some may need Rstudio Cloud!
https://posit.cloud/

Command-line Bootcamp

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

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. It was originally developed as part of a Bioinformatics Core Workshop taught at UC Davis (Using the Linux Command-Line for Analysis of High Throughput Sequence Data).

Why Unix?

The Unix operating system 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 Unix (including Linux), 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.

Typeset Conventions

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

```
$ ls -lrh
```

Sometimes the accompanying text will include a reference to a Unix command. Any such text will also be in a fixed-width, boxed font. E.g.

Type the `pwd` command again.

From time to time this documentation will contain web links to pages that will help you find out more about certain Unix commands. Usually, the *first* mention of a command or function will be a hyperlink to Wikipedia. Important or critical points will be styled like so:

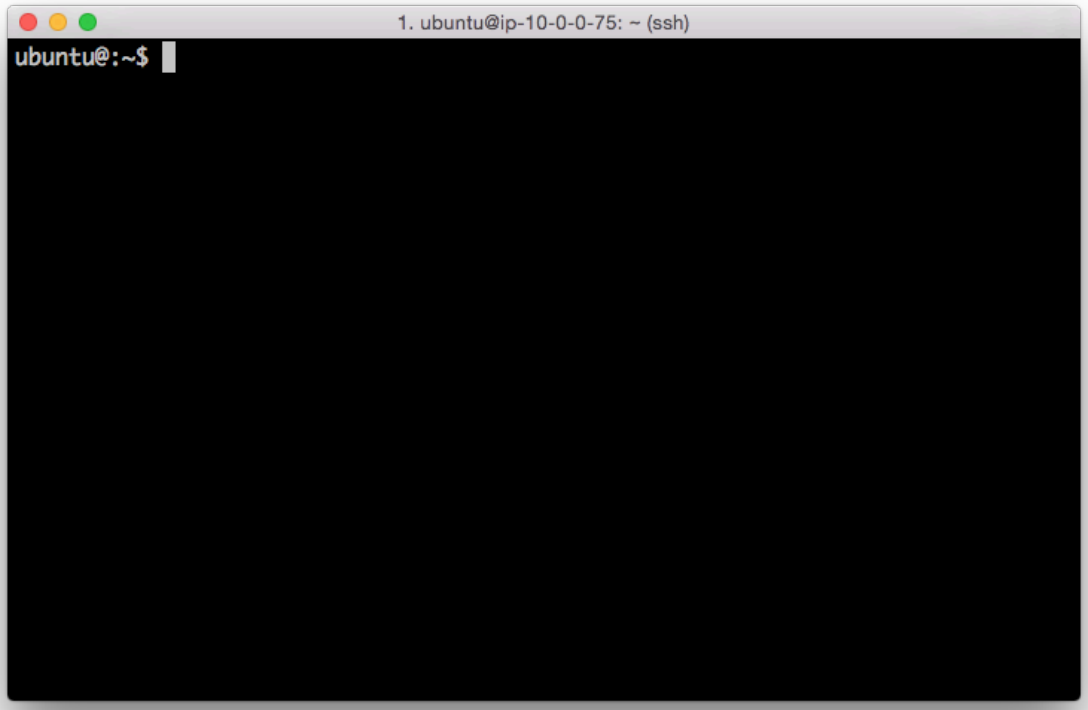
This is an important point!

---

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:

