Linux Systems in the School of Earth and Environment

cemac-support@leeds.ac.uk

CEMAC
School of Earth and Environment
University of Leeds

November 2019

This document intends to provide an introduction to using Linux based computer systems within the School.

Contents

1 Introduction

1.1 Background

Linux is a UNIX like operating system.

Though commonly referred to simply as 'Linux', the complete operating system is made up of various components developed by different projects.

In particular, many of the core user tools were developed and maintained by the GNU project, which was founded by Richard Stallman, and is sponsored by the Free Software Foundation.

The Linux Kernel is maintained by the Linux Foundation, and overseen by its creator Linus Torvalds.

Much of the most common and core software developed for Linux is free.

There are many different Linux distributions available, with some of the most well known being Ubuntu, Fedora, Debian and openSUSE.

At the University of Leeds, the standard distribution used is CentOS, which is based on Red Hat Enterprise Linux.

1.2 School Systems

The School Linux systems rely heavily on the network. For example, though many core software packages are installed on the local system disk, much of the additional and useful third party software (e.g. Matlab, IDL..) is accessed from network file systems. This allows a wide range of software to be easily made available to a large number of machines.

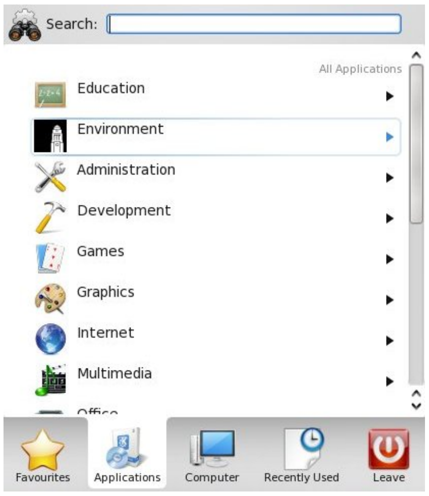
User files will generally be stored on networked file systems, which allows users to easily move between machines, and still have access to the same software, data and settings.

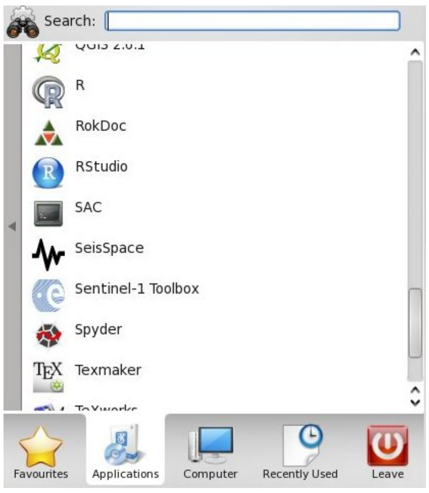
2 Desktop Environment

The default desktop environment on the School Linux systems is KDE.

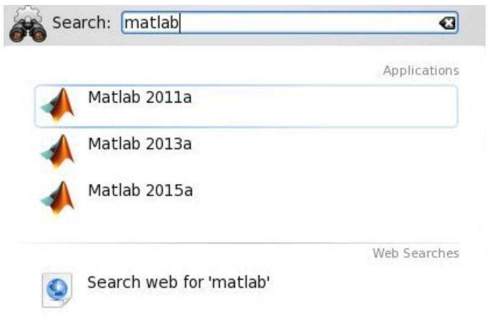
2.1 Menus

Many programs and features can be found in the system menus, with some local software available in the 'Environment' section of the Applications menu.



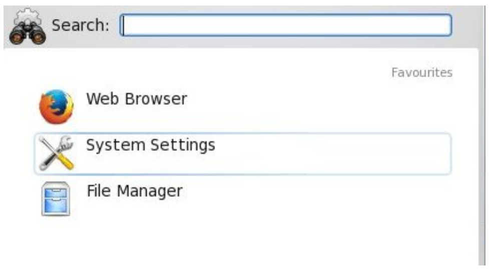


The menu can be accessed with the shortcut keys **Alt+F1**, and the search box can be used to find and access applications.



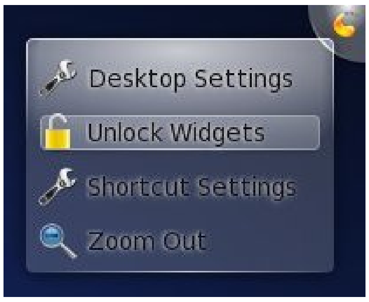
2.2 Customising The Desktop

The desktop environment is highly customisable, with many of the settings (appearance, windows behaviour ...) being found within the 'System Settings' within the menus.

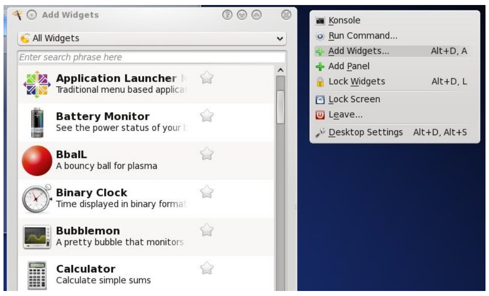


This can also be launched with the command `systemsettings`.

The KDE desktop is made up of 'Widgets'. Many Widgets are available to be added to and removed from the desktop. To Manipulate the Widgets, it is necessary to first unlock the Widgets. This can be done by right clicking on the desktop, or from the icon in the top right corner of the screen.



Once unlocked, the desktop can be manipulated, and additional Widgets can be added, for example by right clicking on the desktop, and selecting 'Add Widgets'.



When settings have been altered as required, it is wise to then lock the widgets again, to avoid accidentally changing things.

2.3 KDE Keyboard Shortcuts

Some useful keyboard shortcuts for the KDE desktop environment:

Alt+F1	Open the desktop menu
Alt+F2	Run a command
Ctrl+Alt+L	Lock the screen
Ctrl+Esc	Launch the system monitor
Ctrl+F1	Switch to workspace 1
Ctrl+F2	Switch to workspace 2
Ctrl+F3	Switch to workspace 3
Ctrl+F4	Switch to workspace 4

These window manipulation keyboard shortcuts require 'Desktop Effects' to be enabled in the session, which is enabled by default. However, these will not generally work when connecting to a desktop session on a remote machine:

Ctrl+F8	Show all workspaces
Ctrl+F9	Show all windows in current workspace
Ctrl+F10	Show all windows in desktop session
Ctrl+F11	Show all workspaces on a cube

3 Basic Commands

Getting started with a Linux system usually requires learning some of the most common and basic commands. To do this, we will need access to a shell.

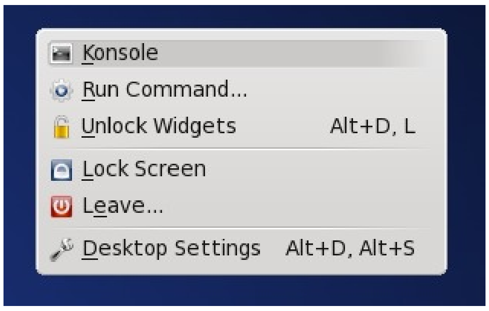
3.1 The Shell

A shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out by the computer.

Though many applications and functions can be accessed from a graphical desktop environment, a lot of work in Linux is performed from a shell.

From a desktop session a terminal window can be opened to access the command line.

When using the KDE desktop environment, the program `konsole` can be used - this can be found in the software menu, or by right-clicking on the desktop.



Opening konsole

3.2 The Basic Commands

These are some of the most commonly used, and useful commands.

3.2.1 pwd

Show the **p**ath to the current **w**orking **d**irectory:

```
$ pwd
/nfs/foe-fs-00_users/earabc
```

3.2.2 ls

List files and their properties:

```
$ ls
Documents data00.tar.gz file.txt
```

Many commands can take extra options which change the way they behave. To list files with extra information add the option `-l` (long listing).

This will display:

- file permissions
- number of links which point to that file (not particularly important to know about)
- file owner
- file group ownership
- file size
- time stamp (last modified)

```
$ ls -l
total 12516
-rw-r--r-- 1 earabc users 12805577 Nov  5 10:07 data00.tar.gz
drwxr-xr-x 2 earabc users     4096 Feb  8 14:34 Documents
-rw-r--r-- 1 earabc users       15 Oct 11 17:22 file.txt
```

Add a `-h` to list file size in 'human readable' format:

```
$ ls -lh
total 13M
-rw-r--r-- 1 earabc users  13M Nov  5 10:07 data00.tar.gz
drwxr-xr-x 2 earabc users 4.0K Feb  8 14:34 Documents
-rw-r--r-- 1 earabc users   15 Oct 11 17:22 file.txt
```

The `-a` option will show files which are usually hidden (files and directories whose names start with a '`.`'). Many programs save settings and preferences in 'dotfiles' within the home directory.

To list hidden files in your home directory:

```
$ ls -a ~
```

3.2.3 cd

Change **d**irectory:

```
$ cd Documents
$ pwd
/nfs/foe-fs-00_users/earabc/Documents
```

Running `cd` on its own, with no additional arguments, will take you back to your home directory.

3.2.4 mkdir

Make a directory:

```
$ mkdir dir00
$ ls -l
drwxr-xr-x 2 earabc users     4096 Nov  5 13:09 dir00
```

3.2.5 cp

Copy a file:

```
$ cp ../file00.txt ./file01.txt
$ ls
dir00 file01.txt
```

Note: Two dots (`..`) means the parent directory/up one level, and a single dot (`.`) means the current directory.

3.2.6 rm

Remove a file:

```
$ rm file00.txt
```

A directory and its contents can be removed by adding a `-r` (recursive delete) to the rm command. This should prompt for confirmation for deletion of each file by default. To avoid the confirmation, you can add a `-f` (force delete), but **use only with extreme caution!**:

```
$ rm -fr ./dir00
```

An empty directory can be removed with `rmdir`:

```
$ rmdir dir00
```

3.2.7 mv

Move/rename a file:

```
$ mv file01.txt file00.txt
```

3.2.8 less

View the contents of a file:

```
$ less file00.txt
```

Press **Q** to exit, and use up and down arrows to scroll. The program `more` works in a similar way.

3.2.9 man

Many commands have manual pages, which provide detailed information on usage and available options, via the man command.

