https://ryanstutorials.net/linuxtutorial/

Section 1: Command Line - Introduction

The command line is an interesting beast, and if you've not used one before, can be a bit daunting. Don't worry, with a bit of practice you'll soon come to see it as your friend. Don't think of it as leaving the GUI behind so much as adding to it.

- While you can leave the GUI altogether, most people open up a command line interface just as another window on their desktop (in fact you can have as many open as you like). This is also to our advantage as we can have several command lines open and doing different tasks in each at the same time. We can also easily jump back to the GUI when it suits us. Experiment until you find the setup that suits you best.
    - For Example: I will typically have 3 terminals open: 1 in which I do my working, another to bring up ancillary data and a final one for viewing Manual pages (more on these later).

Section 1: Command Line - What is it?

A command line, or terminal, is a text based interface to the system. You are able to enter commands by typing them on the keyboard and feedback will be given to you similarly as text.

The command line typically presents you with a prompt. As you type, it will be displayed after the prompt. Most of the time you will be issuing commands. Here is an example:

```
1. user@bash: ls -l /home/ryan
2. total 3
3. drwxr-xr-x  2 ryan users 4096 Mar 23 13:34 bin
4. drwxr-xr-x 18 ryan users 4096 Feb 17 09:12 Documents
5. drwxr-xr-x  2 ryan users 4096 May 05 17:25 public_html
6. user@bash:
```

Let's break it down:

- **Line 1** presents us with a prompt (*user@bash*). After that we entered a command (*ls*). Typically a command is always the first thing you type. After that we have what are referred to as command line arguments (*-l /home/ryan*). Important to note, these are separated by spaces (there must be a space between the command and the first command line argument also). The first command line argument (*-l*) is also referred to as an option. Options are typically used to modify the behaviour of the command. Options are usually listed before other arguments and typically start with a dash (-).
- **Lines 2 - 5** are output from running the command. Most commands produce output and it will be listed straight under the issuing of the command. Other commands just perform their task and don't display any information unless there was an error.
- **Line 6** presents us with a prompt again. After the command has run and the terminal is ready for you to enter another command the prompt will be displayed. If no prompt is displayed then the command may still be running (you will learn later how to deal with this).
- Your terminal probably WON'T have line numbers on it. I have just included them here to make it easier to refer to different parts of the material.

Section 1: Command Line - Opening a Terminal

Opening a terminal is fairly easy. I can't tell you exactly how to do it as every system is different but here are a few places to start looking:

- If you're on a **Mac** then you'll find the program Terminal under Applications -> Utilities. An easy way to get to it is the key combination 'command + space' which will bring up Spotlight, then start typing Terminal and it will soon show up.

- If on **Linux** then you will probably find it in Applications -> System or Applications -> Utilities. Alternatively you may be able to 'right-click' on the desktop and there may be an option 'Open in terminal'.

- If you are on **Windows** and intend to remotely log into another machine then you will need an SSH client. A rather good one is Putty (free) .

Section 1: Command Line - The Shell, Bash

Within a terminal you have what is known as a shell. This is a part of the operating system that defines how the terminal will behave and looks after running (or executing) commands for you. There are various shells available but the most common one is called bash which stands for Bourne again shell. This tutorial will assume you are using bash as your shell.

If you would like to know which shell you are using you may use a command called echo to display a system variable stating your current shell. echo is a command which is used to display messages.

```
user@bash: echo $SHELL
/bin/bash
user@bash:
```

As long as it prints something to the screen that ends in bash then all is good.

- **Mac** has the zsh Shell also known as the Z-shell

Section 1: Command Line - Shortcuts

The terminal may seem daunting but don't fret. Linux is full of shortcuts to help make your life easier. You'll be introduced to several of them throughout this tutorial. Take note of them as not only do they make your life easier, they often also save you from making silly mistakes such as typos.

- Here's your first shortcut. When you enter commands, they are actually stored in a history. You can traverse this history using the **up and down arrow keys**. So don't bother re-typing out commands you have previously entered, you can usually just hit the up arrow a few times. You can also edit these commands using the left and right arrow keys to move the cursor where you want.

Section 2: Basic Navigation - Introduction

In this section, we'll learn the basics of moving around the system. Many tasks rely on being able to get to, or reference the correct location in the system. As such, this stuff really forms the foundation of being able to work effectively in Linux. **Make sure you understand it well**.

Section 2: Basic Navigation - So Where are we?

The first command we are going to learn is pwd which stands for Print Working Directory. (You'll find that a lot of commands in linux are named as an abbreviation of a word or words describing them. This makes it easier to remember them.) The command does just that. It tells you what your current or present working directory is. **Give it a try now**.

```
user@bash: pwd
/home/ryan
user@bash:
```

A lot of commands on the terminal will rely on you being in the right location. As you're moving around, it can be easy to lose track of where you are at. **Make use of this command** often to remind yourself where you presently are.

Section 2: Basic Navigation - What’s in Our Current Location?

It's one thing to know where we are. Next we'll want to know what is there. The command for this task is ls. It's short for list. Let's give it a go.

```
user@bash: ls
bin Documents public_html
user@bash:
```

Whereas pwd is just run by itself with no arguments, ls is a little more powerful. We have run it here with no arguments in which case it will just do a plain listing of our current location. We can do more with ls however. Below is an outline of its usage:

- ls [options] [location]

In the above example, the square brackets ([ ]) mean that those items are optional, we may run the command with or without them. In the terminal below I have run ls in a few different ways to demonstrate.

```
user@bash: ls
bin Documents public_html
user@bash:
user@bash: ls -l
total 3
drwxr-xr-x  2 ryan users 4096 Mar 23 13:34 bin
drwxr-xr-x 18 ryan users 4096 Feb 17 09:12 Documents
drwxr-xr-x  2 ryan users 4096 May 05 17:25 public_html
user@bash:
user@bash: ls /etc
a2ps.cfg aliases alsa.d cups fonts my.conf systemd
...
user@bash: ls -l /etc
total 3
-rwxr-xr-x  2 root root 123 Mar 23 13:34 a2ps.cfg
-rwxr-xr-x 18 root root 78 Feb 17 09:12 aliases
drwxr-xr-x  2 ryan users 4096 May 05 17:25 alsa.d
...
user@bash:
```

Let's break it down:

- **Line 1** - We ran ls in its most basic form. It listed the contents of our current directory.
- **Line 4** - We ran ls with a single command line option (-l) which indicates we are going to do a long listing. A long listing has the following:
    - First character indicates whether it is a normal file (-) or directory (d)
    - Next 9 characters are permissions for the file or directory (we'll learn more about them in section 6).
    - The next field is the number of blocks (don't worry too much about this).
    - The next field is the owner of the file or directory (ryan in this case).
    - The next field is the group the file or directory belongs to (users in this case).
    - Following this is the file size.
    - Next up is the file modification time.
    - Finally we have the actual name of the file or directory.
- **Line 10** - We ran ls with a command line argument (/etc). When we do this it tells ls not to list our current directory but instead to list that directories contents.
- **Line 13** - We ran ls with both a command line option and argument. As such it did a long listing of the directory /etc.
- Lines 12 and 18 just indicate that I have cut out some of the commands normal output for brevity's sake. When you run the commands you will see a longer listing of files and directories.

Section 2: Basic Navigation - Paths

In the previous commands we started touching on something called a path. I would like to go into more detail on them now as they are important in being proficient with Linux. Whenever we refer to either a file or directory on the command line, we are in fact referring to a path. ie. A path is a means to get to a particular file or directory on the system.

Absolute & Relative Paths

There are 2 types of paths we can use, **absolute** and **relative**. Whenever we refer to a file or directory we are using one of these paths. Whenever we refer to a file or directory, we can, in fact, use either type of path (either way, the system will still be directed to the same location).

To begin with, we have to understand that the file system under linux is a hierarchical structure. At the very top of the structure is what's called the root directory. It is denoted by a single slash (/). It has subdirectories, they have subdirectories and so on. Files may reside in any of these directories.

- **Absolute paths** specify a location (file or directory) in relation to the root directory. You can identify them easily as they always begin with a forward slash (**/**)
- **Relative paths** specify a location (file or directory) in relation to where we currently are in the system. They will not begin with a slash.

Here's an example to illustrate:

```
1. user@bash: pwd
2. /home/ryan
3. user@bash:
4. user@bash: ls Documents
5. file1.txt file2.txt file3.txt
6. ...
7. user@bash: ls /home/ryan/Documents
8. file1.txt file2.txt file3.txt
9. ...
10.   user@bash:
```

- **Line 1 -** We ran pwd just to verify where we currently are.
- **Line 4 -** We ran ls providing it with a relative path. Documents is a directory in our current location. This command could produce different results depending on where we are. If we had another user on the system, bob, and we ran the command when in their home directory then we would list the contents of their Documents directory instead.
- **Line 7 -** We ran ls providing it with an absolute path. This command will provide the same output regardless of our current location when we run it.

More on Paths

You'll find that a lot of stuff in Linux can be achieved in several different ways. Paths are no different. Here are some more building blocks you may use to help build your paths:

- **~ (tilde)** - This is a shortcut for your home directory.
    - eg, if your home directory is /home/ryan then you could refer to the directory Documents with the path /home/ryan/Documents or ~/Documents
- **. (dot)** - This is a reference to your current directory.
    - eg in the example above we referred to Documents on line 4 with a relative path. It could also be written as ./Documents (Normally this extra bit is not required but in later sections we will see where it comes in handy).
- **.. (dotdot)**- This is a reference to the parent directory. You can use this several times in a path to keep going up the hierarchy.
    - eg if you were in the path /home/ryan you could run the command ls ../../ and this would do a listing of the root directory.

So now you are probably starting to see that we can refer to a location in a variety of different ways. Some of you may be asking the question, which one should I use? The answer is that you can use any method you like to refer to a location. Whenever you refer to a file or directory on the command line you are actually referring to a path and your path can be constructed using any of these elements. The best approach is whichever is the most convenient for you. Here are some examples:

```
user@bash: pwd
/home/ryan
user@bash:
user@bash: ls ~/Documents
file1.txt file2.txt file3.txt
...
user@bash: ls ./Documents
file1.txt file2.txt file3.txt
...
user@bash: ls /home/ryan/Documents
file1.txt file2.txt file3.txt
...
user@bash:
user@bash: ls ../../
bin boot dev etc home lib var
...
user@bash:
user@bash: ls /
bin boot dev etc home lib var
```

```
20.  ...
```

After playing about with these on the command line yourself they will start to make a bit more sense.

- **<u>Make sure you understand how all of these elements of building a path work</u>** as you'll use all of them in future sections.

Section 2: Basic Navigation - Let’s Move Around a Bit

In order to move around in the system we use a command called cd which stands for change directory. It works as follows:

- cd [location]

Shortcut:

- If you run the command cd without any arguments then it will always take you back to your home directory.

The command cd may be run without a location as we saw in the shortcut above but usually will be run with a single command line argument which is the location we would like to change into. The location is specified as a path and as such may be specified as either an absolute or relative path and using any of the path building blocks mentioned above. Here are some examples:

```
user@bash: pwd
/home/ryan
user@bash: cd Documents
user@bash: ls
file1.txt file2.txt file3.txt
...
user@bash: cd /
user@bash: pwd
/
user@bash: ls
bin boot dev etc home lib var
...
user@bash: cd ~/Documents
user@bash: pwd
/home/ryan/Documents
user@bash: cd ../../
user@bash: pwd
/home
user@bash: cd
user@bash: pwd
/home/ryan
```

Tab Completion

Typing out these paths can become tedious. If you're like me, you're also prone to making typos. The command line has a nice little mechanism to help us in this respect. It's called Tab Completion.

- When you start typing a path (anywhere on the command line, you're not just limited to certain commands) you may hit the Tab key on your keyboard at any time which will invoke an auto complete action. If nothing happens then that means there are several possibilities. If you hit Tab again it will show you those possibilities. You may then continue typing and hit Tab again and it will again try to auto complete for you.
    - It's kinda hard to demonstrate here so it's probably best if you try it yourself. If you start typing cd /hTab/<beginning of your username>Tab you'll get a feel for how it works.

