Introduction to Linux

David Robertson

david.m.robertson@ncl.ac.uk

July 1, 2016

Abstract

'Linux' is a freely available operating system which runs on a variety of computers. It's relatively unknown in the world of desktop computing, despite running on a vast number of servers, embedded systems and mobile phones (Android). This course is a hands-on introduction to Linux which assumes no prior experience; we'll show why it's a good environment for technical computing.

We'll start by explaining what's different and what's the same to Windows and macOS. The bulk of this course is on getting familiar with the 'terminal', and learning how to use it to manage programs efficiently.

Contents

0 What is Linux, anyway?

Linux is an computer operating system, widely used in academia and technical industry. First released in 1991, it has its roots in the 1970s UNIX operating system. Because Linux is open-source, absolutely anyone is free to download, copy and modify its source code. This allows programmers and researchers to test, improve and even attack its code; the result is an operating system that's been well-tested and critically analysed over a number of years. By comparison, the source code for Windows and macOS is closed: only Apple and Microsoft get to know what goes on behind the curtain.

Strictly speaking, 'Linux' refers only to the Linux kernel, the glue between the computer's software and hardware. The kernel is responsible for

Hardware management Allowing applications to use the computer's hardware, e.g. hard drives or solid state drives; keyboard and mouse input; USB memory sticks; outputting to the screen; communicating over the internet.

Process management The kernel distributes the processor's time between all the different applications currently running. It also has to facilitate a means for processes to communicate with each other.

Memory management The kernel allocates memory (RAM) to processes as required, and also ensures that processes cannot read or modify memory that they shouldn't be able to. It also has to handle the situation where not enough memory is available to meet the running programs' demands!

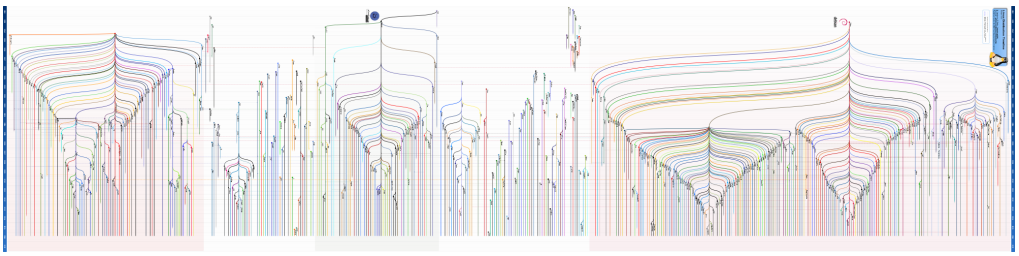

Figure 1: A timeline of Linux distributions. There's a vast number of them! Image: `https://commons.wikimedia.org/w/index.php?curid=2556373`

This is all low-level stuff—very close to the bare metal!

A lot of what gets referred to as Linux is really a set of tools which run *on top of* the kernel. Many of these programs were written in the 1980s by the GNU Project, and so many people prefer to refer to the entire system as 'GNU/Linux'. Various organisations distribute Linux together with curated collections of software—resulting in what's called a *Linux distribution*.

For today's session we'll be using the School of Computer Science's media room, whose (non-Mac) machines are equipped with the Fedora Linux distribution; but there are many others available. For instance, Maths and Stats uses the Ubuntu distribution on their desktops and laptops.

If you want to try Linux on your own machine, both of these distributions are good places to get started. These—along with many others—offer a 'Live USB' version, so you can try before you buy into Linux. It's also possible to install Linux directly, even alongside an existing Windows installation. If you choose to do this, I recommend you find a Linux guru and seek their advice!

Today Linux is hidden in plain sight—it's on smartphones, satnavs, TVs, wireless routers, web servers, supercomputers,... the list goes on and on. Microsoft has even announced plans to run a version of Ubuntu on top of Windows 10!

0.1 What's in it for me?

The main focus of this tutorial is on the advantages the Linux environment has for technical computing. In particular, the *command-line interface* provided by `bash` is extremely powerful, and lends itself naturally to automating tasks. To many this way of interacting with a computer seems foreign and unintuitive at first; but there are advantages. For instance, the `ssh` command allows us to interact with a remote computer via a terminal. While graphical desktops are sometimes available over a network connection, they're often sluggish as every input and response has to be transmitted over a wire. Some machines—such as `topsy`, the SAgE high performance computing cluster—don't offer this: command-line is the only way.

Another strong point of Linux is its suite of automation tools. Have you ever written a complicated program which needs frequent testing to make sure you haven't broken anything? Shell scripts and `make` are your friend, allowing you to process as many instructions as you like in one command. For another example, imagine you need to do the same repetitive task over and over again to multiple files. Maybe they need to be renamed to a specific name format; perhaps you need to compile 200 LaTeX documents and send them all to the printer.

The final advantage of Linux that we'll mention is its reliability. In Maths and Stats we've had staff and students run vast number-crunching simulations that took many months to complete. Thanks to the Linux server infrastructure these ran successfully

without interruption or any errors. (Though if you're running a long job, you should really save data periodically to file so that computations can be resumed if the program suddenly encounters a problem!)

All of the stuff we'll describe could be done on Windows and macOS via their own shells and and tools; in particular macOS shares a lot of the same command-line tools. Linux isn't the only way; it's just a tool (albeit one you don't have to pay for!) You should pick the tool you find easiest to use and which allows you to be the most productive.

0.2 About this document

By all means read through at your own pace. When software, e.g. LINUX is introduced it'll be typeset in small caps. Code snippets are typeset `like this`, and if I want to point something out in particular it'll **`also be bold`**. When a *term* is introduced, or when I'm explaining an abbreviation I'll use *italics*.

```
Longer code snippets are typeset as their own paragraph;
to set them apart they're usually indented slightly.
They typically span multiple lines.
```

Exercise. To test your understanding and gain some hands-on experience, I've included a number of short exercises throughout. They're not meant to be stressful or challenging—just a silly hoop-jumping task to try and get commands and information to stick in your brain. Feel free to ask if something isn't clear: we're not trying to grill you!

Top tip! Helpful tips are included throughout the document. You don't need them to understand everything, but they will make your life a lot easier!

Maths & Stats only: Some information is specific to the School of Maths & Stats. It may apply to other Linux systems as well, if they've chosen to adopt the same conventions as Maths' computing officer.

1 Using a graphical interface

Let's start by getting used to the graphical desktop in Linux. For many day-to-day tasks, you can use a graphical interface just like you're already used to. The biggest difference is that the programs you're familiar with probably won't be available. Some software is written for a specific operating system: e.g. MICROSOFT OFFICE, Apple's PAGES, NUMBERS and KEYNOTE. There are Linux replacements for many of these—for instance LIBRE OFFICE for the two office suites above. Quite often the replacements are *cross-platform*: designed to run on as many operating systems as possible. Libre Office is such an example, and can be installed free of charge on your own computer. Probably the best-known examples of cross-platform software are Google's CHROME/CHROMIUM and MOZILLA FIREFOX.

Top tip! On both Ubuntu and Fedora, the most recent text selection that you've made with your mouse can be pasted by pressing the middle-mouse button.

Top tip! If you find that your screen suddenly becomes entirely black with white text asking for a login, you have probably accidentally switched to a 'tty'. To get back to the normal graphical interface, press `Control-Alt-F1` on Fedora and `Control-Alt-F7` on Ubuntu.

There are a multitude of other desktop environments available; we'll only discuss the two you're most likely to encounter in Maths and Stats.

1.1 Fedora: GNOME shell

At the time of writing, the Computer Science machines are running Fedora 21. You should be able to login using your usual Newcastle username and password. Once successful, an information message about GNOME shell will pop up; you can safely ignore this. You should be presented with a blank desktop (Figure 2), along with a 'getting started' setup guide. Again, the setup screen can be safely ignored in today's session.

The default desktop in Fedora 21 consists only of a 'panel' at the top of the screen. From left to right, this contains

- a hotspot to access the *activities overview*;
- a menu for the currently active application;
- the system's date and time; and
- notification icons: network connectivity, volume control, and the logout menu.

The *activities overview* is available in the top-left of the screen. The default shortcut for this is the Windows key (also called the *Super* or *Meta* key). It contains

- the *Dash* (left), which is similar to MacOS' Dock;
- a search bar (top);
- a summary of the current desktop's windows (centre); and
- an overview of the virtual desktops (right).

(See Figures 3 and 4.) To access the full list of available applications, press the 3-by-3 grid of dots at the bottom of the dock (Figure 5).

Multiple workspaces, or virtual desktops allow you to group together related windows without having them take up the same screen space as everything else. Drag-and-drop windows on the right-hand side of the activities overview to move them between virtual desktops. The default shortcuts for moving between these are `Control-Alt-Up` and `Control-Alt-Down`.

We'll quickly mention that Mozilla Firefox and Libre Office are available for web browsing and office software. For handling plain-text files there's gedit. Both files and folders, as well as the holding place for deleted files can be handled in the Files application.

Top tip! The same shortcuts and mouse gestures used to (half-)maximise windows in Windows 7 also work in Fedora. Press `Super-Up` or drag a window's titlebar to the top of the screen to maximise; press `Super-Left/Right` or drag a window's titlebar to the side of the screen to half-maximise.

Figure 2: The default desktop in Fedora 21.

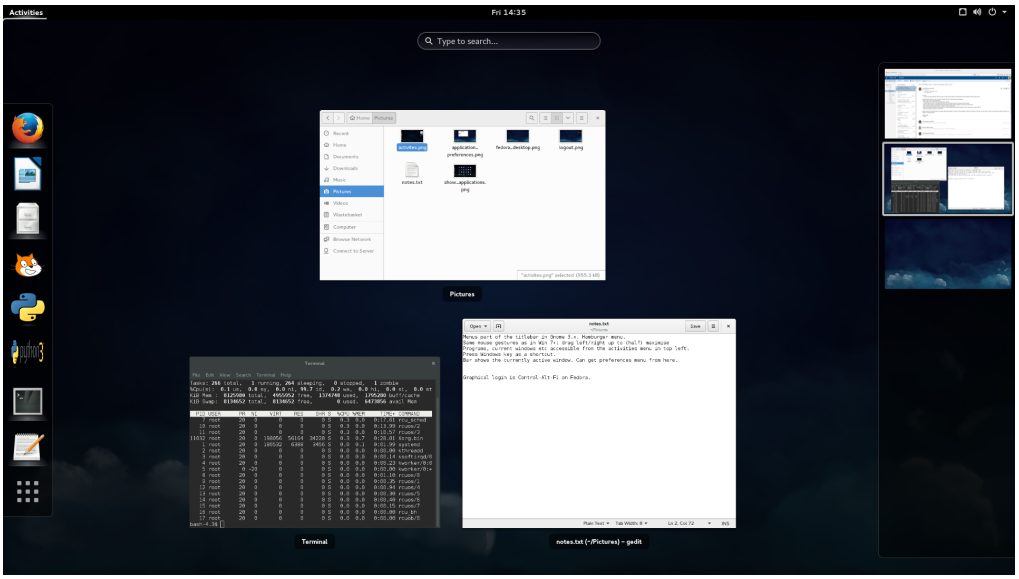

Figure 3: GNOME shell's activities overview.

Figure 4: Searching in the activities overview.

Figure 5: The full list of applications.

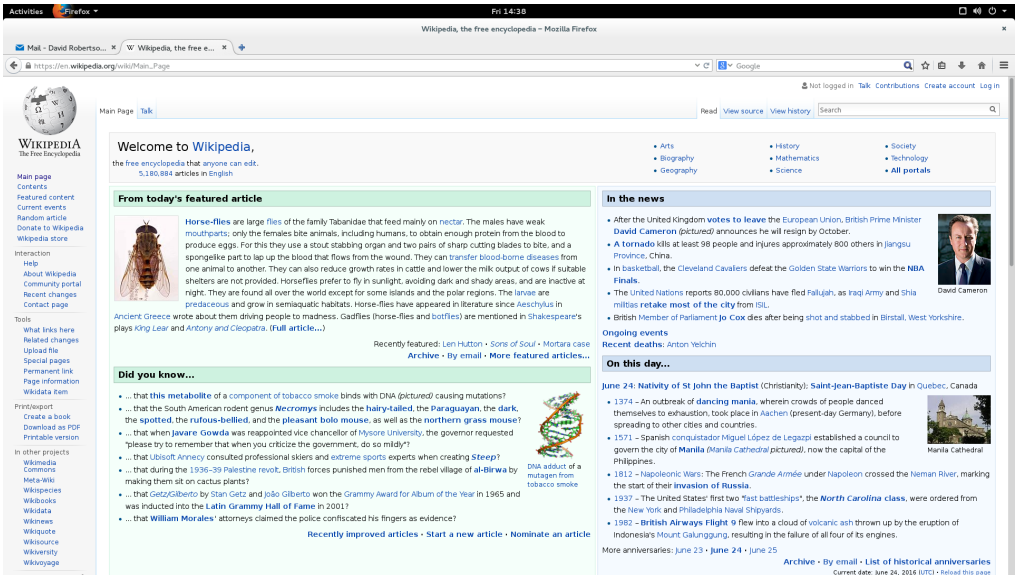

Figure 6: Firefox web browser.