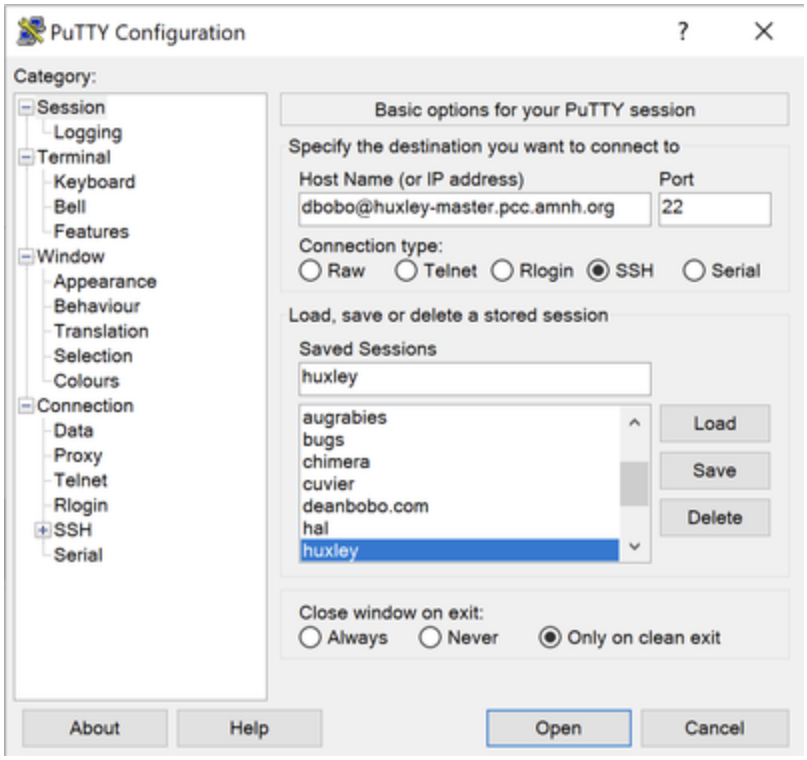


Terminal application

connect over ssh

```
ssh dbobo@huxley-master.pcc.amnh.org
```

if you're using putty:



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).

For MacOS, I recommend iTerm2.

For Windows I recommend PuTTy (for SSH connections)

---

2. Your first Unix command

It's 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 our home directory, we 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
project.Rproj
$
```

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 computer setup. Don't worry about that for now.
2. The `/cloud/project$` that you see is the Unix command prompt. In this case, it contains the path of the current /cloud/project. Note that the command prompt might not look the same on different Unix systems. In this case, the `$` sign marks the end of the prompt.
3. The output of the `ls` command lists one thing. In this case, it is the configuration file for your Rstudio project. You can ignore this for now.
4. After the `ls` command finishes it produces a new command prompt, ready for you to type your next command.

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 /
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 appearing 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 working with your files in your home directory, and this is often inside a directory called ‘users’ or ‘home’.

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 will Print the Working Directory and that’s pretty much all this command does:

```
$ pwd
/home/ubuntu
```

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 ‘ubuntu’ which itself is a *subdirectory* of another directory called ‘home’. Conversely, ‘users’ is the *parent* directory of ‘clmuser’. 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 ‘home’ and ‘ubuntu’) delimits the various parts of the directory hierarchy. If you ever get ‘lost’ in Unix, remember the `pwd` command.

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.

5: Making new directories

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

```
$ mkdir Learning_unix
$ ls
command_line_course  Learning_unix  linux_bootcamp
```

6: Getting from ‘A’ to ‘B’

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

```
$ cd Learning_unix
```

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

```
$ mkdir Outer_directory
$ cd Outer_directory
```

```
$ mkdir Inner_directory
$ cd Inner_directory/
```

We created the two directories in separate steps, but it is possible to use the `mkdir` command 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 Outer_directory/Inner_directory
```

Note the spaces either side the `-p`!

7: The root directory

Let’s change directory to the root directory, and then into the usr directory and then into the bin directory

```
$ cd /
$ cd usr
$ cd bin
```

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

```
cd /usr/bin/
```

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

```
cd /usr/bin/ # absolute path
cd usr/bin/   # relative path
```

The first command says go to the `bin` directory that is beneath the `usr` directory that is at the top level (the root) of the file system. There can only be one `/usr/bin` directory on any Unix system.
The second command says go to the `bin` directory that is beneath the `usr` directory that is located **wherever I am right now**. There can potentially be many `usr/bin` directories on a Unix system (though this is unlikely).

Learn and understand the difference between these two commands.

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 `Learning_unix` directory and then navigate up two levels:

```
$ cd
$ cd ~/Learning_unix/
$ cd ..
```

What if you wanted to navigate up *two* levels in the file system in one go? It's very simple, just use two sets of the `..` operator, separated by a forward slash:

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

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 **`/cloud/project/Learning_unix`** directory and you then want to change to the **`/tmp`** directory, then you could do either of the following:

`$ cd ../../../tmp`

or…

`$ cd /tmp`

They both achieve the same thing, but the 2nd example requires that you know about the full *path* from the root level of the computer to your directory of interest (the 'path' is an important concept in Unix). Sometimes it is quicker to change directories using the relative path, and other times it will be quicker to use the absolute path.