General usage is:

```
man commandname
```

For example, to see all the options available to the `ls` command:

```
$ man ls
```

Related to man, the command `apropos` can be used to search for keywords in the system documentation, though you may find it can provide more results than is sometimes useful:

```
$ apropos list
...
ls                   (1)  - list directory contents
...
```

The program `whatis` can also give a provide more information about a command:

```
$ whatis cp
cp                   (1)  - copy files and directories
```

3.2.10 *

The asterisk is a special character which is used to match anything:

```
$ ls p*
```

This will list any files or directories starting with the letter p.

4 Files and Processes

Everything on a Linux system is either a file or a process.

A process is an executing program identified by a unique PID (process identifier).

A file is a collection of data, which could be:

- A Text document
- An executable binary file
- A directory

4.1 Directory Structure

The directory structure on a Linux system is a hierarchical tree like structure, with the root/base of the structure being '/'.

For example, the program `perl` may be found at the location `/usr/bin/perl`:

```
/
|-- etc/
|-- usr/
|    |-- bin/
|    |    |-- gcc
|    |    |-- perl
|    |    `-- vim
|    |-- lib/
|    `-- share/
`-- var/
```

In reality there would be many more directories and files at each level, but here we can see:

- At the root level, we have directories `/etc`, `/usr`, `/tmp`.
- Within `/usr` are the directories `/usr/bin`, `/usr/lib` and `/usr/share`.
- Within `/usr/bin` are the files `/usr/bin/gcc`, `/usr/bin/perl` and `/usr/bin/vim`.

4.2 Home Directory

Your home directory is your personal file space, stored on a file server, accessed via the network, and backed up nightly.

On Linux systems this directory is where all of your settings, preferences and such get stored (for example, your Firefox profile), and is the directory you will be in by default, when you open a terminal or log in to a machine.

Within the School, as the home directory is stored on the network, rather than a local machine, user files, preferences and configuration (Firefox settings, shell settings..) follow the user between systems, and are common between all systems.

The home directory can be accessed in different ways, for example the tilde character (~) and the variable `${HOME}` can be used to refer to the home directory.

`echo` (print to screen) the value of the `HOME` variable:

```
$ echo ${HOME}
/nfs/foe-fs-00_users/earabc
```

List home directory contents using the `HOME` variable:

```
  $ ls ${HOME}
  bin Desktop private public_html
```

List home directory contents using ~:

```
  $ ls ~
  bin Desktop private public_html
```

If you access a Windows system in the School, your home directory will appear as the 'Z:' drive.

4.3 Quotas

Space in the home directory is restricted by user quotas. You can see your current usage and limits with the `quota` command (the `-s` causes units to be displayed in megabytes/gigabytes, rather than bytes).

```
$ quota -s
Disk quotas for user earabc (uid 12345):
     Filesystem  blocks   quota   limit   grace   files   quota   limit
foe-fs-00:/export/users
                   290M   2048M   2548M            6691       0       0
```

It is wise to keep an eye on disk quota usage. Exceeding the quota limit can cause problems logging in to desktop sessions, corrupt Firefox profiles, cause file content to be erased ...

Quota increases can be requested by contacting IT support.

4.4 File Systems

Linux file systems can compromise a mixture of local, network and virtual file systems, with the root of all of these being `/`.

If you were to run the command:

```
  $ ls /
```

You would see a number of directories. Some of these are common Linux directories, for example:

```
/etc
```

is the common location for configuration files, such as printer configuration files, authentication configuration, and so on.

```
/tmp
```

is a temporary directory, to which all users can write, various programs save temporary information and files, and is **automatically cleaned up regularly** so it not a suitable location to store important files.

It is important to note that any location which is named *scratch*, *nobackup* or similar, should be used with the assumption that files are not backed up, and should only be used for temporary storage.

4.4.1 Network File Systems

Within the School, the directory `/nfs` is used as the root for network mounted file systems - i.e. file systems which are stored on remote file servers and accessed over the network.

All of the network file systems accessible under `/nfs` are *automounted*, that is, they are automatically connected and disconnected when required.

For example, if someone asked you to access a file at `/nfs/a999/earabc/file00.nc`, if you were to first look at the content of the `/nfs/` directory, it is likely that the `a999` directory would not be visible.

However, if you were to try and access the file (`ls /nfs/a999/earabc/file00.nc`), the file system will be mounted and the file will be accessible.

4.4.2 File System Information

It may not always be obvious which areas are on the local machine, and which are mounted over the network, or how much disk space is available in various locations.

The `df` command can be useful here, and is used to display information about file systems:

```
$ df
$ df -h
$ df -lh
```

Running `df` on its own will display information about all mounted file systems, adding a `h` displays the units in 'human readable' format, and adding `l` will restrict the output to local file systems only.

The `mount` command can also be used to show information about file systems:

```
$  mount
```

will display which file systems are mounted on the machine, along with additional information, such as the file system type.

4.4.3 Relative and Absolute Paths

Files and directories can be addressed by relative or absolute paths.

The absolute path is the full location to the file or directory, and will always start at the root of the file system `/`.

The relative path is the location relative to the current directory, using a single dot '`.`' to represent the current directory, and two dots '`..`' to represent the parent directory (i.e. 'up' one level).

For example, the full path to `/usr/bin` is precisely that:

```
/usr/bin
```

If you change directory to `/usr/local`:

```
$ cd /usr/local
```

the directory `/usr/bin` then has the relative path of:

```
../bin
```

If you were then to change to the `/usr` directory:

```
$ cd /usr
```

`/usr/bin` has the relative path of:

```
./bin
```

This can be useful for quickly moving around file systems and accessing files:

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

would move 'up' the file system structure one directory, and 'up' two directories respectively.

4.5 Processes

Anything which is running on a Linux system (web browser, compiler..) is a process, and will be allocated a unique *PID* (process identifier).

4.5.1 top

To see what is currently happening on your machine, the `top` command can be used. By default, this will return the processes using most CPU resources on the system:

```
Tasks: 735 total,   1 running, 733 sleeping,   0 stopped,   1 zombie
Cpu(s):  5.9%us, 21.1%sy, 11.8%ni, 61.1%id,  0.1%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  65962068k total,  7905036k used, 58057032k free,   432864k buffers
Swap: 68019200k total,  6895832k used, 61123368k free,  3814188k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
21902 earabc    17   0 9789m 1964 1948 S 95.3  0.0  62048:43 MATLAB
 2152 ee09a2b   16   0 12192 1632  724 S  1.8  0.0 226:20.41 gcc
```

As well as process information, we can see some system information, such as the CPU usage and how much memory is available and being used.

Pressing **1** will expand the CPU information, and show more information about the activity on each individual processor.

By default top orders processes by CPU usage, to instead order processes by memory usage press **Shift+M**.

To switch back to ordering by CPU usage, press **Shift+P**.

To filter the processes so that only your own are displayed, press **U**, and then enter your username.

To exit top, press **Q**.

4.5.2 ps

Another useful command for displaying process information is `ps`. If run alone:

```
  $ ps
```

Will display the processes running in the current shell.

If you want to display all processes which are running on the system, you can run:

```
  $ ps -efH
```

The option `-e` means *everything*, i.e. all processes are displayed. The `-f` means the *full* information about the processes will be displayed (user, process id, start time…), and the `-H` displays the information in a *hierarchical* format, so you can see which processes have launched other processes.

To see information about all of your own processes on a system, you can run:

```
  $ ps -fH -u username
```

replacing *username* with your own username. It may return something similar to:

```
UID        PID  PPID  C STIME TTY          TIME CMD
earabc  31946 31935  0 14:09 ?        00:00:00 sshd: earabc@pts/0
earabc  31947 31946  0 14:09 pts/0    00:00:00   -csh
earabc   3771 31947  0 16:08 pts/0    00:00:00     ps -fH -u earabc
```

Here, we can see a shell (`-csh`) was launched by a `ssh` process, and within the shell, the `ps` command is being run. You should also be able to see the relationship between PID (process identifier) and PPID (parent process identifier).

5 The Shell

As mentioned previously, the shell is the user interface to the Linux operating system, and within the School, the default shell for new accounts is `bash`, though `tcsh` (also known as `csh`, or *c-shell*) was the default for many years, and is still widely used.

Various different shells exist, and the default may vary on different systems that you may access outside the School. The `bash` shell is genenrally the most popular. Be aware, if you find documentation written for the bash shell, that the syntax may be different in `csh`, and vice-versa. You can see the value of your default shell with:

```
$ echo ${SHELL}
```

If you wish to change your default shell on the School Linux systems, contact IT Support.

The shell is a powerful interface, and can be customised to the wishes of the user, and has many shortcuts available to help maximise efficiency.

One of the most useful keyboard shortcuts to know about is:

Ctrl+C

Pressing these two keys together will cancel the current operation, and return you to a prompt. If things start to go wrong, hammering **Ctrl+C** many times, can sometimes make them better.

The best resource for information about the shell is the manual page, though it is quite a long read:

```
$ man bash
```

or, if you prefer:

```
% man csh
```

5.1 Auto Completion

The shell can automatically complete command and file names, with use of the **Tab** key.

If you wanted to change to the directory `~/Desktop`, you should be able to type:

```
$ cd ~/Desk
```

and press the **Tab** key, to auto complete the directory name.

If you wished to run the `gfortran` compiler, you should be able to tap in:

```
$ gfor
```

and press **Tab**, to have the command name automatically completed.

If there is more than one auto complete option available, you can press **Tab** twice to see the options:

```
$ gfortran #- press Tab twice:
gfortran   gfortran44
```

Here we can see two different versions of `gfortran` are available.

5.2 Aliases

Aliases can be used to create customised commands to perform various actions. You can see all currently assigned aliases by simply running:

```
$ alias
rm='rm -i'
```

Here, we can see that when the `rm` (remove a file) command is run, the action performed will actually be `rm -i`, this means the command will be interactive, and will prompt for confirmation before removing a file.