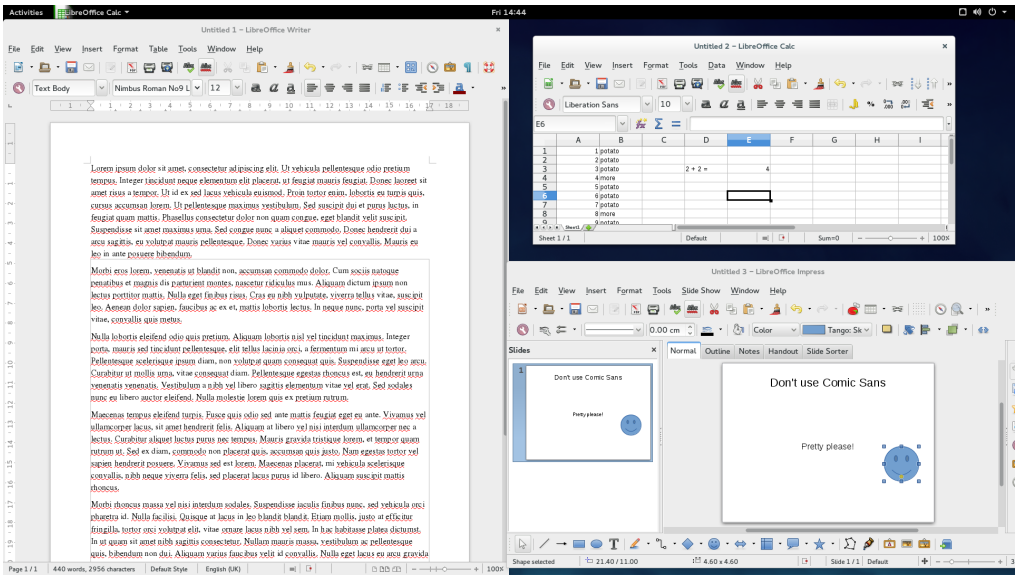

Figure 7: Libre Office's Writer, Calc and Impress.

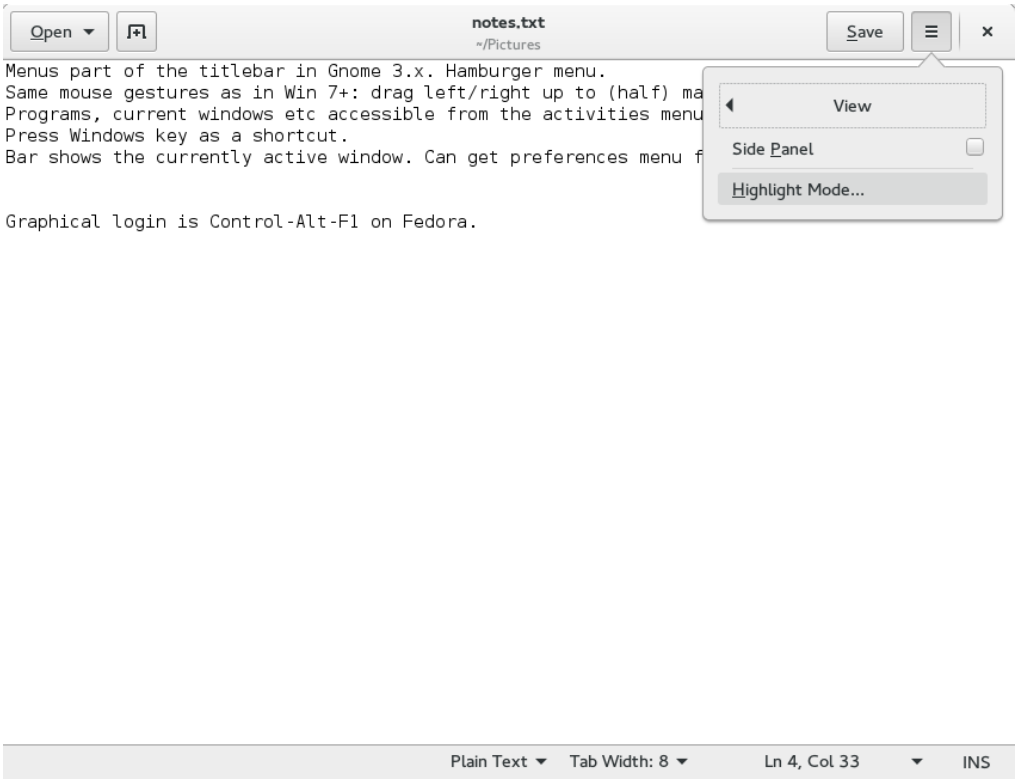

Figure 8: GNOME's text editor GEDIT.

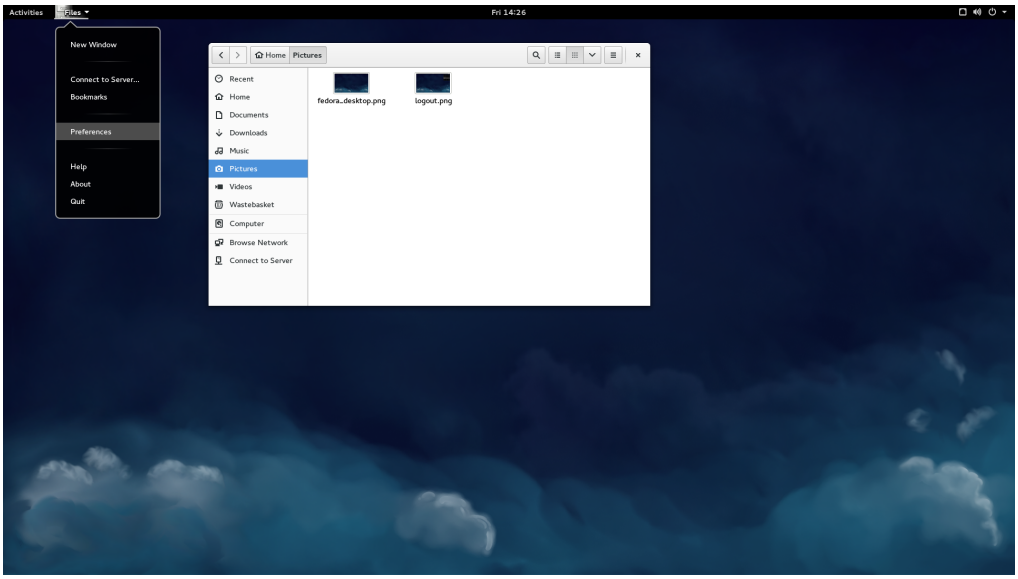

Figure 9: The file browser. On Fedora, note how the application preferences are available from the top panel.

1.2 Ubuntu: GNOME classic desktop

Maths & Stats only: At the time of writing, the Maths & Stats machines are running a mixture of Ubuntu 14.04 and 16.04. This is the default desktop environment provided on Maths & Stats Linux desktops.

You should be able to login using your usual Newcastle username and password. You should be presented with a blank desktop (Figure 10), which consists of two 'panels'. All in all, the desktop contains

- a 'show desktop' button, which minimises all windows on the current desktop (bottom left);
- a area to display the windows which open on the current desktop (bottom);
- a widget indicating which *virtual desktop* is currently being used (bottom right);
- a notification area, including network, volume, and login controls (top right); and
- the applications and places menu, along with some shortcut icons (top left).

Multiple workspaces, or virtual desktops allow you to group together related windows without having them take up the same screen space as everything else. Drag-and-drop the boxes corresponding to windows in the bottom-right corner of the screen to move them between virtual desktops. The default shortcuts for moving between these are `Control-Alt-Left` and `Control-Alt-Right`.

We'll quickly mention that Mozilla Firefox and Libre Office are available for web browsing and office software. For handling plain-text files there's gedit. Both files and folders, as well as the holding place for deleted files can be handled in the Files application. This can be opened by selecting a destination from the *places menu* (Figure 11).

Figure 10: Ubuntu 14.04's default desktop.

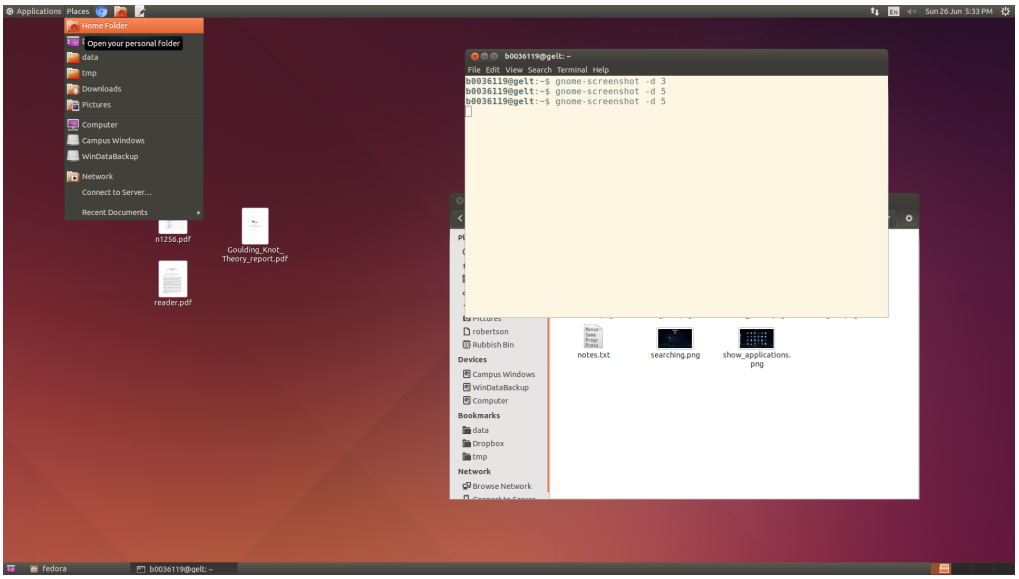

Figure 11: Ubuntu's places menu.

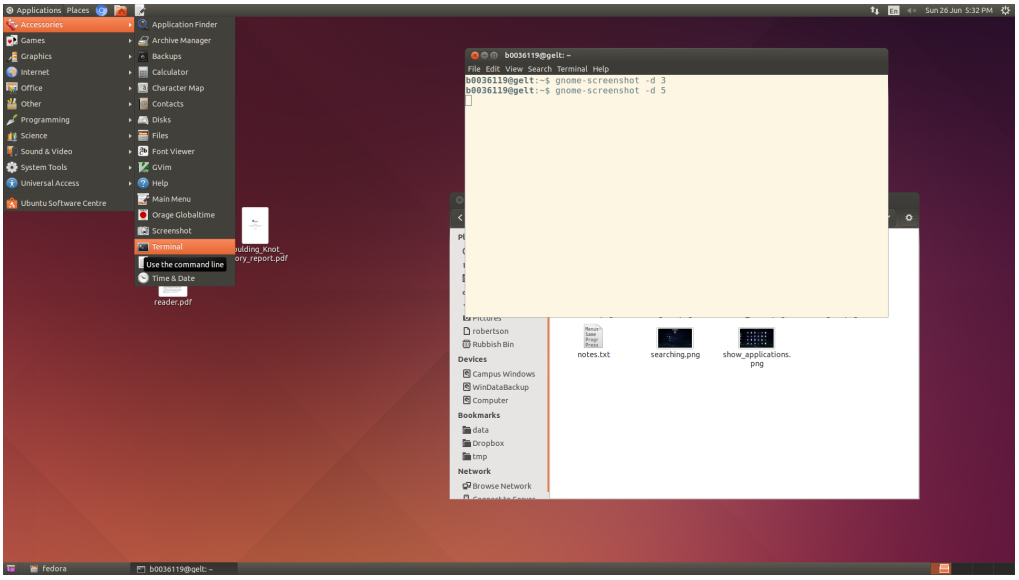

Figure 12: The applications menu, showing how to start the terminal.

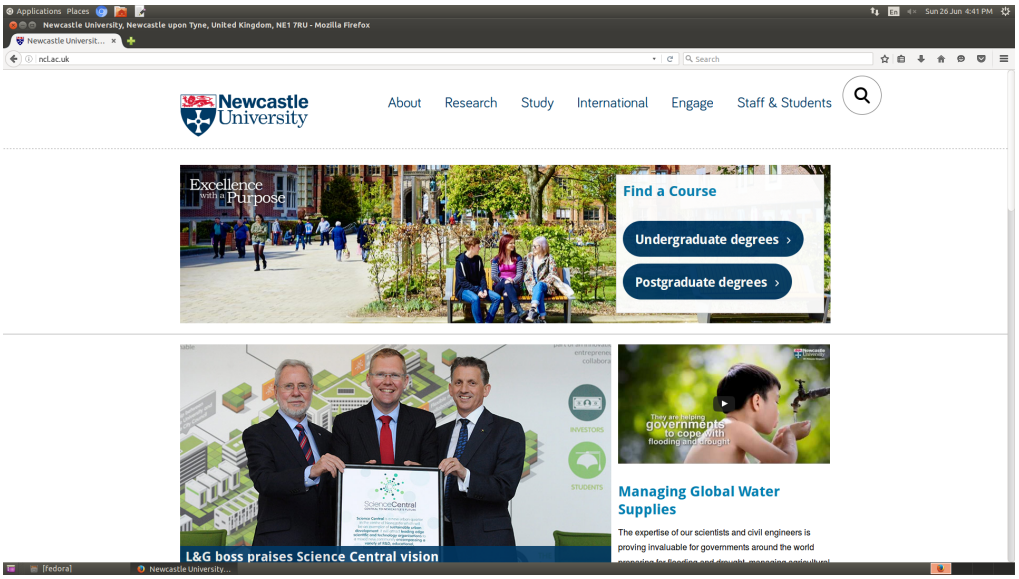

Figure 13: Firefox web browser.

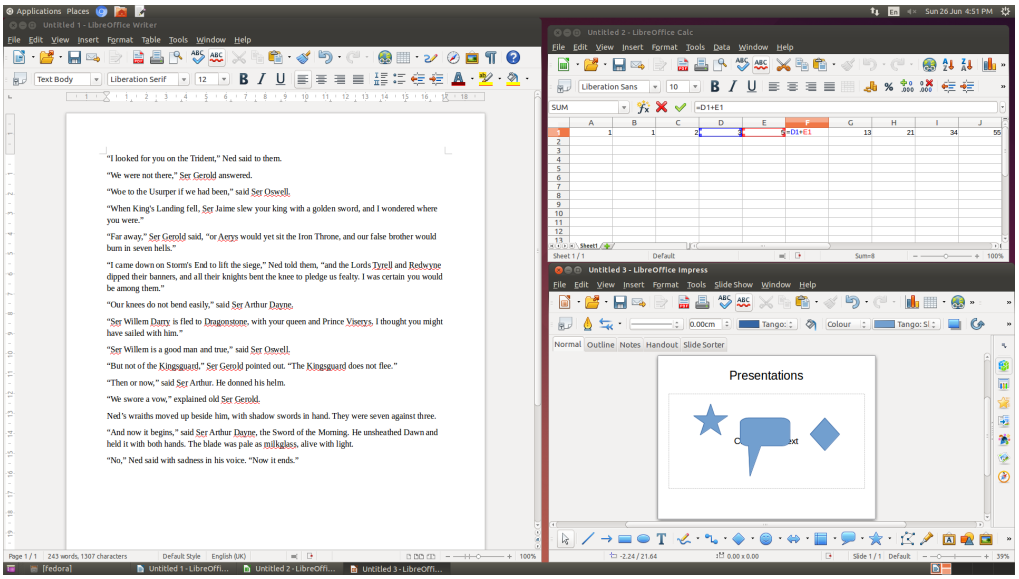

Figure 14: Libre Office's Writer, Calc and Impress.