Section 2: Basic Navigation - Summary

Stuff we learned:

- **pwd** - print working directory
    - i.e where we currently are in the file system

- **ls** - list the contents of a directory

- **cd** - change directories
    - i.e move to another directory

- **Relative Path** - A file or directory location relative to where we currently are in the file system.

- **Absolute Path** - A file or directory location in relation to the root of the file system.

Section 3: More About Files - Introduction

After the previous section I'm sure you're keen and eager to get stuck into some more commands and start doing some actual playing about with the system. We will get to that shortly but first we need to cover some theory so that when we do start playing with the system you can fully understand why it is behaving the way it is and how you can take the commands you learn even further. That is what this section and the next intend to do. After that it will start getting interesting, I promise.

Section 3: More About Files - Everything is a File

Ok, the first thing we need to appreciate with linux is that under the hood, everything is actually a file.

- A text file is a file
- A directory is a file
- Your keyboard is a file (one that the system reads from only)
- Your monitor is a file (one that the system writes to only) etc.

To begin with, this won't affect what we do too much but keep it in mind as it helps with understanding the behaviour of Linux as we manage files and directories.

Section 3: More About Files - Linux is an Extensionless System

This one can sometimes be hard to get your head around but as you work through the sections it will start to make more sense. A file extension is normally a set of 2 - 4 characters after a full stop at the end of a file, which denotes what type of file it is. The following are common extensions:

- file.exe - an executable file, or program
- file.txt - a plain text file
- file.png, file.gif, file.jpg - an image

In other systems such as Windows the extension is important and the system uses it to determine what type of file it is. Under Linux the system actually ignores the extension and looks inside the file to determine what type of file it is. So for instance I could have a file myself.png which is a picture of me. I could rename the file to myself.txt or just myself and Linux would still happily treat the file as an image file. As such it can sometimes be hard to know for certain what type of file a particular file is. Luckily there is a command called **file** which we can use to find this out.

- file [path]

Now you may be wondering why I specified the command line argument above as path instead of file. If you remember from the previous section, **whenever we specify a file or directory on the command line it is actually a path**. Also because directories (as mentioned above) are actually just a special type of file, it would be more accurate to say that a path is a means to get to a particular location in the system and that location is a file.

Section 3: More About Files - Linux is Case Sensitive

This is very important and a common source of problems for people new to Linux. Other systems such as Windows are case insensitive when it comes to referring to files. Linux is not like this. As such it is possible to have two or more files and directories with the same name but letters of different cases.

```
user@bash: ls Documents
FILE1.txt File1.txt file1.TXT
...
user@bash: file Documents/file1.txt
Documents/file1.txt: ERROR: cannot open 'file1.txt' (No such file or directory)
```

Linux sees these all as distinct and separate files.

Also be aware of case sensitivity when dealing with command line options. For instance with the command ls there are two options s and S both of which do different things. A common mistake is to see an option which is upper case but enter it as lower case and wonder why the output doesn't match your expectation.

Section 3: More About Files - Spaces in Names

Spaces in file and directory names are perfectly valid but we need to be a little careful with them. As you would remember, a space on the command line is how we separate items. They are how we know what is the program name and can identify each command line argument. If we wanted to move into a directory called Holiday Photos for example the following would not work.

```
user@bash: ls Documents
FILE1.txt File1.txt file1.TXT Holiday Photos
...
user@bash: cd Holiday Photos
bash: cd: Holiday: No such file or directory
```

What happens is that Holiday Photos is seen as two command line arguments. cd moves into whichever directory is specified by the first command line argument only. To get around this we need to identify to the terminal that we wish Holiday Photos to be seen as a single command line argument. There are two ways to go about this, either way is just as valid.

Quotes

The first approach involves using quotes around the entire item. You may use either single or double quotes (later on we will see that there is a subtle difference between the two but for now that difference is not a problem). Anything inside quotes is considered a single item.

```
user@bash: cd 'Holiday Photos'
user@bash: pwd
/home/ryan/Documents/Holiday Photos
```

Escape Characters

Another method is to use what is called an escape character, which is a backslash (\ ). What the backslash does is escape (or nullify) the special meaning of the next character.

```
user@bash: cd Holiday\ Photos
user@bash: pwd
/home/ryan/Documents/Holiday Photos
```

In the above example the space between Holiday and Photos would normally have a special meaning which is to separate them as distinct command line arguments. Because we placed a backslash in front of it, that special meaning was removed.

Short Cut

If you use Tab Completion before encountering the space in the directory name then the terminal will automatically escape any spaces in the name for you.

Section 3: More About Files - Hidden Files & Directories

Linux actually has a very simple and elegant mechanism for specifying that a file or directory is hidden. If the file or directory name begins with a . (full stop) then it is considered to be hidden. You don't even need a special command or action to make a file hidden. Files and directories may be hidden for a variety of reasons. Configuration files for a particular user (which are normally stored in their home directory) are hidden for instance so that they don't get in the way of the user doing their everyday tasks.

To make a file or directory hidden all you need to do is create the file or directory with it's name beginning with a . or rename it to be as such. Likewise you may rename a hidden file to remove the . and it will become unhidden. The command ls which we have seen in the previous section will not list hidden files and directories by default. We may modify it by including the command line option -a so that it does show hidden files and directories.

```
user@bash: ls Documents
FILE1.txt File1.txt file1.TXT
...
user@bash: ls -a Documents
. .. FILE1.txt File1.txt file1.TXT .hidden .file.txt
...
```

In the above example you will see that when we listed all items in our current directory the first two items were . and ..

Section 3: More About Files - Summary

Stuff we learned:

- file - obtain information about what type of file a file or directory is.
- ls -a - List the contents of a directory, including hidden files.
- Everything is a file in Linux!
- Linux is an extensionless system
- Linux is Case Sensitive

Section 4: Manual Pages - Introduction

The Linux command line offers a wealth of power and opportunity. If your memory is like mine then you find it hard to remember a large number of details. Fortunately for us there is an easy to use resource that can inform us about all the great things we can do on the command line. That's what we're going to learn about in this section. I know you're keen and eager to get stuck into doing stuff, and we'll get started on that in the next section, I promise, first we need to learn how to use Manual pages however.

Section 4: Manual Pages - So What Are They Exactly

The manual pages are a set of pages that explain every command available on your system including what they do, the specifics of how you run them and what command line arguments they accept. Some of them are a little hard to get your head around but they are fairly consistent in their structure so once you get the hang of it it's not too bad. You invoke the manual pages with the following command:

- **man** <command to look up>

```
1. user@bash: man ls
2. Name
3.     ls - list directory contents
4.
5. Synopsis
6.     ls [option] ... [file] ...
7.
8. Description
9.     List information about the FILEs (the current directory by
   default). Sort entries alphabetically if none of -cftuvSUX nor
   --sort is specified.
10.
11.       Mandatory arguments to long options are mandatory for
   short options too.
12.
13.       -a, --all
14.           do not ignore entries starting with .
15.
16.       -A, --almost-all
17.           do not list implied . and ..
```

Let's break it down:

- **Line 3** tells us the actual command followed by a simple one line description of it's function.
- **Lines 6** is what's called the synopsis. This is really just a quick overview of how the command should be run. Square brackets ([ ]) indicate that something is optional. (option on this line refers to the command line options listed below the description)
- **Line 9** presents us with a more detailed description of the command.
- **Line 11 onwards** Below the description will always be a list of all the command line options that are available for the command.

Tip: To exit the man pages press 'q' for quit.

Section 4: Manual Pages - Searching

It is possible to do a keyword search on the Manual pages. This can be helpful if you're not quite sure of what command you may want to use but you know what you want to achieve. To be effective with this approach, you may need a few goes. It is not uncommon to find that a particular word exists in many manual pages.

- man -k <search term>

If you want to search within a manual page this is also possible. To do this, at the same time you’re in the particular manual page you would like to search press forward slash '/' followed by the term you would like to search for and hit 'enter' If the term appears multiple times you may cycle through them by pressing the 'n' button for next.

Section 4: Manual Pages - More on the Running of Commands

A lot of being proficient at Linux is knowing which command line options we should use to modify the behaviour of our commands to suit our needs. A lot of these have both a longhand and shorthand version. eg. Above you will notice that to list all directory entries (including hidden files) we can use the option -a or --all (remember from last section what files and directories beginning with a . are?). The long hand is really just a more human readable form. You may use either, they both do the same thing. One advantage of using longhand is that it can be easier for you to remember what your commands are doing. One advantage of using shorthand is that you can chain multiple together easier.

```
user@bash: pwd
/home/ryan
user@bash: ls -a
user@bash: ls --all
user@bash: ls -alh
```

As you can see, long hand command line options begin with two dashes (--) and shorthand options begin with a single dash (-). When we use a single dash we may invoke several options by placing all the letters representing those options together after the dash. (There are a few instances where a particular option requires an argument to go with it and those options generally have to be placed separately along with their corresponding argument. Don't worry too much about these special cases for now though. We'll point them out as we encounter them.

Section 4: Manual Pages - Summary

Stuff we learned:

- man <command> - Look up the manual page for a particular command.

- man -k <search terms> - Do a keyword search for all manual pages containing the given search term.

- /<term> - Within a manual page, perform a search for 'term'

- n - After performing a search within a manual page, select the next found item.

- The man pages are your friend
  - Instead of trying to remember everything, instead remember you can easily look stuff up in the man pages.

Section 5: File Manipulation - Introduction

We've got some basic foundation stuff out of the way. Now we can start to play around. To begin with we'll learn to make some files and directories and move them around. Future sections will deal with putting content in them and more interesting manipulation.

Section 5: File Manipulation - Making a Directory

Linux organises it's file system in a hierarchical way. Over time you'll tend to build up a fair amount of data (storage capacities are always increasing). It's important that we create a directory structure that will help us organise that data in a manageable way. Develop the habit of organising your stuff into an elegant file structure now and you will thank yourself for years to come. Creating a directory is pretty easy. The command we are after is mkdir which is short for Make Directory.

- Mkdir [options] <Directory>

In its most basic form we can run **mkdir** supplying only a directory and it will create it for us.

```
1. user@bash: pwd
2. /home/ryan
3. user@bash:
4. user@bash:ls
5. bin Documents public_html
6. user@bash:
7. user@bash: mkdir linuxtutorialwork
8. user@bash:
9. user@bash: ls
10.   bin Documents linuxtutorialwork public_html
```

Let's break it down:

- **Line 1** Let's start off by making sure we are where we think we should be. (In the example above I am in my home directory)

- **Lines 2** We'll do a quick listing so we know what is already in our directory.

- **Line 7** Run the command mkdir and create a directory linuxtutorialwork (a nice place to put further work we do relating to this tutorial just to keep it separate from our other stuff).

Remember that when we supply a directory in the above command we are actually supplying a path. Is the path we specified relative or absolute?

Here are a few more examples of how we can supply a directory to be created:

- mkdir /home/ryan/foo
- mkdir ./blah
- mkdir ../dir1
- mkdir ~/linuxtutorialwork/dir2

There are a few useful options available for mkdir. Can you remember where we may go to find out the command line options a particular command supports?

- **-p** which tells mkdir to make parent directories as needed
- **-v** which makes mkdir tell us what it is doing

mkdir with the -p option

```
user@bash: mkdir -p linuxtutorialwork/foo/bar
user@bash:
user@bash: cd linuxtutorialwork/foo/bar
user@bash: pwd
/home/ryan/linuxtutorialwork/foo/bar
```

And now the same command but with the -v option

```
user@bash: mkdir -pv linuxtutorialwork/foo/bar
mkdir: created directory 'linuxtutorialwork/foo'
mkdir: created directory 'linuxtutorialwork/foo/bar'
user@bash:
user@bash: cd linuxtutorialwork/foo/bar
user@bash: pwd
/home/ryan/linuxtutorialwork/foo/bar
```

Section 5: File Manipulation - Removing a Directory

Creating a directory is pretty easy. Removing or deleting a directory is easy too. One thing to note, however, is that there is no undo when it comes to the command line on Linux (Linux GUI desktop environments typically do provide an undo feature but the command line does not). Just be careful with what you do. The command to remove a directory is rmdir, short for remove directory:

- rmdir [options] <Directory>

Two things to note. Firstly, rmdir supports the -v and -p options similar to mkdir. Secondly, a directory must be empty before it may be removed (later on we'll see a way to get around this).

```
user@bash: rmdir linuxtutorialwork/foo/bar
user@bash:
user@bash: ls linuxtutorialwork/foo
```

Section 5: File Manipulation - Create a Blank File

A lot of commands that involve manipulating data within a file have the nice feature that they will create a file automatically if we refer to it and it does not exist. In fact we can make use of this very characteristic to create blank files using the command touch:

- touch [options] <filename>

```
user@bash: pwd
/home/ryan/linuxtutorialwork
user@bash:
user@bash: ls
foo
user@bash:
user@bash: touch example1
user@bash:
user@bash: ls
example1 foo
```

touch is actually a command we may use to modify the access and modification times on a file (normally not needed but sometimes when you're testing a system that relies on file access or modification times it can be useful). What we are taking advantage of here is that if we touch a file and it does not exist, the command will do us a favor and automatically create it for us.

