Introduction to the Linux Command Line

16 October 2018

Paul Sumption ps459@cam.ac.uk
Mark Sharpley mcs94@cam.ac.uk

Based on the Introduction UNIX command line course developed by: Bob Dowling and Julian King

Table of Contents

Notation

Warnings!

These sections are used to highlight common mistakes.

Exercises

You are expected to complete all exercises. Some of them do depend on previous exercises being successfully completed.

An indication is given as to how long we expect the exercise to take. Do not panic if you take longer than this. If you are stuck, ask a demonstrator.

Input and output

Material appearing in a terminal is presented like this:

```
$ more lorem.txt
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
```

The material you type is presented like this: **`ls`**. (Bold face, typewriter font.)

The material the computer responds with is presented like this: "`Lorem ipsum`". (Typewriter font again but in a normal face.)

Keys on the keyboard

Keys on the keyboard will be shown as the symbol on the keyboard surrounded by square brackets, so the "A key" will be written "[A]". Note that the return key (pressed at the end of every line of commands) is written "[□]", the shift key as "[□]", and the tab key as "[□]". Pressing more than one key at the same time (such as pressing the shift key down while pressing the A key) will be written as "[□]+[A]". Note that pressing [A] generates the lower case letter "a". To get the upper case letter "A" you need to press [□]+[A].

Content of files

The content[1] of files (with a comment) will be shown like this:

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur.                                    This is a comment about the line.
```

1 The example text here is the famous "lorem ipsum" dummy text used in the printing and typesetting industry. It dates back to the 1500s. See `http://www.lipsum.com/` for more information.

Course History

The material for this course has been modified so that today's session can be taught in the Bioinformatic Training Facility. Normally when delivered by my UIS colleagues it is taught at the UIS managed MCS training rooms.

A map of the UIS run MCS facilty's can be found here - https://help.uis.cam.ac.uk/service/devices-networks-printing/managed-desktops/mcs/mcr-rooms

Details of the MCS Linux service can be found here - https://help.uis.cam.ac.uk/service/devices-networks-printing/managed-desktops/mcs/basiclinux

The course has been designed as 'self paced', the idea was that you could either:

a) Obtain an MCS account, download the course and then start teaching yourself using the notes.
b) Book a place on a course run by UIS, there is an instructor present to assist you if you get stuck rather than to lecture or provide an instructor lead course.

I have adapted this course and split the self paced material into several sections. At the start of each section I will present some slides to introduce the topic before you move onto the self paced material.

With a copy of the notes and exercise files most if not all exercise should be repeatable on your own Linux machine (other than the remote file transfer as this is a machine built just for today's course)

The future aim for this course is:

- To adapt the material you find useful
- Convert it to a web based 'read the docs' style website
- Record exercises as 'terminal videos' basically screen captures of the completed exercises so you can check your work
- Provide a short video on how to make a Linux virtual machine in Virtual Box on your laptop. This would let give you a Linux machine to do the exercises on (assuming its not already got Linux on it!)

Booting & logging in

“Booting” is the name given to the process that happen when we power on on restart our compute. It gives us a fully functioning computer ready to help you with your work. It takes its name from “bootstrapping”, the magical ability to lift yourself up by your own boot straps.
In the simplest case a computer will boot into its one and only operating system, be it Microsoft Window or Linux

Some computers, such as those in this classroom, can be set up so that they can boot into either Windows or Linux, with the choice being made by the user at boot time.

If the computer is already running Windows then we need to reboot from Windows to Linux.

Rebooting from Windows to Linux on the training PC’s

1. Log out from Windows to return to the login window.
2. Click the “Shutdown” button.
3. Select “Shutdown and restart” from the menu offered.
4. As the system boots you will be offered a menu of Windows and Linux.
5. Use the down arrow [↓] to select Linux.
6. Press Return, [□].
7. Click the [OK] button to continue.
8. You will then be asked for your user name. This will be issued to you by your instructor. The demonstrator has used the user name ‘y250’ in all examples in the notes; when you attempt the practical you should change ‘y250’ to the username you have been issued.
9. Enter your user name and press return [□].
10. You will be asked for your password. For this course you have all been given the same (not very good) password. Enter it and press [□] again. (For your own account you should have a better password and should not share it with anybody else.)
11. After about 15 seconds you should be logged in and should see the “message of the day” window in the middle of your screen.

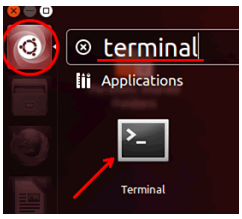
Exercise 1 (5 minutes):

Reboot into Linux and log in with the user name you have been given.

Section 1: Terminal windows and text consoles

To launch a terminal window in Ubuntu Linux:

Select the Terminal icon from the launcher sidebar. 

Or click in the top left corner on the search icon and then type 'terminal'. 

You can launch multiple terminals if you want. Each runs an independent command line interpreter ("shell").

When the terminal opens you will see a set of text that looks like this in the graphical terminal window or the plain text console:

```
workstation:~$
```

The text at the start of the line is called the "prompt" and its purpose is to prompt you to enter some commands.

The prompt can be changed (see below) but the default prompts on Linux have these components:

`workstation`	Your computer
`:`	separator
`~`	The directory your session is "in", also known as the "current working directory". "~" is shell short hand for "your home directory".
`$`	Final separator.

To issue a command at the prompt simply type the name of the command and press the Return key, [□]. For example, the `ls` command lists the files in the current working directory:

```
workstation:~$ ls
Desktop  Library  My Music  My Pictures  My Video  Linux Intro LinuxIntro.tgz

workstation:~$
```

Note that the prompt is repeated after the `ls` command has completed.

Logging out

Once we are finished with a terminal or a terminal window we need to quit. We will illustrate three ways to do this.

Close the window

In the graphical environment the terminal window is just another window. At its top right corner are the three buttons for minimising, maximising and closing. If you click in the [×] button the window is closed and the session cleanly ended.

The exit command

In either a terminal window or a text console you can issue the command "`exit`"; this will end the session. In the graphical environment ending the session running in a window closes the window too. In a text console the console is typically cleared and a fresh login prompt presented.

[Ctrl]+[D]

Recall that "[Ctrl]+[D]" means to press down the [Ctrl] key at the same time as the [D] key. In practice we press the [Ctrl] key down, press and release the [D] key, and then release the [Ctrl] key.

On a Linux system [Ctrl]+[D] means “end of input”. We will meet it later when we are entering data into a command and want to signal that we have finished. Here it signals to the shell that we have no more input for it so it might as well quit. And quit it does.

Text console

If you are a Linux server administrator it is common to dispense with the graphical environment entirely (assuming one is installed) and just use a text mode console.

Read all of this section before attempting the exercise! You can press [Ctrl]+[Alt]+[F2] to get a pure text login console. If we enter our login and password again we can log in here too. Note that we can be logged in both through the graphical interface and the text interface(s) simultaneously. In fact, we could be logged in to one interface and our neighbour could log in through another. Linux is a fully multi-user operating system. Identical interfaces are available by using [F3], [F4], [F5], or [F6] in place of [F2]. [F1] has a text console that tends to be more colourful.

Using [Ctrl]+[Alt]+[F7] returns you to the graphical interface.

Warning!

When you switch between consoles there is often a several second black screen as the console switches over. Please be patient and let the switch complete before you start switching back again. If you try to switch while the consoles are in mid-switch you can jam things badly.

Exercise 2: Multiple logins (5 minutes)

1. Log in to the graphical interface if you are not already logged in.
2. Start up two terminal windows using the terminal icon.
3. Switch to the [Ctrl]+[Alt]+[F2] console and log in there too.
4. Run the command “`w`”. This shows who is logged in and where. You should get something like this:
5. Type exit
6. Now press [Ctrl]+[Alt]+[F7] and you should be back at your desktop environment.

```
workstation:~$ w
 10:20:00 up 16 min,  5 users,  load average: 0.01, 0.08, 0.16
USER  TTY   LOGIN@  IDLE   JCPU   PCPU   WHAT
y250  tty2  10:10   10:00  0.05s  0.05s  -bash
y250  tty3  10:10   0.00s  0.07s  0.00s  w
y250  tty7  10:06   10:07  3.39s  0.47s  /usr/bin/gnome-session
y250  pts/0 10:09   10:27  0.02s  0.02s  bash
y250  pts/1 10:09   10:20  0.04s  0.04s  bash
workstation:~$
```

The user column will show your ID rather than `y250`, of course.
The `w` command summarizes various bits of data which can also be seen individually. Try the “`who`” and “`uptime`” commands as well.
Log out of one of the text consoles when you are done.

Warning!

It is easy to forget to log out of sessions that aren't right in front of your eyes. Logging out of the graphical interface will not log you out of any of the text interfaces.

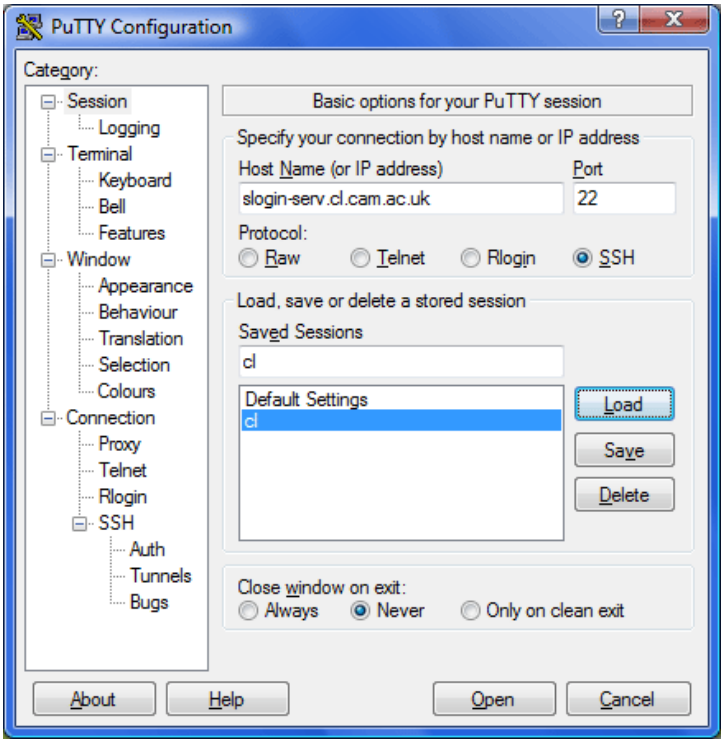
SSH clients for Windows and Mac OS X

This is a Linux course so we won't talk about Windows much. However, now that we have seen how to connect to a remote Linux command line from a Linux box with `ssh`, it is quite like that you might ask how to connect from a Windows or OS X machine.

Windows

A popular Windows application for this purpose is called "`putty`":

The `putty` application can be downloaded free from `http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`, courtesy of its author, Simon Tatham.



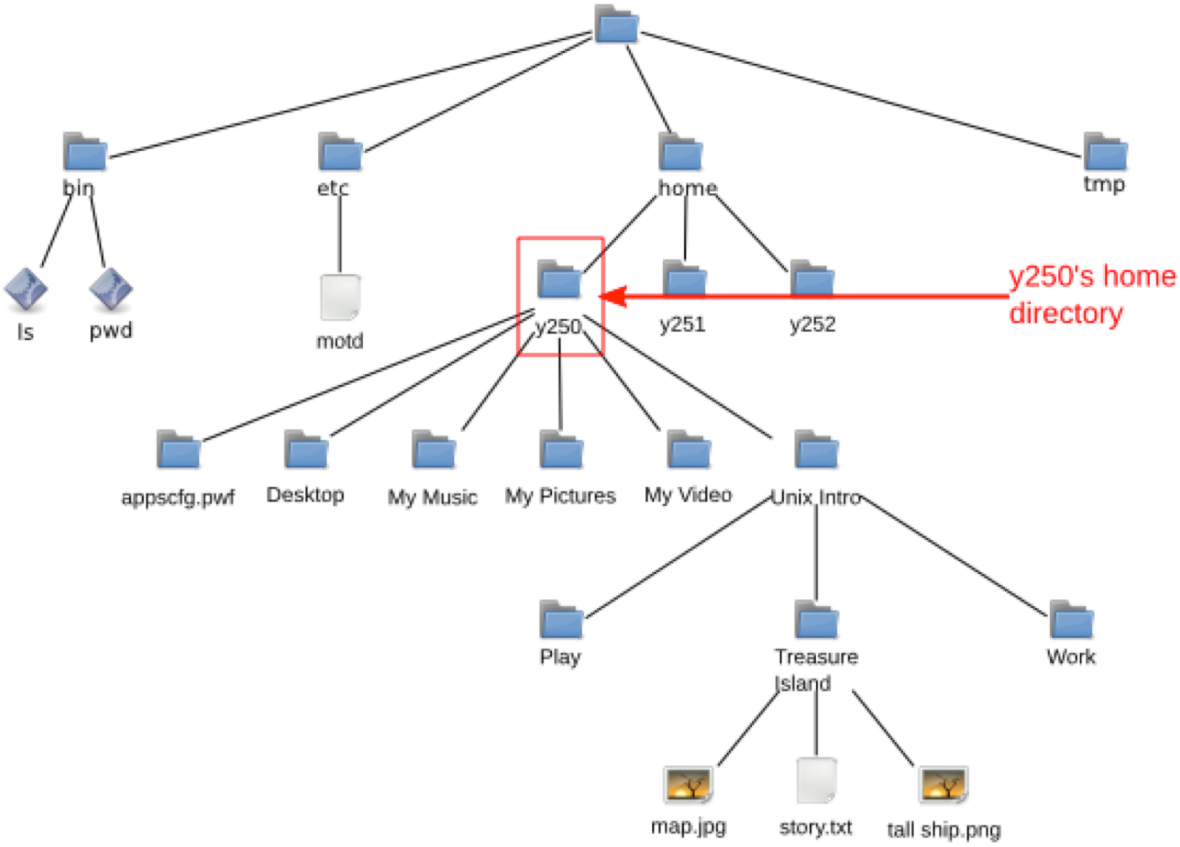
OS X

OS X has a built in terminal as its underlying operating system is Unix based. The terminal application can be found under application > utilities. Command line SSH and SFTP are also installed by default.

Section 2: Navigating the file system in the CLI

All the elements of a Linux system form a hierarchy called the “file system”. This hierarchy is a tree of folders (typically called “directories” in the Linux world) and files containing content. (Actually, there are other types of thing in the file system but we don't need to worry about them here.)

A subset of that hierarchy belongs to you and takes the form of a subset of the hierarchy hanging off your “home directory”. The various logins you have done so far all start you off in your home directory. This is the standard place to start. If you go looking for files or creating them then the path to them starts here.



Directories

Working directory

To know what directory we are in we use the command `pwd`, “print working directory”:

```
workstation:~$ pwd
/home/y250

workstation:~$
```

Directory contents

To see what is in the current directory we use the `ls` (“**ls**t”) command:

```
workstation:~$ ls
Desktop  My Music  My Pictures  My Video  Linux Intro

workstation:~$
```

These are all directories. The `ls` on most Linux shells uses colours to indicate types of files, and directories are coloured blue. Also note that with the exception of “`Desktop`” all the names of directories have spaces in them.

We can get more information from `ls` about the contents of the directory by asking for its “long” output. We

do this by adding an option, "`-l`" (for "**l**ong"), to the command. Note that there is a space between the "`ls`" and the "`-l`":