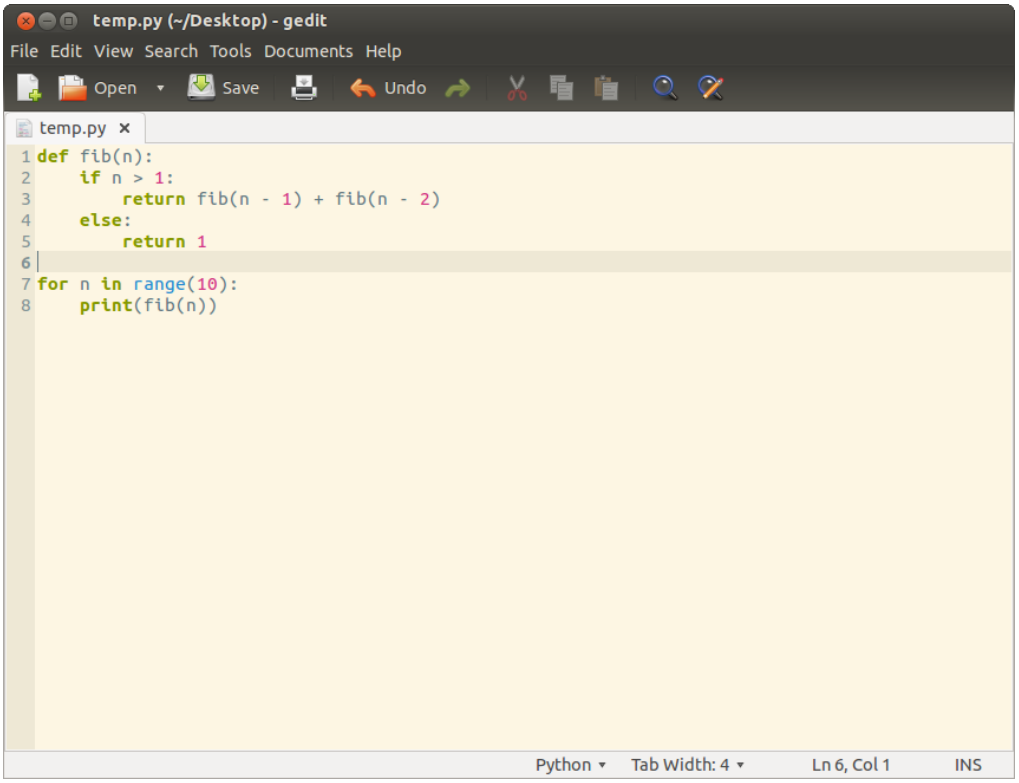

Figure 15: GNOME's text editor gedit.

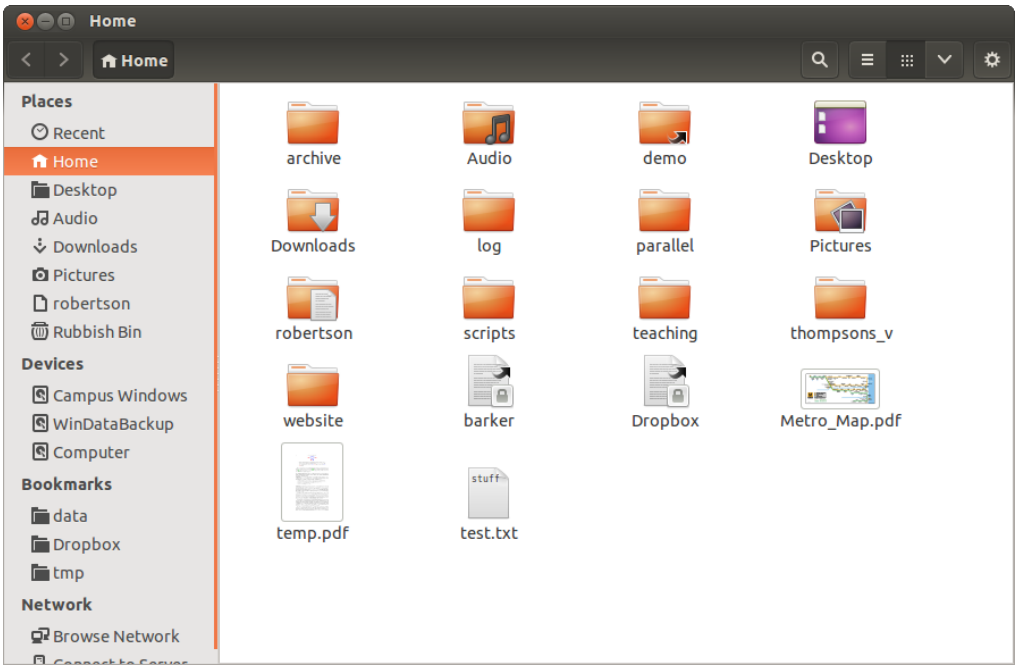

Figure 16: The file browser.

2 Getting used to the terminal

Figure 17: Left: A Teletype Model 33 ASR teleprinter, manufactured from 1965 to 1976. Centre: A DEC VT100, introduced in 1978. Right: A terminal emulator GNOME-TERMINAL. This is what 'terminal' typically refers to nowadays.
Image: `https://commons.wikimedia.org/wiki/File:ASR-33_at_CHM.agr.jpg`
Image: `https://commons.wikimedia.org/wiki/File:DEC_VT100_terminal.jpg`

The word *terminal* originally referred to a hardware device for entering data into and displaying output from a computer. The computers of the 70s were big, bulky and expensive, so they were often used as servers, time-shared between multiple users who would each use their own terminal. As time progressed; computers became quicker, smaller, and crucially cheaper, so that each user could have their own personal computer—a terminal and a server all in one.

Strictly speaking, what we call a terminal today is really a *terminal emulator*. This is a piece of software which simulates a terminal inside a window on your desktop. The terminal emulator sends your input to a a program called the *shell* (typically `bash`). In turn, the shell is responsible for running other commands and relaying their output to the terminal.

Enough history and jargon; let's launch a terminal. In Fedora, press the Windows key to open the activities view; type '`term`' to begin searching and click the terminal option that appears—see Figure 4. In Ubuntu the terminal is in the applications menu under *Accessories* → *Terminal*—see Figure 12.

Top tip! In Ubuntu you can also use the shortcut `Control-Alt-T` to launch a new terminal.

You should get something a window containing text that looks like:

```
b0036119@caldew:~$
```

I've written up these notes working on my own machine in Maths & Stats, so the terminal snippets may look a bit different to how they appear in the Computer Science cluster. The key thing to look out for is the `$` prompt, which tells us the terminal is waiting for its next command to run.

Commands are executed by typing their name followed by a list of arguments. Press the `Return` or `Enter` keys to run the command.

```
b0036119@caldew:~$ echo Hello world!
Hello world!
```

The list of arguments may be empty.

```
b0036119@caldew:~$ whoami
b0036119
b0036119@caldew:~$ date
Mon Jun 27 11:40:02 BST 2016
```

Arguments are separated or *delimited* by spaces. If you want to pass an argument which contains a space you should enclose the argument in "quotes" or escape the space by typing a backslash before it. There are other special characters like `|<>&$^?*~[]‘#` which need to be escaped with a backslash too.

```
b0036119@caldew:~$ echo a b  c   d    e
a b c d e
b0036119@caldew:~$ echo "a b  c   d    e"
a b  c   d    e
b0036119@caldew:~$ echo a\ b\  c\   d\    e
a b  c  d  e
b0036119@caldew:~$ echo a\ b\ \ c\ \ \ d\ \ \ \ e
a b  c   d    e
```

In the example above, I'm trying to print[1] the string `"a b c d e"` (including spaces). The `echo` command prints out its arguments separated by a space, so I have to tell the shell that these spaces are part of an argument, rather than an argument delimiter.

Exercise. For this minicourse I've prepared a collection of small demo files to give you something to work with. These are available online from `https://goo.gl/k7JiwX`. Download them to your home folder (see below) by running the command:

```
curl -L https://goo.gl/k7JiwX | tar -x
```

The `|` character is a 'pipe', available on a UK keyboard on the same key that contains backslash. When successful, you should have a `demo` folder inside your current directory.

For the moment just take this command on faith. By the end of this course you should hopefully understand (how to find out) what this string of gibberish means!

2.1 Navigation

One big difference between Linux and Windows is hierarchy the use to store files. (macOS is quite similar to Linux here.) Rather than have multiple drives containing folders and files, there is one *root* directory denoted by `/`. We can inspect the contents of `/` at any time by using the `ls` command.

```
b0036119@caldew:~$ ls /
backup  etc         lost+found  proc     space  var
bin     home        media       root     srv    vmlinuz
boot    initrd.img  mnt         run      sys    websites
data    lib         net         sbin     tmp    WinData
dev     lib64       opt         scratch  usr
```

`ls` stands for *list*: it shows the files and folders contained within the path you supply to it—in this case, `/`. It's an extremely useful command and we'll be seeing lots of it! In particular, notice the **`home`** directory in the output—this is where each user's personal files, folders and settings are kept. The equivalent on recent versions of Windows is `C:\Users`, on older Windows versions it's `C:\Documents and Settings`, and on Mac it's `/Users`.

[1] In computing, *print* means to output to the screen. Before computers actually had screens, the only way to output something human-readable was to physically print it on paper; hence the name.

Your own *home area* is typically a subfolder with the same name as your username within `/home`. For instance, mine is `/home/b0036119`:

```
b0036119@caldew:~$ pwd
/home/b0036119
```

Here I've used the **`pwd`** command to *print* out the current *working directory*. When you start a new terminal, the working directory is always your home folder. Within this directory you can read, create and edit files and folders as you please. No one else (barring system administrators) has these access privileges to your home area. There's also `/tmp`, a place to store temporary files which are not required to remain after the computer is rebooted.

Maths & Stats only: Home areas are actually kept on a network drive, so there is some overhead when reading and writing files. There's also the `/data` folder which can be used to keep files on your physical PC without accessing the network.

If I don't tell `ls` to list a specific folder, it defaults to printing the working directory.

```
b0036119@caldew:~$ ls
archive  Desktop    Metro_Map.pdf  robertson  temp.py~
Audio    Downloads  parallel       scripts    test.txt
barker   Dropbox    Pictures       teaching   thompsons_v
bin      log        R              temp.pdf   website
```

I could ask to see what's inside a specific directory—for instance, the `robertson` directory. However this does *not* change the current directory.

```
b0036119@caldew:~$ ls robertson
Experiments        Periodic 2-simultaneous conjugacy
FP Centralisers    Pond example
HNC                Posters
LaTeX              Presentations
Misc notebooks     Progression
Papers             Reading
Paperwork archive
b0036119@caldew:~$ pwd
/home/b0036119
```

This is an example of using a *relative path*. Paths starting with a `/` are *absolute*—they always start from the root directory `/` and work their way down through subfolders. Paths like `robertson` which don't start with a `/` are interpreted as if they were glued onto the end of the path to the working directory. In this example, `robertson` got interpreted as `/home/b0036119/robertson`.

If we want to work within a particular folder, we can make our lives easier by *changing* the current *directory*. For instance, I could have done the above by first moving into `robertson` and then running `ls`.

```
b0036119@caldew:~$ cd robertson
b0036119@caldew:~/robertson$ ls
Experiments        Periodic 2-simultaneous conjugacy
FP Centralisers    Pond example
HNC                Posters
LaTeX              Presentations
Misc notebooks     Progression
Papers             Reading
```

```
Paperwork archive
b0036119@caldew:~/robertson$ pwd
/home/b0036119/robertson
```

Note that my prompt—the bit before the `$`—changes to remind me which directory I'm currently working in. Your system may or may not have this; don't worry if it doesn't look exactly the same.

What about if you want to go back up a level? Use the `..` notation, which stands for 'the previous level'. The exception is when you're in the root directory `/`. Here `..` refers to the root again, simply because there is no previous level! In my example, I can see the contents of my home directory by running `ls ..` in my terminal.

```
b0036119@caldew:~/robertson$ ls ..
archive  Desktop    Metro_Map.pdf  robertson  temp.py~
Audio    Downloads  parallel       scripts    test.txt
barker   Dropbox    Pictures       teaching   thompsons_v
bin      log        R              temp.pdf   website
b0036119@caldew:~/robertson$ pwd
/home/b0036119/robertson
```

If I want to refer to something *inside* the parent directory, I need to add a forwardslash to the double dots.

```
b0036119@caldew:~/robertson$ ls ../website
css        fonts   index.html   js         TODO
downloads  images  index.html~  README.md
```

Without the extra slash, I'd be asking for a file or folder in the present directory (`robertson`) called `..website`. If I want to refer to two levels above, I need to use `../..` (two double-dots glued together with a slash). Three levels would be `../../..` and so on.

```
b0036119@caldew:~/robertson$ ls ../..
b0036119
b0036119@caldew:~/robertson$ ls ../../..
backup  etc         lost+found  proc     space  var
bin     home        media       root     srv    vmlinuz
boot    initrd.img  mnt         run      sys    websites
data    lib         net         sbin     tmp    WinData
dev     lib64       opt         scratch  usr
```

The first directory listing above is `/home`; the second is the root `/`.