Section 5: File Manipulation - Copying a File or Directory

There are many reasons why we may want to make a duplicate of a file or directory. Often before changing something, we may wish to create a duplicate so that if something goes wrong we can easily revert back to the original. The command we use for this is cp which stands for copy:

- cp [options] <source> <destination>

There are quite a few options available to cp. I'll introduce one of them further below but it's worth checking out the man page for cp to see what else is available.

```
user@bash: ls
example1 foo
user@bash:
user@bash: cp example1 barney
user@bash: ls
barney example1 foo
```

Note that both the source and destination are paths. This means we may refer to them using both absolute and relative paths. Here are a few examples:

- cp /home/ryan/linuxtutorialwork/example2 example3
    - cp example2 ../../backups
        - cp example2 ../../backups/example4
            - cp /home/ryan/linuxtutorialwork/example2 /otherdir/foo/example5

When we use cp the destination can be a path to either a file or directory. If it is to a file (such as examples 1, 3 and 4 above) then it will create a copy of the source but name the copy the filename specified in destination. If we provide a directory as the destination then it will copy the file into that directory and the copy will have the same name as the source.
In it's default behaviour cp will only copy a file (there is a way to copy several files in one go but we'll get to that in section 6. Wildcards). Using the -r option, which stands for recursive, we may copy directories. Recursive means that we want to look at a directory and all files and directories within it, and for subdirectories, go into them and do the same thing and keep doing this.

```
user@bash: ls
barney example1 foo
user@bash: cp foo foo2
cp: omitting directory 'foo'
user@bash: cp -r foo foo2
user@bash: ls
barney example1 foo foo2
```

In the above example any files and directories within the directory foo will also be copied to foo2.

Section 5: File Manipulation - Moving a File or Directory

To move a file we use the command mv which is short for move. It operates in a similar way to cp. One slight advantage is that we can move directories without having to provide the -r option:

- mv [options] <source> <destination>

```
1. user@bash: ls
2. barney example1 foo foo2
3. user@bash: mkdir backups
4. user@bash: mv foo2 backups/foo3
5. user@bash: mv barney backups/
6. user@bash: ls
7. backups example1 foo
```

Let's break it down:

- **Line 3** We created a new directory called backups.
- **Line 4** We moved the directory foo2 into the directory backups and renamed it as foo3
- **Line 7** We moved the file barney into backups. As we did not provide a destination name, it kept the same name.
    - Note that again the source and destination are paths and may be referred to as either absolute or relative paths.

Renaming Files and Directories

Now just as above with the command touch, we can use the basic behaviour of the command mv in a creative way to achieve a slightly different outcome. Normally mv will be used to move a file or directory into a new directory. As we saw on line 4 above, we may provide a new name for the file or directory and as part of the move it will also rename it. Now if we specify the destination to be the same directory as the source, but with a different name, then we have effectively used mv to rename a file or directory.

```
1. user@bash: ls
2. backups example1 foo
3. user@bash: mv foo foo3
4. user@bash: ls
5. backups example1 foo3
6. user@bash: cd ..
7. user@bash: mkdir linuxtutorialwork/testdir
8. user@bash: mv linuxtutorialwork/testdir
   /home/ryan/linuxtutorialwork/fred
9. user@bash: ls linuxtutorialwork
10.   backups example1 foo3 fred
```

Let's break it down:

- **Line 3** We renamed the file foo to be foo3 (both paths are relative).

- **Line 6** We moved into our parent directory. This was done only so in the next line we can illustrate that we may run commands on files and directories even if we are not currently in the directory they reside in.
- **Line 8** We renamed the directory testdir to fred (the source path was a relative path and the destination was an absolute path).

Section 5: File Manipulation - Removing a File & Non-Empty Directories

As with rmdir, removing a file is an action that may not be undone so be careful. The command to remove or delete a file is rm which stands for remove:

- rm [options] <file>

```
user@bash: ls
backups example1 foo3 fred
user@bash: rm example1
user@bash: ls
backups foo3 fred
```

Removing Non-Empty Directories

Like several other commands introduced in this section, rm has several options that alter it's behaviour. I'll leave it up to you to look at the man page to see what they are but I will introduce one particularly useful option which is -r. Similar to cp it stands for recursive. When rm is run with the -r option it allows us to remove directories and all files and directories contained within.

```
user@bash: ls
backups foo3 fred
user@bash: rmdir backups
rmdir: failed to remove 'backups': Directory not empty
user@bash: rm backups
rm: cannot remove 'backups': Is a directory
user@bash: rm -r backups
user@bash: ls
foo3 fred
```

A good option to use in combination with r is i which stands for interactive. This option will prompt you before removing each file and directory and give you the option to cancel the command.

Section 5: File Manipulation - One Final Note

I know I've stressed the point but by now hopefully you can appreciate that whenever we refer to a file or directory on the command line it is in fact a path. As such it may be specified as either an absolute or relative path. This is pretty much always the case so remember this important point. In future sections I will not keep reminding you of this and the given examples will usually not illustrate this. Remember to experiment with both absolute and relative paths in the commands as sometimes they give subtle but useful differences in output. (The output of the command may be slightly different but the action of the command will always be the same.) (I will illustrate an example of this in the section on Wildcards)

Section 5: File Manipulation - Summary

Stuff we learned:

- mkdir - Make Directory
    - i.e. Create a directory

- rmdir - Remove Directory
    - i.e. Delete a directory

- touch - Create a Blank File

- cp - Copy
    - i.e. Copy a file or directory

- mv - Move
    - i.e. Move a file or directory (Can also be used to rename)

- rm - Remove
    - i.e. Delete a file

- No undo
    - The Linux command line does not have an undo feature.
    - **<u>Perform destructive actions carefully.</u>**

- Command line options
    - Most commands have many useful command line options.
    - Make sure you skim the man page for new commands so you are familiar with what they can do and what is available.

Section 6: Vi Text Editor - Introduction

Master the Vi text editor and learn how to make complex edits on your files with less time and effort.

- Vi is a text editor that is most likely very different to any editor you have used before.
- It will take a while to get your head around but once you do you will realise it is actually quite powerful.
    - It's kinda like touch typing, initially learning is awkward but once you get the hang of it you will not want to go back.
    - Even if you don't use Vi all the time you will definitely find that the work patterns you develop in learning the editor can be transferred easily to other programs and to great effect.

This section and the next few sections are actually forming the foundation for the last few sections where we will put them all together and start doing some really funky stuff. Vi is a very powerful tool.

- In this section my aim is not to cover everything that Vi can do but to get you up and running with the basics.
- At the end of the section I'll provide some links to resources where you can learn Vi further. I highly recommend you look into a few of them.

Section 6: Vi Text Editor - Command Line Editor

Vi is a command line text editor. As you would be quite aware now, the command line is quite a different environment to your GUI. It's a single window with text input and output only. Vi has been designed to work within these limitations and many would argue, is actually quite powerful as a result. Vi is intended as a plain text editor (similar to Notepad on Windows, or Textedit on Mac) as opposed to a word processing suite such as Word. It does, however, have a lot more power compared to Notepad or Textedit.

- Everything in Vi is done via the keyboard, get rid of the mouse.

There are two modes in Vi. Insert (or Input) mode and Edit mode. In input mode you may input or enter content into the file. In edit mode you can move around the file, perform actions such as deleting, copying, search and replace, saving etc. A common mistake is to start entering commands without first going back into edit mode or to start typing input without first going into insert mode. If you do either of these it is generally easy to recover so don't worry too much.

- When we run vi we normally issue it with a single command line argument which is the file you would like to edit:
    - vi <file>

If you forget to specify a file then there is a way to open a file within vi but it is easiest to just quit vi and have another go. Also remember that when we specify the file it can be with either an absolute or relative path. Let’s edit our first file:

```
user@bash: vi firstfile
```

When you run the command above, it opens up the file. If the file does not exist then it will create it for you then open it up. (no need to touch files before editing them) Once you enter vi it will look something like this (though depending on what system you are on it may look slightly different).

```
~
~
~
~
~
"firstfile" [New File]
```

You always start off in edit mode so the first thing we are going to do is switch to insert mode by pressing i. You can tell when you are in insert mode as the bottom left corner will tell you.

```
~
~
~
~
~
-- INSERT --
```

Now type in a few lines of text and press Esc which will take you back to edit mode.

Section 6: Vi Text Editor - Saving and Exiting

There are a few ways to go about doing this. They all do essentially the same thing so pick whichever way you prefer. For all of these, make sure you are in edit mode first:

- ZZ (Note: capitals) - Save and exit
- :q! - discard all changes, since the last save, and exit
- :w - save file but don't exit
- :wq - again, save and exit

Most commands within vi are executed as soon as you press a sequence of keys. Any command beginning with a colon (:) requires you to hit <enter> to complete the command.

Tip:
If you are unsure if you are in edit mode or not you can look at the bottom left corner. As long as it doesn't say INSERT you are fine. Alternatively you can just press Esc to be sure. If you are already in edit mode, pressing Esc does nothing so you won't do any harm.

Section 6: Vi Text Editor - Other Ways to View Files

vi allows us to edit files. If we wanted, we could use it to view files as well, but there are two other commands which are a bit more convenient for that purpose. The first one is cat which actually stands for concatenate. It's main purpose is to join files together but in its most basic form it is useful for just viewing files.

- cat <file>

Tip:

If you accidentally run cat without giving it a command line argument you will notice that the cursor moves to the next line and then nothing happens. Because we didn't specify a file, cat instead reads from something called STDIN (which we'll learn about in the section 'Piping and redirection' which defaults to the keyboard. If you type something then hit <enter> you will see cat mirror your input to the screen. To get out of here you may press <Ctrl> + c which is the universal signal for Cancel in Linux.

- In fact, whenever you get in trouble you can generally press <Ctrl> + c to get yourself out of trouble.

```
user@bash: cat firstfile
here you will see
whatever content you
entered in your file
user@bash:
```

This command is nice when we have a small file to view but if the file is large then most of the content will fly across the screen and we'll only see the last page of content. For larger files there is a better suited command which is less.

- less <file>

less allows you to move up and down within a file using the arrow keys. You may go forward a whole page using the SpaceBar or back a page by pressing b. When you are done you can press q for quit.

Section 6: Vi Text Editor - Navigating a File in Vi

In insert mode you may use the arrow keys to move the cursor around then you can hit Esc to go back to edit mode.

Below are some of the many commands you may enter to move around the file:

- **Arrow keys** - move the cursor around
- **j**, **k**, **h**, **l** - move the cursor down, up, left and right (similar to the arrow keys)
- **^** (caret) - move cursor to beginning of current line
- **$** - move cursor to end of the current line
- **nG** - move to the nth line (eg 5G moves to 5th line)
- **G** - move to the last line
- **w** - move to the beginning of the next word
- **nw** - move forward n word (eg 2w moves two words forwards)
- **b** - move to the beginning of the previous word
- **nb** - move back n word
- **{** - move backward one paragraph
- **}** - move forward one paragraph

Tip:

If you type **:set nu** in edit mode within vi it will enable line numbers.