10: Finding your way back home

Unix uses the tilde character as a short-hand way of specifying a home directory.

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

```
cd /
cd ~ # takes you home
cd   # takes you home
```

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 the root directory to your `Learning_unix` directory is as follows:

```
$ cd /
$ cd ~/Learning_unix
```

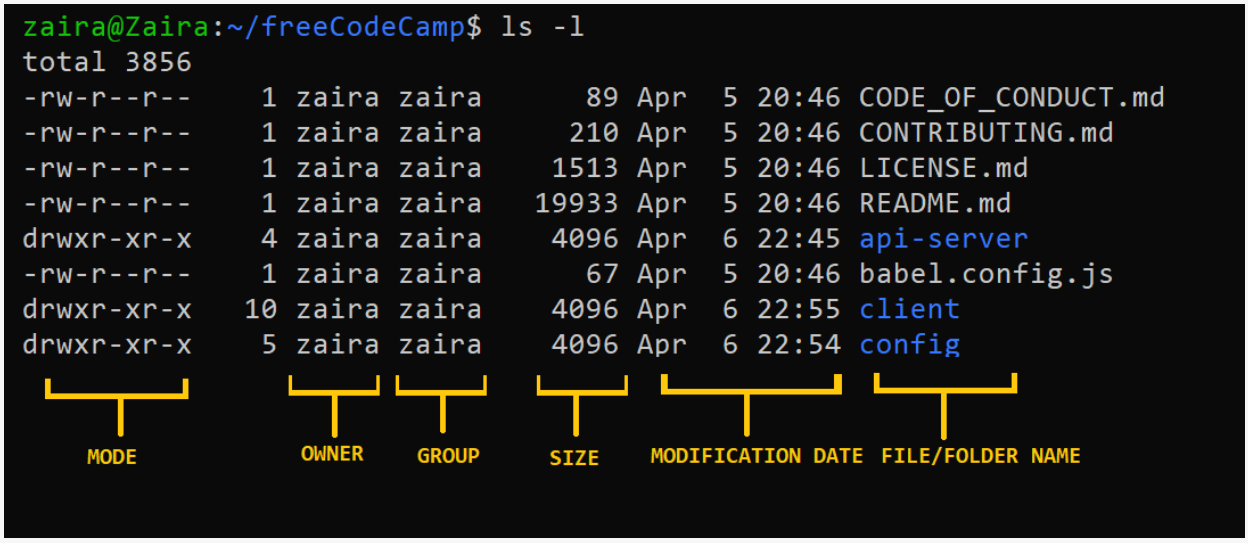
11: Making the `ls` command more useful

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 ~/Learning_unix/Outer_directory/
$ ls ../../
Learning_unix  project.Rproj
```

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 /home
total 0
lrwxrwxrwx 1 root root 20 Jan  9 11:37 r1819497 -> /cloud/home/r1819497
```



Mode, owner, and group explained in more detail in section 33.

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 combine multiple options but only use 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’s time to learn about Unix documentation….

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.

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 ~/Learning_unix/Outer_directory/
$ rmdir Inner_directory/
$ cd ..
$ rmdir Outer_directory/
$ ls
$
```

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

14: Using tab completion

Saving keystrokes may not seem important, 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 that 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 `Learning_unix` directory. Use tab completion to complete directory name. If there are no other directories starting with 'L' in your home directory, then you should only need to type 'cd' + 'L' + '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.

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 Learning_unix/
$ touch heaven.txt
$ touch earth.txt
$ ls
earth.txt  heaven.txt
```

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
$ ls Temp/
earth.txt  heaven.txt
```

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 letters 't' (if there was, 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.

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
rags  Temp
$ mv rags Temp/riches
$ ls Temp/
```

```
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 have to use `mv` to do this as Unix does not have a separate ‘rename’ command):

```
$ mv Temp/riches Temp/rags
```

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/riches Temp/rags
$ mkdir Temp2
$ mv Temp2 Temp
$ ls Temp/
earth.txt  heaven.txt  rags  Temp2
```

This step moves the Temp2 directory inside the Temp directory.