Similarly I can use a single full stop `.` to refer to the current directory.

```
b0036119@caldew:~/teaching/linux$ ls
build       demo         demo.tar.gz  images  intro_to_linux.tex
publish.sh  publish.sh~  tex
b0036119@caldew:~/teaching/linux$ ls .
build       demo         demo.tar.gz  images  intro_to_linux.tex
publish.sh  publish.sh~  tex
b0036119@caldew:~/teaching/linux$ ls build
intro_to_linux.aux  intro_to_linux.out  intro_to_linux.synctex.gz
intro_to_linux.log  intro_to_linux.pdf  intro_to_linux.toc
b0036119@caldew:~/teaching/linux$ ls ./build
intro_to_linux.aux  intro_to_linux.out  intro_to_linux.synctex.gz
intro_to_linux.log  intro_to_linux.pdf  intro_to_linux.toc
```

These shortcuts `.` and `..` might seem a little odd at first, but they do come in handy. For instance, some commands will accept a directory as their argument, for instance `cp` (see Section 2.3). Also, when executing your own programs, you need to explicitly refer to the current directory (Section 2.3.5).

Let's move back to our home directory. Because you want to refer to your home area very often, the terminal (strictly speaking, the shell) expands the tilde character `~` to the absolute path of your home directory. (On a UK keyboard this is typically above the hash key, which itself is above the right shift key.) So we can go home with just four characters:

```
b0036119@caldew:~/robertson$ cd ~
b0036119@caldew:~$ pwd
/home/b0036119
```

Top tip! Typing out long file and folder names is tedious, especially when you make a mistake. To quicken things up, we can use *tab completion*. For instance, inside my home directory I can type `ls rob` and press the Tab key; the shell will autocomplete to `ls robertson` because there's no other file or folder beginning with `rob`. On the other hand, if I type `ls b` and then press Tab, the shell doesn't know what to do. Do I mean `ls barker` or `ls bin`? As there's no obvious answer, the shell does nothing. You can ask (the default) shell to show you all the options by pressing Tab twice.

```
b0036119@caldew:~$ ls b
barker/ bin/
```

Inside the home area are lots of hidden folders and files; they're typically used to store your application's preferences and data caches. Any file name or directory starting with a `.` isn't shown by default. We can override this by passing an extra argument to `ls`—the `-a` flag.

```
b0036119@caldew:~$ ls -a
.                       .julia_history
..                      .jupyter
.adobe                  .lesshst
.alias                  .local
archive                 log
.atom                   .macromedia
[listing truncated]
```

The 'a' in `-a` stands for 'all': it shows you everything—even the `.` and `..` which refers to the current and parent directory.

Exercise. Within the `demo` folder, navigate to the `easter_egg_hunt` subfolder. Somewhere inside is a file called `secret.txt`—can you find it? The file and folder names along the way should give you some help, but be warned: some of them are decoys! If you succeed, you can print the secret file to screen by running `cat secret.txt`.

2.2 General command-line advice

Probably the single most useful thing you can do is grab yourself a cheat sheet. I'd recommend the one available from FOSSwire,[2] which has been included at the end of these notes.

[2] `https://fosswire.com/post/2007/08/unixlinux-command-cheat-sheet/`

2.2.1 Getting help

There are three main ways to get help if you don't know how to use a command:

1. See if you can print a help message with **`-h`** or **`--help`**;
2. Read the **`man`** page for the command; or
3. Consult an expert, boffin or guru; even better try asking the INTERNET.

Let's go through the first two.

The vast majority of commands have some sort of help option, typically accessed by running the command and passing the `--help` option (sometimes `-h`). Let's use our old friend `ls`.

```
b0036119@caldew:~$ 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 is specified.

Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  do not ignore entries starting with .
  -A, --almost-all           do not list implied . and ..
      --author               with -l, print the author of each file
  -b, --escape               print C-style escapes for non-graphic characters
[..output truncated]
```

At the top is the usage pattern:

```
Usage: ls [OPTION]... [FILE]...
```

All of the stuff listed that starts with a dash is considered an option, like `-a` from before. The pattern specifics that these options come after the command name but before the file or folder name. The ellipses `...` indicate that more than one option and more than one file/folder can be given. The square brackets enclose *optional* arguments: if we like we can leave them out—that's why `ls` by itself is a perfectly valid command.

Next comes a complete list of all the options specified. Some of them have a short form (one dash and one letter); some have an explicit long form (two dashes); and some commands even have both! These are case-sensitive: note that `-a` does something different to `-A`. Somewhere in that list of options should be `--help`, which we've just used to print the help message!

Exercise. Have a look through the options for `ls` for anything interesting or potentially useful; give them a try! I'd suggest `-S`, `-l` and `-t`. What does the `-h` flag do? (It doesn't print help in this case!)

Exercise. The FIREFOX browser is available in a terminal via the `firefox` command. Check the help message to work out how to open a new tab pointing to `http://example.com`. Can you open two tabs at the same time? Three windows? A mix of tabs, windows and private browsing windows?

Onto the second option: consult the manual. Try running `man ls`—you should end up with something like this.

```
LS(1)                    User Commands                    LS(1)
```

```
NAME
       ls - list directory contents

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

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

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

Manual page ls(1) line 1/272 5% (press h for help or q to quit)
```

`man` searches your machine for relevant documentation and displays it to you in a scrollable window.[3] You can scroll the page using the arrow keys, mouse wheel, or Page Up/Down keys. In this case the manual page is virtually identical to the help page. However if you scroll to the bottom you'll notice that there's some extra information that wasn't present before: 'Author', 'Reporting bugs', 'Copyright' and 'See also'.

Top tip! Even `man` has its own documentation. Try `man --help` or `man man` if you're feeling brave.

Man doesn't print out text to your terminal—instead, it 'takes over' the entire terminal temporarily. At the bottom is the helpful instruction '`press q to quit`', which leads us onto the next section.

2.2.2 Quitting a command and exiting the terminal.

There are plenty of commands that 'take over' your terminal; this section is here just as a reminder that you can often quit by pressing the `Q` key. If that happens, most programs accept an 'interruption signal', or `SIGINT` for short. This tells the program to stop running immediately. You can send this by pressing `Control-C` in a terminal—note that this does *not* copy the current selection to the clipboard!

Applications get the chance to 'respond' to this signal, so they can clean up any resources they're using and exit gracefully. If this goes wrong then `Control-C` might not stop the program; in this case you could quit the terminal, or refer to section 5 for what to do next.

Exercise. If you want to test this out, try running the `ping` command:

```
b0036119@caldew:~$ ping bbc.co.uk
PING bbc.co.uk (212.58.246.79) 56(84) bytes of data.
64 bytes from 212.58.246.79: icmp_seq=1 ttl=51 time=9.09 ms
64 bytes from 212.58.246.79: icmp_seq=2 ttl=51 time=9.19 ms
64 bytes from 212.58.246.79: icmp_seq=3 ttl=51 time=9.15 ms
64 bytes from 212.58.246.79: icmp_seq=4 ttl=51 time=9.11 ms
[output continues...]
```

[3] This will typically be passed displayed through the program `less`, which we'll in Section 2.3.2.

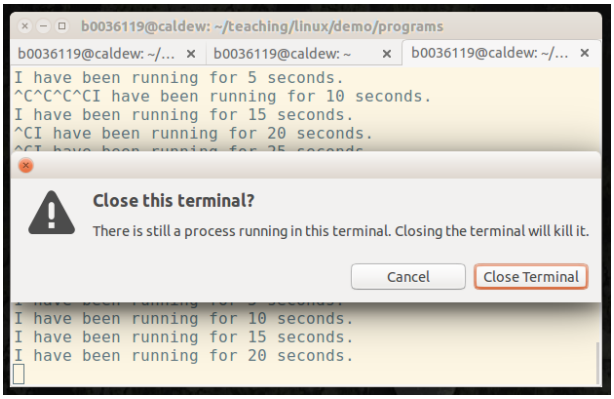

Figure 18: Terminals will typically ask you to confirm closing them if they are running a process.

Without any other arguments, this command runs indefinitely, testing that the given internet address can be reached over the network. Once you're satisfied that your network connection to the BBC (or any website of your choice) is working, you can quit with `Control-C`.

So how do we quit the terminal? If your terminal is graphical, you can close it by closing the window it's contained in. This will kill any processes currently running, though you may be asked for confirmation. (See Figure 18).

Top tip! Some terminals can run multiple shells each in their own tab, which can be handy if you need to alternate between working in one directory and another. Check the terminal's menus to see how to enable this. In this case closing the window will close the shells running in all tabs, and any processes the shells are running. Closing one tab will only close the corresponding shell.

If you're not using a graphical shell, for instance if you are logged into another machine via `ssh`, you can quit the current shell by using the `exit` command. Beware that if this is a 'login shell' (as is the case with `ssh`) you will be logged out from your current session!

2.3 File handling

2.3.1 Creation

To make a new file in the current directory, use the `touch` command. If the files you give it don't exist, they're empty files with the given names are created. That's why the `cat` below (*concatenate*—among other things, it prints the given file to the screen) doesn't appear to do anything.

```
b0036119@caldew:~$ touch example.file
b0036119@caldew:~$ cat example.file
b0036119@caldew:~$ ls -l example.file
-rw-r--r--+ 1 b0036119 nma 0 Jun 22 21:01 example.file
```

Let me wait a minute before I type some more commands. (I checked the file's modification time with `ls -l`.) Note that `touch` will not overwrite an existing file. In this case, its purpose is to update the file's (files') access and modification time to the current time. It doesn't actually edit the file; it only very gently *touches* it to update its metadata.

```
b0036119@caldew:~$ date
Wed Jun 22 21:02:11 BST 2016
b0036119@caldew:~$ touch example.file
b0036119@caldew:~$ ls -l example.file
-rw-r--r--+ 1 b0036119 nma 0 Jun 22 21:02 example.file
```

The analogous command for directories is `mkdir` (*make directory*), which creates new empty directories. If the directory already exists, you'll get an error message.

```
b0036119@caldew:~$ ls
archive  Downloads      parallel   teaching     website
Audio    Dropbox        Pictures   temp.pdf
barker   example.file   R          temp.py~
bin      log            robertson  test.txt
Desktop  Metro_Map.pdf  scripts    thompsons_v
b0036119@caldew:~$ mkdir example_dir
b0036119@caldew:~$ ls
archive  Downloads     Metro_Map.pdf  scripts   thompsons_v
Audio    Dropbox       parallel       teaching  website
barker   example_dir   Pictures       temp.pdf
bin      example.file  R              temp.py~
Desktop  log           robertson      test.txt
b0036119@caldew:~$ ls example_dir
b0036119@caldew:~$ mkdir website
mkdir: cannot create directory 'website': File exists
```

Exercise. (This one isn't particularly important but you might find it interesting.) Make a new file using `touch`; then use `touch` to change its access/modify date to your birthday (consult the help files to work out how.) What's the earliest date you can set using `touch`? What's the latest date? Why is the answer to the last question worrying?

Top tip! Before I forget, here's some advice about how to name your files.

- By default, tab completion is case-sensitive. For example, `ro` would expand to `robertson` but not `Robertson`. Some people get around this by advocating that you only use lowercase filenames. With that said, tab completion can be made case-INsensitive. If you don't mind getting your hands dirty, see e.g. `http://superuser.com/a/435127/550042`.
- Spaces are also a bit of a pain on the command line. If I type `rm my file`, the shell would interpret that as passing two arguments to `rm`: the first being the string `"my"` and the second being `"file"`. I'd have to either quote the argument (`rm "my file"` or escape the space with a backslash (`rm my\file`). People advocate getting around this by using underscores instead.
- The same goes for other characters like `&`, `>`, `<` and `|` which have special meanings in a shell. Escape them with a backslash; quote them; or else avoid them in file and folder names!

2.3.2 Reading and writing files

Let's start with writing. We're going to make life easy for ourselves by using a graphical editor called `gedit`, so-called because it's the GNOME Desktop's text editor. This can be launched from the applications menu, or in the terminal by typing

```
b0036119@caldew:~$ gedit &
[1] 20637
```

The reason there's an ampersand (`&`) here is that we need to run `gedit` in the background. Without the ampersand otherwise it will take over our terminal until we quit with `Control-C`. The number that's displayed is a 'process ID': see Section 5 for more details.

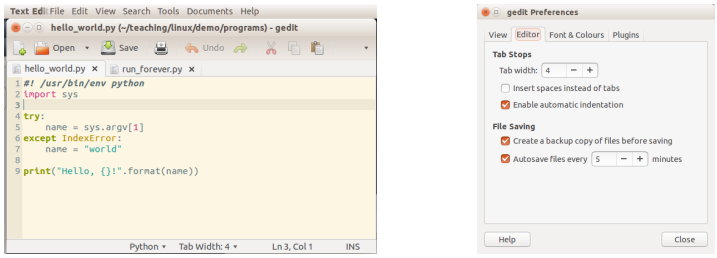

Figure 19: I used `gedit` to write most of the demo files. In the setup I'm using (Ubuntu's UNITY desktop) the menus are kept on the top of the screen, similar to a Mac.

Gedit works just like NOTEPAD on Windows and TEXTEDIT on Mac: it handles reading, editing and writing of plain text files. Let me briefly a few tips:

- Unlike Notepad, `gedit` can edit multiple files by placing each in their own tab. This is helpful when you need to edit two or more related files in the same project, for instance.
- The user settings[4] can be configured by selecting *Edit* → *Preferences*. If you're planning to do any programming I'd recommend you use the display line numbers and the highlighting options.
- One very useful feature is 'syntax highlighting'. `gedit` will try to turn this on automatically by examining the extension of the file you're editing. If it guesses wrong, you can override this by selecting *View* → *Highlight mode*.
- If you're editing `example.file` in gedit, an backup file `example.file~` with a tilde `~` appended will be automatically created. These are perfectly normal and nothing to worry about! If you don't want them cluttering your filespace, you can disable them in the options—see Figure 19.

Exercise. Create a new text file in your home area which contains your favourite joke. Don't forget to save it!