Aliases can easily be created, for example if we wanted to change to the `~/Desktop` directory by simply pressing `d`:

```
$ alias 'd'='cd ~/Desktop'
```

If we wanted this alias to change to `~/Desktop`, and then clear the screen:

```
$ alias 'd'='cd ~/Desktop; clear'
```

The semi colon is used to separate the two commands.
This alias is arguably of limited use, so we can remove it with `unalias`:

```
$ unalias d
```

5.3 History

The shell keeps a history of commands which have been run, this can be viewed with the `history` command:

```
$ history
```

When you log out of shell, the history from that session is saved into the file:

```
~/.bash_history
```

In `csh`:

```
~/.history
```

Therefore, when you inspect your shell history, you may see commands from previous sessions as well as the current session.

It is possible to browse through your command history with the up and down arrow keys, and also to access the history with some useful shortcuts.

If you run `history`, it will show an index number before each command:

```
$ history
   ...
   322 man bash
   323 cd ~/Desktop
   324 history
```

In this case, if we wanted to run the command `man bash` again, we can use the exclamation mark, followed by the index number:

```
$ !322
```

And to run `cd ~/Desktop`, it would be:

```
$ !323
```

The previous command can be referenced with two exclamation marks:

```
$ ls ~
$ !!
```

Will re-run the command `ls ~`.
To re-run the last command which started with `ls`, you can use:

```
$ !ls
```

The `ls` can be replaced with other text as desired:

```
$ echo "hello"
hello
$ cd ~
$ !echo
echo "hello"
hello
```

There is an extremely useful keyboard shortcut available to help search through your history - if you press:

Ctrl+R

and type the text to search for, you can then browse through previous commands which contained this text by again pressing:

Ctrl+R

until you find the command that you were searching for.

5.4 Variables

Variables can be used in many ways by different programs and commands. Earlier we saw that the `HOME` variable will set the location of the home directory:

```
$ echo ${HOME}
/nfs/foe-fs-00_users/earabc
```

Here we see that when a variable is referenced, it should be prefixed with a `$`. The curly brackets aren't always necessary:

```
$ echo $HOME
```

will produce the same results, though it is generally recommended to use the curly bracket, as this example illustrates:

```
$ day=fri
$ echo $dayday

$ echo ${day}day
friday
```

You can see all environment variables which are currently set with:

```
$ env
```

Try inspecting these variables:

```
$ echo ${USER}
$ echo ${HOSTNAME}
```

Some variables are specific to certain programs, for example, if set, the variable `MATLABPATH` can be used to point the program Matlab, at directories which contain functions you wish to use.

If you had created some excellent Matlab programs in your home directory at `~/matlab/programs`, which you wished to be available from the Matlab command prompt, you can set the variable using `export`:

```
export MATLABPATH="${HOME}/matlab/programs"
```

You can then add another directory separated by a colon:

```
export MATLABPATH="${MATLABPATH}:${HOME}/matlab/moreprograms"
```

Here, we have appended the directory `${HOME}/matlab/moreprograms` to the existing value of the `MATLABPATH` variable.

If you change your mind, you can use `unset` to remove the variable:

```
unset MATLABPATH
```

For reference, in a `csh` shell, the same thing would be done with:

```
setenv MATLABPATH "${HOME}/matlab/programs"
unsetenv MATLABPATH
```

5.5 The PATH Variable

The `PATH` variable is very important - this sets the locations which are searched for executable files and programs, and allows these to be called by name rather than entering the full path.

To examine the current value of the `PATH` variable:

```
echo ${PATH}
```

Because the `gcc` compiler program is located at `/usr/bin/gcc`, and the directory `/usr/bin` is located in the `PATH` variable it is possible to run:

```
$ gcc
```

rather than having to type:

```
$ /usr/bin/gcc
```

It is possible to have multiple versions of the same program in different locations - the first directory listed in the `${PATH}` variable takes precedence.

The `which` command will tell you which version of a program will be run. Let's see which version of `gfortran`, the GNU Fortran compiler, would be run, if we simply type `gfortran`:

```
$ which gfortran
```

The `whereis` command will tell us if the `gfortran` command can be found elsewhere in the `PATH`:

```
$ whereis gfortran
```

When altering the PATH variable, it is wise to be cautious.

If we had installed a new program within ${HOME}/program/bin, and wanted this to be available in the PATH, it is always sensible to either append or prepend to the PATH than reset it completely.

These methods are not recommended:

```
$ export PATH="${HOME}/program/bin"
$ export PATH="/usr/bin:${HOME}/program/bin"
```

The first example would remove all directories from the PATH, and re-set the value to only include ${HOME}/program/bin.

The second example explicitly sets the value to include only /usr/bin and ${HOME}/program/bin.

Depending on which system you log in to, other directories may be automatically added to the PATH when you log in, and these would be removed, therefore it is always advisable to append or prepend to the existing value, rather than set it explicitly.

To prepend to the PATH, use:

```
$ export PATH="${HOME}/program/bin:${PATH}"
```

To append to the PATH:

```
$ export PATH="${PATH}:${HOME}/program/bin"
```

Prepending to the path is useful, if you want to replace the default version of a program with an updated or customised version.

Appending is recommended if you want to make sure your program does not take precedence over another program with the same name.

In csh, if you add a new file to a directory within your PATH, it may not be picked up straight away - to get the shell to rescan the PATH directories, run:

```
% rehash
```

Some other *'special'* variables which you may come across:

LD_LIBRARY_PATH	Locations where programs look for run time libraries.
LIBRARY_PATH	Locations where compilers may search for link time libraries.
MANPATH	Locations of manual pages.

5.6 Quoting Variables

When using variables in commands which involve quotes, it is important to know that single quotes are used to mean the contents should be interpreted literally, whereas variables will be expanded within double quotes:

```
$ echo '$HOME'
$HOME
$ echo "$HOME"
/nfs/foe-fs-00_users/earabc
```

5.7 Shell Variables

The shell has its own set of specific options which change how the shell behaves, and which you can view with:

```
$ shopt
```

In `csh`:

```
% set
```

The shell also has special variables which change the way it behaves, for example to change the shell prompt, the `PS1` setting can be altered, for example:

```
$ export PS1="HELLO > "
HELLO >
```

The prompt in `csh` can be altered with `set prompt`:

```
% set prompt="HELLO > "
HELLO >
```

There are various other shell settings which can be tweaked - you can find all the information within:

```
$ man bash
```

For `csh`

```
% man csh
```

5.8 Shell Customisation Files

The shell has special files which are read upon start up, and which can be used to change the way the shell behaves.

The two which are probably of most interest are:

```
~/.bash_profile
~/.bashrc
```

The `~/.bash_profile` file is read at login time, and contains settings which generally only need to be set once, when first logging in to a machine.

The `~/.bashrc` file is re-read each time a shell is opened.

Default versions of these files, with some useful settings are provided in your home directory.

Generally, most users will customise their settings using the `~/.bashrc` file.

To inspect what's in there:

```
$ less ~/.bashrc
```

For example, if you wanted to add an alias to always be available in your sessions, you could add a line to the end of the file such as:

```
alias 'x'='exit'
```

If you edit your `~/.bashrc` file, and want the changes to immediately be available you can use the `source` command to re-read and apply the settings:

```
source ~/.bashrc
```

For reference, the equivalent files for a `csh` shell are:

```
~/.login
~/.cshrc
```

5.9 Shell Keyboard Shortcuts

Knowing a few handy keyboard shortcuts can really make life easier when working in a shell.

These are some of the most of the useful ones:

Navigation:

Ctrl+A	Move to the start of the line
Ctrl+E	Move to the end of the line
Alt+B	Move backward one word
Alt+F	Move forwards one word
Ctrl+XX	Skip between the beginning of the line and the current location
Shift+Page Up / Shift+Page Down	Scroll up and down
Ctrl+R	Search through command history

Command Line Manipulation:

Alt+Backspace	Delete previous word
Ctrl+W	Delete line backwards from current location
Ctrl+K	Delete line forwards from current location
Ctrl+U	Delete the entire line
Alt+D	Delete the word forward from current location
Ctrl+Y	Paste text that has been deleted/cut
Alt+_	Insert the last word from the previous line
Alt+R	Revert any changes made to current command
Ctrl+T	Switch the character under the cursor with the previous character
Alt+U / Alt+L / Alt+C	Set word to uppercase/lowercase/capitalised, forward from current position

Terminal Control:

Tab	Autocomplete command and file names
Ctrl+C	Cancel current operation
Ctrl+L	Clear the terminal
Ctrl+Z	Suspend a running process

6 Exercises

6.1 Example Files

For the next section, we are going to need to grab some example files.

We will make a directory to work in, and download the files from the web using the wget command:

```
$ cd
$ mkdir linux_intro
$ cd linux_intro
$ wget "https://cemacrr.github.io/linux_intro/example_files.zip"
```

It looks like we downloaded a *.zip* file, but we can use the `file` command to verify this - it examines a file's contents to determine the file type:

```
$ file examplefiles.zip
```

If all looks correct, we can view the contents of the *.zip* file:

```
$ unzip -l examplefiles.zip
```

and then extract:

```
$ unzip examplefiles.zip
```

Let's take a look at what we have:

```
$ ls
```

What's the difference between the output from these two commands:

```
$ ls -l s*
$ ls -ld s*
```

6.2 File and directory size

We have seen previously that we can see the size of files using the `ls -l` and `ls -lh` commands.

Sometimes it is useful to find out the size of a directory and it's contents - for this we can use the `du` command. Try these and see how the arguments alter the output:

```
$ du
$ du -h
$ du -ch
$ du -chs
```

If no directory is specified with the `du` command (e.g. `du -chs doc`), then the current directory (`.`) will be used.

The meaning of the different arguments can be found within the manual page:

```
$ man du
```

6.3 File Permissions

Change in to the `bin` directory:

```
$ cd bin
```

There should be a single file named *hello*, examine the file properties with:

```
$ ls -l
-rw-r--r-- 1 earabc staff 33 Oct 13 17:47 hello
```

The first character would be a `d` if this was a directory, as this is a file it is a dash `-`. The following nine characters display the permissions on the file.

These can be broken down in to three groups of three, to represent:

User	Permissions for owner of the file (`rw-`)
Group	Permissions for the group (`r--`)
Others	Permissions for everyone else (`r--`)

The file owner and group are also displayed in the listing. Here, the file owner is `earabc`, and the group is `staff`.