Try creating a ‘Temp3’ directory inside ‘Learning_unix’ and then `cd` to `/tmp`. Can you move `Temp3` inside `Temp` without changing directory?

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 `Learning_Unix/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
earth.txt  heaven.txt  rags  Temp2
$ rm -i earth.txt heaven.txt rags
rm: remove regular empty file 'earth.txt'? y
rm: remove regular empty file 'heaven.txt'? y
rm: remove regular empty file 'rags'? y
$ ls
Temp2
```

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. Let's finish cleaning up:

```
rmdir Temp2/Temp3
rmdir Temp2
cd ..
rmdir Temp
```

20: Copying files

Copying files with the cp (copy) command is very similar to moving them. Remember to always specify a source and a target location. Let's create a new file and make a copy of it:

```
~/Learning_unix$ touch file1
~/Learning_unix$ cp file1 file2
~/Learning_unix$ ls
file1  file2
```

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 current directory (`Learning_unix`):

```
~/Learning_unix$ touch ~/file3
~/Learning_unix$ ls ~
command_line_course  file3  Learning_unix  linux_bootcamp
~/Learning_unix$ cp ~/file3 .
~/Learning_unix$ ls
file1  file2  file3
```

This last step introduces another new concept. In Unix, the current directory can be represented by a `.` (dot) character. You will mostly use this only for copying files to the current 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. You can use `rm` to remove the temporary files.

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*.

22: Echo, redirect, and less

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

```
~/Learning_unix$ echo "Call me Ishmael."
Call me Ishmael.
~/Learning_unix$ echo "Call me Ishmael." > opening_lines.txt
~/Learning_unix$ ls
opening_lines.txt
~/Learning_unix$ less 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 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 less, press `q` (for quit). The `less` program also does about a million other useful things (including text searching).

23: Viewing files with cat; append to file

Let's add another line to the file:

```
~/Learning_unix$ echo "The primroses were over." >> opening_lines.txt
~/Learning_unix$ cat opening_lines.txt
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
```

24: Counting characters in a file

```
~/Learning_unix$ ls
opening_lines.txt

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

~/Learning_unix$ wc opening_lines.txt
 2  7 42 opening_lines.txt

~/Learning_unix$ wc -l opening_lines.txt
2 opening_lines.txt
```

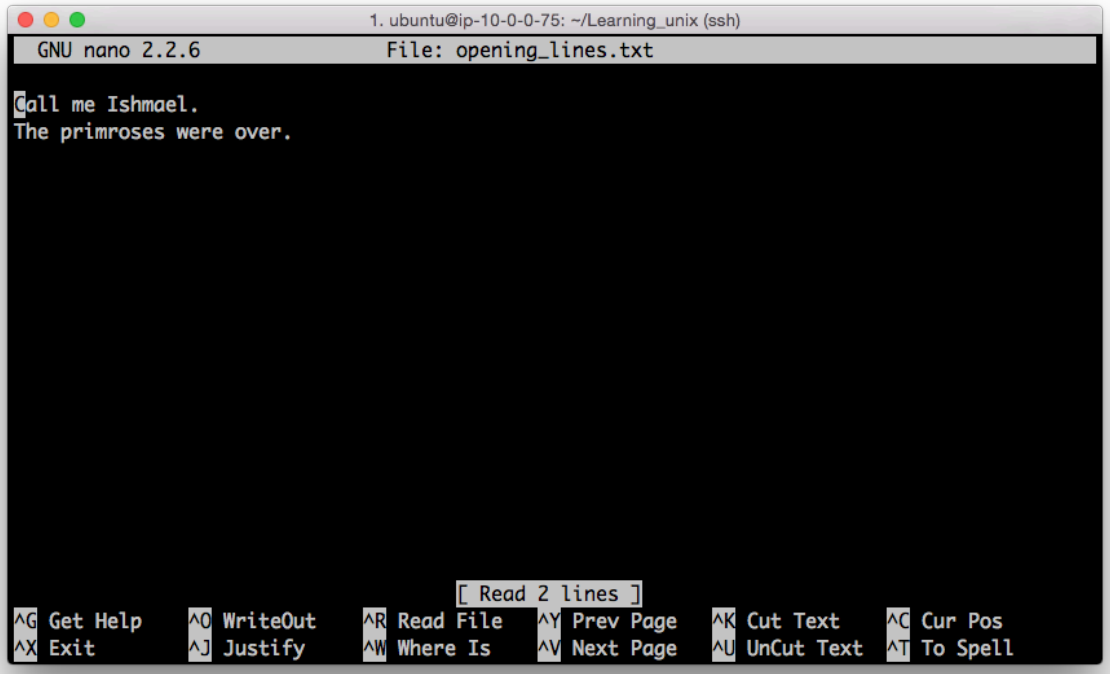
The `ls -l` option shows us a long listing, which includes the size of the file in bytes (in this case '42'). Another way of finding this out is by using Unix's wc command (word count). By default this tells you many lines, words, and characters are in a specified file (or files), but you can use command-line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