There's nothing stopping you from using `gedit` to read files, but let me mention a few terminal-based utilities to do so. The big one is `less`, which displays as much of a file as it can in the terminal and lets you scroll with the arrow, Page Up and Page Down keys. (The name is a quirk of history: it's an improvement of a previous program called `more`, which did the same thing but only allowed you to scroll forward.)

```
b0036119@caldew:~/teaching/linux$ less intro_to_linux.tex
\documentclass[a4paper, draft]{article}
\usepackage[margin=1.5in]{geometry}
\usepackage{amsmath}
\usepackage{hyperref}
\usepackage{graphicx}
\usepackage{xcolor}
\usepackage{placeins}
\usepackage{framed}
\usepackage{lmodern}
\usepackage{pdfpages}
```

[4]On Windows the conventional menu location for this is *Tools* → *Options*; on a Mac the settings are contained in the menu whose label is the application's name.

```
\title{Introduction to Linux}
\author{David Robertson}
\setcounter{section}{-1}
intro_to_linux.tex lines 1-14/1394 0%
```

Press `Q` to quit `less` when you're finished reading the file.

`less` has too many features and options to go into now, but let me briefly mention how to search. Activate the search mode by pressing forwardslash `/`. Type the string you're searching for and press enter. To find the next instance of your search term, press `N`; to find the previous insteance press `Shift-N`.

It's also worth mentioning `head` and `tail`, which show the first (respectively last) few lines of a file.

```
b0036119@caldew:~/demo/poems$ head chocolate_cake.poem
Chocolate Cake

I love chocolate cake.
And when I was a boy
I loved it even more.

Sometimes we used to have it for tea
and Mum used to say,
'If there's any left over
you can have it to take to school
b0036119@caldew:~/demo/poems$ tail chocolate_cake.poem
not before you've washed your dirty sticky face.'
I went upstairs
looked in the mirror
and there it was,
just below my mouth,
a chocolate smudge.
The give-away.
Maybe she'll forget about it by next week.

        -- Michael Rosen
```

Exercise. Have a look at the `poems` folder inside the collection of demo files. Use `less` to read through them at your leisure. You should now feel mentally reinvigorated and culturally energised. Apply this energy to your next challenge: can you work out a way to get `tail` to show the authors of all three poems using one command?

2.3.3 Only for the brave: `nano`, `vim` and `emacs`

Feel free to skip this section. It describes *terminal-based* text editors and is **not for the faint of heart**. Learning to use them is more of a learning cliff than a learning curve; they have many powerful features and are highly configurable. Indeed some people prefer to use them over a graphical editor—once you've learned all the tips and tricks you can manipulate files very efficiently.

I'm including this section mainly as reference material. Even if you don't like their interface, these editors can be useful if you want to edit files on a command-line only remote computer, e.g. via `ssh`. (They can also be the default editor used by programs like `git commit`.) Unfortunately I'm only aware of how to use `vim`; you'll have to defer to the INTERNET for `nano` or `emacs` advice. So with that said, here's an extreme crash course in `vim`.

- Open a file for editing by typing `vim <filename>`.
- Initially you're in *normal mode*, which is used to navigate the file and edit its contents. You can use the arrow keys to move the cursor around the file.
- Press `I` once to enter *insert mode*. This mode is very similar to a normal text editor: typing inserts characters into the file; delete and backspace work as normal.

- Once you're done with insert mode, press `Escape` to return to normal mode.
- In normal mode, type a colon `:` followed by a `w`, then enter to save the file. (The w stands for write.)
- To quit `vim` and return to the terminal, type `:q`. If you have any unsaved changes, `vim` will ask you to confirm that you want to quit by typing `:q!`.

If this hasn't put you off and you'd like to learn more, I'd recommend finding yourself a good tutorial or cheat sheet[5].

2.3.4 Moving, copying and deleting

WARNING! All of these commands will overwrite or delete the target file, so think before you type.

The commands you want are `mv` for *moving*, `cp` for *copying*, and `rm` for *removing*. The first two move/copy their first 'source' argument to their second 'destination' argument. As in the warning, if the destination file already exists, moving or copying something onto it will overwrite the destination file.

Let's see this with an example. I've made two text files in a folder called `example`. First a copy operation:

```
b0036119@caldew:~/example$ ls
file1.txt  file2.txt
b0036119@caldew:~/example$ cat file1.txt
file 1
b0036119@caldew:~/example$ cat file2.txt
file 2
b0036119@caldew:~/example$ cp file1.txt file3.txt
b0036119@caldew:~/example$ cat file3.txt
file 1
```

Now we'll do another copy operation which overwrites `file3.txt`.

```
b0036119@caldew:~/example$ ls
file1.txt  file2.txt  file3.txt
b0036119@caldew:~/example$ cp file2.txt file3.txt
b0036119@caldew:~/example$ cat file3.txt
file 2
```

Now we'll do a move operation (or 'rename' if you prefer) which also happens to overwrite the destination file.

```
b0036119@caldew:~/example$ mv file2.txt file3.txt
b0036119@caldew:~/example$ ls
file1.txt  file3.txt
b0036119@caldew:~/example$ cat file3.txt
file 2
```

Finally let's delete everything to tidy up after ourselves. The `rm` command deletes all its arguments. We can see this because the second `ls` command outputs nothing below.

```
b0036119@caldew:~/example$ ls
file2.txt  file3.txt
b0036119@caldew:~/example$ rm file2.txt file3.txt
b0036119@caldew:~/example$ ls
```

[5]Try `http://www.viemu.com/a_vi_vim_graphical_cheat_sheet_tutorial.html`

Alternatively we could have tried to remove the entire directory, but this would have given us an error. To delete a directory, we need to pass the `-r` (for *recursive*) option. Note that this will ruthlessly delete anything in the target directory and all of its subdirectories—it's a big red nuclear button of death!

```
b0036119@caldew:~/example$ rm ..
rm: cannot remove '..': Is a directory
b0036119@caldew:~/example$ rm -r .
rm: cannot remove directory: '.'
```

I can't remove the current directory by referring to it as `.`, because afterwards the working directory would be a directory that no longer exists! Instead, I need to move up a level and delete the `demo` folder explicitly.

```
b0036119@caldew:~/example$ cd ..
b0036119@caldew:~$ rm -r example
b0036119@caldew:~$ ls
archive  Desktop    Metro_Map.pdf  robertson  temp.py~
Audio    Downloads  parallel       scripts    test.txt
barker   Dropbox    Pictures       teaching   thompsons_v
bin      log        R              temp.pdf   website
```

No `example` folder in sight!

Exercise. Open the `jokes` folder inside the demo files. At the time of writing it contains 15 jokes of varying quality—some of them are awful puns from the internet, whereas others are award-winning jokes from the Edinburgh Festival.

Make three new folders called `good`, `okay` and `poor`. Delete the worst three jokes, and categorise the remaining 10 jokes by moving them into the folders you've just created. Copy the joke you created two exercises ago into the appropriate folder. Delete the `poor` folder, and rename `okay` to '`barely_acceptable`'.

Top tip! The `scp` (*secure copy*) command can be used to copy files or folders to another computer via `ssh`.

2.3.5 Running your own programs

This section is mostly here for reference purposes in the future.

In the `programs` folder there's a toy program called `hello_world.py`. Let's run it to see what it does.

```
b0036119@caldew:~/demo/programs$ ls
hello_world.py  potatoes.py  run_forever.py
b0036119@caldew:~/demo/programs$ hello_world.py
hello_world.py: command not found
```

That's odd–we've run every command before by just typing its name and pressing enter. What's different?

The list of paths that the shell searches on for programs is kept in an environment variable called `PATH`. (This is also true on Mac and Windows.) We'll learn more about environment variables later; for now I'll just show you how to print it out.

```
b0036119@caldew:~/demo/programs$ echo $PATH
/data/bin:
/home/b0036119/bin:
```

```
/usr/local/vapor/x86_64/bin:
/usr/local/maple15/x86_64/bin:
/usr/local/bin:
/usr/sbin:
/usr/bin:
/sbin:
/bin:
/usr/local/auto/07p/cmds:
/usr/local/auto/07p/bin
```

Because the PYTHON program we just tried to run isn't in one of these directories, it the shell didn't run it when we asked it to. (In a terminal these paths won't be shown on new lines; I've added them for clarity.) The reason for this is to stop someone leaving a malicious program called `ls` or `pwd` which secretly deletes your home directory whenever you run it. You want to be confident that `ls` always refers to the system program `ls`!

To get around this problem, tell the shell 'I'm sure I want to execute this thing in the current directory' by adding a `./` to the front of the file name.[6]

Let's try the example again.

```
b0036119@caldew:~/demo/programs$ ./hello_world.py
bash: ./hello_world.py: Permission denied
```

Hmm, this still isn't working. The problem is that this file isn't marked as 'safe to execute'. Part of a file's metadata includes whether or not it can be executed. This stops you from asking the computer to interpret a random file as an executable program, which could go badly! We can get around this by marking the file as executable with `chmod`.

```
b0036119@caldew:~/demo/programs$ ls -l hello_world.py
-rw-r--r--+ 1 b0036119 nma 129 Jun 22 17:27 hello_world.py
b0036119@caldew:~/demo/programs$ chmod +x hello_world.py
b0036119@caldew:~/demo/programs$ ls -l hello_world.py
-rwxr-xr-x+ 1 b0036119 nma 129 Jun 22 17:27 hello_world.py
b0036119@caldew:~/demo/programs$ ./hello_world.py
Hello, world!
```

(Note how an `x` for *execute* appeared in the output of `ls -l`.) If I want to execute a program in another directory that works fine. The key thing is that I have to give a specific path to the program I want to execute, not just its name by itself.

```
b0036119@caldew:~/demo/programs$ cd ..
b0036119@caldew:~/demo$ programs/hello_world.py
Hello, world!
```

`chmod` is used more generally to alter file and folder permissions. This metadata is present on every file and folder; it controls who can read, write and execute a given file. Have a look at the `man` or `--help` pages for `chmod` if you're curious.

If you have to run a long numerical simulation, it can be useful to measure how long your program takes to complete. Enter the `time` command! To use it, pass the name of a program and any arguments you want to send to it. For instance I could see how long it takes to `echo` back a short message.

```
b0036119@caldew:~$ time echo "How long will this take to print?"
How long will this take to print?
```

[6] Windows differs here—files in the current working directory are considered for execution without the `./` notation.

```
real    0m0.000s
user    0m0.000s
sys     0m0.000s
```

The answer is 'not very long'. A slightly more instructive example is `curl`, which can be used to retrieve files from the internet.

```
b0036119@caldew:~$ time curl example.com
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
        [... output trimmed ...]
</html>

real    0m0.178s
user    0m0.006s
sys     0m0.009s
```

`time` reports three durations.

`real` This is 'clock time', the number of seconds between starting your program and receiving its final output.

`user` This is the amount of time the computer is executing your code, including any libraries of packages you're using. It doesn't include time when your program is waiting, e.g. for a file to be read from disk or for a file to download.

`sys` The system time measures how much time the kernel spent facilitating your process, e.g. assigning memory, writing files to disk.

We always have `user`+`sys` < `real`. This is because the kernel is constantly multitasking: for instance, it has to run all the other programs on your machine, and wait for files or network connections to become available.

Exercise. The program `nth_fibonacci.py` accepts two arguments n and `display`. It computes the nth Fibonacci number, defined by the recurrence $f_{m+2} = f_{m+1} + f_m$ with initial conditions $f_0 = f_1 = 1$. If `display` is the string "`print`", the program prints f_n; otherwise it displays nothing.

```
b0036119@caldew:~/demo/programs$ ./nth_fibonacci.py 10
b0036119@caldew:~/demo/programs$ ./nth_fibonacci.py 10 print
55
```

Use the program to compute the first, tenth, hundredth and thousandth Fibonacci number. Then use `time` to measure how long it takes to do this. Can you describe the relationship between the `real` runtime and n—e.g. is it linear, logarithmic, quadratic, exponential in n? Can you explain why?

Disclaimer: this program was just something I quickly cobbled together—it's by no means optimised or efficient.

2.4 The filesystem

This section is mostly for reference; it's not particularly relevant or exciting.

2.4.1 Links

Have you ever created a shortcut on Windows to an important file or folder? It points to another piece of data on your system, but can be kept in a separate place. This is implemented as a special `.lnk` file which Windows' EXPLORER knows how to interpret. If you're not Explorer, however, all you you can see is this `.lnk` file, which looks just like an ordinary text file!

Shortcuts are built on top of the filesystem; but 'links' are a bit more transparent. If you try read from a file via shortcut, you'll see the data within the `.lnk` file describing the target. By contrast, if you try to open a link in your program you'll see the data within the target file. The links we'll discuss are called *symbolic links*, which we can create by calling `ln -s <target> <link_location>`. The `-s` stands for symbolic.

```
b0036119@caldew:/tmp$ mkdir stuff
b0036119@caldew:/tmp$ echo "Example file contents" > stuff/example.txt
b0036119@caldew:/tmp$ ls
hsperfdata_condor  stuff
b0036119@caldew:/tmp$ ln -s stuff/example.txt my_great_link.ext
b0036119@caldew:/tmp$ ls
hsperfdata_condor  my_great_link.ext  stuff
b0036119@caldew:/tmp$ cat my_great_link.ext
Example file contents
```

(the `>` is a quick way to add output to a file; see Section 3.2.3.) If I delete the original file, the link becomes broken.

```
b0036119@caldew:/tmp$ rm stuff/example.txt
b0036119@caldew:/tmp$ cat my_great_link.ext
cat: my_great_link.ext: No such file or directory
```

If you want to see if a file or folder is really a link, you can use `ln -l`:

```
b0036119@caldew:~$ ls -l
total 795
drwxr-xr-x+ 10 b0036119 nma     12 Apr 25 11:33 archive
drwxr-xr-x+  2 b0036119 nma      3 Dec 17  2015 Audio
lrwxrwxrwx   1 b0036119 nma     12 Nov 27  2015 barker -> /data/barker
lrwxrwxrwx   1 b0036119 nma     10 Oct 19  2015 bin -> .local/bin
lrwxrwxrwx   1 b0036119 nma      5 Jun 28 16:00 data_drive -> /data
lrwxrwxrwx   1 b0036119 nma     20 Jun 23 17:14 demo -> teaching/linux/demo/
[output truncated]
```

There's also the `readlink` command:

```
b0036119@caldew:~$ readlink data_drive
/data
```

Notice how you're not allowed to make a cycle using links.

```
b0036119@caldew:/tmp$ touch a
b0036119@caldew:/tmp$ ln -s a b
b0036119@caldew:/tmp$ ln -s b c
b0036119@caldew:/tmp$ ln -s c d
b0036119@caldew:/tmp$ ln -s d a
ln: failed to create hard link 'a': File exists
```

In general, the `linkeaster_location` must not already exist, or else an error will be thrown.