Section 6: Vi Text Editor - Deleting Content

We just saw that if we want to move around in vi there are quite a few options available to us. Several of them also allow us to precede them with a number to move that many times. Deleting works similar to movement, in fact several delete commands allow us to incorporate a movement command to define what is going to be deleted.

Below are some of the many ways in which we may delete content within vi:

- **x -** delete a single character

- **nx** - delete n characters
  - i.e. 5x deletes five characters

- **dd** - delete the current line

- **dn** - d followed by a movement command. Delete to where the movement command would have taken you.
  - i.e. d5w means delete 5 words

Section 6: Vi Text Editor - Undoing

Undoing changes in vi is fairly easy. It is the character **u**.

- **u** - Undo the last action (you may keep pressing u to keep undoing)
- **U** (Note: capital) - Undo all changes to the current line

Section 6: Vi Text Editor - Taking it Further

We can now insert content into a file, move around the file, delete content and undo it then save and exit. You can now do basic editing in vi. This is just touching the surface of what vi can do however. I won't go into all the details here (I think I've thrown enough at you already) but I will give you a few things you may want to look into to further your expertise in vi. A basic search in your search engine of choice for vi <insert concept here> will find you many pages with useful information. There are many vi cheat sheets out there too which list all the commands available to you.

- copy and paste
- search and replace
- buffers
- markers
- ranges
- settings

Section 6: Vi Text Editor - Summary

Stuff we learned:

- vi - Edit/Create a file
- cat - View a file
- less - Convenient way of viewing large files
- No mouse
    - vi is a text editor where everything is done on the keyboard
- Edit commands
    - There are many of them. Practice and search on your favorite browser

Section 7: Wildcards - Introduction

In the section on File Manipulation we learnt about a few commands to do interesting things. The problem was that they all operated on a single file at a time, not very efficient. Now I'm going to introduce a means to play about with a set of files at once.

Section 7: Wildcards - What Are Wildcards?

Wildcards are a set of building blocks that allow you to create a pattern defining a set of files or directories. As you would remember, whenever we refer to a file or directory on the command line we are actually referring to a path. Whenever we refer to a path we may also use wildcards in that path to turn it into a set of files or directories.

Here is the basic set of wildcards:

- *** -** represents zero or more characters
- **?** - represents a single character
- **[]** - represents a range of characters

As a basic first example we will introduce the *. In the example below we will list every entry beginning with a b.

```
user@bash: pwd
/home/ryan/linuxtutorialwork
user@bash:
user@bash: ls
barry.txt blah.txt bob example.png firstfile foo1 foo2
foo3 frog.png secondfile thirdfile video.mpeg
user@bash:
user@bash: ls b*
barry.txt blah.txt bob
```

Section 7: Wildcards - Under the Hood

The mechanism here is actually kinda interesting. On first glance you may assume that the command above (ls) receives the argument b* then proceeds to translate that into the required matches. It is actually bash (The program that provides the command line interface) that does the translation for us. When we offer it this command it sees that we have used wildcards and so, before running the command (in this case ls) it replaces the pattern with every file or directory (ie path) that matches that pattern. We issue the command:

- ls b*

Then the system translates this into:

- ls barry.txt blah.txt bob

and then executes the program. The program never sees the wildcards and has no idea that we used them. This is funky as it means we can use them on the command line whenever we want. We are not limited to only certain programs or situations.

Section 7: Wildcards - Some More Examples

Some more examples to illustrate their behaviour. For all the examples below, assume we are in the directory linuxtutorialwork and that it contains the files as listed above. Also note that I'm using ls in these examples simply because it is a convenient way to illustrate their usage. Wildcards may be used with any command.

Every file with an extension of txt at the end. In this example we have used an absolute path. Wildcards work just the same if the path is absolute or relative.

```
user@bash: ls /home/ryan/linuxtutorialwork/*.txt
/home/ryan/linuxtutorialwork/barry.txt /home/ryan/linuxtutorialwork/blah.txt
user@bash:
```

Now let's introduce the ? operator. In this example we are looking for each file whose second letter is i. As you can see, the pattern can be built up using several wildcards.

```
user@bash: ls ?i*
firstfile video.mpeg
user@bash:
```

Or how about every file with a three letter extension. Note that video.mpeg is not matched as the path name must match the given pattern exactly.

```
user@bash: ls *.???
barry.txt blah.txt example.png frog.png
user@bash:
```

And finally the range operator ([ ]). Unlike the previous 2 wildcards which specified any character, the range operator allows you to limit to a subset of characters. In this example we are looking for every file whose name either begins with a s or v.

```
user@bash: ls [sv]*
secondfile video.mpeg
user@bash:
```

With ranges we may also include a set by using a hyphen. So for example if we wanted to find every file whose name includes a digit in it we could do the following:

```
user@bash: ls *[0-9]*
foo1 foo2 foo3
user@bash:
```

We may also reverse a range using the caret (^) which means look for any character which is not one of the following:

```
user@bash: ls [^a-k]*
secondfile thirdfile video.mpeg
user@bash:
```

Section 7: Wildcards - Some Real World Examples

The examples above illustrate how the wildcards work but you may be wondering what use they actually are. People use them everywhere and as you progress I'm sure you'll find many ways in which you can use them to make your life easier. Here are a few examples to give you a taste of what is possible. Remember, these are just a small sample of what is possible, and they can be used whenever you specify a path on the command line. With a little creative thinking you'll find they can be used in all manner of situations

Find the file type of every file in a directory:

```
user@bash: file /home/ryan/*
bin: directory
Documents: directory
frog.png: PNG image data
public_html: directory
user@bash:
```

Move all files of type either jpg or png (image files) into another directory:

```
user@bash: mv public_html/*.??g public_html/images/
user@bash:
```

Find out the size and modification time of the .bash_history file in every users home directory. (.bash_history is a file in a typical users home directory that keeps a history of commands the user has entered on the command line. Remember how the . means it is a hidden file?) As you can see in this example, we may use wildcards at any point in the path.

```
user@bash: ls -lh /home/*/.bash_history
-rw------- 1 harry users 2.7K Jan 4 07:32 /home/harry/.bash_history
-rw------- 1 ryan users 3.1K Jun 12 21:16 /home/ryan/.bash_history
user@bash:
```

Section 7: Wildcards - Summary

Stuff we learned:

- Anywhere in any PATH - Wildcards may be used at any part of a path.

- Wherever a PATH is used - Because wildcard substitution is done by the system, not the command, they may be used wherever a path is used.

Section 8: Permissions - Introduction

In this section we'll learn about how to set Linux permissions on files and directories. Permissions specify what a particular person may or may not do with respect to a file or directory. As such, permissions are important in creating a secure environment. For instance you don't want other people to be changing your files and you also want system files to be safe from damage (either accidental or deliberate). Luckily, permissions in a Linux system are quite easy to work with.

Section 8: Permissions - So What Are They?

Linux permissions dictate 3 things you may do with a file, read, write and execute. They are referred to in Linux by a single letter each:

- r (Read) - you may view the contents of the file.

- w (Write) - you may change the contents of the file.

- x (Execute) - you may execute or run the file if it is a program or script.

For every file we define 3 sets of people for whom we may specify permissions:

- owner - A single person who owns the file.
    - Typically the person who created the file but ownership may be granted to someone else by certain users

- group - Every file belongs to a single group.

- others - Everyone else who is not in the group or the owner.

Three permissions and three groups of people. That's about all there is to permissions really. Now let's see how we can view and change them.

Section 8: Permissions - View Permissions

To view permissions for a file we use the long listing option for the command ls.

- ls -l [path]

```
user@bash: ls -l /home/ryan/linuxtutorialwork/frog.png
-rwxr----x 1 harry users 2.7K Jan 4 07:32 /home/ryan/linuxtutorialwork/frog.png
user@bash:
```

In the example above, the first 10 characters of the output are what we look at to identify permissions:

- The first character identifies the file type.
  - If it is a dash (-) then it is a normal file.
  - If it is a d then it is a directory.

- The following 3 characters represent the permissions for the owner.
  - A letter represents the presence of a permission
  - A dash (-) represents the absence of a permission.
    - In this example the owner has all permissions (read, write and execute).

- The following 3 characters represent the permissions for the group.
  - In this example the group has the ability to read but not write or execute.
    - **Note** that the order of permissions is always read, then write then execute.

- Finally the last 3 characters represent the permissions for others (or everyone else).
  - In this example they have the execute permission and nothing else.

Section 8: Permissions - Change Permissions

To change permissions on a file or directory we use a command called chmod It stands for change file mode bits.

- chmod [permissions] [path]

chmod has permission arguments that are made up of 3 components:

- Who are we changing the permission for?
    - [ugoa] - user (or owner), group, others, all

- Are we granting or revoking the permissions/
    - They're indicated with either a plus (+) or minus (-)

- Which permission are we setting?
    - read (r), write (w) or execute (x)

The following examples will make their usage clearer.

Grant the execute permission to the group. Then remove the write permission for the owner.

```
user@bash: ls -l frog.png
-rwxr----x 1 harry users 2.7K Jan 4 07:32 frog.png
user@bash:
user@bash: chmod g+x frog.png
user@bash: ls -l frog.png
-rwxr-x--x 1 harry users 2.7K Jan 4 07:32 frog.png
user@bash:
user@bash: chmod u-w frog.png
user@bash: ls -l frog.png
-r-xr-x--x 1 harry users 2.7K Jan 4 07:32 frog.png
user@bash:
```

Don't want to assign permissions individually? We can assign multiple permissions at once.

```
user@bash: ls -l frog.png
-rwxr----x 1 harry users 2.7K Jan 4 07:32 frog.png
user@bash:
user@bash: chmod g+wx frog.png
user@bash: ls -l frog.png
-rwxrwx--x 1 harry users 2.7K Jan 4 07:32 frog.png
user@bash:
user@bash: chmod go-x frog.png
user@bash: ls -l frog.png
-rwxrw---- 1 harry users 2.7K Jan 4 07:32 frog.png
user@bash:
```

It may seem odd that as the owner of a file we can remove our ability to read, write and execute that file but there are valid reasons we may wish to do this. Maybe we have a file with data in it we wish not to accidentally change for instance. While we may remove these permissions, we may not remove our ability to set those permissions and as such we always have control over every file under our ownership.

Section 8: Permissions - Shorthand

The method outlined above is not too hard for setting permissions but it can be a little tedious if we have a specific set of permissions we would like to apply regularly to certain files. Luckily, there is a shorthand way to specify permissions that makes this easy.

To understand how this shorthand method works we first need a little background in number systems. Our typical number system is decimal. It is a base 10 number system and as such has 10 symbols (0 - 9) used. Another number system is octal which is base 8 (0-7). Now it just so happens that with 3 permissions and each being on or off, we have 8 possible combinations (2^3). Now we can also represent our numbers using binary which only has 2 symbols (0 and 1). The mapping of octal to binary is in the table below.

Octal	**Binary**
0	0 0 0
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1

Now the interesting point to note is that we may represent all 8 octal values with 3 binary bits and that every possible combination of 1 and 0 is included in it. So we have 3 bits and we also have 3 permissions. If you think of 1 as representing on and 0 as off then a single octal number may be used to represent a set of permissions for a set of people. Three numbers and we can specify permissions for the user, group and others. Let's see some examples. (refer to the table above to see how they match)

```
user@bash: ls -l frog.png
-rw-r----x 1 harry users 2.7K Jan 4 07:32 frog.png
user@bash:
user@bash: chmod 751 frog.png
user@bash: ls -l frog.png
-rwxr-x--x 1 harry users 2.7K Jan 4 07:32 frog.png
user@bash:
user@bash: chmod 240 frog.png
user@bash: ls -l frog.png
--w-r----- 1 harry users 2.7K Jan 4 07:32 frog.png
user@bash:
```

People often remember commonly used number sequences for different types of files and find this method quite convenient. For example 755 or 750 are commonly used for scripts.

Section 8: Permissions - Permissions for Directories

The same series of permissions may be used for directories but they have a slightly different behaviour.