The additional information displays the file size (33 bytes), the date of last modification, and the file name.

In this case we can see that the file owner has permissions `r` and `w`, which stand for *read* and *write* - the owner can view and modify the file.

The group and others have only `r` (read) permission - they can view the file, but not modify.

As this directory is named `bin`, we're going to presume that this is a program which we should be able to run, so let's try:

```
$ ./hello
```

What happens?

It appears we also need to make this file executable, see what effect this has:

```
$ chmod +x hello
$ ls -l
$ ./hello
```

The program should now run, and have the permissions:

```
-rwxr-xr-x
```

We can see that three `x` characters are now displayed - this means that the user, group and others all now have emphexecute permissions.

r	Read permission - the permission to read and copy the file
w	Write permission - the permission to change a file
x	Execute permission - the permission to execute a file

Directories need to have the execute permission set to allow a user access.

More examples:

`drwxrwxrwx`	Indicates permissions for a directory to which anybody can write
`-rw-------`	Indicates a file which is only readable or writeable by the owner

File permissions can be altered using the chmod command, using the following values:

u	user
g	group
o	other
a	all
r	read
w	write (and delete)
x	execute (and access directory)
+	add permission
-	take away permission

We can remove all permissions on the file hello for the group and others with:

```
$ chmod go-rwx hello
```

To allow all users to read and write to the file:

```
$ chmod a+rw hello
```

It is also possible to set file permissions using numeric values. Each permission is set a value:

r	4
w	2
x	1

These can be added together to add up to a maximum of 7 for each of *user*, *group* and *other*, to give a 3 digit file permission value.

If we wanted to set the file *hello* to have **r**ead, **w**rite and e**x**ecute permissions for the owner, **r**ead and e**x**ecute permissions for the group, and **r**ead only permissions for others, we can do some quick sums:

user	r + w + x = 4 + 2 + 1 = **7**
group	r + x = 4 + 1 = **5**
other	r = **4**

Giving us a value of 754:

```
$ chmod 754 hello
$ ls -l
-rwxr-xr-- 1 earabc staff 33 Oct 13 17:47 hello
```

What do these do?:

```
$ chmod 700 hello
$ chmod 0 hello
```

6.4 File Manipulation

6.4.1 less

`less` is a useful tool for viewing the contents of a text file.

If we change to the *texts* directory, within the example files, we should see some text files:

```
$ cd ../texts
$ ls
data.txt  fruit.txt  manx.txt
```

Take a look at the file *manx.txt*:

```
$ less manx.txt
```

The up and down arrows can be used to move up and down the text, and the text can be searched using the `/` character:

```
/kitten
```

should take you to the next occurrence of the word *kitten*. You can press **N** to go to the next match, or **Shift+N** to go to the previous match.

Ctrl+F can be used to move down one page, and **Ctrl+B** can be used to move back one page.

Pressing **GG** (**G** twice) will take you to the top of the file and **Shift+G** should take you to the bottom of the file.

Pressing **Q** will exit `less`.

What happens if you open the file with:

```
$ less -N manx.txt
```

6.4.2 grep

`grep` is powerful tool for searching through text files, and quite simple to use:

```
$ grep kitten manx.txt
```

will display any line in the file containing the word *kitten*.

`grep` is case sensitive, try:

```
$ grep many manx.txt
$ grep Many manx.txt
```

The search can be made case insensitive with the `-i` option:

```
$ grep -i many manx.txt
```

Sometimes it can be useful to have any matches highlighted with the `--color` option:

```
$ grep --color dog manx.txt
Many people call the Manx the "dog cat" because of its strong desire to be
attack by a Manx that is very protective. Strange dogs are especially a target
```

If we want to search for lines which don't match the pattern, we can add the `-v` option:

```
$ grep -iv manx manx.txt
```

To also display the line number add `-n`:

```
$ grep -in kittens manx.txt
```

If searching for a phrase, this can be quoted:

```
$ grep 'Great Britain' manx.txt
```

The `-c` option can be used to show how many lines match:

```
$ grep -ic 'manx cat' manx.txt
5
```

or, combined with `-v`, to show how many lines do not match:

```
$ grep -ivc 'manx cat' manx.txt
210
```

6.4.3 wc

`wc` (Word Count) is a useful tool for finding out more information about a file:

```
$ wc manx.txt
  215  2301 12983 manx.txt
```

This shows us the file contains 215 lines, 2301 words and 12983 characters.

These values can be found individually with the options `-l` (*line*s), `-w` (*words*), `-c` (*characters*):

```
$ wc -l manx.txt
$ wc -w manx.txt
$ wc -c manx.txt
```

6.4.4 sort

The sort command can be used to, as expected, sort data. Have a quick look at the contents of the file *fruit.txt*:

```
$ less fruit.txt
```

It is a list fruits, in no particular order.

Running the command:

```
$ sort fruit.txt
```

Will display the contents of the file, sorted in alphabetical order.

As there are duplicates in the file, we can perform a *unique* sort, with the `-u` option:

```
$ sort -u fruit.txt
```

The file *data.txt* contains:

```
00      aa      500
01      bb      400
02      cc      300
03      dd      200
04      ee      100
```

We can sort these lines in *reverse* order, using the `-r` option:

```
$ sort -r data.txt
```

If we wanted to sort by values in the third column, we can use the `-k` (*key*) option:

```
$ sort -k3 data.txt
```

or the second column, in reverse order:

```
$ sort -k2 -r data.txt
```

6.4.5 head and tail

`head` and `tail` can be used to view the beginning or end of a file. The default is to display 10 lines:

```
$ head manx.txt
$ tail manx.txt
```

The number of lines can be altered by adding the `-n` option followed by the desired number of lines:

```
$ head -n 5 manx.txt
$ tail -n 20 manx.txt
```

Another useful feature of tail is the ability to follow the contents of a file as it is being created. For example, if you had a program which was writing data to the file *output.txt*, you could see what was being written to the file with:

```
$ tail -f output.txt
```

6.4.6 cat

The `cat` command can be used in various ways. The description of the command says:

```
$ whatis cat
cat      (1) - concatenate files and print on the standard output
```

In its simplest form, it can be used to quickly display the content of a file:

```
$ cat fruit.txt
banana
apple
mango
pear
peach
orange
apple
```

or multiple files:

```
$ cat data.txt fruit.txt
00      aa      500
01      bb      400
02      cc      300
03      dd      200
04      ee      100
banana
apple
mango
pear
peach
orange
apple
```

6.5 Redirection

Many programs will display the output on the screen, as with the `cat` commands above. We can use redirection to instead output to a file. Output can be redirected with the > character:

```
$ echo "1234" > file00.txt
```

Will redirect the output from the `echo` command (*1234*) in to the file *file00.txt*.

It is also possible to append to the end of a file using >>. Try running:

```
$ echo "5678" >> file00.txt
```

The file should now contain two lines:

```
$ cat file00.txt
1234
5678
```

If we append an extra line to *fruit.txt*:

```
$ echo "grape" >> fruit.txt
```

we can then `sort` the output, and redirect this to a new file:

```
$ sort -u fruit.txt > fruit-sorted.txt
```

will output the uniquely sorted list to the new file *fruit-sorted.txt*.
The input to some commands can also be redirected with the < character:

```
$ sort < fruit.txt
```

This would ask the sort command to read in the file *fruit.txt* for sorting.

6.6 The Pipe

The pipe character | can be used to redirect the output from one command to the input of another command. If we wanted to sort the file *fruit.txt* and then search for any line containing the letter *e*:

```
$ sort fruit.txt | grep 'e'
apple
apple
orange
peach
pear
```

The pipe can be used many times in a single line - if we then wanted to count how many lines we have:

```
$ sort fruit.txt | grep 'e' | wc -l
5
```

The pipe can be a very powerful tool for quickly performing complex tasks.

6.7 Editing Text Files

There are many different text editors available on Linux systems, and most people have a preference for one or the other.

There are many graphical editors which may be available, such as `kwrite`, `kate`, `gvim`, `emacs`..

If working in a session without graphical output, or working on a system with no graphical text editor available (for example if you are working on a high performance cluster system), it is useful to know how to use a terminal based text editor.

The editor `vi` is generally available on all Linux and UNIX systems, so it is wise to know at least some basics of how it can be used.

Open the text editor with:

```
$ vi
```

or to open an existing file:

```
$ vi filename
```

To be able to enter text, you need to enter *insert* mode, by pressing the **I** key, and to stop editing press the **Esc** key.

When not in *insert* mode (press **Esc**), navigating around the file is similar to using the `less` command - The arrow keys can be used to move up, down, left and right.

Pressing **GG** (pressing **G** twice) will move to the top of the file, and pressing **Shift+G** will move to the bottom of the file.

Ctrl+F will move down one page, and **Ctrl+B** will move up one page.

Text can be searched with the / character:

```
/word
```

would perform a case sensitive search, adding `\c` to the beginning of the word will make the search case insensitive:

```
/\cword
```

Pressing **N** will move to the next match, and **Shift+N** moves to the previous match.

A file can be saved by pressing **Esc** and then entering:

```
:w
```

To specify the file name to be saved:

```
:w newfile.txt
```

To save the file and exit:

```
:wq
```

To exit `vi`:

```
:q
```

To exit without saving:

```
:q!
```

If things start to go wrong in a `vi` session, it often helps to press **Esc** several times, and then enter `:q!`.

Those are the very basic commands which should be enough to get started, but there are many, many more things which can be done with the `vi` text editor.

The following commands may be of use (Note: You may have to press **Esc** before running any of these, to exit editing mode):

Shift+R	Replace mode replace existing text
YY (**Y** twice)	’Yank’ (copy) a line
5 YY	Copy five lines, including the current line (replace 5 with any number as required)
P	Paste below the current line
Shift+P	Paste above the current line
O	Open a new line below the current line
Shift+O	Open a new line above the current line
DD (**D** twice)	Delete the current line (deleted lines can be pasted with **P**)
10 DD	Delete ten lines, including the current line.
DW (**D** then **W**)	Delete the next word
U	Undo last command
Ctrl+R	Redo an ’undone’ action
`:13`	Go to line 13
`:set number`	Turn on line numbers
`:set nonumber`	Turn off line numbers
`:s/old/new`	Replace the first instance of *old* and replace with *new* on the current line.
`:s/old/new/g`	Replace all instances of *old* with *new* on the current line.
`:%s/old/new/g`	Replace all instances of *old* with *new* in the current file.
`:%s/old/new/gc`	Replace all instances of *old* with *new* in the current file, asking for confirmation each time.