Exercise. Make a link in `easter_egg_hunt` to the `secret.txt` file from a previous exercise. Try making a link from a directory to its parent directory. You should then be able to make an infinite path which goes round in circles! Use this to create a daft path such as

`~/around/the/world/around/the/world/around/the/world/around/`

which could potentially repeat forever. Why does the existence of this trick make it harder to search through a file system?

Maths & Stats only: In Maths and Stats, many users often make a symbolic link to important data drive areas in their home area, such as an SVN repository.

Links can be made on Mac and Windows (Vista or later) machines too. `ln` works the same on Mac, but Windows users have to use the `mklink` command with administrator privileges. The Wikipedia page[7] gives a thorough overview.

2.4.2 Disk usage

The `du` (*disk usage*) command prints out the folders inside a directory and shows how much space they're occupying on disk, including any hidden files they contain. It can take a while to process everything, particularly if you have a lot of small files in the given directory. The command `df` (*disk free*) is complementary; it shows how much free space is available on each filesystem available to your system.

```
b0036119@caldew:~/demo$ \du
3        ./easter_egg_hunt/down_down
[... output trimmed ..]
17        ./jokes
8        ./programs
8        ./poems
62        .
b0036119@caldew:~/demo$ \df
Filesystem                          1K-blocks     Used Available Use% Mounted on
udev                                  4026468       12   4026456   1% /dev
tmpfs                                  807464     1212    806252   1% /run
/dev/sda5                            19092180  8070016  10029296  45% /
[... output trimmed ..]
/dev/sda9                           336301224 16948396 302246652   6% /data
mas-home-pg:/tank/home/b0036119       2097152  1330176    766976  64% /home/b0036119
```

We'll quickly mention the `-h` flag, which shows turns the numbers above into human-readable numbers with units attached. For instance, part of `df -h`'s output is

```
mas-home-pg:/tank/home/b0036119           2.0G  1.3G  749M  64% /home/b0036119
```

which I think you'll agree is easier to read.

3 More advanced terminal trickery

Now that we've got the basics, this section introduces commands and notation to make us more efficient when using a terminal.

[7] `https://en.wikipedia.org/wiki/Symbolic_link`

3.1 Searching and replacing

3.1.1 Searching for files and folders

With no argument, the `find` command prints out a list of the files and folders contained in and below the current working directory. This includes hidden files and folders as well.

```
b0036119@caldew:~/teaching/linux$ find
.
./.git
./.git/refs
./.git/refs/heads
./.git/refs/heads/master
./.git/refs/tags
./.git/hooks
[...output truncated]
```

In general, `find` searches in a directory for a files matching a certain series of *tests*. For instance, if I only wanted to find `.pdf` files I'd use the `-name` test.

```
b0036119@caldew:~/teaching/linux$ find -name *.pdf
./build/intro_to_linux.pdf
./images/fwunixref.pdf
```

Strictly speaking I should be using `-iname` here, which is a case-insensitive name test. The asterisk `*` is part of a *glob pattern* and stands for 'anything'. Thus we can read `*.pdf` as 'anything followed by `.pdf`'. See section 4.2.5 for more on glob patterns.

Another useful test is `-type f` for finding files only, or `type d` for directories only. In the example below I've combined these with the `-maxdepth 1` test to stop `find` from inspecting the entire contents of my DROPBOX.

```
b0036119@caldew:~/Dropbox$ find -maxdepth 1 -type d
.
./Project Euler
./Keys and certificates
./Screenshots
./Maths
./Public
./Python
./.dropbox.cache
./GoodNotes
./CV
./Outlook
./Camera Uploads
./LaTeX
b0036119@caldew:~/Dropbox$ find -maxdepth 1 -type f
./Subtle is the Lord - Einstein Biography.pdf
./.dropbox
./Dynamic Storage Allocation.pdf
```

For each successful match an *action* is made. If no action is specified, the default is `-print`, which prints each file found to `stdout`. Other actions that might be handy include `-ls`, which prints the same style output as `ls -l`.

```
b0036119@caldew:~/teaching/linux$ find -name *.pdf
./build/intro_to_linux.pdf
./images/fwunixref.pdf
```

There's also the `-delete` action, but just like `rm` this is an action to be used with extreme caution!

```
b0036119@caldew:/tmp$ touch 1.good 2.bad 3.bad 4.good 5.bad 6.good
b0036119@caldew:/tmp$ touch 7.bad 8.good 9.good 10.good 11.bad
b0036119@caldew:/tmp$ ls
10.good  11.bad 1.good  2.bad  3.bad  4.good  5.bad  6.good  7.bad  8.good  9.good
b0036119@caldew:/tmp$ find -name "*.bad"
./11.bad
./5.bad
./7.bad
./2.bad
./3.bad
b0036119@caldew:/tmp$ find -name "*.bad" -delete
b0036119@caldew:/tmp$ ls
10.good  1.good  4.good  6.good  8.good  9.good
```

`find` does much more than these examples show. In fact it does far too much to cover here, so I'll defer to `find --help` and `man find` if you want to find out more.

> **Exercise.** Use `find` to cheat at the previous exercise; find the location `secret.txt` inside the `demo` folder. Starting from the root `/`, which directories do you have permission to write to?

3.1.2 Searching within files

`grep` stands for *global regular expression print*. Its job is to look through a stream of text and look for a run of text which matches a given pattern. These patterns are the 'regular expressions' we mentioned. They can be more complicated than the 'glob patterns' we used earlier; but we'll start by keeping things simple. The simplest version is `grep <pattern> <file>`, which tests each line of FILE to see if it matches PATTERN; if so, the line is printed and the match is highlighted.

```
b0036119@caldew:~/thompsons_v$ ls
docs  notebooks  README.md  requirements.txt  scripts  thompson
b0036119@caldew:~/thompsons_v$ grep group README.md
[Nathan Barker](https://www.dpmms.cam.ac.uk/~nb443/), [Andrew Duncan]
(http://www.mas.ncl.ac.uk/~najd2/) and [David Robertson]
(https://DMRobertson.github.io) are currently preparing a paper entitled
*The power conjugacy problem in Higman-Thompson groups*
which describes an algorithm to solve a certain equation in the groups
named $G_n,r$.
- the automorphism group $G_n,r = \Aut V_n,r$.
```

Notice how `grep` doesn't look for words in their own right—just substrings which match the given pattern.

```
b0036119@caldew:~/demo$ grep so poems/chocolate_cake.poem
so I thought,
so it must have been really late
so I lick my finger and run my finger all over the crumbs
but it's so nice,
'And don't forget to take some chocolate cake with you.'
```

> **Exercise.** Use `grep` to count how many times the word 'cake' appears in `poems/chocolate_cake.poem`. Make sure you find the option to ignore the case, so that 'Cake', 'CAKE' and 'cake' would all get counted.

Within a regular expression, a caret `^` stands for 'the start of the line' and a dollar `$` means 'the end of the line'. So I can search in a file for a line starting with 'so', instead of just all instances of 'so'.

```
b0036119@caldew:~/demo$ grep ^so poems/chocolate_cake.poem
so I thought,
so it must have been really late
so I lick my finger and run my finger all over the crumbs
```

The `-r` flag is particularly useful: it recursively searches for a pattern in the lines of every file inside the current directory and its subdirectories.

```
b0036119@caldew:~/demo$ grep -r if
jokes/joke5.txt:Crime in multi-storey car parks. That is wrong on so many different levels.
programs/run_forever.py:        if seconds % 5 == 0:
programs/delay.py:if __name__ == "__main__":
programs/potatoes.py:                if ticks % 4 != 0:
programs/nth_fibonacci.py:if __name__ == "__main__":
programs/nth_fibonacci.py:        if display:
poems/chocolate_cake.poem:what if I go downstairs
poems/chocolate_cake.poem:So I take a knife
poems/chocolate_cake.poem:Take the knife
poems/chocolate_cake.poem:This time the knife makes a little cracky noise
poems/chocolate_cake.poem:Knife -
poems/chocolate_cake.poem:and the knife
poems/chocolate_cake.poem:'You don't know. You don't know if you've eaten a whole
poems/jabberwocky.poem:Came whiffling through the tulgey wood,
```

> **Exercise.** How many files in `demo` contain the substring 'cake', ignoring upper and lower case? How hungry do you think I was when I was writing this section?

Passing the `-E` flag (for *extended grep*) to `grep` interprets special characters like `()?+{}|()` with their meaning in regular expression syntax, allowing you to use the full power of regular expressions. We'll see a little bit more in the next section, but a full discussion is beyond the scope of an introductory course.

Instead let me point you to the website `http://regexr.com/`. This is an excellent learning resource and features an interactive regular expression builder. For more information about the mathematical theory of regular expressions and 'finite state automata', I'd recommend the book *Formal languages and automata theory* by Vladimir Drobot (1989).

3.1.3 Replacing within files

The `sed` command (*stream editor*) typically takes two pieces of input: a path pointing to a text file, and a *script* which describes a text transformation. The program applies this transformation to each line and prints out the result of doing so. By default the input file is unedited. By far the most commonly used transformation is a souped-up version of 'find and replace'; but `sed` does much more that this. We'll just touch on the basics.

The find-and-replace syntax is `sed "s/`P`/`R`/" `I, where P is the search pattern, R is the replacement pattern and I is the input file. In this example I replace every instance of `"Humpty"` I can find with the string `"Henry"`.

```
b0036119@caldew:~/demo/nursery_rhymes$ cat humpty_dumpty
Humpty Dumpty sat on a wall,
Humpty Dumpty had a great fall.
All the king's horses and all the king's men
Couldn't put Humpty together again.
b0036119@caldew:~/demo/nursery_rhymes$ sed "s/Humpty/Henry/" humpty_dumpty
Henry Dumpty sat on a wall,
Henry Dumpty had a great fall.
All the king's horses and all the king's men
Couldn't put Henry together again.
```

By default the output of `sed` goes to `stdout`, so if you want to write to a file you need to tell the shell to redirect `stdout` using a `>` symbol (see Sections 3.2.1 and 3.2.3).

```
b0036119@caldew:~/demo/nursery_rhymes$ sed "s/Humpty/Henry/" humpty_dumpty > henry
b0036119@caldew:~/demo/nursery_rhymes$ ls
henry_dumpty  humpty_dumpty
b0036119@caldew:~/demo/nursery_rhymes$ cat henry
Henry Dumpty sat on a wall,
Henry Dumpty had a great fall.
All the king's horses and all the king's men
Couldn't put Henry together again.
```

I enclose the script in quotes out of habit, because the patterns P and R are allowed to contain spaces. If I don't quote the script, the shell will break up my script into multiple arguments and make `sed` unhappy.

```
b0036119@caldew:~/demo/nursery_rhymes$ sed s/ and / ANDANDAND / humpty_dumpty
sed: -e expression #1, char 2: unterminated `s' command
b0036119@caldew:~/demo/nursery_rhymes$ sed "s/ and / ANDANDAND /" humpty_dumpty
Humpty Dumpty sat on a wall,
Humpty Dumpty had a great fall.
All the king's horses ANDANDAND all the king's men
Couldn't put Humpty together again.
```

Just like `grep`, P is allowed to be a regular expression. For example, I can use `^` and `$` to refer to the beginning and end of a line.

```
b0036119@caldew:~/demo/nursery_rhymes$ sed "s/^Humpty/Davos/" humpty_dumpty
Davos Dumpty sat on a wall,
Davos Dumpty had a great fall.
All the king's horses and all the king's men
Couldn't put Humpty together again.
```

In the pattern P below, I'm using a full stop which stands for "any single character". It matches the command and full stop at the end of the first and second line. I had to escape the exclamation mark `!` with a backslash, as `!` has a special meaning in `sed` scripts.

```
b0036119@caldew:~/demo/nursery_rhymes$ sed "s/all.$/ALL\!/" humpty_dumpty
Humpty Dumpty sat on a wALL!
Humpty Dumpty had a great fALL!
All the king's horses and all the king's men
Couldn't put Humpty together again.
```

By default, `sed` will only replace $P \rightarrow R$ once per line (the first time it sees a match for P). To get around this, pass the `g` (*global*) option to the `s` command by placing it after the last `/`.

```
b0036119@caldew:~/demo/nursery_rhymes$ sed "s/the/of those fancy/" humpty_dumpty
Humpty Dumpty sat on a wall,
Humpty Dumpty had a great fall.
All of those fancy king's horses and all the king's men
Couldn't put Humpty togeof those fancyr again.
b0036119@caldew:~/demo/nursery_rhymes$ sed "s/the/of those fancy/g" humpty_dumpty
Humpty Dumpty sat on a wall,
Humpty Dumpty had a great fall.
All of those fancy king's horses and all of those fancy king's men
Couldn't put Humpty togeof those fancyr again.
```

Unfortunately the last example matched `"the"` in `together` when I really wanted P to match the word 'the' by itself. I could get around this by adding spaces to the pattern P, like I did with `"s/ and / ANDANDAND /"` above. Another option is to specify an *address* before the `s` command. For instance, I can ask `sed` to replace a matches only on line 3.

```
b0036119@caldew:~/demo/nursery_rhymes$ sed "3 s/the/of those fancy/" humpty_dumpty
Humpty Dumpty sat on a wall,
Humpty Dumpty had a great fall.
```

```
All of those fancy king's horses and all the king's men
Couldn't put Humpty together again.
```

Exercise. This one's a challenge—you'll need to look through documentation. I'd suggest running `info sed`, or alternatively browse the same thing online at `https://www.gnu.org/software/sed/manual/sed.html`. There's also a nice tutorial avilable at `http://www.grymoire.com/Unix/Sed.html`

Browse through the `nursery_rhymes` folder at your leisure. Once you've satisfied your desire for childish nostalgia, use `sed` to make the following edits to `london_bridge`.