- r (Read) - you have the ability to read the contents of the directory
  - i.e. do an ls

- w (Write) - you have the ability to write into the directory
  - i.e. create files and directories

- x (Execute) - you have the ability to enter that directory
  - i.e. cd

Let's see some of these in action:

```
user@bash: ls testdir
file1 file2 file3
user@bash:
user@bash: chmod 400 testdir
user@bash: ls -ld testdir
dr-------- 1 ryan users 2.7K Jan 4 07:32 testdir
user@bash:
user@bash: cd testdir
cd: testdir: Permission denied
user@bash: ls testdir
file1 file2 file3
user@bash:
user@bash: chmod 100 testdir
user@bash: ls -ld testdir
---x------ 1 ryan users 2.7K Jan 4 07:32 testdir
user@bash:
user@bash: ls testdir
user@bash: cd testdir
user@bash: pwd
/home/ryan/testdir
ls: cannot open directory testdir/: Permission denied
user@bash:
```

Note, on lines 5 and 14 above when we ran ls I included the -d option which stands for directory. Normally if we give ls an argument which is a directory it will list the contents of that directory. In this case however we are interested in the permissions of the directory directly and the -d option allows us to obtain that.

These permissions can seem a little confusing at first. What we need to remember is that these permissions are for the directory itself, not the files within. So, for example, you may have a directory which you don't have the read permission for. It may have files within it which you do have the read permission for. As long as you know the file exists and it's name you can still read the file.

```
user@bash: ls -ld testdir
--x------- 1 ryan users 2.7K Jan 4 07:32 testdir
user@bash:
user@bash: cd testdir
user@bash:
user@bash: ls
ls: cannot open directory .: Permission denied
user@bash:
user@bash: cat samplefile.txt
Kyle 20
Stan 11
Kenny 37
user@bash:
```

Section 8: Permissions - The Root User

On a Linux system there are only 2 people usually who may change the permissions of a file or directory. The owner of the file or directory and the root user. The root user is a superuser who is allowed to do anything and everything on the system. Typically the administrators of a system would be the only ones who have access to the root account and would use it to maintain the system. Typically normal users would mostly only have access to files and directories in their home directory and maybe a few others for the purposes of sharing and collaborating on work and this helps to maintain the security and stability of the system.

Section 8: Permissions - Basic Security

Your home directory is your own personal space on the system. You should make sure that it stays that way.

Most users would give themselves full read, write and execute permissions for their home directory and no permissions for the group or others however some people for various reasons may have a slightly different set up.

Normally, for optimal security, you should not give either the group or others write access to your home directory, but execute without read can come in handy sometimes. This allows people to get into your home directory but not allow them to see what is there. An example of when this is used is for personal web pages.

- It is typical for a system to run a webserver and allow users to each have their own web space. A common set up is that if you place a directory in your home directory called public_html then the webserver will read and display the contents of it. The webserver runs as a different user to you however so by default will not have access to get in and read those files. This is a situation where it is necessary to grant execute on your home directory so that the webserver user may access the required resources.

Section 8: Permissions - Summary

Stuff we Learned:

- chmod - Change permissions on a file or directory.

- ls -ld - View the permissions for a specific directory.

- Security - Correct permissions are important for the security of a system.

- Usage - Setting the right permissions is important in the smooth running of certain tasks on Linux.

Section 9: Filters - Introduction

One of the underlying principles of Linux is that every item should do one thing and one thing only and that we can easily join these items together. Think of it like a set of building blocks that we may put together however we like to build anything we want. In this section and the next we will learn about a few of these building blocks. Then in section 11 and 13 we'll look at how we may build them into more complex creations that can do useful work for us.

- This section looks quite long but it is mostly examples so it's not as scary as it looks.

Section 9: Filters - What Are They

A filter, in the context of the Linux command line, is a program that accepts textual data and then transforms it in a particular way. Filters are a way to take raw data, either produced by another program, or stored in a file, and manipulate it to be displayed in a way more suited to what we are after.

- These filters often have various command line options that will modify their behaviour so it is always good to check out the man page for a filter to see what is available.

In the examples below we will be providing input to these programs by a file but in the section Piping and Redirection we'll see that we may provide input via other means that add a lot more power.

- Let's dive in and introduce you to some of them.
    - **Remember**, the examples here will only give you a taste of what is possible with these commands. Make sure you explore and use your creativity to see what else you may do with them.

For each of the demonstrations below I will be using the following file as an example. This example file contains a list of content purely to make the examples a bit easier to understand but realise that they will work the same with absolutely any other textual data. Also, remember that the file is actually specified as a path and so you may use absolute and relative paths and also wildcards.

```
user@bash: cat mysampledata.txt
Fred apples 20
Susy oranges 5
Mark watermellons 12
Robert pears 4
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
Oliver rockmellons 2
Betty limes 14
user@bash:
```

Section 9: Filters - head

Head is a program that prints the first so many lines of it's input. By default it will print the first 10 lines but we may modify this with a command line argument:

- head [-number of lines to print] [path]

```
user@bash: head mysampledata.txt
Fred apples 20
Susy oranges 5
Mark watermellons 12
Robert pears 4
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
user@bash:
```

Above was head's default behaviour. And below is specifying a set number of lines.

```
user@bash: head -4 mysampledata.txt
Fred apples 20
Susy oranges 5
Mark watermelons 12
Robert pears 4
user@bash:
```

Section 9: Filters - tail

Tail is a program that prints the last so many lines of it's input. By default it will print the last 10 lines but we may modify this with a command line argument.

- tail [-number of lines to print] [path]

```
user@bash: tail mysampledata.txt
Mark watermellons 12
Robert pears 4
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
Oliver rockmellons 2
Betty limes 14
user@bash:
```

Above was tail's default behaviour. And below is specifying a set number of lines.

```
user@bash: tail -3 mysampledata.txt
Greg pineapples 3
Oliver rockmellons 2
Betty limes 14
user@bash:
```

Section 9:Filters - sort

Sort will sort it's input, nice and simple. By default it will sort alphabetically but there are many options available to modify the sorting mechanism. Be sure to check out the man page to see everything it may do.

- sort [-options] [path]

```
user@bash: sort mysampledata.txt
Anne mangoes 7
Betty limes 14
Fred apples 20
Greg pineapples 3
Lisa peaches 7
Mark grapes 39
Mark watermellons 12
Oliver rockmellons 2
Robert pears 4
Susy oranges 12
Susy oranges 5
Terry oranges 9
user@bash:
```

Section 9: Filters - nl

nl stands for number lines and it does just that.

- nl [-options] [path]

```
user@bash: nl mysampledata.txt
1 Fred apples 20
2 Susy oranges 5
3 Mark watermellons 12
4 Robert pears 4
5 Terry oranges 9
6 Lisa peaches 7
7 Susy oranges 12
8 Mark grapes 39
9 Anne mangoes 7
10 Greg pineapples 3
11 Oliver rockmellons 2
12 Betty limes 14
user@bash:
```

The basic formatting is ok but sometimes you are after something a little different. With a few command line options, nl is happy to oblige.

```
user@bash: nl -s '. ' -w 10 mysampledata.txt
         1. Fred apples 20
         2. Susy oranges 5
         3. Mark watermellons 12
         4. Robert pears 4
         5. Terry oranges 9
         6. Lisa peaches 7
         7. Susy oranges 12
         8. Mark grapes 39
         9. Anne mangoes 7
        10. Greg pineapples 3
        11. Oliver rockmellons 2
        12. Betty limes 14
user@bash:
```

In the above example we have used 2 command line options. The first one -s specifies what should be printed after the number while the second one -w specifies how much padding to put before the numbers. For the first one we needed to include a space as part of what was printed. Because spaces are normally used as separator characters on the command line we needed a way of specifying that the space was part of our argument and not just in between arguments. We did that by including the argument surrounded by quotes.

Section 9: Filters - wc

wc stands for word count and it does just that (as well as characters and lines). By default it will give a count of all 3 but using command line options we may limit it to just what we are after.

- wc [-options] [path]

```
user@bash: wc mysampledata.txt
12 36 195 mysampledata.txt
user@bash:
```

Sometimes you just want one of these values.

- -l will give us lines only,
    - -w will give us words and
        - -m will give us characters.

The example below gives us just a line count.

```
user@bash: wc -l mysampledata.txt
12 mysampledata.txt
user@bash:
```

You may combine the command line arguments too. This example gives us both lines and words.

```
user@bash: wc -lw mysampledata.txt
12 36 mysampledata.txt
user@bash:
```

Section 9: Filters - cut

cut is a nice little program to use if your content is separated into fields (columns) and you only want certain fields.

- cut [-options] [path]

In our sample file we have our data in 3 columns, the first is a name, the second is a fruit and the third an amount. Let's say we only wanted the first column.

```
user@bash: cut -f 1 -d ' ' mysampledata.txt
Fred
Susy
Mark
Robert
Terry
Lisa
Susy
Mark
Anne
Greg
Oliver
Betty
user@bash:
```

cut defaults to using the TAB character as a separator to identify fields. In our file we have used a single space instead so we need to tell cut to use that instead. The separator character may be anything you like, for instance in a CSV file the separator is typically a comma (,). This is what the -d option does (we include the space within single quotes so it knows this is part of the argument). The -f option allows us to specify which field or fields we would like. If we wanted 2 or more fields then we separate them with a comma as below.

```
user@bash: cut -f 1,2 -d ' ' mysampledata.txt
Fred apples
Susy oranges
Mark watermellons
Robert pears
Terry oranges
Lisa peaches
Susy oranges
Mark grapes
Anne mangoes
```

```
Greg pineapples
Oliver rockmellons
Betty limes
user@bash:
```

Section 9: Filters - sed

sed stands for Stream Editor and it effectively allows us to do a search and replace on our data.

- sed <expression> [path]

A basic expression is of the following format:

- s/search/replace/g

Let’s break that syntax down:

- The initial s stands for substitute
  - specifies the action to perform.

- In between the first and second slashes (/) we place what it is we are searching for.

- Then between the second and third slashes, what it is we wish to replace it with.

- The g at the end stands for global and is optional.
  - If we omit it then it will only replace the first instance of search on each line.
  - With the g option we will replace every instance of search that is on each line.

```
user@bash: sed 's/oranges/bananas/g' mysampledata.txt
Fred apples 20
Susy bananas 5
Mark watermellons 12
Robert pears 4
Terry bananas 9
Lisa peaches 7
Susy bananas 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
Oliver rockmellons 2
Betty limes 14
user@bash:
```

It's important to note that sed does not identify words but strings of characters. Try running the example above yourself but replacing oranges with es and you'll see what I mean. The search term is also actually something called a regular expression which is a means to define a pattern (similar to wildcards we looked at in section 7).

Also note that we included our expression within single quotes. We did this so that any characters included in it which may have a special meaning on the command line don't get interpreted and acted upon by the command line but instead get passed through to sed.

Tip:
A common mistake is to forget the single quotes in which case you may get some strange behaviour from the command line. If this happens you may need to press CTRL+c to cancel the program and get back to the prompt.

Section 9: Filters - uniq

uniq stands for unique and it's job is to remove duplicate lines from the data. One limitation however is that those lines must be adjacent (ie, one after the other). ((sometimes this is not the case but we'll see one way we can fix this in Section 11 Piping and Redirection)).

- uniq [-options] [path]

Let's say that our sample file was actually generated from another sales program but after a software update it had some buggy output.

```
user@bash: cat mysampledata.txt
Fred apples 20
Susy oranges 5
Susy oranges 5
Susy oranges 5
Mark watermellons 12
Robert pears 4
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Mark grapes 39
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
Oliver rockmellons 2
Betty limes 14
user@bash:
```

No worries, we can easily fix that using uniq.

```
user@bash: uniq mysampledata.txt
Fred apples 20
Susy oranges 5
Mark watermelons 12
Robert pears 4
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
```

```
Oliver rockmellons 2
Betty limes 14
user@bash:
```

Section 9: Filters - tac

The program tac is actually cat in reverse. It was named this as it does the opposite of cat. Given data it will print the last line first, through to the first line.