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.

We recommend Vim or Emacs text editors. They are much more feature-rich.

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:

```
~/Learning_unix$ echo $USER
ubuntu
~/Learning_unix$ echo $HOME
/home/ubuntu
~/Learning_unix$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

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).

env

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 elbow.
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 `[]`)

```
$ 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.

$ grep -v was opening_lines.txt
Call me Ishmael.
The primroses were over.
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 all opening_lines.txt
Call me Ishmael.

grep -i all opening_lines.txt
Call me Ishmael.
All children, except one, grow up.

grep in opening_lines.txt
Now is the winter of our discontent.
The Galactic Empire was dying.
In a hole in the ground there lived a hobbit.
It was a bright, cold day in April, and the clocks were striking thirteen.
```

```
I am an invisible man.
Marley was dead, to begin with.

grep -w in opening_lines.txt
In a hole in the ground there lived a hobbit.
It was a bright, cold day in April, and the clocks were striking thirteen.

grep -w o.. opening_lines.txt
Now is the winter of our discontent.
All children, except one, grow up.

grep [aeiou]t opening_lines.txt
In a hole in the ground there lived a hobbit.
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.

grep -w -i [aeiou]t opening_lines.txt
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.
```

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:

```
~/Learning_unix$ grep was opening_lines.txt | wc -c
328
```

```
~/Learning_unix$
grep was opening_lines.txt | sort | head -n 3 | wc -c
136
```

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 (328).

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!

29: Screen

GNU Screen is a terminal multiplexer that allows you to manage multiple terminal sessions within a single terminal window or remote terminal session. Here are some key features and aspects of GNU Screen:

Multiple Sessions: You can have multiple terminal sessions running simultaneously, each within its own "window". This is useful when working on different tasks or projects concurrently.

Session Detachment and Reattachment: You can detach a screen session and then reattach it later, even from a different terminal. This is very useful for long-running processes on remote servers.

Session Sharing: Multiple users can connect to the same screen session, making collaborative work easier.

Customization and Scripting: GNU Screen is customizable and scriptable, allowing for a tailored user experience.

Persistent Sessions: If your connection drops or you need to disconnect from a server, your screen sessions remain active and you can pick up right where you left off.

By utilizing GNU Screen, you can significantly enhance your terminal-based workflows, making it easier to manage multiple tasks and maintain long-running processes, especially on remote servers.

to start a screen session, type the command

```
$ screen
```

to see if you're in a screen session

```
$ echo $TERM
```

if you see something that says "xterm", you're not in screen.
If you see something that says "screen", you're in a screen session.

to see a list of screen sessions

```
$ screen -ls
There is a screen on:
        26760.pts-0.huxley-head (Detached)
1 Socket in /home/dbobo/.screen.
```

to detach from a screen session