1. Remove every instance of the letter `e`.
2. Make all letter `i`-s upper case.
3. Remove both upper and lower case `I` and `i` with one command.
4. Make every vowel upper case.
5. Insert a blank line between every line in the file.
6. Insert a blank line after every even numbered line in the file.
7. Transform into an early 2000s text message: remove all vowels and punctuation. Make the first remaining character of every line lowercase.

Top tip! We've seen that the `s` command takes the form `s/P/R/O`, where P and R are as before and O is a list of options, e.g. the `g`lobal option. I actually told you a white lie: you can use any character, not just `/` to separate out the arguments to `s`; `/` is just a convention. If you want to find-and-replace something involving `/`, say part of a file pathfor instance, you can use another character like `|` as your separator. For example:

```
b0036119@caldew:/tmp$ cat path.txt
/data/bin:/home/b0036119/bin:[...truncated]
b0036119@caldew:/tmp$ sed "s|/bin|/binaries_go_here|g" path.txt
/data/binaries_go_here:/home/b0036119/binaries_go_here:[truncated]
```

The alternative would be escaping the forwardslashes, as in

```
sed "s/\/bin/\/binaries_go_here/g" path.txt
```

which is a lot less readable.

3.2 Pipelines

3.2.1 Streams

In Linux (and other Unix-like operating systems, e.g. macOS), absolutely everything which can send or receive data is treated in the same way as if it were a file stored on disk. This abstraction includes processes (try looking inside `/proc`), ordinary files, network connections; even devices like CD/DVD drives, printers, mice and keyboards. People often say that 'everything is a file', though Linus Torvalds explained that this really means 'everything is a stream of bytes'.

So what is a stream? Computers typically send and receive data like water flowing down a pipe—they receive small chunks information over time, rather than getting it all in one go. For instance, we download or upload files over the internet as fast as we can send them, but unless the file is very small it'll get sent via multiple packets.

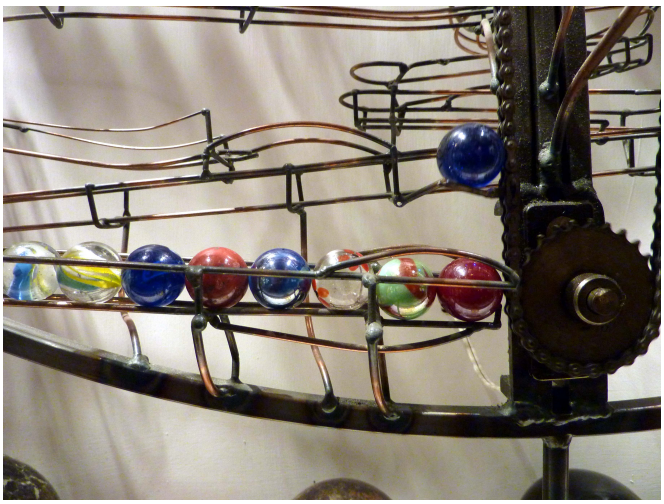

Figure 20: Streams are like kids' marble run toys. Marbles get sent along the track, just like data gets sent through a stream. You can put more marbles into the beginning of the track in the same way you can send data into a stream. We have to wait for marbles to reach the bottom of the track so they can be collected. We collect one marble at a time and the next falls into place as soon as possible. Analogously, streams read one byte (or sometimes bigger units) of data at a time, or else wait for data to become available.
Image: `https://www.flickr.com/photos/glamhag/5565546722`

Every process on Linux (and Mac, and Windows, and many other operating systems) comes pre-connected to three streams called the *standard streams*; it's free to use other streams too, but you're always guaranteed to have these.

`stdin` The *standard input* usually represents instructions or input data for the process in question. By default it consists of the characters you type on your keyboard. The commands we've been using so far received their instructions from command-line arguments, so it might seem like we haven't been using this. However, the shell process itself had to capture your keyboard input; it did so by reading from `stdin`.

`stdout` *Standard output* is the opposite—it's where the program sends all of its results. By default this is shown in the terminal when you run a command, so you've already seen loads of `stdout`put.

`stderr` *Standard error* is much the same, but is specifically intended for displaying error messages only. The reason this is kept separate from standard output is that you might want to pass a program's results into another program. It's much easier to keep any error messages or warnings separate; otherwise we'd have to teach the receiving program to ignore the extra error information.

Not every program gets its input from `stdin`; some do so exclusively via their arguments. Others are `stdin` only, and other still can accept input from both.

Exercise. The example below features the `factor` program, which accepts a positive integer n and returns its prime factors $p_1 \leq p_2 \leq \cdots \leq p_r$. Which parts of the terminal output are `stdin`, `-out` and `-err`?

Figure 21: Repeatedly redirecting program output into another program's input leaves us with a chain of programs called a *pipeline*.

```
b0036119@caldew:~$ factor
12345
12345: 3 5 823
-1
factor: '-1' is not a valid positive integer
182374
182374: 2 67 1361
0
0:
8
8: 2 2 2
```

Top tip! Processes also provide an *error code* or *return code* when they complete. If you've dabbled in C programming this is the `int`eger returned by `int main()`. By convention, a value of 0 means that no errors occurred.

3.2.2 Pipes

The shell introduces several operators which 'redirect' the standard streams. This allows us to plug the output of one command into the input of another; or to send the output of a command to a file and suppress it in the terminal. The process of arranging this is called *piping*; we pipe the first program into the second. To create a pipe we use the vertical bar character `|` (also called a pipe) between two commands. The shell knows that it needs to interpret the `|` as a pipe instruction, rather than sending a literal `|` character as an argument to the first command.

It's probably best to demonstrate with examples.

```
b0036119@caldew:~$ ls -al | grep py
```

```
drwxr-xr-x+  6 b0036119 nma        6 Feb  2 14:22 .ipython
drwxr-xr-x+  2 b0036119 nma        4 Apr 13 10:34 .jupyter
drwxr-xr-x+  2 b0036119 nma        4 May 25 12:53 .pylint.d
-rw-------+  1 b0036119 nma     2574 Jun 23 12:16 .python_history
-rw-r--r--+  1 b0036119 nma      450 Jun 22 17:24 .pythonstartup.py
-rw-r--r--+  1 b0036119 nma      449 Mar  7 16:52 .pythonstartup.py~
-rw-r--r--+  1 b0036119 nma       60 Jun 15 15:00 temp.py~
```

Say I was looking for Python files and folders in my home directory. The first program is `ls`, which I've used to print a detailed listing of everything in my home directory. (Notice that I can combine the flags `-a` and `-l` into `-al`; this always works for options with one letter names.) Unfortunately I've got loads of files there, so I filtered the output using `grep` (Section 3.1.2) to just those lines which contain the string `'py'`. Notice here that if I only provide a search expression, `grep` will filter the text it receives from its standard input.

How many files did I find? We can find out programatically by piping the output of `grep` into another program.

```
b0036119@caldew:~$ ls -al | grep py | wc -l
7
```

The `wc` program stands for *word count*. It counts the number of bytes, words and lines in is standard input (if no file name is given). I've used the `-l` option here to only ask for the number of lines. If we were feeling really adventurous we could try to prime factorise this number.

```
b0036119@caldew:~$ ls -al | grep py | wc -l | factor
7: 7
```

It turns out that I have a prime number's worth of Python files in my home directory.

This is a silly example but it drives home an important part of Unix's approach to computing. Rather than have lots of programs for general purpose use, the command line tools are a set of small, tailored programs which do one job really well. It's the user's job to chain these programs together as needed for their specific problem.

Exercise. The `curl` (*see URL*) command can be used to retrieve a file from the internet. It places the file's contents on `stdout` and prints some network statistics on `stderr`; try `curl www.example.com` for instance. Use `curl`, `grep` and `wc -l` to count how many times the word "Linux" occurs in the Wikipedia article on Linux. Can you find an article which (ignoring upper- and lowercase) mentions its own name more than a thousand times?

3.2.3 Redirecting output to file

Any command's `stdout` can be written to a file instead of displayed using the screen by using the `>` operator.

```
b0036119@caldew:~/demo$ grep -i sword "Tower of Joy.txt"
"When King's Landing fell, Ser Jaime slew your king with a golden sword, and I
wondered where you were." Ned's wraiths moved up beside him, with shadow swords in
hand. They were seven against three.
"And now it begins," said Ser Arthur Dayne, the Sword of the Morning. He unsheathed
Dawn and held it with both hands. The blade was pale as milkglass, alive with light.
b0036119@caldew:~/demo$ grep -i sword "Tower of Joy.txt" > sword.txt
b0036119@caldew:~/demo$ cat sword.txt
```

```
"When King's Landing fell, Ser Jaime slew your king with a golden sword, and I
wondered where you were." Ned's wraiths moved up beside him, with shadow swords in
hand. They were seven against three.
"And now it begins," said Ser Arthur Dayne, the Sword of the Morning. He unsheathed
Dawn and held it with both hands. The blade was pale as milkglass, alive with light.
```

If the target file already exists, it is emptied before being filled with the process' `stdout`.

```
b0036119@caldew:~/demo$ echo I've got a big sword > sword.txt
b0036119@caldew:~/demo$ cat sword.txt
I've got a big sword
```

To change this behaviour, use two greater-than signs `>>` to *append* to the end of the target file.

```
b0036119@caldew:~/demo$ echo \"My liege lord has granted me two swords,\" spoke
Ser Dual of House Wield. >> sword.txt
b0036119@caldew:~/demo$ cat sword.txt
I've got a big sword
"My liege lord has granted me two swords," spoke Ser Dual of House Wield.
```

(The backslashes are needed to send the quote characters to `echo` and stop them from being interpreted by the shell.)

A common trick to silence a program with lots of output is to send it to the special file `/dev/null`, which ignores all of its input. Fortunately, any error messages are preserved.

```
b0036119@caldew:~/demo/programs$ python lots_of_output.py > /dev/null
Fizz
Buzz
Fizz
Fizz
Buzz
Fizz
Fizzbuzz!
[...output repeats for a long time, before ending with an error...]
Traceback (most recent call last):
  File "lots_of_output.py", line 12, in <module>
    raise Exception("Look, standard error isn't redirected")
Exception: Look, standard error isn't redirected
b0036119@caldew:~/demo/programs$ python lots_of_output.py > /dev/null
Traceback (most recent call last):
  File "lots_of_output.py", line 12, in <module>
    raise Exception("Look, standard error isn't redirected")
Exception: Look, standard error isn't redirected
```

Exercise. Another useful trick for a long-running process is to redirect it to a file but view its output in another window. Open two terminals A and B. In terminal A, run `potatoes.py` in the `demo/programs` folder and redirect its output to a file inside `/tmp`. In terminal B, use `tail -f` to view the output file and watch it update in real time. (The `-f` stands for follow.)

Top tip! Similarly you can use `<` to use a file as a process' `stdin`, even if the program won't normally read from files. Here's an example with the `factor` command.

```
b0036119@caldew:/tmp$ cat numbers.dat
5
```

```
6
7
b0036119@caldew:/tmp$ factor numbers.dat
factor: 'numbers.dat'         is not a valid positive integer
b0036119@caldew:/tmp$ factor < numbers.dat
5: 5
6: 2 3
7: 7
```

3.3 Other useful shell functions

Again this section is mostly for reference.

3.3.1 Boolean && operator

As well as piping `|`, another means to chain commands together is the double-ampersand `&&`. This is a binary operator which takes two commands A `&&` B. It runs A, and if it successfully completes returns B. If A terminates with an error, the B command isn't run. For instance, maybe A creates a file that B reads from. If there's a problem with A, rather than running B and getting twice as many error messages, `&&` allows you to stop as soon as something goes wrong.

Here's an example. Maybe I've been writing a C program, and I want to compile and run it in one command. There are two ways to do this: by separating my commands with a semicolon `;` or by using `&&`.

```
gcc adding_in_c.c -o adding ; ./adding
0.1 + 0.2 = 0.3000000119209290
gcc adding_in_c.c -o adding && ./adding
0.1 + 0.2 = 0.3000000119209290
```

No difference this time, as both commands complete successfully. But if I try to compile a file with a syntax error, I'm going to get different results.

```
b0036119@caldew:~/demo/programs$ gcc syntax_error.c -o example; ./example
syntax_error.c: In function 'main':
syntax_error.c:5:2: error: expected ',' or ';' before 'int'
  int c = "blah blah blah;
  ^
syntax_error.c:5:10: warning: missing terminating " character [enabled by default]
  int c = "blah blah blah;
          ^
syntax_error.c:5:2: error: missing terminating " character
  int c = "blah blah blah;
  ^
bash: ./example: No such file or directory
b0036119@caldew:~/demo/programs$ gcc syntax_error.c -o example && ./example
syntax_error.c: In function 'main':
syntax_error.c:5:2: error: expected ',' or ';' before 'int'
  int c = "blah blah blah;
  ^
syntax_error.c:5:10: warning: missing terminating " character [enabled by default]
  int c = "blah blah blah;
          ^
syntax_error.c:5:2: error: missing terminating " character
  int c = "blah blah blah;
  ^
```

3.3.2 The `clear` command

If you've got a terminal full of output and you want to completely discard it so you can concentrate on something else, the `clear` command is your friend. It simply prints

a stream of empty lines to the terminal, so that you have the appearance of a blank workspace. (The history will still be there if you need to scroll back to retrieve it.)

3.3.3 Aliases

If you have to use all sorts of commands all day long, it's easy to forget all the common options and flags. Or maybe you often chain commands together and can't be bothered to re-type all the pipework. Enter aliases: these are a way to define abbreviations that can be used in place of commands.