- tac [path]

Maybe our sample file is generated by writing each new order to the end of the file. As a result, the most recent orders are at the end of the file. We would like it the other way so that the most recent orders are always at the top.

```
user@bash: tac mysampledata.txt
Betty limes 14
Oliver rockmellons 2
Greg pineapples 3
Anne mangoes 7
Mark grapes 39
Susy oranges 12
Lisa peaches 7
Terry oranges 9
Robert pears 4
Mark watermellons 12
Susy oranges 5
Fred apples 20
user@bash:
```

Section 9: Filters - Others

Here are two other programs that are worth investigating if you want to take your knowledge even further. They are quite powerful but also more complex than the programs listed above.

- awk
- diff

Section 9: Filters - Summary

Stuff we learned:

- head - View the first n lines of data.
- tail - View the last n lines of data.
- sort - Organise the data into order.
- nl - Print line numbers before data.
- wc - Print a count of lines, words and characters.
- cut - Cut the data into fields and only display the specified fields.
- sed - Do a search and replace on the data.
- uniq - Remove duplicate lines.
- tac - Print the data in reverse order.
- Processing - Filters allow us to process and format data in interesting ways.
- man pages - Most of the programs we looked at have command line options that allow you to modify their behaviour.

Section 10: Grep & Regular Expressions - Introduction

In this section we will look at another filter which is quite powerful when combined with a concept called regular expressions or re's for short. Re's can be a little hard to get your head around at first so don't worry if this stuff is a little confusing. I find the best approach is to go over the material and experiment on the command line a little, then leave it for a day or 3, then come back and have another go. You will be surprised but it will start to make more sense the second time. Mastering re's just takes practice and time so don't give up.

Section 10: Grep & Regular Expressions - What Are They

Regular expressions are similar to the wildcards that we looked at in section 7. They allow us to create a pattern. They are a bit more powerful however. Re's are typically used to identify and manipulate specific pieces of data.

- i.e. we may wish to identify every line which contains an email address or a url in a set of data.

Re's are used all over the place. We will be demonstrating them here with grep but many other programs use them (including sed and vi which you learned about in previous sections) and many programming languages make use of them too.

Tip:
The characters used in regular expressions are the same as those used in wildcards. Their behaviour is slightly different however. A common mistake is to forget this and get their functions mixed up.

Section 10: Grep & Regular Expressions - eGrep

egrep is a program which will search a given set of data and print every line which contains a given pattern. It is an extension of a program called grep. It's name is odd but based upon a command which did a similar function, in a text editor called ed. It has many command line options which modify it's behaviour so it's worth checking out it's man page.

- i.e. the -v option tells grep to instead print every line which does not match the pattern.
    - egrep [-options] <pattern> [path]

In the examples below we will use a similar sample file as in the last section. It is included below as a reference.

```
user@bash: cat mysampledata.txt
Fred apples 20
Susy oranges 5
Mark watermellons 12
Robert pears 4
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
Oliver rockmellons 2
Betty limes 14
user@bash:
```

Let's say we wished to identify every line which contained the string mellon

```
user@bash: egrep 'melon' mysampledata.txt
Mark watermelons 12
Oliver rockmelons 2
user@bash:
```

The basic behaviour of egrep is that it will print the entire line for every line which contains a string of characters matching the given pattern. This is important to note, we are not searching for a word but a string of characters.

- Also note that we included the pattern within quotes. This is not always required but it is safer to get in the habit of always using them. They are required if your pattern contains characters which have a special meaning on the command line.

Sometimes we want to know not only which lines matched but their line number as well.

```
user@bash: egrep -n 'melon' mysampledata.txt
3:Mark watermelons 12
11:Oliver rockmelons 2
user@bash:
```

Or maybe we are not interested in seeing the matched lines but wish to know how many lines did match.

```
user@bash: egrep -c 'mellon' mysampledata.txt
2
user@bash:
```

Section 10: Grep & Regular Expressions - Understanding Regular Expressions

The best way to learn regular expressions is to give the examples a try yourself, then modify them slightly to test your understanding. It is common to make mistakes in your patterns while you are learning. When this happens typically every line will be matched or no lines will be matched or some obscure set. Don't worry if this happens you haven't done any damage and you can easily go back and have another go. Remember you may hit the up arrow on your keyboard to get at your recent commands and also modify them so you don't need to retype the whole command each time.

If you're not getting the output you would like then here are some basic strategies.

- First off, **check for typo's**. If you're like me then you're prone to making them.

- **Re read the content here**. Maybe what you thought a particular operator did was slightly different to what it actually does and re reading you will notice a point you may have missed the first time.

- **Break your pattern down** into individual components and test each of these individually.
    - This will help you to get a feel for which parts of the patterns are right and which parts you need to adjust.

- **Examine your output**. Your current pattern may not have worked the way you want but we can still learn from it.
    - Looking at what we actually did match and using it to help understand what actually did happen will help us to work out what we should try changing to get closer to what we actually want.

Debuggex is an on-line tool that allows you to experiment with regular expressions and allows you to visualise their behaviour. It can be a good way to better understand how they work.

Section 10: Grep & Regular Expressions - Regular Expression Overview

I will outline the basic building blocks of re's below then follow on with a set of examples to demonstrate their usage.

. (period) - a single character.

? - the preceding character matches 0 or 1 times only.

* - the preceding character matches 0 or more times.

+ - the preceding character matches 1 or more times.

{n} - the preceding character matches exactly n times.

{n, m} - the preceding character matches at least n times and not more than m times.

[agd] - the character is one of those included within the square brackets.

[^agd] - the character is not one of those included within the square brackets.

[c-f] - the dash within the square brackets operates as a range.

- In this case it means either the letters c, d, e or f.

() - allows us to group several characters to behave as one.

| (pipe symbol) - the logical OR operation.

^ - matches the beginning of the line.

$ - matches the end of the line.

Section 10: Grep & Regular Expressions - Some Examples

Let's say we wish to identify any line with two or more vowels in a row. In the example below the multiplier {2,} applies to the preceding item which is the range.

```
user@bash: egrep '[aeiou]{2,}' mysampledata.txt
Robert pears 4
Lisa peaches 7
Anne mangoes 7
Greg pineapples 3
user@bash:
```

How about any line with a 2 on it which is not the end of the line. In this example the multiplier + applies to the . which is any character.

```
user@bash: egrep '2.+' mysampledata.txt
Fred apples 20
user@bash:
```

The number 2 as the last character on the line.

```
user@bash: egrep '2$' mysampledata.txt
Mark watermelons 12
Susy oranges 12
Oliver rockmelons 2
user@bash:
```

And now each line which contains either 'is' or 'go' or 'or'.

```
user@bash: egrep 'or|is|go' mysampledata.txt
Susy oranges 5
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Anne mangoes 7
user@bash:
```

Maybe we wish to see orders for everyone who's name begins with A - K.

```
user@bash: egrep '^[A-K]' mysampledata.txt
Fred apples 20
Anne mangoes 7
Greg pineapples 3
Betty limes 14
user@bash:
```

Section 10: Grep & Regular Expressions - Summary

Stuff we learned:

- egrep - View lines of data which match a particular pattern.

- Regular Expressions - A powerful way to identify particular pieces of information.

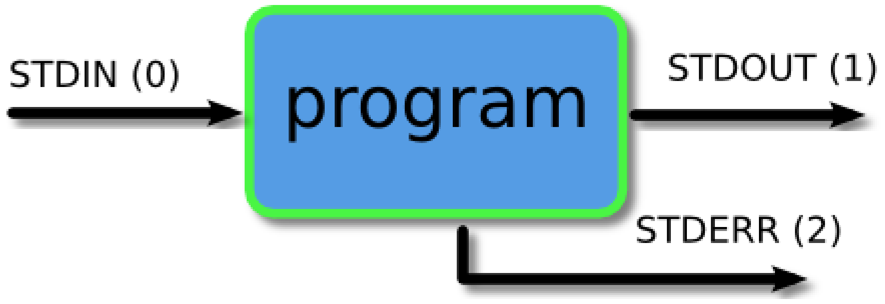
Section 11: Piping & Redirection - Introduction

In the previous two sections we looked at a collection of filters that would manipulate data for us. In this section we will see how we may join them together to do more powerful data manipulation. There is a bit of reading involved in this section. Even though the mechanisms and their use are quite simple, it is important to understand various characteristics about their behaviour if you wish to use them effectively.

Section 11: Piping & Redirection - What Are They

Every program we run on the command line automatically has three data streams connected to it.

- STDIN (0) - Standard input
  - Data fed into the program
- STDOUT (1) - Standard output
  - Data printed by the program, defaults to the terminal
- STDERR (2) - Standard error
  - For error messages, also defaults to the terminal



Piping and redirection is the means by which we may connect these streams between programs and files to direct data in interesting and useful ways.

- We'll demonstrate piping and redirection below with several examples but these mechanisms will work with every program on the command line, not just the ones we have used in the examples.

Section 11: Piping & Redirection - Redirecting to a File

Normally, we will get our output on the screen, which is convenient most of the time, but sometimes we may wish to save it into a file to keep as a record, feed into another system, or send to someone else. The greater than operator (>) indicates to the command line that we wish the program's output (or whatever it sends to STDOUT) to be saved in a file instead of printed to the screen.

```
1. user@bash: ls
2. barry.txt bob example.png firstfile foo1 video.mpeg
3. user@bash: ls > myoutput
4. user@bash: ls
5. barry.txt bob example.png firstfile foo1 myoutput video.mpeg
6. user@bash: cat myoutput
7. barry.txt
8. bob
9. example.png
10.   firstfile
11.   foo1
12.   myoutput
13.   video.mpeg
14.   user@bash:
```

Let's break it down:

- **Line 1** Let's start off by seeing what's in our current directory.
- **Line 3** Now we'll run the same command but this time we use the > to tell the terminal to save the output into the file myoutput.
  - You'll notice that we don't need to create the file before saving it. The terminal will create it automatically if it does not exist.
- **Line 4** As you can see, our new file has been created.
- **Line 6** Let's have a look at what was saved in there.

Some Observations

You'll notice that in the above example, the output saved in the file was one file per line instead of all across one line when printed to the screen. The reason for this is that the screen is a known width and the program can format its output to suit that. When we are redirecting, it may be to a file, or it could be somewhere else, so the safest option is to format it as one entry per line. This also allows us to easier manipulate that data later on.

- You'll also notice that the file we created to save the data into is also in our listing.
  - The way the mechanism works, the file is created first (if it does not exist already) and then the program is run and output saved into the file.

Tip:
When piping and redirecting, the actual data will always be the same, but the formatting of that data may be slightly different to what is normally printed to the screen. Keep this in mind.

Saving to an Existing File

If we redirect to a file which does not exist, it will be created automatically for us. If we save into a file which already exists, however, then it's contents will be cleared, then the new output saved to it.

```
user@bash: cat myoutput
barry.txt
bob
example.png
firstfile
foo1
myoutput
video.mpeg
user@bash: wc -l barry.txt > myoutput
user@bash: cat myoutput
7 barry.txt
user@bash:
```

We can instead get the new data to be appended to the file by using the double greater than operator (>>).

```
user@bash: cat myoutput
7 barry.txt
user@bash: ls >> myoutput
user@bash: cat myoutput
7 barry.txt
barry.txt
bob
example.png
firstfile
foo1
myoutput
video.mpeg
user@bash:
```

Section 11: Piping & Redirection - Redirecting From a File

If we use the less than operator (<) then we can send data the other way. We will read data from the file and feed it into the program via it's STDIN stream.

```
user@bash: wc -l myoutput
8 myoutput
user@bash: wc -l < myoutput
8
user@bash:
```

A lot of programs (as we've seen in previous sections) allow us to supply a file as a command line argument and it will read and process the contents of that file. Given this, you may be asking why we would need to use this operator. The above example illustrates a subtle but useful difference. You'll notice that when we ran wc supplying the file to process as a command line argument, the output from the program included the name of the file that was processed. When we ran it redirecting the contents of the file into wc the file name was not printed. This is because whenever we use redirection or piping, the data is sent anonymously. So in the above example, wc received some content to process, but it has no knowledge of where it came from so it may not print this information. As a result, this mechanism is often used in order to get ancillary data (which may not be required) to not be printed.