```
$ screen -d
```

to reattach to a screen session

```
$ screen -r 26760
```

The escape key sequence is **ctrl + a**

[ctrl] + [a] then **[c]** = create a new window

["] = show all windows
[n] = move to next window
[p] = move to previous window

Screen cheat sheet:
https://kapeli.com/cheat_sheets/screen.docset/Contents/Resources/Documents/index

30: Keyboard Shortcuts

Shortcut	Action
Bash Navigation	
Ctrl + A	Move to the start of the command line
Ctrl + E	Move to the end of the command line
Ctrl + F	Move one character forward
Ctrl + B	Move one character backward
Ctrl + XX	Switch cursor position between start of the command line and the current position
Ctrl +] + x	Moves the cursor forward to next occurrence of x
Alt + F / Esc + F	Moves the cursor one word forward
Alt + B / Esc + B	Moves the cursor one word backward
Alt + Ctrl +] + x	Moves cursor to the previous occurrence of x
Bash Control/Process	
Ctrl + L	Similar to clear command, clears the terminal screen
Ctrl + S	Stops command output to the screen
Ctrl + Z	Suspends current command execution and moves it to the background

Ctrl + Q	Resumes suspended command
Ctrl + C	Sends SIGI signal and kills currently executing command
Ctrl + D	Closes the current terminal
Bash History	
Ctrl + R	Incremental reverse search of bash history
Alt + P	Non-incremental reverse search of bash history
Ctrl + J	Ends history search at current command
Ctrl + _	Undo previous command
Ctrl + P / Up arrow	Moves to previous command
Ctrl + N / Down arrow	Moves to next command
Ctrl + S	Gets the next most recent command
Ctrl + O	Runs and re-enters the command found via Ctrl + S and Ctrl + R
Ctrl + G	Exits history search mode
!!	Runs last command
!*	Runs previous command except its first word
!*:p	Displays what !* substitutes
!x	Runs recent command in the bash history that begins with x

!x:p	Displays the x command and adds it as the recent command in history
!$	Same as OPTION+., brings forth last argument of the previous command
!^	Substitutes first argument of last command in the current command
!$:p	Displays the word that !$ substitutes
^123^abc	Replaces 123 with abc
!n:m	Repeats argument within a range (i.e, m 2-3)
!fi	Repeats latest command in history that begins with fi
!n	Run nth command from the bash history
!n:p	Prints the command !n executes
!n:$	Repeat arguments from the last command (i.e, from argument n to $)
Bash Editing	
Ctrl + U	Deletes before the cursor until the start of the command
Ctrl + K	Deletes after the cursor until the end of the command
Ctrl + W	Removes the command/argument before the cursor
Ctrl + D	Removes the character under the cursor
Ctrl + H	Removes character before the cursor
Alt + D	Removes from the character until the end of the word
Alt + Backspace	Removes from the character until the start of the word

Alt + . / Esc+.	Uses last argument of previous command
Alt + <	Moves to the first line of the bash history
Alt + >	Moves to the last line of the bash history
Esc + T	Switch between last two words before cursor
Alt + T	Switches current word with the previous
Bash Information	
TAB	Autocompletes the command or file/directory name
~TAB TAB	List all Linux users
Ctrl + I	Completes the command like TAB
Alt + ?	Display files/folders in the current path for help
Alt + *	Display files/folders in the current path as parameter

31: Downloading files from the web

wget is a free utility for non-interactive download of files from the web. It supports HTTP, HTTPS, and FTP protocols, as well as retrieval through HTTP proxies. With wget, you can download entire websites, parts of websites, or individual files from websites. It's known for its

ability to resume interrupted downloads, which can be very useful in scripts or automated processes.

```
$ wget
https://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/ALL.chrMT.phase
3_callmom-v0_4.20130502.genotypes.vcf.gz
```

32: File compression & Tarballs

ALL.chrMT.phase3_callmom-v0_4.20130502.genotypes.vcf.gz is a gzipped compressed VCF file.

`less` can view many compressed files natively.

