with pipes

One of the most powerful features of Unix is that you can send the output from one command or program to any other command (as long as the second command accepts input of some sort). We do this by using what is known as a **pipe**. This is implemented using the ‘|’ character (which is a character which always seems to be on different keys depending on the keyboard that you are using). Think of the pipe as simply connecting two Unix programs. Here’s an example which introduces some new Unix commands:

```
grep was opening_lines.txt | wc -c
# 316

grep was opening_lines.txt | sort | head -n 3 | wc -c
# 130
```

The first use of `grep` searches the specified file for lines matching ‘was’. It sends the lines that match through a pipe to the `wc` program. We use the `-c` option to just count characters in the matching lines (316).

The second example first sends the output of `grep` to the Unix `sort` command. This sorts a file alphanumerically by default. The sorted output is sent to the `head` command which by default shows the first 10 lines of a file. We use the `-n` option of this command to only show 3 lines. These 3 lines are then sent to the `wc` command as before.

Whenever making a long pipe, test each step as you build it!

16.32 Miscellaneous Unix power commands

The following examples introduce some other Unix commands, and show how they could be used to work on a fictional file called `file.txt`. Remember, you can always learn more about these Unix commands from their respective man pages with the `man` command. These are not all real world cases, but rather show the diversity of Unix command-line tools:

- View the penultimate (second-to-last) 10 lines of a file (by piping `head` and `tail` commands):

```
#tail -n 20 gives the last 20 lines of the file, and piping that to head will show  
tail -n 20 file.txt | head
```

- Show the lines of a file that begin with a start codon (ATG) (the `^` matches patterns at the start of a line):

```
grep "^ATG" file.txt
```

- Cut out the 3rd column of a tab-delimited text file and sort it to only show unique lines (i.e. remove duplicates):

```
cut -f 3 file.txt | sort -u
```

- Count how many lines in a file contain the words ‘cat’ or ‘bat’ (`-c` option of `grep` counts lines):

```
grep -c '[bc]at' file.txt
```

- Turn lower-case text into upper-case (using `tr` command to ‘transliterate’):

```
cat file.txt | tr 'a-z' 'A-Z'
```

- Change all occurrences of ‘Chr1’ to ‘Chromosome 1’ and write changed output to a new file (using `sed` command):

```
cat file.txt | sed 's/Chr1/Chromosome 1/' > file2.txt
```