A more complete `vi` reference sheet can be found at https://www.cs.cornell.edu/courses/cs312/2006fa/software/quick-vim.pdf.

6.8 find

The `find` command is a tool for searching for files and directories.

When run without any additional arguments, it will find all files and directories in the current location, and print their names.

In the directory containing the example files, try:

```
$ find
```

We could pipe the output through `grep` to search for a particular file:

```
$ find | grep 'fruit'
```

but it is more efficient to use the built in functionality of `find`. For example, to search for any file which has a name ending in *.txt*:

```
$ find -name '*.txt'
```

`-name` is case sensitive, the case insensitive option is `-iname`:

```
$ find -iname '*.jpg'
```

would find the files *photo01.jpg* and *PHOTO01.JPG*.

To find only directories, we can specify `-type d`:

```
$ find -type d
```

or to find only files:

```
$ find -type f
```

`find` can search based on many other criteria - to find files modified in the last day:

```
$ find -mtime -1
```

and options can be added together - to find directories only, that have names that begin with an *s*:

```
$ find -type d -iname 's*'
```

much more information can be found in the manual page:

```
$ man find
```

The `find` command is capable of many clever things - don't worry if these examples look a bit confusing now, they may be of use in the future.

This command will search within the *texts* directory for any file, with a name ending with *.txt*, and then execute the command `ls -l` on these files:

```
$ find texts/ -type f -name '*.txt' -exec ls -l '{}' \;
-rw-r--r-- 1 earabc staff 12983 Oct 14 21:16 texts/manx.txt
-rw-r--r-- 1 earabc staff 50 Oct 13 18:39 texts/data.txt
-rw-r--r-- 1 earabc staff 43 Oct 13 18:37 texts/fruit.txt
```

This command will search search the directory *texts*, for any files with names ending *.txt*, print out the results, which are then passed, via the `xargs` command, to the `grep` command, which searches for *banana*, and prints out the results:

```
$ find texts/ -type f -name '*.txt' -print0 | xargs -0 grep -i banana
texts/manx.txt:Some Manx cats like bananas. Some do not.
texts/fruit.txt:banana
```

6.9 Regular Expressions

Regular expressions (often referred to as *regex* or *regexp*) are used to match patterns, and can be used in many programs, for example grep, `vi`, `less`, and many programming languages.

Different characters have special meanings, such as ^ to represent the start of a line, and *$* to match the end of a line:

```
$ grep '^a' fruit.txt
apple
apple
$ grep 'a$' fruit.txt
banana
```

The first example above matches any lines which start with *a* and the second, any lines which end with *a*.

The dot . can be used to match any character:

```
$ grep '^pea.' fruit.txt
pear
peach
```

The asterisk * is used to match the preceding character zero or more times:

```
$ grep 'an.*' fruit.txt
banana
mango
orange
```

Note how the dot is also required - the dot matches any character, and the asterisk allows this to be repeated any number of times.

Groups of characters to be matched can be specified inside square brackets *[]*:

```
$ grep 'ang[eo]' fruit.txt
mango
orange
```

Here we have searched for the letters *ang* followed by either an *e* or an *o*. This would also match *banger*, but not *language*.

Ranges of numbers and letters can also be specified within square brackets:

[0-9]   [a-z]   [A-Z]

should match any number, any lower case letter, and any upper case letter in turn.

Depending on the program being used, these may have to be specified as:

[:digit:]   [:lower:]   [:upper:]

Using `grep`, these have to be within square brackets:

```
$ grep '[[:digit:]]' data.txt
00       aa       500
01       bb       400
02       cc       300
03       dd       200
04       ee       100
```

A mass of information regarding regular expressions can be found at http://www.regular-expressions.info/.

6.10 sed

The `sed` command can be used to quickly and powerfully manipulate text files.

There are many, many ways in which `sed` can be used, these are some useful basics.

Input can be piped into `sed`, for example, to search and replace:

```
$ echo "Hello ian" | sed 's/ian/Ian/g'
Hello Ian
```

Here, we have chosen to search (`s`) for the word *ian* and replace with *Ian* globally (`g`).

To perform a case insensitive search, we can add an insensitive (`i`) option:

```
$ echo "Xello xoward" | sed 's/x/H/gi'
Hello Howard
```

`sed` can also perform operations directly on text files. By default, this will output the results to the screen:

```
$ sed 's/peach/pineapple/g' fruit.txt
banana
apple
mango
pear
```

```
pineapple
orange
apple
```

This output could then be redirected to another file:

```
$ sed 's/peach/pineapple/g' fruit.txt > fruit_new.txt
```

It is also possible to edit the file directly, with the `-i` option:

```
$ sed -i 's/peach/pineapple/g' fruit.txt
```

Some more quick examples of `sed` usage . . . Delete lines containing the word *apple*:

```
$ sed '/apple/d' fruit.txt
```

Delete lines 2 to 4 inclusive:

```
$ sed '2,4d' fruit.txt
```

Print lines 3 to 5 inclusive:

```
$ sed -n '3,5p' fruit.txt
```

6.11 awk

`awk` is another powerful tool for working with text files. Like `sed`, input can be 'piped' in, or taken from a file. One of the most common uses is to print out specified fields:

```
$ awk 'print $3' data.txt
500
400
300
200
100
```

The instructions for `awk` appear within the single quotes. Here, we have told `awk` that we wish to print the third field from the file *data.txt*.

If the separator in the file is not white space, this can be specified with the `-F` option:

```
$ echo "00:01:02:03:04" | awk -F ':' 'print $2"-"$3"-"$4'
01-02-03
```

We have told `awk` to separate the input wherever there is a colon, and print out field 2, 3 and 4, separated by a dash.

`awk` can also be used to perform some quick calculations:

```
$ echo "00:01:02:03:04" | awk -F ':' 'print $2/$4'
0.333333
```

6.12 Loops

When tasks have to be repeated multiple times, these can often be made much easier with a loop.

Here is a simple loop, to `echo` the specified values:

```
$ for name in bob ian jane
> do
> echo "hello ${name}"
> done
```

The same can be done in `csh` in a slighlty different way:

```
% foreach name ( bob ian jane )
foreach? echo "hello ${name}"
foreach? end
hello bob
hello ian
hello jane
```

Here, we have create a variable `${name}`, which will in turn have the values *bob*, *ian* and *jane*.

We then specify what actions we would like to be take for each value - in this case echo *hello* followed by the current value of `${name}`.

Multiple actions can take place inside a loop - if we wanted to create a number of directories, change their permissions, and then list their properties, we can save having to repeatedly type these commands:

```
$ for dir in directory00 directory01 directory02
> do
> mkdir ${dir}
> chmod 700 ${dir}
> ls -ld ${dir}
> done
```

The three directories have been created with the specified names, their permissions changed, and their properties listed.

For reference, in a `csh` shell, the same loop would look like this:

```
% foreach dir ( directory00 directory01 directory02 )
foreach? mkdir ${dir}
foreach? chmod 700 ${dir}
foreach? ls -ld ${dir}
foreach? end
drwx------ 2 earabc staff 4096 Oct 15 14:30 directory00
drwx------ 2 earabc staff 4096 Oct 15 14:30 directory01
drwx------ 2 earabc staff 4096 Oct 15 14:30 directory02
```

7 Scripts

Shell scripts are, at the basic level, a number of instructions in a text file, which can be run as a single command, and can be used to automate tasks, perform repetitive tasks, etc.

A script is just a text file, with a special first line, which specifies the program which will interpret the commands.

The first line of a script to be interpreted by `bash` would begin:

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

The characters `#!` followed by the path to the program which will interpret the commands.

A `csh` script would have:

```
#!/bin/csh
```

and a `perl` script:

```
#!/usr/bin/perl
```

Presuming these were the correct paths to the `bash`, `csh` and `perl` programs on the system.

If we created a script *mkdirs.sh* (note, the file extension is not required, but may be useful to easily identify what type of file this is) to create a few directories, the contents might look like this:

```
#!/bin/bash

#- specify the year and some months:
year="2015"
months="January February March"

#- make the directory for the year:
mkdir /tmp/${year}

#- loop through the months, and create
#  a directory within the year directory
#  for each month:
for month in ${months}
do
  mkdir /tmp/${year}/${month}
done
```

Note how we can add comment lines, by starting the line with a 'hash' (`#`) character, and how indentation is used to help keep things tidy and easy to read.

Once the file has been created, it will need to be made executable:

```
$ chmod 755 mkdirs.sh
```

If it is then run:

```
$ ./mkdirs.sh
```

It should create the following directories:

```
/tmp/2015/
/tmp/2015/January
/tmp/2015/March
/tmp/2015/February
```

This, and other example scripts can be found within the example files that were downloaded earlier, within the *scripts* directory.

7.1 Moving files between Linux and Windows systems

If moving text files between Linux and Windows systems, particularly scripts, you have to be aware of how they may be handled differently on the different systems.

The two systems handle line breaks differently, so if you create a file on a Linux system and view it in a Windows text editor such as notepad, it may display all the text on a single line, and if you create a script on Windows system to be run on a Linux system, you may see some strange errors:

```
$ ./windowsScript.csh
./windowsScript.csh: Command not found.
```

Luckily, there are some tools which can be used to easily convert text files for moving between the two systems:

```
unix2dos
dos2unix
```

If we run our script from the Windows system through `dos2unix`, that should make things work:

```
$ dos2unix windowsScript.csh
dos2unix: converting file windowsScript.csh to UNIX format ...
$ ./windowsScript.csh
hello earabc
```

`unix2dos` can be used in much the same way to convert text file created on a Linux system to be compatible with a Windows text editor.

There are also text editors available for Windows which are able to handle Linux/UNIX text files, such as the program *notepad++*.

8 Job Control

Programs or commands which take a long time to run can be set to run in the background, which allow you to work on other things in the terminal.