```
workstation:~$ ls -l
total 3
drwxr-xr-x 1 y250 y250 512 2018-04-28 12:51 Desktop
drwxr-x--- 1 y250 y250 512 2018-04-28 13:11 My Music
drwxr-x--- 1 y250 y250 512 2018-04-28 13:11 My Pictures
drwxr-x--- 1 y250 y250 512 2018-04-28 13:11 My Video
drwxr-xr-x 1 y250 y250 512 2018-04-28 13:22 Linux Intro

workstation:~$
```

(We will not usually put in spaces explicitly as "␣" in future.)

We can analyse the output of `ls -l` by looking at the `Desktop` line in detail. It will be easier to explain if we work through it right to left, though.

`Desktop`	This is the name of the file or directory.
`2018-04-24 11:37`	This is the date and time of the last update to the file, or the date it was created if it has not been updated since. The format of this time stamp varies between Linux distributions; some spell out the date less numerically.
`512`	This is the number of bytes taken by the file. Directories have a database structure within them and their sizes are typically multiples of 512 bytes. General files will have arbitrary sizes.
`y250 y250`	The first instance of `y250` is the owner of the file. This is typically the user who created the file (or for whom it was created in this case). Users can be lumped together into groups and it is common practise to place each user into a group of their own. The second `y250` is the group associated with the file. In this specific case the user `y250` is the only member of the group `y250`. On some Linux systems all the users are placed in a group called "`users`" and that group is used.
`1`	A property of the Linux file system is that more than one name can correspond to the same file. This number, called the "reference count", is the number of names that correspond to this file or directory. In our simple case our files all have just one name.
`rwxr-xr-x`	These are the permissions on the file or directory. They form three triplets, each a letter or a dash. The letter shows the permission is granted and the dash that it is denied. We can consider the permissions one triple at a time:
`rwx`	The first triple identify the permissions granted to the owner of the file (user `y250`). The user may **r**ead from, **w**rite to and e**x**ecute the file. Read and write permissions mean exactly what they say. "Execute" permission means that the owner may run this file as a program. In the case of a directory, the read permission means that the owner may look to see the names of the files within the directory, the write permission means that the owner may add or remove things in the directory, and the execute permission means that the owner can change directory into it.
`r-x`	The second triple indicates the permissions granted to members of the group who are not the owner. The middle dash means that the write right is not granted.
`r-x`	The third triple indicates the permissions granted to any user who is not in the group or the owner of the file.
`d`	The first character indicates that this is a directory. If it was a plain file the symbol would be a dash.

Changing directory

Now we know we have some directories, let's use one of them, "`Linux Intro`". The command to **c**hange **d**irectory is `cd`. But things don't go quite as planned:

```
workstation:~$ cd Linux Intro
-bash: cd: Linux: No such file or directory

workstation:~$
```

What has gone wrong here is that the shell is using the space character to split up the various bits of the command line just as it split the command `ls` from its option `-l`. So here the `cd` command is being given two

words to work on (called “arguments” in the jargon). It's only expecting one, the directory to change to, and ignores the second. It tries to change to a directory called “`Linux`” and fails because no such directory exists.

The error message can be understood as a series of reports:

`bash`	The error came from the shell (`bash`).
`cd`	`bash` was running `cd`.
`Linux`	`cd` had a problem with the `Linux` directory.
`No such file or directory`	The problem was that it didn’t exist.

There are two “proper” ways round this and one neat trick that will avoid the problem for ever.

The proper ways involve informing the shell that the space between “`Linux`” and “`Intro`” is not the same as the space between “`cd`” and “`Linux`”.

Quoting

The first way to do this is to place the name of the directory in quotes. The quotes tell the shell to treat everything inside them as a single word and not to split it up on spaces.

```
workstation:~$ pwd
/home/y250

workstation:~$ cd "Linux Intro"

workstation:Linux Intro$ pwd
/home/y250/Linux Intro
```

Now that we've completed that demonstration we need to know how to get back to our home directory. If you run the command `cd` with no argument then it defaults to your home directory:

```
workstation:Linux Intro$ pwd
/home/y250/Linux Intro

workstation:Linux Intro$ cd

workstation:~$ pwd
/home/y250
```

Also note how the prompt changes to indicate the current directory too.

Escaping

The second approach is to specifically identify the space character as “nothing special, just another character” so the shell doesn't use it for splitting. We do this by preceding the space with a backslash character, “`\`”. This is called “escaping” the character:

```
workstation:~$ pwd
/home/y250

workstation:~$ cd Linux\ Intro

workstation:Linux Intro$ pwd
/home/y250/Linux Intro

workstation:Linux Intro$ cd

workstation:~$ pwd
/home/y250

workstation:~$
```

And now the good news: you will never have to remember to type the quotes or backslash again. In fact you're about to do a lot less typing.

File name completion

Let's start again in the home directory. This time we will start to type the name of our double-barrelled directory, “`Linux Intro`”, but only get as far as the first letter. Then we hit the Tab key, [□].

```
workstation:~$ cd U□
```

At this point the shell determines that there is only one file or directory present that starts with a "U" and automatically extends the file name on the command line, adding any escaping that is necessary:

```
workstation:~$ cd Linux\ Intro/

workstation:Linux Intro$
```

In the case of completion of a directory name it even puts on a terminal slash, "/", in case you want to continue the quoting of a file within that directory. A trailing slash makes no difference to us so we leave it.

It's not always possible for the shell to determine exactly what file or directory you want. We can see this if we return to the home directory and look at some of the other directories:

```
workstation:Linux Intro$ cd

workstation:~$ ls
Desktop  My Music  My Pictures  My Video  Linux Intro
```

This time we type "cd M" and then press [□].

```
workstation:~$ cd M□
```

There are three directories that start with an "M". The shell extends the name as far as it can and beeps to let you know that it needs help to go further:

```
workstation:~$ cd My\□
```

(We have explicitly marked the space it has added with a "□".)

We then press one more letter to distinguish between "My Music", "My Pictures" and "My Videos" (an "M", a "P", or a "V") and press [□] again:

```
workstation:~$ cd My\□P□
```

and the shell can now complete the directory name:

```
workstation:~$ cd My\□Pictures/
```

The use of the Tab key and its automatic escaping of file names will save us a lot of typing and also having to remember the backslashes or quotes.

Directories again

Let's return to the Linux Intro directory and take another look at directory navigation. We have seen how to enter a directory and to return to our home directory. How can we move up the directory tree from our current location? The answer lies in a couple of "hidden" directories in every directory. We can see hidden files and directories with another option to ls, "-a" (for "**a**ll"):

```
workstation:Linux Intro$ pwd
/home/y250/Linux Intro

workstation:Linux Intro$ ls
Play  Treasure Island  Work

workstation:Linux Intro$ ls -a
.  ..  Play  Treasure Island  Work

workstation:Linux Intro$ ls -la
total 3
drwxr-xr-x 1 y250 y250 512 2018-04-28 12:07 .
drwxr-xr-x 1 y250 y250 512 2018-04-28 13:27 ..
drwxr-xr-x 1 y250 y250 512 2018-04-23 18:20 Play
drwxr-xr-x 1 y250 y250 512 2018-04-27 19:06 Treasure Island
drwxr-xr-x 1 y250 y250 512 2018-04-27 19:19 Work

workstation:Linux Intro$
```

The ls options -l and -a can be combined as "ls -al", "ls -a -l", "ls -la", or "ls -l -a". They are all equivalent.

Every Linux directory has the two entries "`.`" and "`..`"; they are created automatically and you can't remove them. The first, "`.`", refers to the directory itself, so "`cd .`" has no effect:

```
workstation:Linux Intro$ pwd
/home/y250/Linux Intro

workstation:Linux Intro$ cd .

workstation:Linux Intro$ pwd
/home/y250/Linux Intro

workstation:Linux Intro$
```

The second, "`..`", refers to the parent directory. Changing directory to "`..`" takes you up one level in the directory tree:

```
workstation:Linux Intro$ pwd
/home/y250/Linux Intro

workstation:Linux Intro$ cd ..

workstation:~$ pwd
/home/y250

workstation:~$
```

We have just been dipping in and out of one directory directly beneath our home directory. As a result, plain "`cd`" is enough to return us to where we were before. Alternatively we have been able to use "`cd ..`" to get back.

More generally, "`cd -`" will take us back to our previous directory:

```
workstation:~$ cd Linux Intro

workstation:Linux Intro$ pwd
/home/y250/Linux Intro

workstation:Linux Intro$ cd -

workstation:~$ pwd
/home/y250

workstation:~$ cd -

workstation:Linux Intro$ pwd
/home/y250/Linux Intro

workstation:Linux Intro$ cd -

workstation:~$ pwd
/home/y250

workstation:~$
```

File paths

We don't need to enter a directory to see what's in it. We don't need to only change one level of directory tree at a time either.

We will start in our home directory. We want to run `ls` on the Treasure Island directory in the "`Linux Intro`" directory. We refer to the directory within another directory by separating their names with a slash, "`/`". Note that there are no spaces around the slash:

```
workstation:~$ ls Linux\ Intro/Treasure\ Island/
map.jpg  story.txt  tall ship.png

workstation:~$
```

(Also note how few keys had to be hit to enter that command: "`ls□U□T□□`" . Tab completion is your friend.)

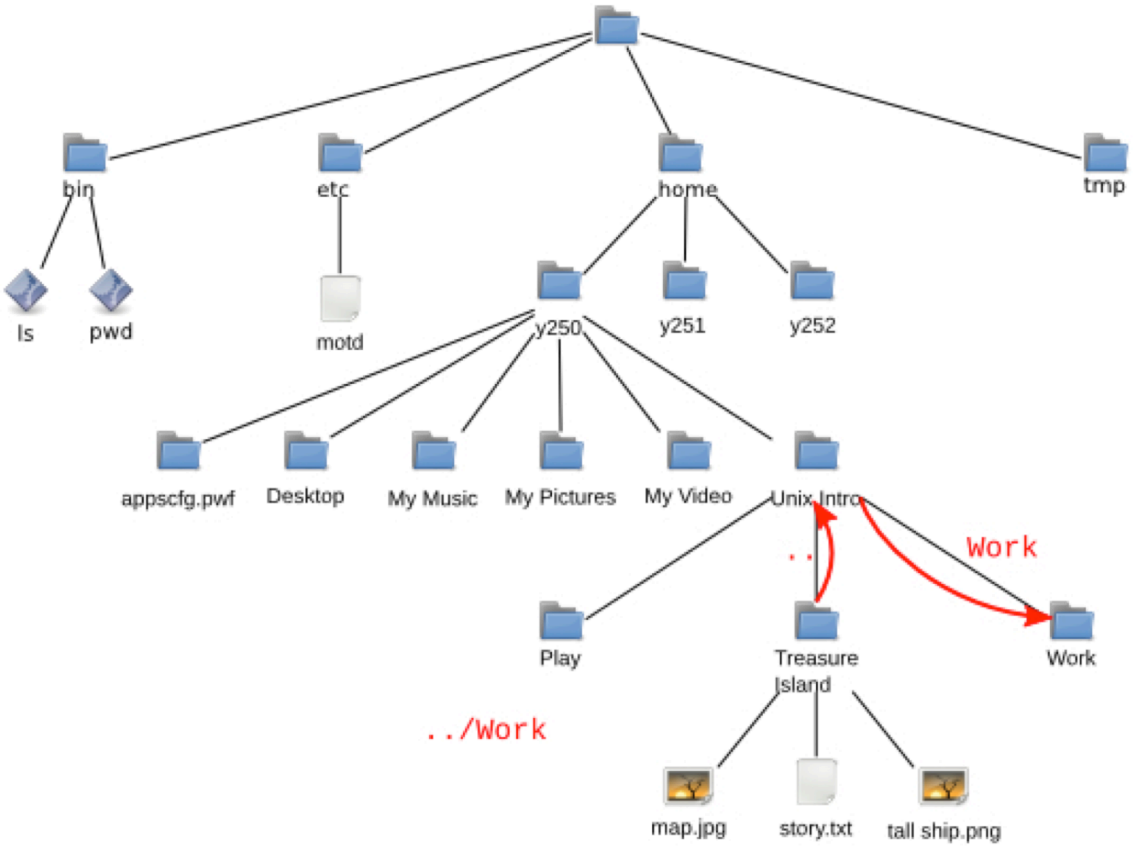
We can also change directory directly rather than having to pass through the intermediate directory:

```
workstation:~$ cd Linux\ Intro/Treasure\ Island/

workstation:Treasure Island$
```

We can also use "`..`" in these file paths:

```
workstation:Treasure Island$ pwd
/home/y250/Linux Intro/Treasure Island

workstation:Treasure Island$ ls -l ../Work/
total 10
-rw-r--r-- 1 y250 y250   36 2018-04-27 19:12 abc.txt
-rw-r--r-- 1 y250 y250   36 2018-04-27 19:13 def.txt
-rw-r--r-- 1 y250 y250   36 2018-04-27 19:13 ghi.txt
-rw-r--r-- 1 y250 y250 3664 2018-04-27 19:14 lorem.txt
-rw-r--r-- 1 y250 y250 3664 2018-04-27 19:15 nonsense.txt
drwxr-xr-x 1 y250 y250  512 2018-04-27 19:18 Project Alpha
```

In this last case "`..`" has taken us up one level and then "`Work`" has taken us back down again into another directory:



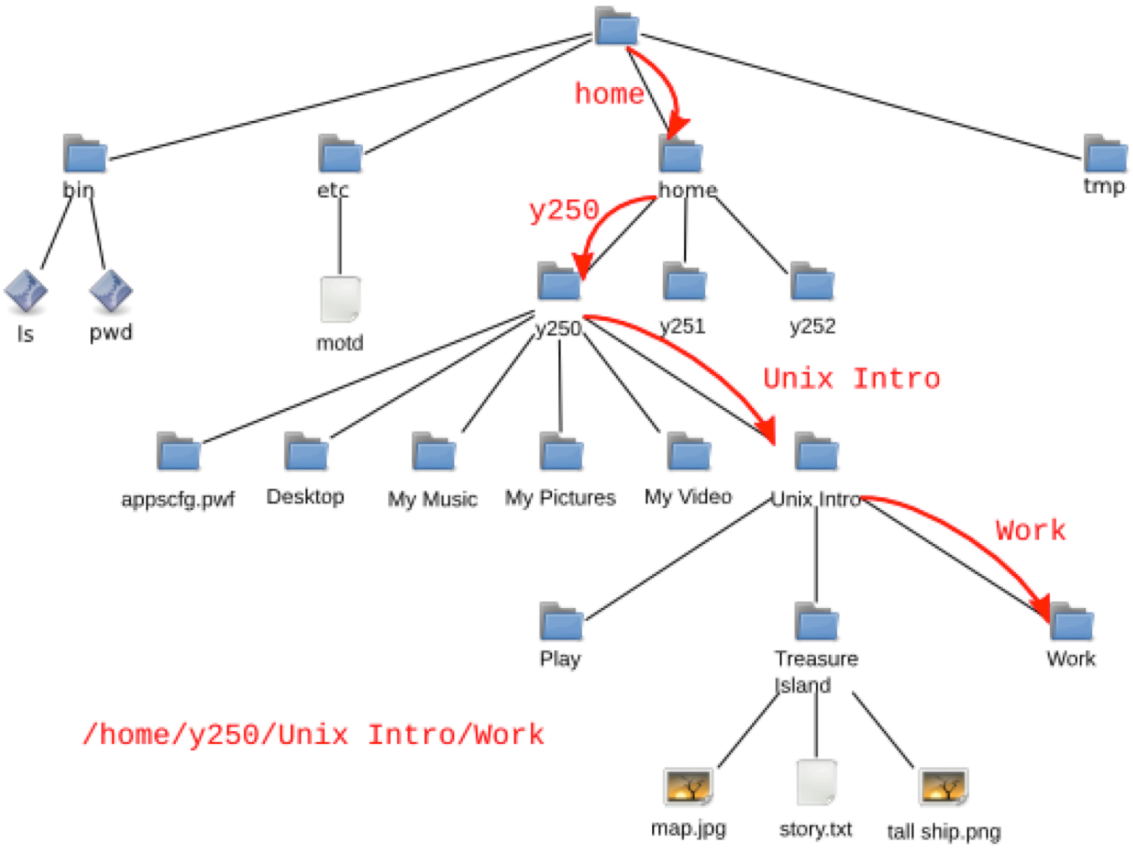
Warning!

Note that a *forward* slash, "`/`" is used in file paths to separate components and a *back* slash, "`\`", is used to escape spaces in commands. They are *not* the same.

File Paths

Our file paths so far have all started with a component in the current working directory. (Recall that "`..`" is technically within the directory even though it points up a level.) These are called "*relative* paths" and refer to files and directories relative to the current working directory.

If a file path starts with a forward slash, "`/`", then it is evaluated relative to the top (the "root") of the file tree. This is called an "*absolute* path".



So if our current working directory is "`Treasure Island`" then we can refer to the "`Work`" directory either as "`../Work`" (relative path) or as "`/home/y250/Linux\ Intro/Work`" (absolute path).

Exercise 3: Navigating your home folder (10 minutes)

1. Run the "`cd`" command on its own to start in your home directory.
2. Run "`pwd`" to check where you are.
3. In a *single* `cd` command, using a relative path, change directory to the `Work` subdirectory of the `Linux Intro` directory.
4. Run "`pwd`" to check where you are.
5. Starting from the `Work` directory, move into the `Play` directory with a single `cd` command and a relative path.
6. What do you think the absolute path is for your current directory?
7. Run "`pwd`". Were you right?

Hint: Remember the various ways to allow for the space in `Linux Intro`.

Renaming, creating and deleting file and directories

Now we can manoeuvre around the file system, we need to know how to manipulate it. We will start in the "`Treasure Island`" directory.