```
$ less ALL.chrMT.phase3_callmom-v0_4.20130502.genotypes.vcf.gz
```

unwrap the lines

```
$ less -S ALL.chrMT.phase3_callmom-v0_4.20130502.genotypes.vcf.gz
```

decompress a gzipped file:

```
$ gunzip ALL.chrMT.phase3_callmom-v0_4.20130502.genotypes.vcf.gz
```

gzip compress a file

```
$ gzip ALL.chrMT.phase3_callmom-v0_4.20130502.genotypes.vcf
```

Other tools:
`zip` to compress a zip file
`unzip` to decompress a zip file

file command to see if a file is compressed

The Linux ‘tar’ stands for tape archive, which is used to create Archive and extract the Archive files. tar command in Linux is one of the important commands which provides archiving

functionality in Linux. We can use the Linux tar command to create compressed or uncompressed Archive files and also maintain and modify them.

extract a tarball:

```
$ tar -zxvf file_name.tar.gz
```

33: File Permissions

The below is taken from: https://www.redhat.com/sysadmin/linux-file-permissions-explained

What are octal values?

When Linux file permissions are represented by numbers, it's called numeric mode. In numeric mode, a three-digit value represents specific file permissions (for example, 744.) These are called octal values. The first digit is for owner permissions, the second digit is for group permissions, and the third is for other users. Each permission has a numeric value assigned to it:

- r (read): 4
- w (write): 2
- x (execute): 1

In the permission value 744, the first digit corresponds to the user, the second digit to the group, and the third digit to others. By adding up the value of each user classification, you can find the file permissions.

For example, a file might have read, write, and execute permissions for its owner, and only read permission for all other users. That looks like this:

- Owner: rwx = 4+2+1 = 7
- Group: r-- = 4+0+0 = 4
- Others: r-- = 4+0+0 = 4

The results produce the three-digit value 744.

What do Linux file permissions actually do?

I've talked about how to view file permissions, who they apply to, and how to read what permissions are enabled or disabled. But what do these permissions actually do in practice?

Read (r)
Read permission is used to access the file's contents. You can use a tool like cat or less on the file to display the file contents. You could also use a text editor like Vi or view on the file to display the contents of the file. Read permission is required to make copies of a file, because you need to access the file's contents to make a duplicate of it.

Write (w)
Write permission allows you to modify or change the contents of a file. Write permission also allows you to use the redirect or append operators in the shell (> or >>) to change the contents of a file. Without write permission, changes to the file's contents are not permitted.

Execute (x)
Execute permission allows you to execute the contents of a file. Typically, executables would be things like commands or compiled binary applications. However, execute permission also allows someone to run Bash shell scripts, Python programs, and a variety of interpreted languages.

34: For Loops

A for loop is a control flow statement that allows code to be executed repeatedly based on a specified condition. In Bash scripting, for loops are used to iterate over a series of values or elements of an array, or the output of a command.

Here's a general syntax for a for loop in Bash:

psuedocode

```
for VARIABLE in ITEM_1 ITEM_2 ... ITEM_N
do
    COMMANDS
done
```

you can also use shell expansion:

```
for VARIABLE in {1..10}
do
    COMMANDS
echo $VARIABLE


done
```

An example:

```
for i in {1..10}; do echo "The number is $i"; done

The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
The number is 10
```

The `for` loop iterates over the list of numbers 1 through 10. The variable i takes on the value of the current item in the list during each iteration of the loop. The echo command is used to print the current value of i to the console.

`for` loops are a fundamental part of Bash scripting, and understanding how to use them will help you automate repetitive tasks and manage collections of data.

34: While Loops

A while loop is another control flow statement used in Bash scripting to execute a block of commands as long as a specified condition is true. Here's the general syntax of a while loop in Bash:

pseudocode:

```
while [ CONDITION ]
do
    COMMANDS
```

```
done
```

CONDITION is a test or comparison, such as checking whether a variable is less than a certain value.
COMMANDS are the commands that will be executed in each iteration of the loop as long as the condition is true.
Here's a simple example of a while loop in Bash that prints numbers 1 to 5:

```
#!/bin/bash

# Initialize a variable
count=1

# A simple while loop that prints numbers 1 to 5
while [ $count -le 5 ]
do
    echo "Number $count"
    ((count++))  # Increment the count variable
done
```

In this script:

A variable named count is initialized with a value of 1.
The while loop checks the condition $count -le 5 (which means "while count is less than or equal to 5").

Inside the loop, the echo command prints (with `echo`) the current value of count to the console.
The ((count++)) statement increments the value of count by 1 in each iteration of the loop.
When you run this script, you will see the following output:

```
Number 1
Number 2
Number 3
Number 4
Number 5
```

This while loop continues executing as long as the value of count remains 5 or less, and it stops once count becomes greater than 5. The while loop is useful in situations where you want to repeat a block of commands indefinitely, or until a certain condition is met.

If you were to run this without a script (i.e. directly at the prompt), here's what it'd look like:

bash code

```
count=1; while [ $count -le 5 ]; do echo "Number $count";  ((count++));
done
```

While can also be used to iterate over a file.

pseudocode:

```
cat FILENAME | while read line; do COMMAND; done
```

bash code:

```
cat opening_lines.txt | while read column1 column2 column3; do echo
$column1 $colum2 $column3 ; done
```

Tip: In Bash, the read command is commonly used to read a line of input from standard input or from a file. When used in conjunction with a loop, read can be employed to process a delimited file line by line, splitting each line into fields based on a specified delimiter.

Here's how you might use read to parse columns in a delimited file:

Specify a Delimiter:

Use the -a option with read to read input into an array, and specify the delimiter using the IFS (Internal Field Separator) variable.
Read Lines from a File:

Use a while loop to read lines from the file, and read to split each line into array elements.
Access Individual Columns:

Access individual columns by referencing the array indices.
Here's an example that demonstrates how to use read to parse a CSV file with comma as the delimiter:

bash script

```
#!/bin/bash

# Define the delimiter
IFS=','

# Read the file line by line
cat opening_lines.txt | while read -a lineArray
do
    # Access columns by index
    column1=${lineArray[0]}
    column2=${lineArray[1]}
    column3=${lineArray[2]}

    echo "Column 1: $column1, Column 2: $column2, Column 3: $column3"

done
```

In this example:

IFS=',' sets the delimiter to comma.
while read -a lineArray reads each line from input.csv into an array named lineArray.
The array indices are used to access individual columns: column1=${lineArray[0]}, column2=${lineArray[1]}, and column3=${lineArray[2]}.
The echo command prints the values of the columns to the console.
In this way, the read command, along with a while loop, can be used to process a delimited file, parsing the columns and performing actions on the data as needed.

35: ~/.bashrc

The ~/.bashrc file is a script file that Bash runs every time it is started interactively, allowing you to configure your shell environment by setting variables, aliases, functions, and other shell attributes. The tilde ~ represents the user's home directory, so ~/.bashrc refers to a file named .bashrc located in the home directory of the user.

Here are some key points regarding the ~/.bashrc file:

- Environment Configuration:
    - You can use this file to set environment variables which will be available to all terminal sessions.
    - For example, you can set the PATH environment variable to include additional directories where executable programs are located.

- Aliases and Functions:
    - You can define aliases and functions to create shortcuts or to encapsulate complex command sequences.
    - For example, you might define an alias to shorten a frequently used command, making it quicker to type.

- Shell Options:
    - You can set shell options to modify the behavior of the shell.
    - For example, you might set options to control how the shell handles errors or job control.

- Prompt Customization:
    - You can customize the appearance of the shell prompt, including its color and the information it displays.
    -

- Script Execution:
    - You can use the ~/.bashrc file to execute scripts automatically whenever a new shell session is started.

Here's an example snippet from a ~/.bashrc file:

bash code

```
# Set PATH
export PATH="$HOME/software/:$PATH"

# Define alias
alias l='ls -ltrh'

# Customize the prompt
PS1='[\u@\h \W]\$ '
```

Miniconda installation

Conda in a virtual environment manager. It can be used to install software.

```
mkdir -p /nas4/$USER/software/miniconda3
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
-O /nas4/$USER/software/miniconda.sh
bash /nas4/$USER/software/miniconda.sh -b -u -p
/nas4/$USER/software/miniconda3
```

make a backup of ~/.bashc because conda will modify it. Whenever a software is going to modify my .bashrc, I usually create a backup of it.

```
cp ~/.bashrc ~/.bashrc.bak
```

run the conda init (to initialize conda)

```
/nas4/$USER/software/miniconda3/bin/conda init bash
```

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 10 lines of a file (using head and tail commands):

```
$ tail -n 20 file.txt | head
```

- Show 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 ‘transliter ate’):

```
$ 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
```