The sleep command will wait a specified number of seconds before continuing:

```
$ sleep 10
```

If we wanted this job to run in the background while we do other things, we can add an ampersand (`&`) at the end of the command:

```
$ sleep 10 &
[1] 20134
$
```

This has told us the process id of the job, and returned us to the command prompt, where we can continue to work.

After the job has finished, output may be displayed on the screen such as:

```
[1]    Done                         sleep 10
```

This can also be useful for launching graphical programs:

```
$ firefox &
```

This would launch the excellent `firefox` browser, and then allow you to continue working in the shell.

Once a job has been started, it can be set to run in the background, by pressing **Ctrl+Z**:

```
$ sleep 10
[press Ctrl+Z]
Suspended
```

So what has happened to this job? We can find out by running `jobs`:

```
$ jobs
[1]  + Suspended                    sleep 10
```

At the moment the process is suspended or frozen - nothing is happening. If we want this to continue in the background, we can run `bg`:

```
$ bg
[1]    sleep 10 &
$ jobs
[1]  + Running                      sleep 10
```

We can see the status of the process has now changed to *Running*. The job can be brought back in to the foreground with `fg`:

```
$ fg
sleep 10
```

It is possible to have multiple jobs running:

```
$ sleep 100 &
[1] 20471
$ sleep 200 &
[2] 20475
$ jobs
[1]  + Running                      sleep 100
[2]  - Running                      sleep 200
```

In these cases, jobs can be referred to using their job number, which is within the square brackets.

If we wanted to bring job 1 into the foreground:

```
$ fg %1
sleep 100
```

The percentage sign is prefixed to the job number.

If we want to end a process we can use the `kill` command:

```
$ kill %2
[2]  - Terminated                   sleep 200
```

`kill` can also be used on process id numbers (if they are your own processes):

```
$ kill 123456
```

Would `kill` the process with id *123456*.

If a process does not want to stop, `kill -9` tries to end the process more forcefully:

```
$ kill -9 %2
$ kill -9 12345
```

In short:

`&`	Run process in the background
Ctrl+Z	Suspend a processes
`fg`	Run a job to the foreground
`bg`	Run a job in the foreground
`kill`	End a process

It is worth noting here that a double ampersand `&&` has a different meaning - this means only run the next command if the previous command has completed successfully.
Test the different behaviour of these two commands:

```
$ ls / && echo "success"
$ ls /null && echo "success"
```

In a similar manner, a double pipe, `||` will instruct the shell to only run the next command if the previous command did not complete successfully. Try adjusting the commands, replacing the double ampersand with a double pipe:

```
$ ls / || echo "fail"
$ ls /null || echo "fail"
```

9 Compressed Files

There are many different methods of compressing files, using slightly different programs in different formats, some of the most common being `zip`, `gzip`, `tar`, and `bzip2`.

For any compressed format, you will generally want to list the file contents, decompress a file, or create a compressed file.

This is a very quick guide to working with these different file formats:

9.1 zip

List the contents of a *zip* file with `unzip -l`:

```
$ unzip -l files.zip
```

Extract the file with `unzip`:

```
$ unzip files.zip
```

Use the `zip` command to create the file *files.zip* with the contents of the directory *files*:

```
$ zip -r files.zip files/
```

9.2 gzip

gzip files usually have the suffix *.gz*, and a *.gz* file will only contain a single file.

When a *gzip* file is created the original no longer exists in an uncompressed format, and when a *gzip* file is extracted, the compressed version will no longer exist.

To *gzip* a file, use `gzip`:

```
$ gzip file.pdf
$ ls
file.pdf.gz
```

To extract a file, use `gunzip`:

```
$ gunzip file.pdf.gz
$ ls
file.pdf
```

9.3 bzip2

`bzip2` works very much like `gzip`, but can usually compress files more efficiently (but can also take longer):

```
$ bzip2 file.pdf
$ ls
file.pdf.bz2
$ bunzip2 file.pdf.bz2
$ ls
file.pdf
```

9.4 xz

`xz` works very much like `gzip` and `bzip2`, but can often compress files even more efficiently (but can also take even longer):

```
$ xz file.pdf
$ ls
file.pdf.xz
$ unxz file.pdf.xz
$ ls
file.pdf
```

9.5 tar

tar files usually have a *.tar* extension, and the `tar` command is often combined with `gzip` or `bzip2` to create efficiently compressed files.

Many software projects will distribute software in *tar* files which have also been compressed via `gzip`, `bzip2` or `xz`, which usually have the extensions *.tar.gz*, *tar.bz2*, *tar.xz*.

To list the contents of a *tar* file, the `t` option is used:

```
$ tar tf files.tar
```

The `f` option specifies that the next option will be the name of the *tar* file.

To extract a *tar* file, use `x`:

```
$ tar xf files.tar
```

To create a *tar* file, *files.tar*, with the contents of the directory *files*:

```
$ tar cf files.tar files/
```

When `gzip` is used in conjunction with `tar`, a `z` is added to the above commands:

```
$ tar tzf files.tar.gz #- list contents
$ tar xzf files.tar.gz #- extract files
$ tar czf files.tar.gz files/ #- create files.tar.gz
```

When combined with `bzip2`, a `j` is used:

```
$ tar tjf files.tar.bz2 #- list contents
$ tar xjf files.tar.bz2 #- extract files
$ tar cjf files.tar.bz2 files/ #- create files.tar.bz2
```

And when combined with `xz`, a `J` is used:

```
$ tar tJf files.tar.xz #- list contents
$ tar xJf files.tar.xz #- extract files
$ tar cJf files.tar.xz files/ #- create files.tar.xz
```

10 Software

There are many useful software programs installed on the Linux systems as part of the operating system - all the standard command line utilities, text editors, office software, and so on.

On the School Linux systems, many of the useful bits of software are installed on network file systems, and accessible using the `module` command.

10.1 modules

The `module` command is provided by the *Environment Modules* software, which is widely used all over the world, particularly on HPC systems.
Available software can be viewed by typing:

```
$ module avail
```

The output from this command will look slightly different on some systems.

To view more information about a particular bit of software, you can use `module help`:

```
$ module help envi
```

Would display some information about the envi software.

To use one of these bits of software, `module load` will set the required system variables:

```
$ module load envi
```

and then the software can then be used:

```
$ envi
```

Another example, if you wished to use the `qgis` program:

```
$ module load qgis
$ qgis
```

Some applications may have dependencies, for example, if you wished to use the additional Python libraries available in the `python-libs` module, you would also need to load a suitable version of Python:

```
$ module load python-libs
python-libs/1.6 depends on one of the module(s) 'python python2 python3 canopy canopy2
$ module load python3 python-libs
```

Running the `module load` command will make the software available in the current shell. If you wished to always have an application available upon login, the `module load` commands can be added to your login files.

If you would like an additional bit of software to be available, please contact IT support.

10.2 Popular Applications

10.2.1 Matlab

The *Matlab* software is set up for all users by default, and can be accessed from a terminal window, by simply running:

```
$ matlab
```

It is also possible to run Matlab software without the graphical interface:

```
$ matlab -nosplash -nodesktop

                              < M A T L A B (R) >
                    Copyright 1984-2013 The MathWorks, Inc.
                      R2013a (8.1.0.604) 64-bit (glnxa64)
                               February 15, 2013


To get started, type one of these: helpwin, helpdesk, or demo.
For product information, visit www.mathworks.com.

>>
```

If Matlab is unavailable for any reason, the required variables can be set up with:

```
$ module load matlab
```

You can see which versions of Matlab are available with:

```
$module av matlab
```

`module av` is equivalent to running `module avail matlab`. To load a specific version of a module, you can includ ethe version information in the `module load` command:

```
$ module load matlab/2016a
$ which matlab
/apps/applications/matlab/2016a/1/default/bin/matlab
```

10.2.2 IDL

The *IDL* software is also set up by default, and can be accessed with:

```
$ idl
IDL Version 8.4 (linux x86_64 m64). ...
Installation number: 4456405397.
Licensed for use by: University of Leeds

IDL>
```

By default IDL opens up in command line mode, but does also have a graphical interface, which can be launched with:

```
$ idlde
```

Similar to the Matlab variable `MATLABPATH`, the `IDL_PATH` variable tells the software where to look for programs. It is worth noting, that this should always include the location of the default programs, `<IDL_DEFAULT>`. If you had some IDL programs in the directory `${HOME}/idl`, these could be added with:

```
expor IDL_PATH="<IDL_DEFAULT>:${HOME}/idl"
```

If a value in the `IDL_PATH` variable begins with a +, then any sub folders will also be searched for IDL programs.

```
export IDL_PATH="<IDL_DEFAULT>:+${HOME}/idl"
```

If IDL is unavailable for any reason, the required variables can be set up with:

```
$ module load idl
```

10.2.3 GMT

On systems within the School, the *GMT* software is installed as part of the system, and should be available without having to change any settings, but the sytem version may be either version 4 or version 5, depending on the version of the Operating System installed.

If you wish to specifically use GMT version 4, you can use:

```
$ module load gmt4
```

For GMT version 5:

```
$ module load gmt5
```

Version information about the operating system can be viewed with the `lsb_release` command:

```
$ lsb_release -d
Description:    CentOS release 6.9 (Final)
$ lsb_release -a
LSB Version:    :base-4.0-amd64:base-4.0-noarch:core-4.0-amd64: ...
Distributor ID: CentOS
Description:    CentOS release 6.9 (Final)
Release:        6.9
Codename:       Final
```

10.2.4 Python

The default operating system version of `python` will generally be an older release, without access to many of the useful libraries.

The *Canopy* Python distribution (https://www.enthought.com/products/canopy/) is available on all the School Linux systems, and provides a version 2.7 or version 3 of `python`, and also provides many useful libraries (*matplotlib*, *scipy*, *numpy*..) as well as a useful graphical interface:

```
$ module load python
```

or:

```
$ module load python3
```

Will setup the required variables, and when you then run:

```
$ python
```

This will launch the Canopy version of `python`, with all the additional libraries which this provides.
There are even more additional `python` libraries available by running:

```
$ module load python-libs
```

Which provides, among others, the SciTools *Iris* library, and *obspy*, and the `spyder` graphical development environment. Full details can be viewed with:

```
$ module help python-libs
```

To launch the `spyder` development environment:

```
$ module load python python-libs
$ spyder
```

or:

```
$ module load python3 python-libs
$ spyder
```

10.2.5 R

The `R` program is available by default for all users, and includes a number of additional useful libraries. You can install additional libraries for `R` using the `install.packages()` command, which will install files within the `~/R` directory.