```
workstation:Work$ pwd
/home/y250/Linux Intro/Treasure Island

workstation:Treasure Island$ ls
map.jpg  story.txt  tall ship.png

workstation:Treasure Island$
```

Renaming and moving items

Suppose we want to rename the file "`tall ship.png`" to "`hispaniola.png`"[2]. We do this with the `mv` ("**mov**e") command. Note the use of tab completion to avoid having to explicitly escape the space in the file's name.

```
workstation:Treasure Island$ ls
map.jpg  story.txt  tall ship.png
workstation:Treasure Island$ mv tall\ ship.png hispaniola.png
workstation:Treasure Island$ ls
hispaniola.png  map.jpg  story.txt
```

The `mv` command's name is more obvious when used to move a file into another directory:

```
workstation:Treasure Island$ mv story.txt ..
workstation:Treasure Island$ ls
hispaniola.png  map.jpg
workstation:Treasure Island$ ls ..
Play  story.txt  Treasure Island  Work
workstation:Treasure Island$
```

We can move files between directories and rename them simultaneously:

```
workstation:Treasure Island$ mv map.jpg ../island.jpeg
workstation:Treasure Island$ ls
hispaniola.png
workstation:Treasure Island$ ls ..
island.jpeg  Play  story.txt  Treasure Island  Work
workstation:Treasure Island$
```

We can use `mv` on directories just as we do for files within our home directory. (Things can get more complicated if you want to move directories to other parts of the system.)

Copying files

To copy a file we use the command `cp` ("**co**p**y**") just like we used `mv`:

```
workstation:Treasure Island$ ls
hispaniola.png
workstation:Treasure Island$ cp hispaniola.png "tall ship.png"
workstation:Treasure Island$ ls
hispaniola.png  tall ship.png
workstation:Treasure Island$
```

Note how we still have to use quotes (or backslashes) when describing new files. They don't exist yet so tab completion can't find them.

There is a slight wrinkle with using `cp`; it cannot be used to copy directories without an extra option. The option is "`-R`" ("**r**ecursive")[3] which means to copy the directory and everything in it:

2 "Hispaniola" was the ship that took Jim Hawkins, Long John Silver *et al* to Treasure Island.

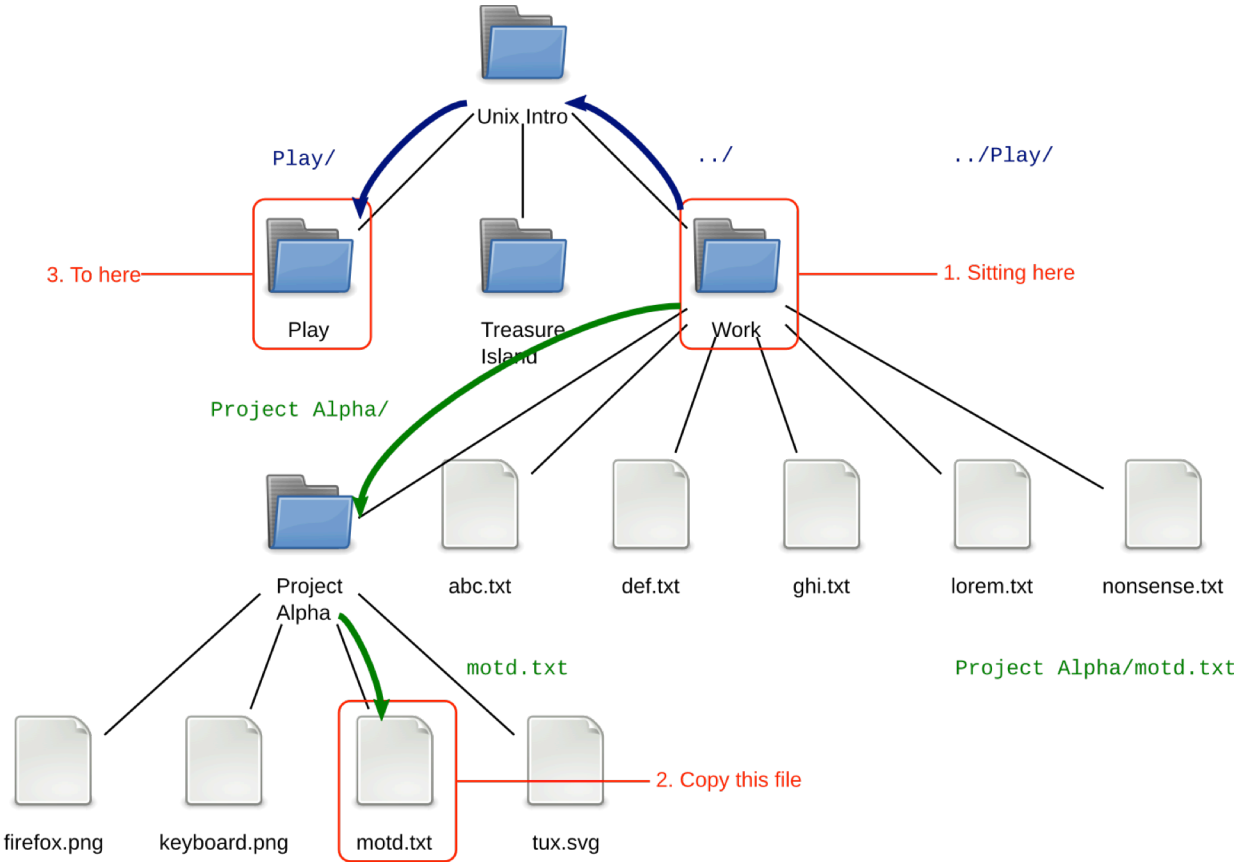
3 On some older versions of Linux this was a lower case letter, "`-r`". However, for consistency between commands capable of recursive behaviour, modern versions have standardised on the upper case "`-R`".

```
workstation:Treasure Island$ cd ..

workstation:Linux Intro$ cp Treasure\ Island/ "Copy of Treasure Island"
cp: omitting directory `Treasure Island/'

workstation:Linux Intro$ cp -R Treasure\ Island/ "Copy of Treasure Island"

workstation:Linux Intro$ ls Copy\ of\ Treasure\ Island/
hispaniola.png  tall ship.png

workstation:Linux Intro$
```

The copy and move commands, `cp` and `mv`, take two arguments[4]: the source and the destination. Each of these is evaluated relative to the current working directory. The destination is *not* evaluated relative to the source.

Warning!

Suppose we were in the Linux Intro/Work directory (i.e. that was our current working directory) and we wanted to copy the motd.txt file in Project Alpha into the Play directory then we would refer to the motd.txt file as Project\ Alpha/motd.txt and its destination location as ../Play/motd.txt.




```
workstation:Work$ pwd
/home/y250/Linux Intro/Work

workstation:Work$ cp Project\ Alpha/motd.txt ../Play/motd.txt

workstation:Work$
```

4 Actually, they can take more but we'll consider the simple case for now.

Creating directories

To create an empty new directory we use the command `mkdir` ("**ma**k**e** **dir**ectory"):

```
workstation:Linux Intro$ pwd
/home/y250/Linux Intro

workstation:Linux Intro$ ls -l
total 428
drwxr-xr-x 1 y250 y250    512 2018-04-28 18:07 Copy of Treasure Island
-rw-r--r-- 1 y250 y250  44928 2008-08-21 18:53 island.jpeg
drwxr-xr-x 1 y250 y250    512 2018-04-23 18:20 Play
-rw-r--r-- 1 y250 y250 390927 2018-04-24 11:38 story.txt
drwxr-xr-x 1 y250 y250    512 2018-04-28 18:06 Treasure Island
drwxr-xr-x 1 y250 y250    512 2018-04-27 19:19 Work

workstation:Linux Intro$ mkdir Fun

workstation:Linux Intro$ ls -l
total 429
drwxr-xr-x 1 y250 y250    512 2018-04-28 18:07 Copy of Treasure Island
drwxr-xr-x 1 y250 y250    512 2018-04-28 18:13 Fun
-rw-r--r-- 1 y250 y250  44928 2008-08-21 18:53 island.jpeg
drwxr-xr-x 1 y250 y250    512 2018-04-23 18:20 Play
-rw-r--r-- 1 y250 y250 390927 2018-04-24 11:38 story.txt
drwxr-xr-x 1 y250 y250    512 2018-04-28 18:06 Treasure Island
drwxr-xr-x 1 y250 y250    512 2018-04-27 19:19 Work

workstation:Linux Intro$ ls -l Fun
total 0

workstation:Linux Intro$
```

Removing files and directories

To remove a file we can use the `rm` ("**rem**ove") command:

```
workstation:Linux Intro$ rm island.jpeg

workstation:Linux Intro$ ls
Copy of Treasure Island  Fun  Play  story.txt  Treasure Island  Work
```

Empty directories can be removed with the `rmdir` ("**rem**ove **dir**ectory") command, but directories with content cannot:

```
workstation:Linux Intro$ rmdir Fun/

workstation:Linux Intro$ ls
Copy of Treasure Island  Play  story.txt  Treasure Island  Work

workstation:Linux Intro$ rmdir Copy\ of\ Treasure\ Island/
rmdir: failed to remove `Copy of Treasure Island/': Directory not empty

workstation:Linux Intro$
```

If you do want to remove a directory and everything within it you need to use the `rm` command with its `-R` ("**r**ecursive" again) option:

```
workstation:Linux Intro$ rm -R Copy\ of\ Treasure\ Island/

workstation:Linux Intro$ ls
Play  story.txt  Treasure Island  Work

workstation:Linux Intro$
```

Warning!

Command line removal with `rm` is for ever. There is no “waste basket” to recover files from.

Exercise 4: Copy and remove files (5 minutes)

1. Start in the `Linux Intro` directory.
2. Copy the file `lorem.txt` from the `Work` directory into the `Linux Intro` directory.
3. In the `Work` directory, create an empty directory called `Project Beta`.
4. Rename `Project Alpha` as `Project Delta`.
5. Copy the file `abc.txt` into `Project Delta`.
6. Copy `Project Delta` and its contents to `Project Epsilon` with a single command.
7. Remove the `Project Delta` directory and its content with a single command.
8. Change directory into the `Treasure Island` directory.
9. Examine its content. The file `story.txt` should no longer be there but be in the parent directory (if you have been following the demonstrator).
10. Without leaving the `Treasure Island` directory, copy the file `story.txt` into the current directory.

Section 3: Anatomy of a command

We will start this section in the Linux Intro directory.

```
workstation:Linux Intro$ pwd
/home/y250/Linux Intro
```

We have seen the use of `ls` on its own to give a simple listing of the content of the current working directory:

```
workstation:Linux Intro$ ls
Fun  lorem.txt  Play  story.txt  Treasure Island  Work
```

and its use with the `-a` and `-l` options to give different output:

```
workstation:Linux Intro$ ls -a -l
total 389
drwxr-xr-x 1 y250 y250    512 2018-11-25 19:39 .
drwxr-x--- 1 y250 y250    512 2018-11-25 19:00 ..
drwxr-xr-x 1 y250 y250    512 2018-11-25 19:37 Fun
-rw-r--r-- 1 y250 y250   3693 2018-11-25 19:39 lorem.txt
drwxr-xr-x 1 y250 y250    512 2018-11-25 19:34 Play
-rw-r--r-- 1 y250 y250 390927 2018-11-25 19:34 story.txt
drwxr-xr-x 1 y250 y250    512 2018-11-25 19:36 Treasure Island
drwxr-xr-x 1 y250 y250    512 2018-11-25 19:40 Work
```

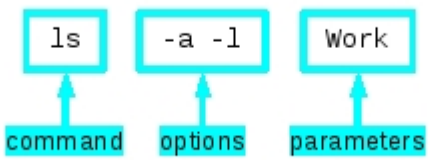
We have also seen it used to peer into another directory:

```
workstation:Linux Intro$ ls Work
abc.txt  ghi.txt    nonsense.txt  Project Delta
def.txt  lorem.txt  Project Beta  Project Epsilon
```

and we can combine these to give:

```
workstation:Linux Intro$ ls -a -l Work
total 12
drwxr-xr-x 1 y250 y250  512 2018-11-25 19:40 .
drwxr-xr-x 1 y250 y250  512 2018-11-25 19:39 ..
-rw-r--r-- 1 y250 y250   36 2018-11-25 19:34 abc.txt
-rw-r--r-- 1 y250 y250   36 2018-11-25 19:34 def.txt
-rw-r--r-- 1 y250 y250   36 2018-11-25 19:34 ghi.txt
-rw-r--r-- 1 y250 y250 3693 2018-11-25 19:34 lorem.txt
-rw-r--r-- 1 y250 y250 3664 2018-11-25 19:34 nonsense.txt
drwxr-xr-x 1 y250 y250  512 2018-11-25 19:39 Project Beta
drwxr-xr-x 1 y250 y250  512 2018-11-25 19:39 Project Delta
drwxr-xr-x 1 y250 y250  512 2018-11-25 19:40 Project Epsilon
```

This is an example of the general case for a Linux command



The difference between an option and a parameter is not as clear cut as we might like. There are optional parameters and some commands even have compulsory options, but for our purposes options start with a dash and parameters don't.

Some options take arguments of their own. For example, `ls` has an option `-w` for setting the width of the output (overriding the width of the terminal). This has to be told what width to use, of course:

```
workstation:Linux Intro$ ls -w 40 Work
abc.txt  lorem.txt     Project Delta
def.txt  nonsense.txt  Project Epsilon
ghi.txt  Project Beta
```

The argument "`40`" counts as part of the options, not the parameters.

Long options

We have seen some options on the command "`ls`". The option "`-a`" lists **a**ll files in a directory. The option "`-h`" shows hidden files, it is normal to use several options with a command.

The short optons approach works so long as you don't need more than 26 options, or perhaps 52 if you use upper and lower case letters. Short options also require you to be able to remember "`a` is for **a**ll" and that "`h` is for **h**idden".

More recent commands have tended to offer long form options as an alternative. This led to the standard form of long options which are introduced with a *double* dash rather than a single one.