We may easily combine the two forms of redirection we have seen so far into a single command as seen in the example below.

```
user@bash: wc -l < barry.txt > myoutput
user@bash: cat myoutput
7
user@bash:
```

Section 11: Piping & Redirection - Redirecting STDERR

Now let's look at the third stream which is Standard Error or STDERR. The three streams actually have numbers associated with them (in brackets in the list at the top of the page). STDERR is stream number 2 and we may use these numbers to identify the streams. If we place a number before the > operator then it will redirect that stream.

- If we don't use a number, like we have been doing so far, then it defaults to stream 1.

```
user@bash: ls -l video.mpg blah.foo
ls: cannot access blah.foo: No such file or directory
-rwxr--r-- 1 ryan users 6 May 16 09:14 video.mpg
user@bash: ls -l video.mpg blah.foo 2> errors.txt
-rwxr--r-- 1 ryan users 6 May 16 09:14 video.mpg
user@bash: cat errors.txt
ls: cannot access blah.foo: No such file or directory
user@bash:
```

Maybe we wish to save both normal output and error messages into a single file. This can be done by redirecting the STDERR stream to the STDOUT stream and redirecting STDOUT to a file. We redirect to a file first then redirect the error stream. We identify the redirection to a stream by placing an & in front of the stream number.

- Otherwise it would redirect to a file called 1.

```
user@bash: ls -l video.mpg blah.foo > myoutput 2>&1
user@bash: cat myoutput
ls: cannot access blah.foo: No such file or directory
-rwxr--r-- 1 ryan users 6 May 16 09:14 video.mpg
user@bash:
```

Section 11: Piping & Redirection - Piping

So far we've dealt with sending data to and from files. Now we'll take a look at a mechanism for sending data from one program to another. It's called piping and the operator we use is (|) (found above the backslash (\ ) key on most keyboards). What this operator does is feed the output from the program on the left as input to the program on the right. In the example below we will list only the first 3 files in the directory.

```
user@bash: ls
barry.txt bob example.png firstfile foo1 myoutput video.mpeg
user@bash: ls | head -3
barry.txt
bob
example.png
user@bash:
```

We may pipe as many programs together as we like. In the below example we have then piped the output to tail so as to get only the third file.

```
user@bash: ls | head -3 | tail -1
example.png
user@bash:
```

Tip:
Any command line arguments we supply for a program must be next to that program.

Tip:
Build your pipes up incrementally then you won't fall into this trap. Run the first program and make sure it provides the output you were expecting. Then add the second program and check again before adding the third and so on. This will save you a lot of frustration.

You may combine pipes and redirection too.

```
user@bash: ls | head -3 | tail -1 > myoutput
user@bash: cat myoutput
example.png
user@bash:
```

Section 11: Piping & Redirection - More Examples

Below are some more examples to give an idea of the sorts of things you can do with piping. There are many things you can achieve with piping and these are just a few of them. With experience and a little creative thinking I'm sure you'll find many more ways to use piping to make your life easier.

- All the programs used in the examples are programs we have seen before. I have used some command line arguments that we haven't covered yet however. Look up the relevant man pages to find out what they do. Also you can try the commands yourself, building up incrementally to see exactly what each step is doing.

In this example we are sorting the listing of a directory so that all the directories are listed first.

```
user@bash: ls -l /etc | tail -n +2 | sort
drwxrwxr-x 3 nagios nagcmd 4096 Mar 29 08:52 nagios
drwxr-x--- 2 news news 4096 Jan 27 02:22 news
drwxr-x--- 2 root mysql 4096 Mar 6 22:39 mysql
...
user@bash:
```

In this example we will feed the output of a program into the program less so that we can view it easier.

```
user@bash: ls -l /etc | less
(Full screen of output you may scroll. Try it yourself to see.)
user@bash:
```

Identify all files in your home directory which the group has write permission for.

```
user@bash: ls -l ~ | grep '^.....w'
drwxrwxr-x 3 ryan users 4096 Jan 21 04:12 dropbox
user@bash:
```

Create a listing of every user which owns a file in a given directory as well as how many files and directories they own.

```
user@bash: ls -l /projects/ghosttrail | tail -n +2 | sed 's/\s\s*/ /g' | cut -d ' ' -f 3 | sort | uniq -c
8 anne
34 harry
37 tina
18 ryan
user@bash:
```

Section 11: Piping & Redirection - Summary

Stuff we learned:

- > - Save output to a file.
- >> - Append output to a file.
- < - Read input from a file.
- 2> - Redirect error messages.
- | (pipe) - Send the output from one program as input to another program.
- Streams - Every program you may run on the command line has 3 streams, STDIN, STDOUT and STDERR.

Section 12: Process Management - Introduction

Linux in general is a fairly stable system. Occasionally, things do go wrong however and sometimes we also wish to tweak the running of the system to better suit our needs. In this section we will take a brief look at how we may manage programs, or processes on a Linux system.

Section 12: Process Management - What Are They

A program is a series of instructions that tell the computer what to do. When we run a program, those instructions are copied into memory and space is allocated for variables and other stuff required to manage its execution. This running instance of a program is called a process and it's processes which we manage.

Section 12: Process Management - What is Currently Running

Linux, like most modern OS's, is a multitasking operating system. This means that many processes can be running at the same time. As well as the processes we are running, there may be other users on the system also running stuff and the OS itself will usually also be running various processes which it uses to manage everything in general. If we would like to get a snapshot of what is currently happening on the system we may use a program called top.

- top

Below is a simplified version of what you should see when you run this program.

```
1. user@bash: top
2. Tasks: 174 total, 3 running, 171 sleeping, 0 stopped
3. KiB Mem: 4050604 total, 3114428 used, 936176 free
4. Kib Swap: 2104476 total, 18132 used, 2086344 free
5.
6.  PID USER %CPU %MEM COMMAND
7. 6978 ryan 3.0  21.2 firefox
8.   11 root 0.3   0.0 rcu_preempt
9. 6601 ryan 2.0   2.4 kwin
10.   ...
```

Let's break it down:

- **Line 2** Tasks is just another name for processes.
    - It's typical to have quite a few processes running on your system at any given time and most of them will be system processes.
        - Many of them will typically be sleeping.
        - This is ok. It just means they are waiting until a particular event occurs, which they will then act upon.

- **Line 3** This is a breakdown of working memory (RAM).
    - Don't worry if a large amount of your memory is used.
    - Linux keeps recently used programs in memory to speed up performance if they are run again.
        - If another process needs that memory, they can easily be cleared to accommodate this.

- **Line 4** This is a breakdown of Virtual memory on your system.
    - If a large amount of this is in use, you may want to consider increasing its size.
    - For most people with most modern systems having gigabytes of RAM you shouldn't experience any issues here.

- **Lines 6 - 10** Finally is a listing of the most resource intensive processes on the system (in order of resource usage). This list will update in real time and so is interesting to watch to get an idea of what is happening on your system.
    - The two important columns to consider are Memory and CPU usage.
    - If either of these is high for a particular process over a period of time, it may be worth looking into why this is so.
        - The USER column shows who owns the process and the PID column identifies a process's Process ID which is a unique identifier for that process.

Top will give you a real time view of the system and only show the number of processes which will fit on the screen.

Another program to look at processes is called ps which stands for processes. In it's normal usage it will show you just the processes running in your current terminal (which is usually not very much). If we add the argument aux then it will show a complete system view which is a bit more helpful.
- ps [aux]

It does give quite a bit of output so people usually pipe the output to grep to filter out just the data they are after. We will see in the next bit an example of this.

Section 12: Process Management - Killing a Crashed Process

It doesn't happen often, but when a program crashes, it can be quite annoying. Let's say we've got our browser running and all of a sudden it locks up. You try and close the window but nothing happens, it has become completely unresponsive. No worries, we can easily kill Firefox and then reopen it. To start off we need to identify the process id.

```
user@bash: ps aux | grep 'firefox'
ryan 6978 8.8 23.5 2344096 945452 ? Sl 08:03 49:53 /usr/lib64/firefox/firefox
user@bash:
```

It is the number next to the owner of the process that is the PID (Process ID). We will use this to identify which process to kill. To do so we use a program which is appropriately called kill.

- kill [signal] <PID>

```
user@bash: kill 6978
user@bash: ps aux | grep 'firefox'
ryan 6978 8.8 23.5 2344096 945452 ? Sl 08:03 49:53 /usr/lib64/firefox/firefox
user@bash:
```

Sometimes you are lucky and just running kill normally will get the process to stop and exit. When you do this kill sends the default signal (1) to the process which effectively asks the process nicely to quit. We always try this option first as a clean quit is the best option. Sometimes this does not work however. In the example above we ran ps again and saw that the process was still running. No worries, we can run kill again but this time supply a signal of 9 which effectively means, go in with a sledge hammer and make sure the process is well and truly gone.

```
user@bash: kill -9 6978
user@bash: ps aux | grep 'firefox'
user@bash:
```

Tip:
Normal users may only kill processes which they are the owner for. The root user on the system may kill anyones processes.

My Desktop has locked up

On rare occasions, when a process crashes and locks up, it can lock up the entire desktop. If this happens there is still hope.

Linux actually runs several virtual consoles.

- Most of the time we only see console 7 which is the GUI but we can easily get to the others. If the GUI has locked up, and we are in luck, we can get to another console and kill the offending process from there.
- To switch between consoles you use the keyboard sequence **CTRL + ALT + F<Console>**.
    - So CTRL + ALT F2 will get you to a console (if all goes well) where you can run the commands as above to identify process ids and kill them.
    - Then CTRL + ALT F7 will get you back to the GUI to see if it has been fixed.
        - The general approach is to keep killing processes until the lock up is fixed.
        - Normally you can look for tell tale signs such as high CPU or Memory usage and start with those processes first.
        - Sometimes this approach works, sometimes it doesn't and you need to restart the computer. Just depends how lucky you are.

Section 12: Process Management - Foreground & Background Jobs

You probably won't need to do too much with foreground and background jobs but it's worth knowing about them just for those rare occasions.

- When we run a program normally (like we have been doing so far) they are run in the foreground.
- Most of them run to completion in a fraction of a second as well.
    - Maybe we wish to start a process that will take a bit of time and will happily do it's thing without intervention from us (processing a very large text file or compiling a program for instance).
    - What we can do is run the program in the background and then we can continue working.
    - We'll demonstrate this with a program called **sleep**.
    - All sleep does is wait a given number of seconds and then quit.
        - We can also use a program called **jobs** which lists currently running background jobs for us.

```
user@bash: sleep 5
user@bash:
```

If you run the above example yourself, you will notice that the terminal waits 5 seconds before presenting you with a prompt again. Now if we run the same command but instead put an ampersand (&) at the end of the command then we are telling the terminal to run this process in the background.

```
user@bash: sleep 5 &
[1] 21634
user@bash:
user@bash:
[1]+ Done sleep 5
```

This time you will notice that it assigns the process a job number, and tells us what that number is, and gives us the prompt back straight away. We can continue working while the process runs in the background. If you wait 5 seconds or so and then hit ENTER you will see a message come up telling you the job has completed.

We can move jobs between the foreground and background as well. If you press CTRL + z then the currently running foreground process will be paused and moved into the background. We can then use a program called fg which stands for foreground to bring background processes into the foreground.

- fg <job number>

```
user@bash: sleep 15 &
[1] 21637
user@bash: sleep 10
(you press CTRL + z, notice the prompt comes back.)
user@bash: jobs
[1]- Running sleep 15 &
[2]+ Stopped sleep 10
user@bash: fg 2
[1] Done sleep 15
user@bash:
```

Tip:
CTRL + z is used in Windows but for the purpose of running the undo command. It is not uncommon for people coming from the Windows world to accidentally hit the key combo (especially in the editor VI for instance) and wonder why their program just disappeared and the prompt returned. If you do this, don't worry, you can use jobs to identify which job it has been assigned to and then fg to bring it back and continue working.

Section 12: Process Management - Summary

Stuff we learned:

- top - View real-time data about processes running on the system.
- ps - Get a listing of processes running on the system.
- kill - End the running of a process.
- jobs - Display a list of current jobs running in the background.
- fg - Move a background process into the foreground.
- Ctrl + Z - Pause the current foreground process and move it into the background.
    - Control - We have quite a bit of control over the running of our programs.