The popular graphical develop environment `rstudio` is available on the Linux systems, and can be launched by simply running:

```
$ rstudio
```

On some older systems, it may be necessary to load `rstudio` with the `module` command:

```
$ module load rstudio
$ rstudio
```

10.2.6 LaTeX

LaTeX commands are available on all of the Linux systems by default. If some packages you wish to use are not available in the default version, there is also a more recent of the *texlive* distribution available with:

```
$ module load texlive
```

There are also various graphical LaTeX editors available. `kile` is provided by the operating system, and can be found in the program menus, or launched with:

```
$ kile
```

The popular `texmaker` and `texworks` editors are also available, and can be launched with the commands `texmaker` and `texworks` respectively.

On some older systems, it may be necessary to first load these prograges using the `module load` command. for texmaker:

```
$ module load texmaker
$ texmaker
```

And for `texworks`:

```
$ module load texworks
$ teworks
```

If you will be using LaTeX, you may be interested in investigating the online LaTeX editor / creater, Overleaf (https://www.overleaf.com/).

There are many excellent LaTeX resources available on the internet, such as:

Wikibooks	https://en.wikibooks.org/wiki/LaTeX/
The Not So Short Introduction to LaTeX	https://tobi.oetiker.ch/lshort/lshort.pdf

10.2.7 Compilers

As well as the *gnu* compilers (`gcc`, `g++`, `gfortran`), the University has licenses available for the Intel (`icc`, `icpc`, `ifort`) and PGI (`pgcc`, `pgCC`, `pgf90` ...) compilers.

These can be accessed with:

```
$ module load pgi
$ module load intel
```

The method for compiling programs can vary depending on how simple or complex the program is, or who created it. There are some simple examples within the downloaded examples, which can be used for testing.

Hello World C program, compiled with `gcc` (`-o` specifies the output file name):

```
$ gcc -o hello helloWorld.c
$ ./hello
Hello World
```

Fortran square root program:

```
$ gfortran -o square squareRoot.f90
$ ./square
 Enter a number, zero to stop:
9
    3.25000000
    3.00961542
    3.00001526
    3.00000000
    3.00000000
 The square root of    9.000000      is approximately    3.000000
```

10.3 File Viewers

Many different file viewers are available for many different types of files. The most common requirements are to view *pdf*, *postscript* and image files.
On the School Linux systems, you can generally view any of these with the `okular` program:

```
$ okular document.pdf
```

`gv` and `gsview` are popular, lightweight choices for viewing postscript files:

```
$ gv file.ps
$ gsview file.ps
```

Other choices for viewing pdf files include `evince` and `xpdf`:

```
$ evince file.pdf
$ xpdf file.pdf
```

`acroread` (Adobe pdf reader) is also available when required:

```
$ acroread document.pdf
```

10.3.1 Converting Image Files

The *ImageMagick* software provides many tools for manipulating images, such as the `convert` command:

```
$ convert image00.jpg image00.png
```

This will convert the *.jpg* file to a *.png* file, keeping the original file.

This works for many different image file types, and many more options are available:

http://www.imagemagick.org/script/convert.php

10.3.2 Firefox

The default browser on the Linux systems is Mozilla Firefox. There are a couple of points worth noting about using Firefox on the School Linux systems.

Firefox stores its profiles information within the home directory (`~/.mozilla/firefox`), and when a browser session is opened, lock files are created within the profile directory. As the home directories are common between machines, it is not possible to have the same Firefox profile open on two different systems. It is, however, possible to have multiple Firefox profiles, so that Firefox can be launched on multiple machines.

To launch the Firefox profiles manager, run:

```
$ firefox -ProfileManager
```

If Firefox refuses to open, complaining that there is already a session running, it is possible that the program may not have been closed cleanly, and may have left some lock files in the profile directory, which are stopping the software from launching. If you are certain that you do not have any `firefox` processes running, there is a script on the systems, which can be used to remove the lock files:

```
$ firefox-remove-lockfiles
```

11 Connecting to Remote Systems

11.1 SSH

Linux is a multi user operating system - many different users can all log into a system at the same time.

You may have access to a remote machine in the School for running jobs which require more resources than are available on your desktop system, you may need to log in to a high performance cluster system within the University or elsewhere, or you may want to connect to a Linux system from a different operating system.

To do this, you can use `ssh`.

In the simplest form, you will be able to run `ssh computername`. Try connecting to the general use Linux systems, *foe-linux*:

```
$ ssh foe-linux
```

You may be prompted for your password, and then be logged in to the remote system.

Within the School Linux systems you will still see the same files within your home directory.

What happens if you try to run a graphical program, such as the post script file viewer `gv`?

You can logout by running:

```
$ exit
```

If you want to connect to the remote system, and run graphical programs, you can add a `-X` option - make sure it is a capital `X`:

```
$ ssh -X foe-linux
```

Once connected, try running:

```
$ gv
```

If you are connecting to a remote system on which your username is different, you will have to specify this when you connect.

This can be done by specifying `username@computername`:

```
$ ssh bob@computer000
```

11.2 Connecting via SSH from Windows

If you are using a Microsoft Windows system, the popular *putty* program (http://www.chiark.greenend.org.uk/~sgtatham/putty/) can be used to make the SSH connection, and the program *xming* (http://sourceforge.net/projects/xming/) can be used to display graphical output these are installed on School Windows desktops by default.

There are some more notes on this method here in the *connectingtolinux.pdf* file, within the example files downloaded earlier.

Another popular choice for connecting from Windows is the program *MobaXterm* which provides a SSH client, X server, and various other utilities. More information is avialable at their web site http://mobaxterm.mobatek.net/.

11.3 Copying Files To And From Remote Systems

If you want to copy files to or from a remote system, the `scp` command can be used.

If your username was *earabc*, and you had a file on a remote system named *computer00*, located at `/data/file00`, which you wanted to copy to your local system, in to the directory `/myfiles/`, the command would be:

```
$ scp earabc@computer00:/date/file00 /myfiles/
```

The source comes first, and then the destination.

On the remote system, the location is separated from the computer name by a colon.

To copy the file on the local system `/myfiles/file02`, to the location `/data/` on the remote system:

```
$ scp /myfiles/file02 earabc@computer00:/data/
```

The default location on the remote system will be your home directory. To copy the same file to your home directory on the remote system:

```
$ scp /myfiles/file02 earabc@computer00:
```

To copy directories, the `-r` (*recursive*) option can be used.

If there was a directory *researchfiles* on a remote system, within the home directory, and we wished to copy this back to the local system, to the location `/myfiles/`, we could use the command:

```
$ scp -r earabc@computer00:researchfiles /myfiles/
```

11.4 Connecting From Outside The Campus Network

SSH access to the School Linux systems is available to all users, by first connecting to the University VPN. There is more information about the VPN, and access to VPN client downloads available on the central IT web pages at http://it.leeds.ac.uk/.

Once connected to the VPN, the systems which are available for remote SSH connections are:

see-gw-01.leeds.ac.uk
foe-linux.leeds.ac.uk

see-gw-01.leeds.ac.uk is available for use by staff and postgraduate students. This system is not suitable for running processing jobs, but can be used for copying files from remote systems, and as a gateway to connect on to other machines within the faculty.

Connecting to *foe-linux.leeds.ac.uk* will log you in to one of four systems (*foe-linux-01* - *foe-linux-04*). These systems are available to all staff and students, and are suitable for running more resource intensive processing jobs.

11.4.1 Direct SSH Access

Direct access to the School Linux systems is not available by default, but can be requested by contacting IT Support. This may be required if you have needs which can't be met by VPN acces, for example if you have been using a remote HPC system, and need to copy some files back to file systems within the University.

Once direct SSH access has been set up for your username, the following machines are available to connect to:

see-gw-01.leeds.ac.uk
foe-linux-01.leeds.ac.uk

As noted above, *see-gw-01.leeds.ac.uk* is available to staff and postgraduate students, and *foe-linux-01.leeds.ac.uk* is available to all staff and students.

Further documentation on remote access to the School Linux systems, including information on how to connect to a remote Linux desktop session can be found at the web page:

https://www.environment.leeds.ac.uk/wiki/view/IT/LinuxRemote

11.5 General Use Linux Systems

The School has some general use Linux systems, available to all staff and students. These are the previously mentioned *foe-linux* systems.

At present, there are four of these machines, each of which has 20 cpu cores, 256GB of memory, and a fast, 10g/s, network connection. These can be useful if you need to run processes with higher requirements than can be provided by a desktop machine, if you don't have a Linux desktop system, if you have long running jobs, if you need to run a process with faster access to network file systems, etc.

If connecting to the generic name *foe-linux* (the *.leeds.ac.uk* suffix should not be required when connecting from a campus network connection), the connections are load balanced across all machines, and a suitable system will be selected:

```
$ ssh foe-linux
Last login: ...
$ hostname
foe-linux-03
```

It is also possible to connect to any of the machines individually if required:

```
$ ssh foe-linux-02
```

11.6 Connecting to Windows

Sometimes, it may be necessary to use a Windows system.

For this purpose, there are a number of Windows Terminal servers to which you can connect, from a Linux system, using the `rdesktop` command.

There are currently three Windows terminal servers available:

foe-rdsh-01.leeds.ac.uk
foe-rdsh-02.leeds.ac.uk
foe-rdsh-03.leeds.ac.uk

In the simplest form, the command to connect to a remote Windows system would be:

```
$ rdesktop -d ds -f foe-rdsh-01
```

`-d ds`	Specifies the name of the University Windows domain (required)
`-f`	Specifies that we would like to make a full screen connection
`foe-rdsh-01`	is the name of the Windows server

Running this command will open a full screen connection to one of the Windows machines, displaying the login screen.

Enter your password and you will be logged in to a desktop session.