```
workstation:~$ ls --all
.               .gconf           My Music         .recently-used.xbel
..              .gconfd          My Pictures      .skel
.gnome2           My Video         .ssh
.bash_history   .gnome2_private  .nautilus        .thumbnails
.dmrc           .ICEauthority    .esd_auth      Library           .pyhistory
.fontconfig     .local           .recently-used

workstation:~$ ls --format=long
total 4
drwxr-xr-x 1 y250 y250 512 2018-11-25 19:48 Desktop
drwxr-x--- 1 y250 y250 512 2018-11-25 19:00 My Pictures
drwxr-x--- 1 y250 y250 512 2018-11-25 19:00 My Video
drwxr-xr-x 1 y250 y250 512 2018-11-25 19:39 Linux Intro
```

Using long options we gain the understanding of what the options mean.

Note that we can't combine these long options the same way we can the single character options. The command "`ls -l -a`" can be written "`ls -la`". The command "`ls --format=long --all`" cannot be written "`ls --format=longall`".

Warning!

The long form of a short option may not always be what you expect it to be. If in doubt check the man pages often the command can show you a list of short and long options i.e. **ls `--help`**

Exercise 5: Long options (10 minutes)

1. Try `ls` --help
2. Try some the long option `--format` sub-options: `across`, `commas`, `horizontal`, `long`, `single-column`, `verbose`, and `vertical`.
3. When do `horizontal` and `vertical` produce different results?

Reading the manual

There are 39 single character options on `ls and` 39 long format options, some of which correspond to the short options. How are we supposed to keep track of this?

Some commands, such as `ls`, offer built in help facilities:

```
workstation:~$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort.

…

Exit status is 0 if OK, 1 if minor problems, 2 if serious trouble.

Report bugs to <bug-coreutils@gnu.org>.
```

Note that the information may go flying off the top of the screen. We will see how to use a utility, "`more`", to solve this problem later.

Not all commands are as well written, though. Linux has a command for printing out the manual pages for commands (and other things too) called "`man`" (short for "**man**ual").

```
workstation:~$ man ls

LS(1)                          User Commands                          LS(1)

NAME

       ls - list directory contents


SYNOPSIS

       ls [OPTION]... [FILE]...

…
```

To advance the output by a screen, press the space bar and to quit press [Q]. This is actually the `more` utility mentioned above but `man` has it built in so we don't need to call it explicitly.

The sections of a manual page are fairly standard:

```
NAME
       ls - list directory contents
```

The name section repeats the name of the command and gives a one-line summary of what the command does. This is used when searching for a command (which we will see soon).

```
SYNOPSIS
       ls [OPTION]... [FILE]...
```

The synopsis gives a very short outline of how to issue the command. Square brackets indicate that something is optional and the ellipsis ("…") indicates that there can be more than one of them. So this says that the "`ls`" command is followed by zero or more options and then zero or more file names. (Directories and files are lumped together for this purpose.)

```
DESCRIPTION
     List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort.

     Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .
```

The description section is where the meat of the manual page lies. This describes what the command does and then lists all the options and explains what they do.

```
AUTHOR
       Written by Richard Stallman and David MacKenzie.

COPYRIGHT
       Copyright © 2008 Free Software Foundation, Inc.  License GPLv3+: GNU GPL
version 3 or later <http://gnu.org/licenses/gpl.html>
       This is free software: you are free to change and redistribute it.  There
is NO WARRANTY, to the extent permitted by law.
```

The author section identifies the person or persons who write the application. The copyright section is typically where the copyright holders state the licence under which the application is released.

```
REPORTING BUGS
       Report bugs to <bug-coreutils@gnu.org>.
```

Many manual pages quote an address where bugs should be reported.

```
SEE ALSO
       The full documentation for ls is maintained as a Texinfo manual.  If the
info and ls programs are properly installed at your site, the command

              info coreutils 'ls invocation'

      should give you access to the complete manual.
```

The final section can be used either to direct the reader to further information or to related commands. This manual page refers the reader to another source of information provided by the "`info`" command. We don't describe this command in this course as its user interface can be rather confusing.

Exercise 6: Long options (5 minutes)

1. Use "`man ls`" to find the short and long options to reverse the order that `ls` lists its files in. Try them.

Warning!

To a new user the man pages do take some getting used to.

It is quite common to just use Google how to use a command. You'll find there are many forums with helpful examples. However one thing to note is that an online example maybe using a different version of a command and some online examples might not work for you.

Most commands have a long option to check the version –**version** and show help for the command **--help** normally this can help you determine if the online example will work for you. Also be careful just cutting and pasting commands from the Internet! Make sure you know what they will do first...

Section 4: Remote access to other Linux systems

Today's course will be taught using the training workstations and these have Ubuntu Linux installed on them. Unless you have made the decision to run Linux on the desktop I would expect that your need for learning Linux commands is because you need to work on remote systems i.e. HPC clusters or departmental servers.

Many Linux systems have an SSH server installed on them to allow a remote log in, provided you have an account on the remote system.

The command for this is "`ssh`" ("secure shell") and as the name "shell" suggests it gives a command line on the remote system.

The first part of `ssh`'s security is to check that you are connecting to the system you think you are. In the `ssh` world every machine has a "fingerprint". The idea is that you can check from a workstation which has never made contact with it before.

Remote login between cooperating systems

Let say we connect from our local machine "workstation" to the server "unix-training.hpc.private.cam.ac.uk

.hpc.private.cam.ac.uk" and our remote username is "y250"

We would use the command "`ssh y250@unix-training.hpc.private.cam.ac.uk`

`.hpc.private.cam.ac.uk`".

If we haven't connected before we will see a "not in list of known hosts" warning.

```
Workstaton:~$
RSA host key for IP address '172.24.47.21' not in list of known hosts.
y250@unix-training.hpc.private.cam.ac.uk's password:
```

We enter the password `for our remote user account`. The password will not be repeated on the screen.

```
workstation:~$ ssh y250@unix-training.hpc.private.cam.ac.uk
y250@unix-training.hpc.private.cam.ac.uk's password:
============================
Welcome to the Introduction for Unix shell scripting test environment.
Information on the course may be found at
https://www.training.cam.ac.uk/event/2496994.
If you require assistance please feel free to ask the trainers.
============================
Last login: Sun May 20 10:28:45 2018 from 131.111.56.111
```

Notice how both user name and machine name in the prompt has changed.

Remote login to a new system

If you connect to a system which doesn't have fingerprint arrangements sorted out in advance you will get a challenge like this:

```
workstation:~$ ssh unix-training.hpc.private.cam.ac.uk
The authenticity of host '172.24.47.21 (172.24.47.21)' can't be established.
ECDSA key fingerprint is SHA256:65jYNEzGK1SL0nJfX5xD3Cp8DjCLgnfB+Jeh/LSVBKc.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '172.24.47.21' (ECDSA) to the list of known hosts.
y250@unix-training.hpc.private.cam.ac.uk's password:
```

It is common for UIS to publish fingerprints online so that you can compare them to the fingerprint seen in the terminal. It is best practice to check a finger print before you accept, if you don't trust the remote host answer '**no**'.

Once we have accepted a fingerprint on a system we will not be challenged again. If the remote system changes its fingerprint (which it should only do if it has hacked or had its operating system re-installed) then our login attempts will be rejected as insecure. The `ssh` command believes that the connection has been hijacked by another computer claiming to be the one you wanted and had been to before.

The `ssh` command assumes we have an account on the remote system with the same name as the account on the workstation. If this is not the case we can give an account name by preceding the machine name with "`username@hostname`".

Exercise 7: SSH for remote login (5 minutes)

Use `ssh` to login to the remote server linux-exercises.hpc.private.cam.ac.uk

1. On your local workstation, open a terminal
2. **ssh y250@unix-training.hpc.private.cam.ac.uk**

   Hint: Replace y250 with the username you have been given for the remote server

3. Say **yes** to accept the remote key
4. Enter your password
5. type **ls**
6. Type **exit** to disconnect

File transfer

In addition to logging on to remote systems we may also want to transfer files to or from them. In addition to `ssh` ("**s**ecure **sh**ell") there is a related program "`scp`" ("**s**ecure **c**o**p**y"). This behaves in exactly the same way as `cp` except that one of the target or destination is actually a reference to a remote system.

There is also "rsync" a program that can remotely synchronise files between computers (in either direction) and "`sftp`" which is an interactive file transfer program that lets you navigate a the far end.

Fetching files and directories

If there was a file on the machine `unix-training.hpc.private.cam.ac.uk` called "`/home/y250/LinuxIntro.tgz`". To fetch it into the current working directory we could run the following command:

```
workstation:~$ scp y250@unix-training.hpc.private.cam.ac.uk:/home/y250/LinuxIntro.tgz LinuxIntro.tgz
LinuxIntro.tgz                                100% 4604KB   4.5MB/s   00:00
workstation:~$ ls -lah LinuxIntro.tgz
-rw-rw-r-- 1 y250 y250 4.5M May 19 23:22 LinuxIntro.tgz
workstation:~$
```

Not that we define a file on a remote computer: `machine_name:file_path` with a colon separating the two components.

Just as with `ssh`, `scp` assumes you have the same login id on the remote system as on the local one. If you have a different name on the remote system (in my example I do) then you specify "`user@`" before the `machine_name:file_path` element.

If we are happy for the file name to remain the same there is a trick to save on the typing. We can say "copy it into this directory" in which case the copy will leave it with the same file name:

```
workstation:~$ scp y250@unix-training.hpc.private.cam.ac.uk:/tmp/LinuxIntro.tgz .
```

Recall that "`.`" means "the current directory".
We can rename the file as we copy it simply by giving a different name as the second argument:

```
workstation:~$ scp y250@unix-training.hpc.private.cam.ac.uk:/tmp/LinuxIntro.tgz NewLinuxIntro.tgz
```

To fetch a directory and everything in it, we must specify a recursive copy. Unfortunately, the `scp` program hasn't moved to the modern upper case "`-R`" option for recursion so we have to use the lower case "`-r`":

```
localworkstation:~$ scp y250@unix-training.hpc.private.cam.ac.uk:/tmp/fetchable .
```

Sending files and directories

To send data rather than to fetch it we simply use the same syntax for specifying a remote file but on the second argument rather than the first.

To copy a file from the current working directory to your home older on the remote location we specify it like this:

```
localworkstation:~$ scp newdata.txt y250@unix-training.hpc.private.cam.ac.uk:/home/y250/my_data.txt
```

Interactive file transfer

So far we have been able to send or fetch files but in both cases we need to know the remote location and we have had no opportunity to send some files and fetch others. The second file transfer program, "`sftp`", will allow us to do that but stops us transferring directories recursively (for no readily apparent reason).

The `sftp` program is interactive, so rather than issue a single command, as with `scp`, you launch the `sftp`

program to connect to a remote system and then issue a series of instructions within the `sftp` program telling it to change directories at either end, to fetch or send files, to list directories at either end etc.

We launch `sftp` by simply identifying the remote computer. Notice that the prompt changes to indicate that we are now inside the `sftp` program rather than the shell. To draw out a particular point we will move into the `Linux Intro` directory before launching the program.

```
workstation:~$ cd Linux\ Intro

workstation:Linux Intro$ sftp y250@unix-training.hpc.private.cam.ac.uk

The authenticity of host 'unix-training.hpc.private.cam.ac.uk (172.24.47.21)'
can't be established.

ECDSA key fingerprint is SHA256:65jYNEzGK1SL0nJfX5xD3Cp8DjCLgnfB+Jeh/LSVBKc.

Are you sure you want to continue connecting (yes/no)? yes

sftp>
```

The examples assume that our local user is different to our remote user which is why we are putting the remote username in the command.

```
workstation:Linux Intro$ sftp y250@unix-training.hpc.private.cam.ac.uk
```

Notice how we do not specify a location. This is the first difference from `scp`. We always start in our home directory at the far end.

```
sftp> pwd
Remote working directory: /home/y250

sftp>
```

The Linux command `pwd` in the `sftp` program gives information about the *remote* end. To get the local working directory, use the `sftp`-only command "`lpwd`" ("**l**ocal **pwd**"):

```
sftp> lpwd
Local working directory: /home/y250/Linux Intro

sftp>
```

This pattern is repeated for several Linux commands. The original Linux command inside sftp works on the remote system and the same command prefixed with an "l" (for "local") works on the local system. For example to change directory at the remote end we use "cd" and to change directory locally we use "lcd":

```
sftp> cd /tmp

sftp> lcd /home/y250

sftp> pwd
Remote working directory: /tmp

sftp> lpwd
Local working directory: /home/y250

sftp>
```

The `ls` and `lls` commands list files at either end and support the `-l` and `-a` options.

To actually transfer files we use two commands within `sftp`: "`get`" and "`put`".

```
sftp> lls
Desktop  My Music My Pictures  My Video  newdata.txt  Linux Intro

sftp> put newdata.txt y250_example.txt
Uploading newdata.txt to /tmp/y250_example.txt
newdata.txt                                   100%   86     0.0KB/s   00:00

sftp> get LinuxIntro.tgz another.txt
Fetching /tmp/LinuxIntro.tgz to another.txt
/tmp/LinuxIntro.tgz                           100%   86     0.1KB/s   00:00

sftp>
```

To exit the `sftp` program, either enter [Ctrl]+[D] or the command "`quit`". Notice how you return to the shell as you left it. The internal `lcd` command only affected the session within `sftp`.

```
sftp> quit

workstation:Linux Intro$
```

A full set of `sftp` commands is given by the `sftp` command "`help`". A set of the most useful ones is given in the appendices to these notes.

```
sftp> help
Available commands:
cd path                     Change remote directory to 'path'
lcd path                    Change local directory to 'path'
…
?                           Synonym for help

sftp>
```

Exercise 8: SSH for remote login (10 minutes)

Make sure you've completed exercise 7 first!

Use sftp transfer a file to the remote server unix-training.hpc.private.cam.ac.uk

1. On your local workstation, open a terminal
2. Start an sftp session:
   sftp y250@unix-training.hpc.private.cam.ac.uk

   Hint: Replace y250 with the username you have been given for the remote server
3. Now transfer a file:
   put LinuxIntro.tgz
4. Check the file has transferred by listing the remote directory:
   ls -lah
   -rw-rw-r-- 1 y250 y250 4.5M May 19 23:22 LinuxIntro.tgz
5. Note we have transferred a zip file so we will need a command to unzip it:
   tar -xvzf LinuxIntro.tgz
6. Check the file has unzipped by listing the remote directory
   ls
   Linux Intro LinuxIntro.tgz
7. You can delete the zip file:
   rm LinuxIntro.tgz
8. Type **exit** to disconnect

   Hint: Check the file **LinuxIntro.tgz** is in your local directory before you try to transfer it

Section 5: Launching graphical applications from the command line

Warning!

This section requires that you be running in a graphical terminal window rather than in a text console. If you are already running a web browser, please kill it off before starting this section.

Launching graphical applications

Now that we have a command line we can use it to launch more useful commands than just `ls`. On remote servers a good example is Rstudio, Matlab or another statistical application. For example we can launch the Firefox web browser.

Background commands

For these examples, please quit any currently running browsers you may have.

We give the command "`firefox`", with a lower case "`f`".

```
workstation:~$ firefox□
```

(For reasons that will become clear soon we are explicitly showing when the Return key, [□], is being pressed.)

The Firefox web browser launches but in the terminal window the prompt has not been returned. We type "`ls`" (and press [□]) but nothing happens (yet).

```
workstation:~$ firefox□

ls□
```

Now we quit the browser from the browser's menus (File→Quit) and we see that the prompt comes back, the `ls` command is repeated at it, and then run:

```
workstation:~$ firefox□

ls□

workstation:~$ ls
Desktop  Library  My Music  My Pictures  My Video  Linux Intro

workstation:~$
```

This means that each application launched is going to tie up a terminal window and any future commands typed in that terminal window will have to wait for the current command to finish before they are run. We can do better than that by running commands "in the background". To do this we follow the command with an ampersand, "`&`":

```
workstation:~$ firefox &
[1] 7941

workstation:~$
```

This time the prompt did come back immediately; the shell did not wait for the command to complete before asking for further instructions. We could now run `ls` again if we wanted for immediate results.

The "`[1]`" means that this is the first job we have in the background for this session. The number, `7941` in the example above, is a numerical identifier (the "process id") for this backgrounded command. We don't need to know about it, but yours will almost certainly be a different number.

Warning!

During the course of this chapter you may see some warning messages appear, for example:

```
workstation:~$ firefox &
[1] 7941

workstation:~$ *** nss-shared-helper: Shared database disabled (set
NSS_USE_SHARED_DB to enable).