Section 13: Bash Scripting - Introduction

So this is the last section in this tutorial. Here we will introduce a concept called scripting. This will be a brief introduction to Bash scripting. There is a lot more you can do but my aim here is to get you started and give you just enough that you can do useful work.

- This section brings together a lot of what we learnt in previous sections (you'll see them referred to often). If some of this stuff doesn't really make sense, you may need to look back over previous sections and refresh your memory.

Section 13: Bash Scripting - What Are They

A Bash script in computing terms is similar to a script in theatrical terms. It is a document stating what to say and do. Here, instead of the script being read and acted upon by a person, it is read and acted upon (or executed) by the computer.

- **A Bash script** allows us to define a series of actions which the computer will then perform without us having to enter the commands ourselves. If a particular task is done often, or it is repetitive, then a script can be a useful tool.
- **A Bash script** is interpreted (read and acted upon) by something called an interpreter. There are various interpreters on a typical linux system but we have been learning the Bash shell so we'll introduce bash scripts here.
    - ***Anything you can run on the command line you may place into a script and they will behave exactly the same. Vice versa, anything you can put into a script, you may run on the command line and again it will perform exactly the same.***
    - The above statement is important to understand when creating scripts. When testing different parts of your script, as you're building it, it is often easiest to just run your commands directly on the command line.
        - A script is just a plain text file and it may have any name you like. You create them the same way you would any other text file, with just a plain old text editor (such as VI which we looked at in section 6).

Section 13: Bash Scripting - A Simple Example

Below is a simple script. I recommend you create a similar file yourself and run it to get a feel for how they work. This script will print a message to the screen (using a program called echo) then give us a listing of what is in our current directory.

- echo <message>

```
1. user@bash: cat myscript.sh
2. #!/bin/bash
3. # A simple demonstration script
4. # Ryan 3/3/2021
5.
6. echo Here are the files in your current directory:
7. ls
8.
9. user@bash: ls -l myscript.sh
10.   -rwxr-xr-x 1 ryan users 2 Jun 4 2012 myscript.sh
11.
12.   user@bash: ./myscript.sh
13.   Here are the files in your current directory:
14.   barry.txt bob example.png firstfile foo1 myoutput video.mpeg
15.   user@bash:
```

Let's break it down:

- **Line 1** Let's start off by having a look at our script. Linux is an extensionless system so it is not required for scripts to have a .sh extension. It is common to put them on however to make them easy to identify.

- **Line 2** The very first line of a script should always be this line. This line identifies which interpreter should be used. The first two characters are referred to as a shebang. After that (<u>important, no spaces</u>) is the path to the interpreter.

- **Lines 3 and 4** Anything following a # is a comment. The interpreter will not run this, it is just here for our benefit. It is good practice to include your name, and the date you wrote the script as well as a one line quick description of what it does at the top of the script.

- **Line 6** We'll use a program called echo It will merely print whatever you place after it, as command line arguments, to the screen. Useful for printing messages.

- **Line 7** The next step of our script is to print the contents of our current directory.

- **Line 9** A script must have the execute permission before it may be run. Here I am just demonstrating that the file does have the right permissions.

- **Line 12** Now we run the script. I'll explain why we need the ./ a bit further down.

- **Lines 13 and 14** The output from running (or executing) our script.

Section 13: Bash Scripting - Important Parts

The Shebang

The very first line of a script should tell the system which interpreter should be used on this file. It is important that this is the very first line of the script. It is also important that there are no spaces. The first two characters #! (the shebang) tell the system that directly after it will be a path to the interpreter to be used. If we don't know where our interpreter is located then we may use a program called which to find out.

- which <program>

```
user@bash: which bash
/bin/bash
user@bash:
user@bash: which ls
/usr/bin/ls
user@bash:
```

If we leave this line out then our Bash script may still work. Most shells (bash included) will assume they are the interpreter if one is not specified. It is good practice to always include the interpreter however. Later on, you, or someone else, may run your script in conditions under which bash is not the shell currently in use and this could lead to undesirable outcomes.

The Name

Linux is an extensionless system. That means we may call our script whatever we like and it will not affect it's running in any way. While it is typical to put a .sh extension on our scripts, this is purely for convenience and is not required. We could name our script above simply myscript or even myscript.jpg and it would still run quite happily.

Comments

A comment is just a note in the script that does not get run, it is merely there for your benefit. Comments are easy to put in, all you need to do is place a hash (#) then anything after that is considered a comment. A comment can be a whole line or at the end of a line.

```
user@bash: cat myscript.sh
#!/bin/bash
# A comment which takes up a whole line
ls # A comment at the end of the line
user@bash:
```

It is common practice to include a comment at the top of a script with a brief description of what the script does and also who wrote it and when. These are just basic things which people often wish to know about a script.

- For the rest of the script, it is not necessary to comment every line. Most lines it will be self explanatory what they do. Only put comments in for important lines or to explain a particular command whose operation may not be immediately obvious.

Why the ./

Linux is set up the way it is, largely for logical reasons. This peculiarity actually makes the system a bit safer for us. First a bit of background knowledge. When we type a command on the command line, the system runs through a preset series of directories, looking for the program we specified. We may find out these directories by looking at a particular variable PATH (more on these in the next section).

```
user@bash: echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/X11R6/bin:/usr/games:/usr/lib/mit/bin:/usr/lib/mit/sbin
user@bash:
```

The system will look in the first directory and if it finds the program it will run it, if not it will check the second directory and so on. Directories are separated by a colon (:).

The system will not look in any directories apart from these, it won't even look in your current directory. We can override this behaviour however by supplying a path. When we do so the system effectively says "Ah, you've told me where to look to find the script so I'll ignore the PATH and go straight to the location you've specified instead." You'll remember from section 2 (Basic Navigation) that a full stop (.) represents our current directory, so when we say ./myscript.sh we are actually telling the system to look in our current directory to find the script. We could have used an absolute path as well (/home/ryan/linuxtutorialwork/myscript.sh) and it would have worked exactly the same, or a relative path if we are not currently in the same directory as the script (../linuxtutorialwork/myscript.sh).

- If it were possible to run scripts in your current directory without this mechanism then it would be easy, for instance, for someone to create a malicious script in a particular directory and name it ls or something similar. People would inadvertently run it if they wanted to see what was in that directory.

Permissions

A script must have the execute permission before we may run it (even if we are the owner of the file). For safety reasons, you don't have execute permission by default so you have to add it. A good command to run to ensure your script is set up right is:

- chmod 755 <script>

Section 13: Bash Scripting - Variables

A variable is a container for a simple piece of data. They are useful if we need to work out a particular thing and then use it later on. Variables are easy to set and refer to but they have a specific syntax that must be followed exactly for them to work.

- When we set a variable, we specify it's name, followed directly by an equals sign (=) followed directly by the value. (So, no spaces on either side of the = sign.)
    - When we refer to a variable, we must place a dollar sign ($) before the variable name.

```
user@bash: cat variableexample.sh
#!/bin/bash
# A simple demonstration of variables
# Ryan 5/3/2021

name='Ryan'
echo Hello $name

user@bash: ./variableexample.sh
  Hello Ryan
  user@bash:
```

Command Line Arguments & More

When we run a script, there are several variables that get set automatically for us. Here are some of them:

- $0 - The name of the script.

- $1 - $9 - Any command line arguments given to the script. $1 is the first argument, $2 the second and so on.

- $# - How many command line arguments were given to the script.

- $* - All of the command line arguments.

There are other variables but these should be enough to get you going for now.

```
user@bash: cat morevariables.sh
#!/bin/bash
# A simple demonstration of variables
# Ryan 5/3/2021

echo My name is $0 and I have been given $# command line arguments
echo Here they are: $*
echo And the 2nd command line argument is $2

user@bash: ./morevariables.sh bob fred sally
My name is morevariables.sh and I have been given 3 command line arguments
Here they are: bob fred sally
And the 2nd command line argument is fred
user@bash:
```

Backticks

It is also possible to save the output of a command to a variable and the mechanism we use for that is the backtick (`) (Note it is a backtick not a single quote. Typically you'll find the backtick on the keyboard to the left of the 1 (one) key.).

```
user@bash: cat backticks.sh
#!/bin/bash
# A simple demonstration of using backticks
# Ryan 5/3/2021

lines=`cat $1 | wc -l`
echo The number of lines in the file $1 is $lines

user@bash: ./backticks.sh testfile.txt
The number of lines in the file testfile.txt is 12
```

A Simple Backup Script

Now let's put the stuff we've learnt so far into a script that actually does something useful. I keep all my projects in separate directories within a directory called projects in my home directory. I regularly take a backup of these projects and keep them in dated folders within a directory called projectbackups also in my home directory.

```
user@bash: cat projectbackup.sh
#!/bin/bash
# Backs up a single project directory
# Ryan 5/3/2021

date=`date +%F`
mkdir ~/projectbackups/$1_$date
cp -R ~/projects/$1 ~/projectbackups/$1_$date
echo Backup of $1 completed

user@bash: ./projectbackup.sh ocelot
Backup of ocelot completed
user@bash:
```

You'll notice that I have used relative paths in the above script. By doing this I have made the script more generic. If one of my workmates wished to use it I could give them a copy and it would work just as well for them without modification. You should always think about making your scripts flexible and generic so they may easily be used by other users or adapted to similar situations. The more reusable your scripts are, the more time goes on, the less work you have to do :)

Section 13: Bash Scripting - if statements

So the above backup script makes my life a little easier, but what if I make a mistake? The script may fall over in a mess of error messages. In the example below I will introduce if statements. I'll only touch on them briefly. You should be able to work out their usage from the example and notes below.

- (If this all seems too confusing, don't worry too much. Even with just the knowledge above you can still write quite useful and practical scripts to make your life easier.)

```
user@bash: cat projectbackup.sh
#!/bin/bash
# Backs up a single project directory
# Ryan 5/3/2021

if [ $# != 1 ]
then
    echo Usage: A single argument which is the directory to backup
    exit
fi
if [ ! -d ~/projects/$1 ]
then
    echo 'The given directory does not seem to exist (possible typo?)'
    exit
fi
date=`date +%F`

# Do we already have a backup folder for todays date?
if [ -d ~/projectbackups/$1_$date ]
then
    echo 'This project has already been backed up today, overwrite?'
    read answer
    if [ $answer != 'y' ]
    then
        exit
    fi
else
```

```
28.         mkdir ~/projectbackups/$1_$date
29.     fi
30.     cp -R ~/projects/$1 ~/projectbackups/$1_$date
31.     echo Backup of $1 completed
32.     user@bash:
```

Let's break it down:

- **Line 6** Our first if statement. The formatting is important. Note where the spaces are as they are required for it to work properly. In this statement we are asking if the number of arguments ($#) is not equal to (!=) one.
- **Line 8** If not then the script has not been properly invoked. Print a message explaining how it should be used.
- **Line 9** Because the script has not been invoked properly we wish to exit the script before going any further.
- **Line 10** To indicate the end of an if statement we have a single line which has fi (if backwards) on it.
- **Line 11** If statements can test a lot of different things. Here the exclamation mark (!) means not, the -d means 'the path exists and is a directory'. So the line reads as 'If the given directory does not exist'
- **Line 22** It is possible to ask the user for input. The command we use for that is read. read takes a single argument which is the variable to store the answer in.
- **Line 23** Let's see how the user responded and act accordingly.

Tip:
You'll notice that certain lines are indented in the above code. This is not necessary but is generally considered good practice as it makes the code a lot easier to read.

Tip:
If statements actually make use of a command called test. If you would like to know all the different comparisons you may perform then have a look at the manual page for test.

Section 13: Bash Scripting - Summary

- #! (Shebang) - Indicates which interpreter a script should be run with.
- echo - Print a message to the screen.
- which - Tells you the path to a particular program.
- $ - Placed before a variable name when we are referring to it's value.
- ` ` (Backticks) - Used to save the output of a program into a variable.
- date - Prints the date.
- if [ ] then else fi - Perform basic conditional logic.
- Behaves the same - Anything you may do on the command line you may do in a script and it will behave exactly the same.
- Formatting - Bash scripts are particularly picky when it comes to formatting. Make sure spaces are put where they are needed and not put when they are not needed.