When using full screen mode, it is possible to switch in and out of full screen mode, by pressing:

Ctrl+Alt+Enter

When you have finished working in the Windows session, it is highly recommend to log out from the Windows 'Start Menu', rather than just closing the rdesktop Window, otherwise, you may leave processes running on the Windows machine.

If you do not wish to open a full screen Windows session, you can specify the geometry of the Window to open with the `-g` option, though recent versions of rdesktop are able to dynamically resize the Windows session:

```
$ rdesktop -d ds -g 1024x768 foe-rdsh-01
```

The dimensions can be changed as desired.

It is also possible to forward the sound from the Windows session to your desktop with the `-r` option. The colour depth can also be increased with the `-a` option:

```
$ rdesktop -d ds -f -r sound -a 32 foe-rdsh-01
```

It may be useful to set up an alias for connecting to windows:

```
$ alias 'win1'='rdesktop -d ds -f -r sound -a 32 foe-rdsh-01'
$ alias 'win2'='rdesktop -d ds -f -r sound -a 32 foe-rdsh-02'
$ alias 'win3'='rdesktop -d ds -f -r sound -a 32 foe-rdsh-03'
```

To find out more about the other options which are available, take a look at the manual page:

```
$ man rdesktop
```

12 Printing

The standard method of printing within the School is to use the *myprint* pull printing system. Print jobs get sent to a queue, and nothing is printed out until you visit the one of the Konica multi function printers, log in, and select the jobs to be printed out.

There are two queues available from the Linux systems:

MyPrint
Staff

The *MyPrint* queue is available to all staff and students, the *Staff* queue is available to staff and postgraduate students.

From graphical applications, print options (single sided/double sided, colour/mono, etc.) can usually be found in the print options for that particular bit of software. It is important to make sure the paper size is set to A4, or A3 when required, otherwise the job is unlikely to print successfully.

Common print options can usually be found under these headings:

Binding Position	For duplex printing. Default is *Left Bind*
Print Type	Duplex option. Default is *2-sided*
Select Color	Switch colour on (*Auto Colour*) or off. Default is *Gray Scale*
Staple	Staple together the output. Default is *Off*. Stapling is not available on all printers

12.1 Printing From The Command Line

There are various commands available for printing from the command line. `lpstat` can be used to show available printers:

```
  $ lpstat -a
```

`lp` can be used to print a document:

```
  $ lp -d Staff document.pdf
```

This would print the file *document.pdf* to the print queue *Staff*, using the default print options for the queue (2 sided, mono).

Additional options can also be specified from the command line:

```
$ lp -o number-up=2 -o ColorModel=Auto -o Duplex=None -o copies=3 file.pdf
```

`-o number-up=2`	Sets 2 pages per sheet
`-o ColorModel=Auto`	Sets colour printing
`-o Duplex=None`	Sets single sided printing
`-o copies=3`	Selects 3 copies to be printed

13 HPC Systems

There are currently three HPC (High Performance Computing) cluster systems in the University. From oldest to newest, these are *arc2*, *arc3*, and *arc4*.

13.1 Requesting Access

Requesting access to the *arc3* and *arc4* HPC systems can be done by filling out a simple online form:

http://arc.leeds.ac.uk/information/getting-an-account/

The documentation for the Leeds HPC systems can be found at this web site:

http://arc.leeds.ac.uk/

The ARC team coordinate various useful training sessions throughout the year. Details can be found through this web page:

http://arc.leeds.ac.uk/training/

13.2 JASMIN

JASMIN is a NERC funded computing facility, providing various services for processing and data analysis, including a large archive of popular data sets.

To find out more about JASMIN, the services they provide and how to gain access, visit http://jasmin.ac.uk/.

14 Anaconda

The Anaconda Distribution provides a popular method of installing Python, R and associated libraries and packages.

The Anaconda Distribution (https://www.anaconda.com/) is available for Linux, Windows and Apple systems, and will install all files within your home directory by default, and as such, installation does not require any elevated / administrative permissions.

14.1 Installation

There are various different Anaconda installers available, with the default installation containing Python version 2 or version 3 and many useful packages (https://www.anaconda.com/distribution/). However, it is worth noting that the installation can require several Gigabytes of disk space.

As an alternative the Miniconda installer (https://docs.conda.io/en/latest/miniconda.html) provides a minimal base installation which can be used to create environments and install required packages.

For Linux systems, the Miniconda 64 bit version 3 installer is recommended, which can be used to create Python version 2 and version 3 as well as R environments:

https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh

As mentioned, Anaconda installations can get quite large, and will install in the home directory by default, so it is worth checking your quota (`quota -s`) or selecting a suitable location (for example, if you have access to some large volume disk space) before installation.

The Miniconda installer can be downloaded and made executable with the following commands:

```
$ wget \
  https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ chmod +x Miniconda3-latest-Linux-x86_64.sh
```

The installer can then be run:

```
$ ./Miniconda3-latest-Linux-x86_64.sh
```

The installation process will ask for the terms of the license to be accepted, and then ask for an installation directory, defaulting to a directory named `miniconda3` in the home directory. If you wish to select an alternative location, enter the full path to the required location. If installing in your home directory, a sensible path to choose might be `/home/username/conda`, replacing *username* with your own username.

After the installer has finished extracting the files in to the selected location, it will ask whether you wish to run *conda init* to initialise the installation. Selecting yes will update the `~/.bashrc` so that the Anaconda programs will be available in future sessions.

Once the installation process has been completed, the installer is no longer required, and can be removed:

```
$ rm Miniconda3-latest-Linux-x86_64.sh
```

14.2 Initial Configuration

Once the installation is complete, if the `~/.bashrc` file was updated successfully, the Miniconda installation will be active when you either run `source ~/.bashrc`, or open a new shell. You should see that the prompt is now prefixed with `(base)`, to indicate that the base environment in the Miniconda installation is active:

```
(base) [see1-234:earabc:1]$
```

At this point the `python` command will now default the version of Python installed with Miniconda, and the `conda` command can be used to install packages in the *base* environment:

```
(base) [see1-234:earabc:3]$ which python
/home/earabc/conda/bin/python
(base) [see1-234:earabc:3]$ python -V
Python 3.7.4
(base) [see1-234:earabc:4]$ conda install numpy
```

However, an alternative method of using may be more advisable. Anaconda environments can include a large number of different programs and libraries which can cause conflicts with system versions of the same files, so it can be better to create environments for specific purposes, activating and deactivating the environments as required.
To stop the *base* environment being automatically activated when a shell is opened, the following command can be run:

```
$ conda config --set auto_activate_base false
```

It may also be worth adding the `conda-forge` channel to configuration. The `conda-forge` channel is an additional repository from which packages can be installed, and includes many popular scientific packages, such as iris. The following commands will add the `conda-forge` channel, and also allow packages from this channel to take precedence over packages in the base channel:

```
$ conda config --add channels conda-forge
$ conda config --set channel_priority strict
```

These changes update the file `~/.condarc`, and after they have been run, and a new shell is opened, the *base* channel will no longer be activated by default, but the `conda` command will be available to allow creating new environments:

```
$ which python
/usr/bin/python
$ which conda
/home/earabc/conda/condabin/conda
$ cat .condarc
auto_activate_base: false
channels:
  - conda-forge
  - defaults
channel_priority: strict
```

14.3 Creating Environments

The command `conda create` can be used to create environments containing the required packages. For example, to create an environment containing Python version 3 and the Spyder graphical development environment:

```
$ conda create -n py3_spyder python=3 spyder
```

Environments are created within the `envs` directory of the installation folder, and the environments can be activated and deactivated with the `conda activate` and `conda deactivate` commands:

```
[see1-234:earabc:12]$ ls conda/envs/
py3_spyder
[see1-234:earabc:12]$ conda activate py3_spyder
(py3_spyder) [see1-234:earabc:13]$ which python
/home/earabc/conda/envs/py3_spyder/bin/python
(py3_spyder) [see1-234:earabc:14]$ python -V
Python 3.7.3
```

```
(py3_spyder) [see1-234:earabc:15]$ which spyder
/home/earabc/conda/envs/py3_spyder/bin/spyder
(py3_spyder) [see1-234:earabc:16]$ conda deactivate
[see1-234:earabc:18]$
```

To create a Python version 2 environment containing the `numpy` and `scipy` packages:

```
$ conda create -n py2_scipy python=2 numpy scipy
```

The available environments can be listed with `conda env list`:

```
$ conda env list
# conda environments:
#
base                  *  /home/earabc/conda
py2_scipy                /home/earabc/conda/envs/py2_spyder
py3_spyder               /home/earabc/conda/envs/py3_spyder
```

14.4 Installing Packages

Once an environment has been activated, additional packages can be installed within that environment with the `conda install` command:

```
(py3_spyder) [see1-234:earabc:24]$ conda install iris
```

The `conda search` command can be used to search for available packages:

```
(py3_spyder) [see1-234:earabc:25]$ conda search 'obsp*'
Loading channels: done
# Name                       Version           Build  Channel
obspy                          1.0.2         py27_0  conda-forge
obspy                          1.0.2         py27_1  conda-forge
...
```

To install a specific version of a package, the version can be specified with ==:

```
(py3_spyder) [see1-234:earabc:26]$ conda install obspy==1.0.3
```

It is also possible to install packages using the `pip` command, if a package is available in the PyPi repositories, but not available in the Anaconda channels (`pip install packagename`).

The packages which are currently installed in an environment, their version information and installation source can be viewed by running:

```
(py3_spyder) [see1-234:earabc:27]$ conda list
```

14.5 Updating Packages

The `conda update` command can be used to update packages. For example, to update `spyder`:

```
(py3_spyder) [see1-234:earabc:28]$ conda update spyder
```

To update all packages in an environment:

```
(py3_spyder) [see1-234:earabc:29]$ conda update --all
```

If an environment is not updated for some time, changes to the environment (such as installing a new package) may cause version conflicts, so it may be wise to either create a new environment, or update all packages in the environment when making changes.

14.6 Installing R

As well as Python environments, the `conda create` command can also be used to create environments for running the `R` program.

To create an environment containing `R` and `rstudio`:

```
  $ conda create -n R R rstudio
```

This will create an environment named `R`, which can be activated with `conda activate R`, and once the environment is active the `R` and `rstudio` programs will be available.