NPP_GetValue()

NPP_GetValue()
```

Don't worry about these. Graphical commands are quite noisy like this because their authors know that you don't get to see the messages if you launch the application graphically.

Note that because the command is running in the background these messages arrive “asynchronously”. This means thet they arrive whenever they want and not in response to you doing anything. They can cause you to lose track of your prompt, if you press [□] you can get a new prompt.

Closing

There is one more feature to observe. We started the Firefox application from this shell and this shell gets informed when it finishes. If we close down Firefox from its menus (File→Quit) then we get a notification the next time a prompt is produced by the shell:

```
workstation:~$

[1]+  Done                    firefox

workstation:~$
```

The message “`Done`” indicates that the program terminated normally. You may get other messages if the program crashes.

Exercise 9: Run xeyes (5 minutes)

1. Run the `xeyes` command in the background. It should get a “`[1]`”.
2. Run Firefox too. It should get a “`[2]`”.

Job control

What can you do if you have already launched a graphical application but forgot to add the ampersand?

There are facilities for taking a running job and moving it into the background. The process comes in two stages: first we stop the running program (“stop” as in “pause” rather than “finish”) and then we restart it in the background.

We will use a second instance of `xeyes` as an example. First we start it in the foreground (i.e. without the ampersand):

```
workstation:~$ xeyes
```

To stop the job we press [Ctrl]+[Z]:

```
workstation:~$ xeyes
^Z
[3]+  Stopped                 xeyes

workstation:~$
```

The “`[3]`” means that this is the third command we have either backgrounded or stopped (if the two commands from the exercise are still running). The “`Stopped`” says that it's stopped, obviously. This is followed by the command itself.

At this point we have the prompt back but the `xeyes` program isn't active; it's stopped. Try moving a terminal window over the `xeyes` window. You will see that the `xeyes` program doesn't track the pointer any more or even redraw itself properly; it's *completely* inactive.

Now we restart it in the background. To do this we issue the command "`bg`" (for "**b**ack**g**round").

```
workstation:~$ bg
[3]+ xeyes &
```

Again we get a response indicating what has happened. The "`[3]`" matches the identifier we saw when we stopped it. We also get the command repeated but this time with a trailing ampersand to indicate that it's running in the background and indeed we do have `xeyes` running in the background as if we had started it with an ampersand in the first place. If you move the terminal window over the `xeyes` window you will see it redraw itself correctly.

If we had stopped the command with [Ctrl]+[Z] and changed our mind we can always restart the command in the foreground with the "`fg`" ("**f**ore**g**round") command.

If you want to know what jobs you currently have running in the background, issue the "jobs" command:

```
workstation:~$ jobs
[1]   Running                 xeyes &
[2]-  Running                 firefox &
[3]+  Running                 xeyes &

workstation:~$
```

The number in square brackets is called the "job number" and we can use it to identify a particular process. If we had thee three jobs running in the background and wanted to foreground the second of them (firefox) then we could do that with "`fg %2`" where the number after the percentage sign is the job number of the job being brought to the foreground. If we wanted to background it again we could do that with [Ctrl]+[Z] and `bg` again. It would still be job number 2. This sort of pushing and pulling of jobs into the background tends to be a minority interest. Typically you will start a job in the background with the ampersand or you won't want it in the background at all.

Killing background jobs

Job control is also tied to another useful facility: killing rogue processes. Suppose a command had gone mad and was refusing to quit as you desperately clicked on the quit button. If a process doesn't die when you click its [×] button in the title bar then usually the graphical environment will wait sixty seconds or thereabouts and prompt you for whether or not you want the process killed more emphatically (while warning you that this will lose any unsaved work). Alternatively, we can use the command line.

We can start with the job numbers in square brackets. At the moment, if you have been following the notes, we have two instances of `xeyes` and one instance of `firefox` running. Note that our first instance of xeyes has job number 1, i.e. is labelled "`[1]`" in the `jobs` output. If we run the command "`kill %1`" then the process corresponding to "[1]" is killed, in our case one of our `xeyes`. A message will appear that the job has been killed the next time you get a prompt.

```
workstation:~$ kill %1

workstation:~$
[1]-  Terminated              xeyes

workstation:~$ jobs
[2]-  Running                 firefox &
[3]+  Running                 xeyes &
```

If a command gets really stuck then there is a stronger version of the "`kill`" command. By default kill politely requests that a command should wind up its business and terminate. If this fails, there is an option "`kill -KILL`" which causes the process to be abruptly killed by the operating system. Note that you can only kill processes which "belong" to you.

```
workstation:~$ jobs
[2]-  Running                 firefox &
[3]+  Running                 xeyes &

workstation:~$ kill -KILL %3

workstation:~$
[3]+  Killed                  xeyes

workstation:~$ jobs
[2]+  Running                 firefox &
```

Exercise 10: Kill Firefox (5 minutes)

1. In the Firefox that you are running in the background navigate to another page.
2. Kill the Firefox instance with the "`kill -KILL`" command and its job number.
3. Start Firefox again. You should get a warning window:
   "Your last Firefox session closed unexpectedly. You can restore the tabs and windows from your previous session, or start a new session if you think the problem was related to a page you were viewing."
4. Select the option to restore the previous session. You should appear back at the last page you were looking at.

Why would you want job control?

What's the point of job control? After all, you can always launch another terminal window.

Backgrounding can be used for much more than just graphical applications however. Any job that is going to take time to complete can be backgrounded. These jobs can be run in a purely text environment where you can't just open another terminal. Alternatively, as we will see later, you may be running the program (graphical or otherwise) on a remote system where you only have one connection established. Backgrounding jobs is often a lot less hassle than establishing another connection.

What would the GUI do?

We can ask the windowing system to open a file with "whatever application it would have used if we had double clicked on the icon in the graphical interface". The command to do this is called "`xdg-open`". (The style of window interface we use on Ubuntu Linux is called "Unity".)

```
workstation:Linux Intro$ pwd
/home/y250/Linux Intro

workstation:Linux Intro$ ls
Play  story.txt  Treasure Island  Work

workstation:Linux Intro$ xdg-open story.txt

workstation:Linux Intro$ xdg-open Treasure\ Island/hispaniola.png

workstation:Linux Intro$
```

This launches `gedit` (the **G**NOME **edit**or) for the text file `story.txt` and `eog` ("**e**ye **o**f **g**nome", the default GNOME picture viewer) for the graphical file `hispaniola.png`.

Because the command is only used for graphical applications it automatically "detaches" the applications (`gedit`, `eog`, etc.) from the terminal so it does not need to be backgrounded.

Warning!

The application cannot usefully open non-existent files. It will not launch the application suitable for a file of that name as a quick way to start an editor with an empty file, for example.

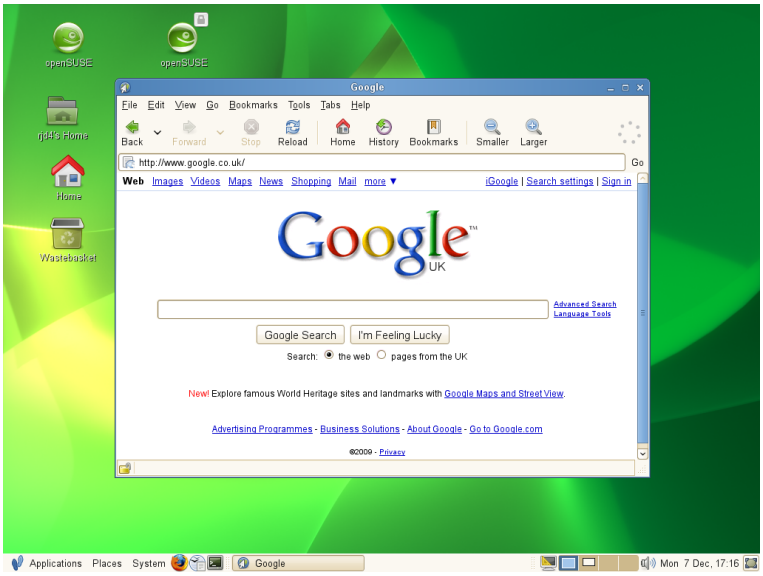
To create an empty file, myfile`.txt` say, use the "`touch`" command: "`touch myfile.txt`".

Exercise 11: Run xdg-open (5 minutes)

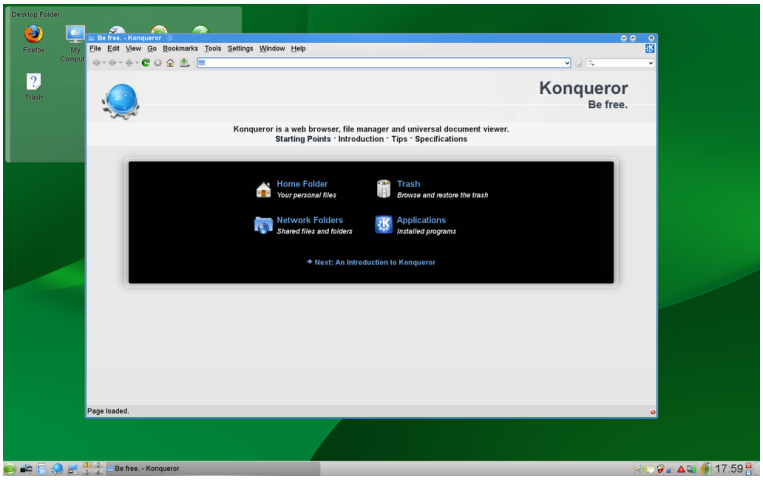
1. Run `xdg-open` on each of the files in `Work/Project Epsilon`. (It will only have this name if you have done the previous exercises. It was originally called `Project Alpha`.)
2. Close down the applications and then try to work out the direct commands to use to run the same applications as `gnome-open` did. (e.g. `gedit story.txt` is the equivalent of `gnome-open story.txt`) Don't forget to background them. (Hint: Help→About in an application typically identifies the application.)

Just for interest

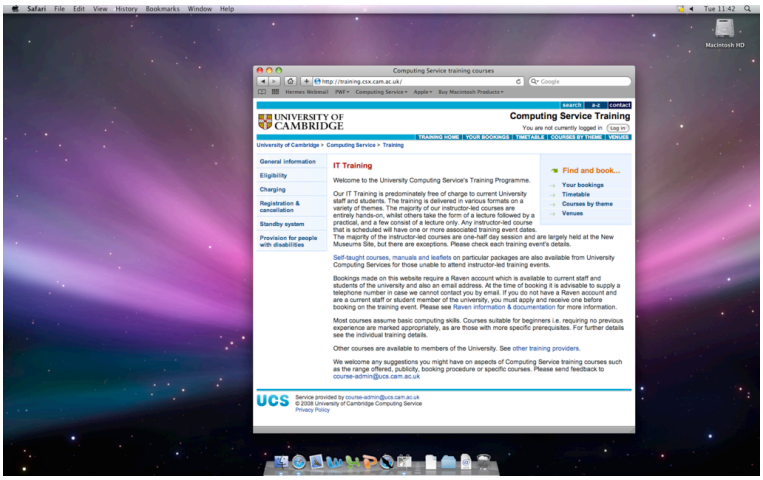
Linux has other windowing environments such as “KDE” and “GNOME”. The command we have used is “xdg-open”, on KDE you can use “`kde-open`” and on GNOME “gnome-oppen”. On MacOS X (a different flavour of Linux) the command is called simply “`open`” as there is no choice of windowing environment.



GNOME



KDE



MAC OS X

Section 6: Command line editing

We have already seen how useful tab completion is. It is not the only assistance that the shell can offer us.

Changing the command line

Suppose we have typed in a command but not hit the [□] key yet:

```
workstation:~$ ls Linux\ Intro
```

(We are showing the cursor as a solid block. Usually it's blinking which is hard to do on paper.) It's at this point we realise that we meant to type "`ls -l`" rather than plain "`ls`". If we press the left arrow key, [←], at this point the cursor moves back. We tap it enough times to move back to just after the "`ls`":

```
workstation:~$ ls Linux\ Intro
```

at this point we simply type "□`-l`" to insert the option. (We must remember to type that leading space to split the command from its options.)

```
workstation:~$ ls -l Linux\ Intro
```

At this point we can hit [□]. There is no need for us to move back to the end of the line.

```
workstation:~$ ls -l Linux\ Intro
total 2
drwxr-xr-x 1 y250 y250 512 2018-04-23 18:20 Play
drwxr-xr-x 1 y250 y250 512 2018-04-28 20:00 Treasure Island
drwxr-xr-x 1 y250 y250 512 2018-04-28 21:34 Work

workstation:~$
```

Alternatively, suppose we had mistyped the "`ls -l`" as "`ls -k`":

```
workstation:~$ ls -k Linux\ Intro
```

We can move the cursor back with the left arrow, [←], to just after the "`-k`":

```
workstation:~$ ls -k Linux\ Intro
```

and press the backspace key, [□], once to delete the "`k`":

```
workstation:~$ ls - Linux\ Intro
```

Then we type the "`l`" that we wanted in the first place:

```
workstation:~$ ls -l Linux\ Intro
```

Then we hit return, [□]:

```
workstation:~$ ls -l Linux\ Intro
total 2
drwxr-xr-x 1 y250 y250 512 2018-04-23 18:20 Play
drwxr-xr-x 1 y250 y250 512 2018-04-28 20:00 Treasure Island
drwxr-xr-x 1 y250 y250 512 2018-04-28 21:34 Work

workstation:~$
```

In addition to moving left you can, of course, move right with [→], but not beyond the end of the line.

Warning!

Be careful to distinguish the left arrow key, [←], from the backspace key, [□], in these notes.

The left arrow key [←] is typically part of the cluster of four arrow keys on the keyboard ([←], [↑], [↓], [→]) between the main keypad and the numeric keypad to the right.

The backspace key, [□], sits on its own at the top right of the main keypad and is shown with a longer arrow.

Other options

There are more options than simply moving forwards and backwards one character at a time. To move to the start of the line press [Home] (or [Ctrl]+[A]), and for the end of the line press [End] (or [Ctrl]+[E]). To move a word at a time use [Ctrl]+[←] and [Ctrl]+[→]. There is a summary of all these movement options at the end of the notes.

History

As well as moving the cursor left and right you can also move it up and down. This gives you access to the shell's history mechanism.

Suppose we type four commands:

```
workstation:Desktop$ cd

workstation:~$ pwd
/home/y250

workstation:~$ cd Linux\ Intro/

workstation:Linux Intro$ ls
Play  Treasure Island  Work
```

These four commands exist in the shell's memory as if they were lines in a file with a fifth, blank line for the command the shell is waiting for that we've not started typing yet:

```
cd

pwd

cd Linux\ Intro/

ls
                                    ← you are here
```

If we press the up arrow, [↑], the command shown on the command line moves back through this history. If we press it three times we end up with the `pwd` command back on our command line.

```
cd

pwd                                 ← you are here

cd Linux\ Intro/

ls
```

We press [□] to run the command:

```
workstation:~$ pwd
/home/y250/Linux Intro/

workstation:~$
```

If we overshoot by typing too many [↑] we can also type [↓] to move back down the list of lines too.

If our command line history has a command which is almost exactly what we want but not quite then we can also scroll back through the commands and then use the other arrow keys and backspace key to edit the historical line to give us the line we want.

If you want to see the "history file" for real, type the command "`history`".

Exercise 12: Using the history part 1 (5 minutes)

1. Change directory to the `Linux Intro` directory.
2. Run `ls -l`.
3. Change directory into the Work subdirectory.
4. Press [Ctrl]+[R].

5. Notice the prompt change to "`(reverse-i-search)`':`"
6. Start to type the "`ls -l`" command as far as the first "`l`" (i.e. just one letter)
7. Notice how the "`ls -l`" command is found and the last "`l`" is indicated by the prompt. ([Ctrl]+[R] triggers a backwards search.)
8. Press the "`s`" key.
9. Notice how the prompt jumps to the "`ls`".
10. Hit [□] to issue the command.

Exercise 13: Using the history part 2 (5 minutes)

1. Issue the command "`cp lorem.txt lorem2.txt`".
2. Press [Ctrl]+[R] again.
3. Press "`l`". Note that the shell finds the last "`l`" in the previous instruction.
4. Press [Ctrl]+[R] again. Note that the shell finds the previous "`l`".
5. Press [Ctrl]+[R] again. Note that this time the prompt jumps back to the last "`l`" in the "`ls -l`" command.
6. Hit [□] to issue the command.

(In practice you would probably just type the "`s`" to jump back to that command.)

Clearing the screen

Not strictly command line editing but related is the facility to clear your screen. To clear your screen you can issue the command "`clear`" which does exactly what you would expect. However, there is a more powerful way to do it: [Ctrl]+[L].

Simply pressing [Ctrl]+[L] at the prompt does exactly the same as the `clear` command but, unlike `clear`, [Ctrl]+[L] can be typed at any point. Suppose we have some command output on the screen already and start to type a command (but don't press [□]):

```
workstation:~$ pwd
/home/y250/Linux Intro/

workstation:~$ ls -
```

We can then press [Ctrl]+[L] at that point:

```
workstation:~$ pwd
/home/y250/Linux Intro/

workstation:~$ ls -[Ctrl]+[L]
```

to clear the screen but leave the partially typed command at the top of the screen ready to be continued:

```
workstation:~$ ls -
```

We can now carry on typing the command on an otherwise clear terminal:

```
workstation:~$ ls -a

.  ..  fubar.txt  Fun  lorem.txt  Play  story.txt  Treasure Island  Work
```

Running applications in the CLI

After our brief excursion into graphical applications, we will return to the pure text world.

Reading plain text files

The classic command for reading a plain text file is called "`more`". We can see this if we move to the `Work` directory and apply it to the `lorem.txt` file:

```
workstation:Work$ pwd
/home/y250/Linux Intro/Work

workstation:Work$ more lorem.txt
TOP OF FILE


Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec at purus sed m
agna aliquet dignissim. In rutrum libero non turpis. Fusce tempor, nulla sit a
met pellentesque feugiat, nibh quam dapibus dui, sit amet ultrices enim odio n
ec ipsum. Etiam luctus purus vehicula erat. Duis tortor lorem, commodo eu, sod
ales a, semper id, diam. Praesent nisl justo, placerat id, rutrum et, vulputat e
ut, metus. Pellentesque habitant morbi tristique senectus et netus et malesu ada
fames ac turpis egestas. Etiam non neque. Curabitur dui. Praesent mi erat,
aliquam eget, aliquet lobortis, pharetra quis, lacus. Nulla facilisis, purus
eget porttitor bibendum, nisi augue auctor lectus, et mollis odio nisi in urna .
Nam felis tortor, porttitor in, ultrices vitae, bibendum non, purus. Cras lu
ctus.

Sed lacus justo, sollicitudin eu, interdum sed, fermentum id, sapien. Class ap
tent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymen
aeos. Nunc purus. In in purus sit amet tellus scelerisque molestie. In in tort
or. Pellentesque viverra, nibh quis feugiat condimentum, metus neque condiment
um lectus, ac commodo turpis justo sit amet nisl. Donec mollis vestibulum feli
s. Aliquam ornare, felis eu suscipit lacinia, neque lectus hendrerit sem, ac v
ulputate est tortor et ligula. Suspendisse quis sapien a urna laoreet elementu
m. Pellentesque nisl ante, tempus ac, porta vel, malesuada vel, sapien. In tor
tor justo, sollicitudin vel, aliquet sed, consequat ac, enim. Proin elit odio,
```

(The author has cheated in the screen above. The `pwd` and `more` commands will both have been scrolled off the top of the screen by the text of the file.)

The line "TOP OF FILE" is in the file. We have added it for your navigational convenience.

The `more` command is named after its prompt which indicates that there is more of the file to follow. We have seen something very similar in the the `man` command pages its output.

If we press the space bar once we get the next screenful:

```
 convallis ac, mollis nec, iaculis et, lectus.
Pellentesque ullamcorper leo eu est. Aliquam metus. Cras sem augue, mattis ege
t, congue vitae, adipiscing in, dolor. Fusce elementum mollis urna. Pellentesq
ue quam. Duis pede tortor, euismod non, varius vitae, consectetuer vel, dui. S
uspendisse potenti. Nullam vehicula, justo euismod imperdiet vulputate, turpis
lacus elementum nulla, ut posuere velit ipsum id elit. Maecenas at justo id r
isus tristique tristique. Vivamus auctor viverra felis. Fusce nonummy commodo
lacus. Morbi et nisi eget nulla iaculis semper. Ut sit amet eros. Quisque in r
isus. Duis id tellus nec magna condimentum facilisis. Fusce feugiat. Curabitur
eleifend tincidunt purus. Etiam ligula mi, mollis vitae, dapibus et, posuere
vel, lacus.


Duis erat. Mauris metus purus, scelerisque ac, pulvinar et, iaculis eu, mauris .
Duis a lectus. Vivamus dolor nisl, aliquet a, venenatis id, consectetuer ut,
lorem. Suspendisse nisi lectus, sollicitudin non, condimentum vel, nonummy ve
l, lacus. Cras nunc justo, tincidunt vel, vulputate non, aliquet euismod, turp
is. Proin sagittis placerat lectus. Donec in lorem. In lacinia, leo ac luctus
tincidunt, nunc pede pulvinar tortor, in molestie tellus sem quis tellus. Maec
enas vel enim. Mauris tincidunt nibh quis mauris ullamcorper pretium. Duis con
sequat commodo risus. Vivamus rutrum. Vivamus dolor augue, imperdiet consectet
uer, dictum at, eleifend et, sem. Suspendisse sed eros. Integer magna purus, e
lementum eget, egestas id, porta id, nunc. Pellentesque habitant morbi tristiq
```

and if we press [B] we go **b**ack a screenful. If we keep pressing the space bar we will eventually reach the end of the file, when `more` will terminate. Alternatively we can just press [Q] to **q**uit immediately.

There is a more modern version of `more` called "`less`" as a pun on the original's name. This is very similar to `more` in that it uses the space bar to page through a document but it does not "fall off the end" when the file reaches the end. Instead you must explicitly press [Q] to quit. The `less` command also reverts the screen to the state it was before the command was run once it is finished.

Searching plain text files

A common requirement is to be able to search through a plain text file for particular words or phrases. For this we will use the text of the story Treasure Island, sitting in `Treasure Island/story.txt` or `Linux Intro/story.txt` depending on whether you have completed the exercises. The search command is called "`grep`":

```
workstation:Linux Intro$ grep Rum story.txt
"Rum," he repeated. "I must get away from here. Rum! Rum!"
but you're on'y a boy, all told. Now, Ben Gunn is fly. Rum wouldn't
fathom and a half of water. We all pulled round again to Rum Cove,

workstation:Linux Intro$
```

Note that the search is case sensitive. Lines containing "rum" with a lower case "r" are not printed. Also note that while "Rum" appears three times in the first line the line is only printed out once.

We can search for more than one word by quoting together the phrase to search for. Recall that quotes lump words together so they are treated as a single item.

```
workstation:Linux Intro$ grep "Ben Gunn" story.txt
"Ben Gunn," he answered, and his voice sounded hoarse and awkward,
like a rusty lock. "I'm poor Ben Gunn, I am; and I haven't spoke with
…
Ben Gunn was on deck alone, and as soon as we came on board he began,
father of a family. As for Ben Gunn, he got a thousand pounds, which
```

However, note that you won't catch lines from the file like this:

```
shame and lies and cruelty, perhaps no man alive could tell. Yet there
were still three upon that island--Silver, and old Morgan, and Ben
Gunn--who had each taken his share in these crimes, as each had hoped in
vain to share in the reward.
```

The two words, “Ben” and “Gunn”, have been split over a line break and have not been detected.

We can search for lower case “rum” also:

```
workstation:Linux Intro$ grep rum story.txt
             Yo-ho-ho, and a bottle of rum!"
called roughly for a glass of rum. This, when it was brought to him,
up my chest. I'll stay here a bit," he continued. "I'm a plain man; rum
…
knuckled under, put up his weapon, and resumed his seat, grumbling like
…
     Yo-ho-ho, and a bottle of rum!"
"this won't do. Stand by to go about. This is a rum start, and I can't
the rum, Darby!"
```

Note that it matched a line containing the word “g**rum**bling”. We can tell `grep` to search for whole words only with the “`-w`” (“**w**ord”) option:

```
workstation:Linux Intro$ grep -w rum story.txt
             Yo-ho-ho, and a bottle of rum!"
called roughly for a glass of rum. This, when it was brought to him,
…
"this won't do. Stand by to go about. This is a rum start, and I
the rum, Darby!"
workstation:Treasure Island$
```

We can search for “Rum” and “rum” (and “rUm” etc.) by requesting a case insensitive search with the option “`-i`” (“**i**nsensitive”):

```
workstation:Linux Intro$ grep -i rum story.txt
             Yo-ho-ho, and a bottle of rum!"
called roughly for a glass of rum. This, when it was brought to him,
…
knuckled under, put up his weapon, and resumed his seat, grumbling
…
"Rum," he repeated. "I must get away from here. Rum! Rum!"
…
"this won't do. Stand by to go about. This is a rum start, and I
the rum, Darby!"
workstation:Linux Intro$
```

If we want just the word “rum” but in either case we combine the `-i` and `-w` options:

```
workstation:Linux Intro$ grep -iw rum story.txt
             Yo-ho-ho, and a bottle of rum!"
called roughly for a glass of rum. This, when it was brought to him,
…
"Rum," he repeated. "I must get away from here. Rum! Rum!"
…
"this won't do. Stand by to go about. This is a rum start, and I
the rum, Darby!"
workstation:Linux Intro$
```

Again, just as with the options to `ls`, the options “`grep -iw`”, “`grep -wi`”, “`grep -i -w`”, and “`grep -w -i`” are all equivalent.

Exercise 14: Using grep

1. How many times does “yo-ho-ho” (regardless of case) appear in the text of Treasure Island?
2. Use `grep` for the search and count the instances manually.

Counting text

Another operation which can be useful on plain text is counting the words, lines or even characters. Linux has a command called "`wc`" ("**w**ord **c**ount") that does this:

```
workstation:Linux Intro$ wc story.txt
  7857  71516 390927 story.txt

workstation:Linux Intro$
```

This tells us that the file contains 7,857 lines, 71,516 words, and 390,927 characters. We can demand just some of the information with the three `wc` options: `-l` for the line count, `-w` for the word count, and `-c` for the character count:

```
workstation:Linux Intro$ wc -l story.txt
7857 story.txt

workstation:Linux Intro$
```

Exercise 15: Counting words (5 minutes)

1. How many words are there in the `lorem.txt` file?.

Editing plain text files

If you want to edit a plain text file then the graphical editors such as `gedit` (the default graphical text editor, as given by `gnome-open`) are probably easiest for you. This is what we will use in this course.

There are two plain text editors which can be used in text consoles. These are "`emacs`" (which also has a graphical form) and "`vi`". These are the grand old men of the Linux world and require training before they can be used. The UCS offers courses on both.

Repeating the command line

The next command seems rather pointless. Its true utility will only become apparent later. The "`echo`" command repeats whatever you give it as arguments.

```
workstation:~$ echo one two three
one two three

workstation:~$
```

We can use it to see how quotes and escaping don't make it through to the command itself. They act purely as instructions to the shell.

```
workstation:~$ echo "one two three"
one two three

workstation:~$ echo 'one two three'
one two three

workstation:~$ echo one\ two\ three
one two three

workstation:~$
```

Telling the time

The "`date`" command seems fairly straightforward at first glance; it gives the date and the time:

```
workstation:~$ date
Tue Apr 28 20:37:12 BST 2018

workstation:~$
```

However, we can modify the format of the output to give just some of that information. The `date` command accepts an argument called a "format string" which controls the look of its output:

```
workstation:~$ date +"%d %m %Y"
28 04 2018

workstation:~$
```

The string in quotes after the plus sign is the format string. The letters after percentage characters are converted to elements of the date and time. `%d` is converted to the day of the month, `%m` to the numerical month of the year and `%Y` to the year. Characters without preceding percentage characters are simply repeated, like the spaces in the example above. The reference sheet at the back of these notes contains a set of useful formatting options.

Warning!

The format string on the date command is one of the places where you are likely to need the quotes. If you have any spaces in your format string then you must quote it. Otherwise the date command will take everything from the "+" to the first space as the format string and either ignore or get confused by everything after that space.

```
workstation:~$ date +"%Y %B %d"
2010 March 01

workstation:~$ date +%Y %B %d
date: extra operand `%B'
Try `date --help' for more information.

workstation:~$
```

Exercise 16: Changing the date format

1. Work out the format string for the `date` command so that the time looks like this:
   `2018-04-28 23:57:37`

   (Do use the crib sheet at the back of these notes.)
2. Change the format string for the `date` command in the previous exercise so that the time looks like this:
   `Date: 2018-04-28`
   `Time: 23:57:37`
3. (This should be the output of a *single* date command. Recall that a `%n` in a format string is converted into a line break.)

Using the date in a shell scripting

It is quite common to write shell scripts that make use of the date command. Often we manipulate the date to help control the script or to create some kind of output such as a log file. The example below is a backup script, it logs the start and end date as well as using the short date for making folder names.

```
#!/bin/bash
#Testing screen recording
########################
# check for lock
if test -f ~/RDS_RSYNC_LOCK
then
    echo ALERT: Rsync is already running! 1>&2
    exit 1
fi

# create lock
touch ~/RDS_RSYNC_LOCK

# output log header

START=$(date +'%F-%T')
RSYNC_OPTIONS="-avz --exclude sshfs-test --progress --delete --stats"
REMOTE="/rds/project/ps459/test/mr-robot-backup"
LOCAL="/home/sumption/"
USER="ps459"
RDS="rds.uis.cam.ac.uk"

#exec &> ~/log/$(date +'%a').backup.$START

touch ~/sumption-backup.log
exec &> ~/sumption-backup.log

        echo "****** STARTING rsync RUN ******"
        echo "Starting server rsync at: $START"
        #All this scripts is doing is the rsync job below but re-directs the jobs output to a logfile
        echo "******"
        echo "$SOURCE $REMOTE"
        rsync -e 'ssh -i ~/.ssh/mr_robot_id_rsa' $RSYNC_OPTIONS $LOCAL $USER@$RDS:$REMOTE
        # output log footer
        echo "******"
        FINISH=$(date +"%F-%T")
        echo "****** FINISHING RSYNC RUN at $FINISH******"

#done

# remove lock
rm ~/RDS_RSYNC_LOCK
exit 1
```

Section 7: Redirecting data and piping commands

The "`cat`" command is a program for con**cat**enating one or more plain text files. You can concatenate a single file but it is rarely useful.

For example in the `Work` directory we have this:

```
workstation:Work$ cat abc.txt def.txt ghi.txt
ABC
ABC
ABC
ABC
…
GHI
GHI
GHI
GHI
GHI

workstation:Work$
```

The output containing the three files combined went to the terminal. What if we wanted to combine them into a new file? We could power up an editor and combine them there but there is a slicker way.

Standard output

All the Linux commands that deal with linear data (such as plain text) typically spit their output to the terminal by default where that linear stream of data typically rushes past you. This output stream is known technically as "standard output" and by default a program's standard output goes to the terminal where it is being run. But it can be redirected.

```
workstation:Work$ cat abc.txt def.txt ghi.txt > combined.txt

workstation:Work$
```

The "`>`" redirection (think of it as an arrow) has taken the standard output of the `cat` command and redirected it from the terminal into a file `combined.txt`. Note that the redirection overwrites any content of the file that was previously there.

```
workstation:Work$ more combined.txt
ABC
…
GHI
GHI

workstation:Work$ cat abc.txt def.txt > combined.txt

workstation:Work$ more combined.txt

ABC
…
DEF
DEF

workstation:Work$
```

There is a variant of "`>`" which *appends* to the end of any pre-existing file (and which still creates the file if it does not): "`>>`".

```
workstation:Work$ cat ghi.txt >> combined.txt

workstation:Work$ more combined.txt
ABC
…
GHI

workstation:Work$
```

Exercise 17: Combining and counting

1. Create a file `lorem2.txt` which is two copies of `lorem.txt`, one after the other.
2. Run `wc` on `lorem.txt` and `lorem2.txt` to check that all three counts have doubled.
3. If you have a `lorem2.txt` file left over from a previous exercise do not worry about over-writing it.

`Do not use an editor! the object is to use the cat command to create and manipulate files.`

Standard input

In addition to standard output, Linux commands also have the concept of “standard input”. This is where commands can get their data from. If a command takes a file name as an argument then it is going to get its data from that file, but those commands can often also take their input from the keyboard directly. In this case we use [Ctrl]+[D] to mark “end of input”.

Warning!

Be careful with [Ctrl]+[D] if you just type it at the shell when there's no command pulling in input the shell interprets it as “no more input” and exits

Using [Ctrl]+[D] to mark “end of input”

We have met `wc` already:

```
workstation:Work$ wc abc.txt
 9  9 36 abc.txt

workstation:Work$
```

We can also tell it to get its input from its standard input (the keyboard) by omitting any file name:

```
workstation:Work$ wc
The quick brown fox jumps over the lazy dog.
The cow jumped over the moon.
[Ctrl]+[D]
      2      15      75

workstation:Work$
```

Note that no file name is quoted in `wc`’s output line. Standard input is anonymous.

Because [Ctrl]+[D] is potentially so dangerous there is a special shell syntax to say “end the input with *this*”:

```
workstation:Work$ wc <<END
The quick brown fox jumps over the lazy dog.
The cow jumped over the moon.
END
      2      15      75

workstation:Work$
```

Note that the ending string, “`END`”, does not contribute to the word count.

Also note that there is nothing special about “`END`”. It is just the string of characters that follows “`<<`”:

```
workstation:Work$ wc <<STOP
The quick brown fox jumps over the lazy dog.
The cow jumped over the moon.
STOP
     2     15     75

workstation:Work$
```

There is no need for the end marker to be a word, either. Punctuation works too and "`!`" is a traditional marker:

```
workstation:Work$ wc <<!
The quick brown fox jumps over the lazy dog.
The cow jumped over the moon.
!
     2     15     75

workstation:Work$
```

Finally we can redirect the standard input from a pre-existing file with the "`<`" redirector. Again, think of it as an arrow but this time pointing *into* the command rather than out of it.

```
workstation:Work$ wc < combined.txt
   27     27    108

workstation:Work$
```

Again, notice that it produces anonymous output for standard input and contrast it with the similar command line:

```
workstation:Work$ wc combined.txt
   27     27    108     combined.txt

workstation:Work$
```

Piping

The full power of standard input and output only becomes clear when we combine the two. We can take the standard output of one command and feed it directly into the standard input of a second. This simple construction allows us to combine Linux commands in very powerful ways.

It is done by placing the "`|`" character (pronounced "pipe") between the two commands:

```
workstation:Work$ cat lorem.txt combined.txt | wc
     40     568    3801

workstation:Work$
```

If we move back to the `Treasure Island` directory we can see a classic use of piping and the `more` command:

```
workstation:Work$ cd ../Treasure\ Island/

workstation:Treasure Island$ grep -iw rum story.txt | more
             Yo-ho-ho, and a bottle of rum!"
called roughly for a glass of rum. This, when it was brought to him,
…
"Rum," he repeated. "I must get away from here. Rum! Rum!"
the stranger. I got the rum, to be sure, and tried to put it down his
```

We no longer have to have output rush past us on the screen.

Exercise 18: Combine grep and wc

1. How many lines in Treasure Island contain the word "rum" in any case?
2. (How should you combine `grep` and `wc`?)

Warning!

Note that “the number of lines containing the word ‘rum’” is *not* the same as “the number of times the word ‘rum’ occurs”. In story.txt `55 lines contain the word rum and the word rumoccurs 67 times.`

```
workstation:$ grep -iw rum story.txt | wc -l
55

workstation:$ grep -c 'rum' story.txt

67
```

More

A very common use of piping is to the more command. If a command's output is too long to fit on a single screen then piping it through more paginates the output.

Exercise 19: More (1 minute)

1. Run these two commands:
2. `ls --help`
3. `ls --help | more`
4. Page through the output of the second command. You have the choice of this and “`man ls`”.
5. This addresses the uncontrollable output issue we met with “`ls --help`” earlier.

Section 8: File name wild cards

The next command line trick we will see is called “wild carding” or “globbing”. This allows us to express a set of files without having to list them all manually.

Asterisk

Observe this use of the `echo` command in the `Work` directory:

```
workstation:Work$ echo abc.txt combined.txt def.txt ghi.txt lorem2.txt lorem.txt
nonsense.txt
abc.txt combined.txt def.txt ghi.txt lorem2.txt lorem.txt nonsense.txt

workstation:Work$
```

The trick behind wild cards is that the shell will take certain special characters and convert them into lists of file names. So, for example, the expression “`*.txt`” is converted into the list of files that end in “`.txt`”. The “`*`” stands for “anything”.

```
workstation:Work$ echo *.txt
abc.txt combined.txt def.txt ghi.txt lorem2.txt lorem.txt nonsense.txt

workstation:Work$
```

Wild cards do not work inside quotes or if the asterisk is preceded by a backslash. (Remember that this makes characters special to the shell, like space, ordinary parts of file names. Well, it works on asterisks too.)

```
workstation:Work$ echo "*.txt"
*.txt

workstation:Work$ echo '*.txt'
*.txt

workstation:Work$ echo \*.txt
*.txt

workstation:Work$
```

Also, if a wild card doesn't match anything it passes through unaltered.

```
workstation:Work$ echo *.foo
*.foo

workstation:Work$
```

It is not `echo` that is doing this; it is the *shell*. All wild cards work equally well with any command:

```
workstation:Work$ wc *.txt
    9     9    36 abc.txt
   27    27   108 combined.txt
    9     9    36 def.txt
    9     9    36 ghi.txt
   26  1082  7386 lorem2.txt
   13   541  3693 lorem.txt
    9   535  3664 nonsense.txt
  102  2212 14959 total

workstation:Work$
```

The asterisk can expand into nothing:

```
workstation:Work$ echo abc.txt*
abc.txt

workstation:Work$
```

Question mark

There are other wild cards. The asterisk, “`*`”, expands into “anything”. The question mark, “`?`”, expands into

“any *one* character”:

```
workstation:Work$ echo abc.t*
abc.txt

workstation:Work$ echo abc.t?
abc.t?

workstation:Work$ echo abc.t??
abc.txt

workstation:Work$ echo abc.t???
abc.t???

workstation:Work$
```

Square brackets — only for the keen

The third wild card is the most difficult. The question mark, “`?`”, expands into “any one character”. The last glob allows us to say “any one character from this set” or even “any one character not from this set”. Obviously this glob is going to be most complex as we have to be able to specify the set.

The glob “`[aeiou]`” means “any one of '`a`', '`e`', '`i`', '`o`', or '`u`'”:

```
workstation:Work$ echo abc.t[xyz]t
abc.txt

workstation:Work$ echo abc.t[abc]t
abc.t[abc]t

workstation:Work$
```

We can negate this membership with the syntax “`[^aeiou]`” to mean “any one character that is *not* '`a`', '`e`', '`i`', '`o`', or '`u`'”:

```
workstation:Work$ echo abc.t[^xyz]t
abc.t[^xyz]t

workstation:Work$ echo abc.t[^abc]t
abc.txt

workstation:Work$
```

Exercise 20: Wild cards (10 minutes)

1. Try any of the commands in this section on text files in your work directory.

Section 9: Environment variables

Certain elements of the Linux including the command line interpreter itself can be influenced by a collection of settings known as “the environment”. Each of these settings is individually referred to as an “environment variable”.

We can see the entire environment with the command “env”. This produces one line of output for each setting, shown in the form

```
VARIABLE=value
```

with the name of an environment variable traditionally given in upper case. There are many environment variables so the output is best piped through more or less:

```
workstation:~$ env | more
XDG_VTNR=7
XDG_SESSION_ID=c2
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/y250
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
GPG_AGENT_INFO=/home/y250/.gnupg/S.gpg-agent:0:1
TERM=xterm-256color
VTE_VERSION=4205
XDG_MENU_PREFIX=gnome-
SHELL=/bin/bash
DERBY_HOME=/usr/lib/jvm/java-8-oracle/db
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
……..and so on…….
```

To illustrate environment variables we will need to use our graphical interface again. Environment variables work everywhere but this demonstration needs a resizable window.

We can search for the string “TERM” (all upper case) in the output of env by piping to the grep command. We see that the TERM environment variable is set to xterm and that the COLORTERM variable is set to gnome-terminal:

```
workstation:~$ env | grep TERM
TERM=xterm-256color

workstation:~$
```

The terminal window is a Gnome Terminal window set to xterm-256color. This is a modern version of an old sort of terminal window called an xterm. Most programs use the value of the TERM environment variable to work out what sort of terminal it is and, therefore, how to talk to it.

Running env and grepping the results is slow and inefficient. There is a simple mechanism to determine the value of any variable (and there is another sort other than environment variables). Recall that the echo command repeats its arguments only after the shell has rewritten them (as happened with file name globbing). The shell also has a syntax for converting the names of variables into their corresponding values.

```
workstation:~$ echo "${TERM}"
xterm

workstation:~$
```

Just this once we will show you a short cut. In this specific case the double quotes and the curly brackets were unnecessary. In this specific case you could have got away with just this:

```
workstation:~$ echo $TERM
xterm-256color

workstation:~$
```

But rather than explain when you do and don't need the double quotes and when you do and don't need the braces (curly brackets) we will get into the good habit of always using them.

You *do* always need the dollar character, though. It is the signal that triggers the conversion from variable name to variable value.

```
workstation:~$ echo TERM
TERM

workstation:~$
```

Let's see the `TERM` environment variable in action. We are going to need a long piece of text and we will use the `story.txt` file in the `Treasure Island` directory. (If your copy is still in the parent directory, `Linux Intro`, then move it back to `Treasure Island`.) We will maximise our terminal window to fill the screen and read the first screenful of `treasure.txt` with `more`:

```
workstation:Treasure Island$ more story.txt

TREASURE ISLAND

by Robert Louis Stevenson

…

     5.  THE LAST OF THE BLIND MAN  . . . . . .  36
     6.  THE CAPTAIN'S PAPERS . . . . . .  41
```

We notice that the number of lines shown matches the increased number of rows in the expanded window. To do this, `more` needed to be able to communicate with the terminal to learn how many rows it had. There are standard ways to do this but they hinge on the `TERM` environment variable identifying the type of terminal correctly.

We are going to unset the `TERM` environment variable with the appropriately named "`unset`" command.

We quit from `more` and run the command "`unset TERM`" and check with `echo` to see that it has no value any more. (Unset variables show up as blank values by default.)

```
workstation:Treasure Island$ unset TERM

workstation:Treasure Island$ echo "${TERM}"


workstation:Treasure Island$
```

Now we repeat the `more` command.

This time we see that only twenty-four lines of output are printed (including the "`--more--(0%)`" prompt) and that the prompt is no longer in inverse video.

```
workstation:Treasure Island$ more story.txt

TREASURE ISLAND

by Robert Louis Stevenson

…

               --So be it, and fall on!  If not,
--More--(0%)
```

The `more` command doesn't know how deep the terminal is so it is guessing at twenty-four lines (a common value for fixed size terminals). It also doesn't know how to generate inverse video any more. It doesn't even know if the terminal *can* produce inverse video.

Now we will see how to set the value of an environment variable. The same command sets the value of a previously unset environment variable and resets the value of an existing one.

```
workstation:Treasure Island$ export TERM=xterm-256color

workstation:Treasure Island$ more story.txt

…
     5.  THE LAST OF THE BLIND MAN  . . . . . .  36
     6.  THE CAPTAIN'S PAPERS . . . . . .  41
```

This time we see a proper screenful again and an inverse video prompt from more.

Why "export"? The word implies (correctly) that the variable is not just for the shell itself but should be "exported to" any other commands the shell launches (such as more). The other kind of variable mentioned above is called a "shell variable" and is used by the shell only and not passed into any of the commands launched from the shell. We will not use shell variables here.

But what are the environment variables used for? There are over a hundred set on a typical PWF Linux session! We've only seen TERM so far. We will only focus on three more.

The PATH environment variable

For this demonstration to work we will need a new terminal window. Again, this is more for the demonstration than for any fundamental property of environment variables.

Almost all the commands in Linux correspond to a file containing the instructions for that command, typically in the computer's machine code. You can find out what file a command corresponds to with the "type" command:

```
workstation:~$ type more
more is /bin/more
```

But how did the shell know to look in the directory "/bin" for a file called "more"?

The PATH environment variable has as its value a list of directories separated by colons. This is the list of directories the shell will go looking in.

```
workstation:~$ echo "${PATH}"
/home/y250/bin:/home/y250/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/u
sr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/usr/lib/jvm/java-8-
oracle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-oracle/jre/bin

workstation:~$
```

As you progress in learning Linux you may find you wish to install or compile software. A common step is to add the path of the software to your PATH environment variable. Note that in my path I have miniconda which is installed in my home.

```
sumption@mr-robot:~$ echo "${PATH}"
home/sumption/miniconda3/bin:/home/sumption/miniconda3/bin:/usr/local/sbin:/usr/
local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/u
sr/lib/jvm/java-8-oracle/bin:/usr/lib/jvm/java-8-
oracle/db/bin:/usr/lib/jvm/java-8-oracle/jre/bin
```

Because it takes time to search through each of these directories the shell remembers where it found each command after the first time it uses it successfully. This is a process called "hashing". (The sort of record the shell uses to remember this information is called a "hash table".) If we use more and then ask about it again we get a slightly different answer:

```
workstation:~$ type more
more is hashed (/bin/more)

workstation:~$
```

This is why we wanted a fresh terminal window; we needed an unused more command.

We will start yet another new terminal window and, before we try to run any command (which will get them hashed), we unset the PATH environment variable. Then we try to use more, or type:

```
workstation:~$ unset PATH

workstation:~$ type more
bash: type: more: not found

workstation:~$ more /etc/motd
bash: more: No such file or directory

workstation:~$
```

Warning!

In the absence of PATH to help it search directories, the shell cannot find programs for the commands you

type. Close the terminal window with the now broken session.

It is actually a major advantage of the environment not being passed back up to the parent or across to fellow child processes that if you corrupt your environment the corruption stays localised.

The HOME environment variable

The last significant environment variable we will mention you should rarely, if ever, modify is `HOME`. This identifies your home directory and where the `cd` command takes you if it is given no argument.

The HOME variable is commonly used in shell scripts.

```
workstation:~$ echo "${HOME}"
/home/y250
workstation:~$ cd
workstation:~$ pwd
/home/y250
workstation:~$
```

Section 10: Trivial shell scripts

Finally, one of the advantages of the command line is that a record or "script" can be made of the commands issued and that script can then be kept for later reuse or given to other people for them to use. Scripting is useful when we have long repeatable tasks.

Exercise 21: Run a script (1 minute)

1. Open a terminal
2. Change to the **Linux Intro/scripts** directory
3. List the directory, you should see a script **'example-script.sh'**
4. Run it with:
   bash example-script.sh
5. What does it do?

Writing a script

We use the text editor gedit and create a file with a few trivial shell commands. We will then save it as a file `commands.sh` in the scripts directory. The following is what should go in the file, not what you should run!

Exercise 22: Write a script (10 minutes)

1. Open a terminal
2. Make a directory called myscripts in the top level of your home i.e. */home/y250/myscripts*
   Hint: mkdir ~/myscripts
3. Close the terminal
4. Open the editor gedit
5. Make a new file with the following content:

```
pwd
date
ls -l
cd "Linux Intro"
ls -l
```

6. Save the file as myscript.sh in your myscripts directory
7. Open a terminal window. This will start in our home directory.
8. **cd** into the **myscripts** directory
9. List the contents of the the directory - **ls**
10. Run the script by using the command **bash myscript.sh**
11. What does the script do?

BASH

Recall that the name of our shell is "`bash`". That's also the command to run another one of these command line interpreters so in our example we run the command "`bash example-script.sh`":

```
workstation:~/myscripts$ bash example-script.sh
hello world
This script is in /home/y250/myscripts and running on workstation
Today is: Sun 20 May 16:55:17 BST 2018
```

This isn't quite a "command", the command was "`bash`" and we told it what file of commands to run.

Making a script executable

Ideally we would like to just be able to give the command "example-script`.sh`".

First, we will do away with having to give the command `bash`.

Recall that Linux has three types of permission of a file, labelled with "`rwx`": a file can be **r**eadable, **w**riteable, and e**x**ecutable. At the moment this file is only readable and writeable by us:

```
workstation:~/myscripts$ ls -l example-script.sh
-rwxrwxr-x 1 y250 y250 200 May 20 16:55 example-script.sh
workstation:~/myscripts$
```

We will make it executable by changing its permissions (also known as its "mode") with the "`chmod`" command ("**ch**ange **mod**e"). The syntax of this command is an extreme example of Linux complex conciseness.

```
workstation:~/myscripts$ chmod a+x example-script.sh
workstation:~$ ls -l example-script.sh
-rwxr-xr-x 1 y250 y250 37 2018-04-28 23:32 commands.sh
workstation:~/myscripts$
```

The "`a+x`" means "for **a**ll classes of user (owner, group member, other) add (**+**) the e**x**ecute permission".

The example-script`.sh` file is now ready to be executed, but the shell will never find it because your home directory is not on your `PATH` and that's where the shell looks. We can tell it where to look by giving the exact file name for the command so that it doesn't have to go looking.

```
workstation:$/home/y250/myscripts/example-script.sh
hello world
This script is in /home/y250/myscripts and running on workstation
Today is: Sun 20 May 17:01:46 BST 2018
```

It's awkward having to keep track of the directory the file it in. Recall that the directory "`.`" means "this directory". We typically use that to tell the shell to run a command from a file this directory:

```
./example-script.sh
hello world
This script is in /home/y250/myscripts and running on workstation
Today is: Sun 20 May 17:04:40 BST 2018
```

Exercise 23: Make your script executable (5 minutes)

1. Open a terminal
2. **cd** into **myscripts**
   Hint: cd ~/myscripts
3. Make the script executable: **chmod a+x example-script.sh**

4. Run it: **./example-script.sh**
5. Run it using the full path: **/home/y250/myscripts/example-script.sh**
   Hint: replace y250 with your username
   To get rid of the leading "`/home/y250/myscripts`" or "`./`". We need to put the executable file somewhere on the `PATH`.

Adding the script directory to your PATH

In the previous exercise we see that we can give the full path or execute the script using ./ if we are in the directory containing the script. What would be better was if the script was in our PATH and we could just type its name to execute it.

```
workstation:~$echo "${PATH}"
/home/y250/bin:/home/y250/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/u
sr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/usr/lib/jvm/java-8-
oracle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-oracle/jre/bin
```

What we can do is export the directory containing our script so that its added to our PATH.

```
workstation:~$export PATH=$PATH:/home/y250/myscripts
/home/y250/bin:/home/y250/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/u
sr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/usr/lib/jvm/java-8-
oracle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-
oracle/jre/bin:/home/y250/myscripts
workstation:~$ example-script.sh

hello world

This script is in /home/y250 and running on workstation

Today is: Sun 20 May 17:33:43 BST 2018
```

We can now see our script directory in our PATH and can run the command example-script.sh without using the full path or ./

Make it permanent .bashrc

When you export a PATH variable in a terminal it will be alive for as long as its parent bash session is alive. Once the parent Bash session is killed the PATH variable no longer exists. Whilst there are ways to set system wide PATH variables, we only want to set the PATH variable for our user so we will follow these steps. We will use the **source** command, to load any functions from a file into the current shell I..e as we are changing a file that contains details about our PATH and this file is read when we open the shell.

Exercise 24: Add your script folder to .bashrc (5 minutes)

1. Open a terminal
2. Open the file .bashrc using nano: **nano ~/.bashrc**
3. Go to the bottom of the file and add this as the last line:
   export PATH="/home/y250/myscripts:$PATH"
4. **Ctrl + x** to save the file
5. Source the bashrc file using this command (reads the changes we have made from the file and updates PATH):
   source ~/.bashrc
6. Check its now in your path: **echo $PATH**
7. Run the script: **example-script.sh**

```
workstation:~$echo $PATH
/home/y250/bin:/home/y250/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/u
sr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/usr/lib/jvm/java-8-
oracle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-
oracle/jre/bin:/home/y250/myscripts
workstation:~$ example-script.sh

hello world

This script is in /home/y250 and running on workstation

Today is: Sun 20 May 17:33:43 BST 2018
```

Congratulations! You have just written your first Linux shell script.

The comments and the shebang

It is good practice when writing scripts to add “comments”. Comments have no effect on what the script does and exist purely to aid you or others to understand what the script is about. They can be used, too, to quote authors and contact details etc.

Comments are defined by a “#” (“hash”) character. Everything from the hash to the end of the line is treated as a comment and ignored by the shell:

```
# My first shell script!

pwd                   # Print the current directory
date                  # Print the date and time
ls -l                 # List the files
cd "Linux Intro"       # Change directory
ls -l                 # List the files in the new directory
```

Scripts can be written in various languages and if there's no special instructions then the operating system assumes that it is written in shell. This is what happened with our script however its is best practise to be explicit.

The “special instructions” lie on the first line which start with “#!” followed by the full path to the interpreter whose language the script is written in. The special line is called the shebang.

Our script will use bash. Many scripting tutorials suggest that we should set “#!/bin/bash” as the first line.

```
#!/bin/bash

# My first shell script!

pwd                   # Print the current directory
date                  # Print the date and time
ls -l                 # List the files
cd "Linux Intro"       # Change directory
ls -l                 # List the files in the new directory
```

Some versions of Linux install bash in a different location so a better practice is to use env to set the bash location.

```
#!/usr/bin/env bash

# My first shell script!

pwd                   # Print the current directory
date                  # Print the date and time
ls -l                 # List the files
cd "Linux Intro"       # Change directory
ls -l                 # List the files in the new directory
```

Note that because “`#`” is the comment character the first line counts as a comment as far as the shell is concerned and won't affect the operation of the script itself.

Incidentally, any scripting language that uses “`#`” as its comment character can have scripts defined this way. So Python (file `/usr/bin/python`) has scripts starting with “`#!/usr/bin/python`” or “#!/usr/bin/env python” Perl has “`#!/usr/bin/perl`” or “#!/usr/bin/env perl”

This course as aimed at getting you comfortable with navigating the Linux command line. If you wish to learn more shell scripting I would recommend the UIS course “Simple Shell Scripting for Scientists” takes it much further.

Appendices

Command summary

This is a very quick summary of all the command line commands we cover in this course.

Command	Example	Action
`bash`	`bash commands.sh`	This is the name of the command line interpreter we've been using all along!
`bc`		Command line calculator.
`bg`		Background a command stopped with [Ctrl]+[Z].
`cat`	`cat abc.txt def.txt`	Concatenate one or more files.
`cd`	`cd ../Work`	Change directory.
`chmod`	`chmod a+x commands.sh`	Change the mode (permissions) of a file. Use "`chmod a+x`" on a shell script.
`clear`		Clear screen. (Better to use [Ctrl]+[L].)
`cp`	`cp island.jpg map.jpg`	Copy a file. (Use the "`-R`" option to recursively copy a directory.)
`date`	`date +"%H:%k:%M"`	Give the date and time. Output can be formatted.
`echo`	`echo *.txt`	Repeat the command line arguments passed to it.
`env`		Print out the set of environment variables.
`exit`		Exit the shell if you don't want to use [Ctrl]+[D].
`export`	`export TERM=xterm`	Set the value of an environment variable.
`fg`		Foreground a command stopped with [Ctrl]+[Z].
`grep`	`grep rum story.txt`	Search a file for a text string.
`history`		List the previous command lines.
`jobs`		List all the backgrounded jobs currently running.
`less`	`less lorem.txt`	Page through a file, a screenful at a time. A recent version of `more`.
`ls`	`ls Work`	List the contents of a directory.
`mkdir`	`mkdir Fun`	Make a directory.
`more`	`more lorem.txt`	Page through a file, a screenful at a time.
`mv`	`mv island.jpg ../map.jpg`	Move (rename) a file or directory.
`rm`	`rm nonsense.txt`	Remove a file. (Use the "`-R`" option to recursively remove a directory.)
`rmdir`	`rmdir Fun`	Remove an *empty* directory.
`scp`		Remotely copy a file or directory tree to or from a remote system.
`sftp`		Interactively transfer files to or from a remote system.
`ssh`		Establish a connection to a remote system to run commands on it.
`touch`	`touch newfile.txt`	Create an empty file.
`type`	`type more`	Find where a command comes from.
`unset`	`unset TERM`[5]	Unset a variable.
`w`		Show who is logged on, what they are doing and how busy the system is.
`wc`	`wc lorem.txt`	Count the lines, words and characters in a file.
`who`		Show who is logged in.

We also met a number of graphical applications:

Command	Default for these file types	
`eog`	GIF, JPEG, PNG, SVG	"Eye of Gnome" graphics viewer
`evince`	PDF	Document viewer
`firefox`	HTML	Firefox web browser
`gedit`	TXT	Text editor
`xeyes`		Silly test application to track the cursor

5 "`unset TERM`": *Don't do this!*

Date formats

Year

`%C`	20	Century
`%Y`	2018	Four digit year
`%y`	09	Two digit year

Month

`%b`	Apr	Abbreviated month name
`%B`	April	Full month name
`%m`	04	Two digit numerical month

Day

`%j`	118	Day of year (1…366)
`%d`	28	Two digit day of month
`%a`	Tue	Abbreviated day of week
`%A`	Tuesday	Full day of week
`%u`	2	Numerical day of week (1…7, 1=Monday)
`%w`	2	Numerical day of week (0…6, 0=Sunday)

Hour

`%H`	21	Hour of the day (0…23)
`%I`	09	Hour of the day (1…12)

Minute

`%M`	07	Minute of the hour

Second

`%S`	23	Second of the minute
`%s`	1240949377	Seconds since 1970-01-01 00:00:00 GMT

Useful

`%n`		New line
`%t`		Tab

There are two useful modifiers. If "`%M`" were to give "`07`" then "`%_M`" would give "□`7`" and "`%-M`" would give "`7`".

Globbing

`*`	Any string of characters, including the empty string.	
	`thing*`	matches: **thing**.txt, **thing**, **thing**s does not match: thin, think
	`th*ing`	matches: **th**ink**ing**, **thing**, **th**e□shin**ing** does not match: ting, think
	`th*in*`	matches: **thin**g.txt, **thin**g, **thin**gs, **th**ank**in**g, **th**e\ shin**in**g, **thin**, **thin**ks
`?`	Any single character.	
	`thing.???`	matches: **thing.**txt, **thing.**dat, **thing.**jpg does not match: thing.jpeg, thing
`[…]`	Any single character from the set in the brackets.	
	`thing.t[xyz]t`	matches: **thing.tx**t does not match: thing.tot
`[^…]`	Any single character not in the set.	
	`thing.t[^xyz]t`	matches: **thing.to**t does not match: thing.txt

The character class globbing expressions `[:…:]` all appear inside the standard `[…]` brackets of the glob, so we get doubly nested square brackets. So "`[N[:digit:]]`" matches "N" or "any one digit".

`[:alnum:]`	Any alphabetic character (upper or lower case) or any digit.
`[:alpha:]`	Any alphabetic character (upper or lower case).
`[:blank:]`	Any horizontal white space (space or tab, essentially).
`[:digit:]`	Any of the ten digits.
`[:lower:]`	Any lower case alphabetic character.
`[:upper:]`	Any upper case alphabetic character.

PS1 codes

These codes can be placed inside the PS1 environment variable. There are more, but these are the useful ones.

Time codes

`\D{format}`		The date where the format is given by a format string using the `%`-codes listed above.
`\d`	`Tue 10 Sep`	The date in this specific format
`\t`	`17:28:26`	24-hour time, with seconds
`\T`	`05:28:26`	12-hour time, with seconds
`\A`	`17:28`	24-hour time
`\@`	`05:28`	12-hour time

Other codes

`\h`	`workstation`	Short machine name
`\H`	`workstation.pwf.cam.ac.uk`	Full machine name
`\l`	`2`	Terminal number, `/dev/pts/2`→2
`\u`	`y250`	User name
`\W`	`Work`	Name of the current working directory
`\w`	`/home/y250/Linux Intro/Work`	Absolute path of the current working directory

Command line cursor control

There are often two ways to do these operations. One way avoids the use of the cursor keys but requires the memorisation of some other letters.

[Ctrl]+[F]	[→]	Move right one character.
[Ctrl]+[B]	[←]	Move left one character.
[Alt]+[F]	[Ctrl]+[→]	Move right one word.
[Alt]+[B]	[Ctrl]+[←]	Move left one word.
[Ctrl]+[A]	[Home]	Move to start of line
[Ctrl]+[E]	[End]	Move to end of line.
[Ctrl]+[W]		Remove the word to the left of the cursor.
[Ctrl]+[K]		Remove the line to the right of the cursor.
[Ctrl]+[U]		Remove the line to the left of the cursor.
[Ctrl]+[P]	[↑]	Go back one line in history.
[Ctrl]+[N]	[↓]	Go forwards one line in history.

sftp commands

Any Linux command can be run on the local system by preceding it with a "`!`". Where there is no `l-` local version of a command we show the `!-` version. The `!-` version always works, except for "`lcd`".

Remote	Local	
`cd`	`lcd`	Change directory
`ls`	`lls`	List directory contents
`pwd`	`lpwd`	Print working directory
`mkdir`	`lmkdir`	Make a directory
`rmdir`	`!rmdir`	Remove empty directory
`rm`	`!rm`	Remove file

`get remote_name`	Fetch a remote file, keeping its name.
`get remote_name local_name`	Fetch a remote file, changing its name.
`put local_name`	Put a file onto the remote system, keeping its name.
`put local_name remote_name`	Put a file onto the remote system, changing its name.
`help`	Show the complete set of `sftp` commands.
`quit`	Quit `sftp`.

Environment variables

HOME

Specifies your home directory.

You should never change this value.

```
workstation:~$ echo "${HOME}"
/home/y220
```

PATH

Specifies the list of directories where the operating system goes looking for executable files to run the commands you issue.

You should only ever add to this value. Removing directories from it that are provided by the system may break some system facilities. On PWF Linux and OpenSUSE Linux your `${HOME}/bin` directory is added to your `PATH` by the system if and only if it exists when you log in.

```
workstation:~$ echo "${PATH}"
/home/y220/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/X11R6/bin:/usr/g
ames:/opt/kde3/bin:/usr/lib/mit/bin:/usr/lib/mit/sbin:/opt/novell/iprint/bin:/
opt/real/RealPlayer
```

TERM

Specifies your terminal type.

This should be set by the system and you should not need to change it. Commands that need to know the parameters of your terminal will fail if this is unset or incorrectly set. (e.g. `more` needs to know how many rows your screen has.)

```
workstation:~$ echo "${TERM}"
xterm
```