They're made using the *alias* command, which is a part of the BASH shell; it's not a separate program sitting somewhere on your computer. The syntax is `alias <short>="<long>"`. For example, maybe you want to abbreviate `ls -l` to `ll`.

```
b0036119@caldew:~/demo/programs$ alias ll="ls -l"
b0036119@caldew:~/demo/programs$ ll
total 8
-rw-r--r--+ 1 b0036119 nma 117 Jun 23 17:49 adding_in_c.c
-rw-r--r--+ 1 b0036119 nma 430 Jun 28 11:50 delay.py
-rw-r--r--+ 1 b0036119 nma 130 Jun 28 11:50 hello_world.py
-rw-r--r--+ 1 b0036119 nma 247 Jun 29 13:39 lots_of_output.py
-rw-r--r--+ 1 b0036119 nma 466 Jun 28 12:32 nth_fibonacci.py
-rw-r--r--+ 1 b0036119 nma 246 Jun 28 11:51 potatoes.py
-rw-r--r--+ 1 b0036119 nma 178 Jun 28 11:51 run_forever.py
-rw-r--r--+ 1 b0036119 nma 118 Jun 29 15:21 syntax_error.c
```

If you run `alias` without any arguments, you'll see the current list of aliases sorted alphabetically.

```
b0036119@caldew:~/demo/programs$ alias
alias calc='gnome-calculator'
alias du='du -h --max-depth 1'
alias ll='ls -l'
alias open='xdg-open'
alias terminal='gnome-terminal'
```

It turns out I like abbreviations; but sometimes I just want a simple name for something that's harder to remember.

There's nothing stopping you from making an alias whose name is a pre-existing command; indeed that's exactly what I do for `du` above. In this situation you can bypass the alias by escaping it with a backslash.

```
b0036119@caldew:~/demo$ du
38K         ./easter_egg_hunt
17K         ./jokes
9.5K         ./programs
8.0K         ./poems
8.0K         ./nursery_rhymes
121K         .
b0036119@caldew:~/demo$ \du
3         ./easter_egg_hunt/down_down
[... output truncated ...]
17         ./jokes
10         ./programs
8         ./poems
```

```
8         ./nursery_rhymes
121         .
```

Without the alias, the filesizes were in blocks rather than human readable units; also I saw the sizes of more deeply nested folders. If you suddenly decide that your alias was a horrible idea, use `unalias` to remove it.

```
b0036119@caldew:~/demo$ unalias ll
b0036119@caldew:~/demo$ alias
alias calc='gnome-calculator'
alias du='du -h --max-depth 1'
alias grep='grep --color=auto'
alias open='xdg-open'
alias terminal='gnome-terminal'
```

You can even include pipes if you want: try `alias la="ls -ahl" | less` to see every file in full detail via `less`' scrollable output pager.

Note that an alias only applies to the current terminal. If you want to make it permanent, add it to your user `~/.bashrc` file;[8] any commands present in that file will automatically be executed when you start a new terminal.

Exercise. Make an alias for a command you've been using a lot today that includes your favourite options. For inspiration, my favourite options for `du` are `-h` for *human readable* and `--max-depth=1` so that I can see file sizes one level at a time.

If you're feeling brave, save it in your `~/.bashrc` file if to reuse it in other terminals and sessions.

4 Automation

This section will tie together all of the terminal skills we've seen so far, by explaining how we can use them to automate boring, repetitive but necessary tasks.

4.1 Scheduling commands

If you check the list of processes (Section 5) running on your machine, you should find that there's one called `cron`.

```
b0036119@caldew:~/teaching/linux$ ps -aux | grep cron
root      1221  0.0  0.0  23652   548 ?        Ss   Jun14   0:01 cron
b0036119 23865  0.0  0.0  11992   944 pts/8    SN+  11:13   0:00 grep cron
```

`cron` is a *job scheduler*: a program which periodically runs other programs at certain times of day. Its name comes from *cronos* (*χρόνος*), the Greek word for 'time'. Maybe you have a bunch of log files that need deleting every week, or perhaps you want to make a backup of your files every night. Maybe you want to check how much memory or disk space is available and keep track of this in a log file. If this sounds similar to what you need, *cron* is probably the tool for you.

Maths & Stats only: Backups of the `/home` area and the `/data` drive are made automatically, so there's no need to use `cron` to make your own backup system.

Let's walk through an example. Say I want to record how much space is available on the `/data` drive every 10 minutes. The first thing we should do is work out what

[8] See `http://askubuntu.com/q/17536`.

command we want to run. `df` is the command to use (Section 2.4.2), but it has a lot of output.

```
b0036119@caldew:~$ df -h
Filesystem                          Size  Used Avail Use% Mounted on
none                                3.9G   52M  3.8G   2% /run/shm
none                                100M  412K  100M   1% /run/user
/dev/sdb2                           325G   17G  292G   6% /backup
/dev/sda3                           128G  4.9G  124G   4% /WinData
/dev/sda6                            28G  1.2G   25G   5% /var
/dev/sda7                            28G   44M   26G   1% /opt
/dev/sda9                           321G   17G  288G   6% /data
mas-home-pg:/tank/home/b0036119     2.0G  1.4G  621M  70% /home/b0036119
[... output trimmed ...]
```

If I'm going to stick this in a log, I'd better only extract the information I need. A pipe to `grep` will do the trick.

```
b0036119@caldew:~$ df -h | grep /data$
/dev/sda9               321G   17G  288G   6% /data
```

I can redirect this to a file with `>`; but since I'm making a log I want to *append* with `>>`.

```
b0036119@caldew:/tmp$ df -h | grep /data$ >> data_drive.log
b0036119@caldew:/tmp$ cat data_drive.log
/dev/sda9               321G   17G  288G   6% /data
```

If we're going to do this properly we should make a note of the time and date each time we log. I'll stick with the default `date` format, which is human-readable.

```
b0036119@caldew:/tmp$ date
Thu Jun 30 11:54:00 BST 2016
```

I'm almost done building my command for `cron` now.

```
b0036119@caldew:/tmp$ rm data_drive.log
b0036119@caldew:/tmp$ date >> data_drive.log && df -h | grep /data$ >> data_drive.log
b0036119@caldew:/tmp$ cat data_drive.log
Thu Jun 30 12:00:02 BST 2016
/dev/sda9                                321G   17G  288G   6% /data
```

In Maths & Stats, commands run by `cron` use your home directory as their working directory. This is a setting that can be changed by system administrators though, so I'd recommend you use one of two approaches to be explicit about where you want to run your commands.

- Always use absolute paths. Don't forget that you can use `~` as a shortcut your home area.
- Include a `cd` command. So if I wanted to run a command C inside `~/logs` I could use `cd ~/logs && C`. If for some reason the `logs` directory doesn't exist, the `&&` operator will raise an error, rather than running C in the wrong place.

Thus the command I want to use is

```
cd ~ && date >> data_drive.log && df -h | grep /data$ >> data_drive.log
```

4.1.1 Specifying the runtime

The list of jobs is stored in 'cron table', or `crontab` for short. To edit this list use the *crontab -e* command. This opens a text file in an editor, in which you enter a cron If you've never used `crontab` before, this file will be empty (or possibly consist of comments after a `#` symbol.)

Top tip! According to `man crontab`, the editor is determined by the environment variables `$VISUAL` and `$EDITOR`. (We're not going to cover environment variables, so I'm going to have to defer to the INTERNET this time.) On both my machine and the Computer Science cluster, the default editor was `vim` (see Section 2.3.3), which can be intimidating for novice users. You can override this in your current terminal session by writing

```
b0036119@caldew:/tmp$ export VISUAL="gedit --new-window --wait"
```

To check this worked correctly, use `echo`:

```
b0036119@caldew:/tmp$ echo $VISUAL
gedit --new-window
```

Then you should be able to edit the `crontab` in `gedit`. The table will be updated when the new `gedit` window closes.

Each non-commented line describes one `cron` command. The format is

```
m h D M Dw C
```

where the symbols stand for the following data.

- m is a number in the range 0–59 specifying a minute,
- h is a number in the range 0–24 specifying an hour,
- D is a number in the range 1–31 specifying an day of the month,
- M is a number in the range 1–12 specifying a month,
- D_w is a number in the range 1–7 specifying a day of the month: 1 for Monday, 2 for Tuesday, ..., through to 7 for Sunday.
- C is the command you want to execute.

Each of the date and time fields can be extended to more than a single value in four ways:

- Use an asterisk `*` to stand for 'anything';
- Use a hyphen `-` to indicate a range;
- Use a comma `,` to stand for 'and'.
- Use a slash `/`n to stand for 'divisible by n'.

See Table 1 for some examples.

Exercise. Use `cron` to write an encouraging or motivational message to a file in your home folder at a time interval of your choice. For bonus points, write two more encouraging or motivational messages, and use the `shuf` command to randomly select one every time the `cron`job runs. For bonus bonus points, can you find a way to print your messages to screen when the `cron`job runs? (Try `wall`, but use it responsibly!)

Important: Before the session ends, remove any `cron` jobs by running `crontab /dev/null`. (Sub-exercise: why does this work?) We're kindly using Computer Science's cluster machines, and since we're only going to use them today we shouldn't leave anything running periodically!

Time specification					Meaning
30	10	*	*	*	10:30 every day
00,15,30,45	*	*	*	*	every fifteen minutes
*/15	*	*	*	*	every fifteen minutes (shorthand)
0	0	*	*	7	midnight every Sunday
0	*/4	*	*	1-5	every four hours Monday to Friday
0	15	25	12	*	3pm Christmas Day
0	0	1	*	*	midnight on the first day of the month
00,20, 40	9-17	*	*	1-3,5	every twenty minutes 9am–5pm on weekdays except Thursday

Table 1: Examples of the `crontab` date/time syntax.

Top tip! Because `cron` isn't running in a terminal, there's no way to feed your commands `stdin`put nor to display any `stdout`put or `stderr`ors. As as result, your commands need to get all of their inputs from their arguments. Any output and errors are sent to you via the system's internal email. (It's not sent over the internet like regular email; but nonetheless it's an electronic message.)

If `cron` has emailed output to you, you may notice a 'you have new mail' message when logging in remotely.

```
b0036119@caldew:/tmp$ ssh caldew
b0036119@caldew's password:
Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 3.13.0-88-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

9 packages can be updated.
4 updates are security updates.

*** System restart required ***
You have new mail.
```

Maths and Stats users can use `mutt` on the command line to read this mail; I'm not sure about Fedora.

4.2 Bash shell scripting

A *shell script* is a text file containing a list of shell commands; one on each line. If there's a complicated operation you need to do frequently involving ten commands, say, it's easier to run one script than run those ten commands each time. Perhaps you want to write a loop to execute a command with different arguments, or to only run some commands if some condition holds. Or you might want to repeatedly run a simulation with different parameters. In other words, maybe you want to do some basic programming. Shell scripts allow you to do all this and more.

Let's see an example of a shell script and go through it line-by-line.

```
b0036119@caldew:~/demo/scripts$ cat simple_shell_script.sh
#! /bin/bash
echo "Script beginning in $(pwd)"
sleep 3
echo "Script finished"
```

The first line is a 'shebang line' (ha*sh* for `#` and *bang* for `!`). This indicates that the file's name is meant to be passed as an argument to `/bin/bash`, which is where the

shell program is conventionally located.[9] Strictly speaking, for maximum compatibility scripts should be marked as running under `/bin/sh`, but `bash` will almost always be available and will make writing shell scripts easier.

Maths & Stats only: The default user shell in Maths & Stats is `tcsh`. You can change this system-wide by talking to a computing officer; change this on your machine by editing your terminal settings. If you just want to play around, just run `bash` in whatever shell you're using.

Once the shell script has been marked as executable with `chmod`, we can use the `./` notation (Section 2.3.5) to run the script.

```
b0036119@caldew:~/demo/scripts$ chmod +x simple_shell_script.sh
b0036119@caldew:~/demo/scripts$ ./simple_shell_script.sh
Script beginning in /home/b0036119/demo/scripts
Script finished
```

The result is the same as if we had called `sh` directly from disk:

```
b0036119@caldew:~/demo/scripts$ /bin/bash ~/demo/scripts/simple_shell_script.sh
Script beginning in /home/b0036119/demo/scripts
Script finished
```

Every other line is an instruction that gets sent to the shell, just like all the commands you've been typing into the terminal. Unlike the normal 'interactive mode' in the terminal, the script doesn't wait for input from you or me—it just executes commands one at a time, as quickly as possible. `sleep` is a new command which takes one argument, a number n. It does nothing for n seconds and then terminates. It's just here as a dummy command to give the impression that the script's doing something.

4.2.1 Substitution

We've used `echo` many times despite never formally introducing it; it prints its arguments to `stdout`, which is why the script has output displayed on screen. There is something new here though: the notation `$(pwd)` which indicates *command substitution*. The shell script spawns a child `sh` process, uses it to run `pwd` and keeps its output (`stdout`). Once the child has finished, control returns to the original `sh` running the script. The substring "`$(pwd)`" will be replaced with the output of the child process.

Here's another example of command substitution.

```
b0036119@caldew:~/demo/scripts$ cat num_files_pwd.sh
#! /bin/bash
echo "Number of files in $(pwd) is $(find -type f | wc -l)."
echo "There are also $(find -type d -not -name . | wc -l) directories".
```

Pipes work just fine! In fact we can even include substitute files' contents using `cat`:

```
b0036119@caldew:~/demo/scripts$ cat ./repeated_sed.sh
#! /bin/bash
echo "This text file contains the letter 't'." > example.file
sed -i "s/t/tt/g" example.file
sed -i "s/t/tt/g" example.file
sed -i "s/t/tt/g" example.file
echo "The result is '$(cat example.file)'."
```

We can do basic arithmetic on integers using two pairs of brackets. This is called *arithmetic expansion*.

[9] The Python files in `demo/programs` have their own shebang lines, although they have a slightly different format.

```
b0036119@caldew:~/demo/scripts$ echo $((2 + 2))
4
b0036119@caldew:~/demo/scripts$ echo $((2 ^ 10))
8
b0036119@caldew:~/demo/scripts$ echo $((2 ** 10))
1024
b0036119@caldew:~/demo/scripts$ echo $((1 / 0))
b0036119@caldew:~/demo/scripts$ echo $((5 / 2))
2
b0036119@caldew:~/demo/scripts$ echo $((-5/2))
-2
```

Note that the power operation is `**` (`^` denotes XOR) and that integer division is rounded towards zero. If you want to work with decimal (floating-point) numbers, take a look at the `bc` (*basic calculator*) command.

```
b0036119@caldew:~/teaching/linux$ echo "0.1 + 0.2" | bc -q
.3
```

4.2.2 Handling arguments

`sh` is just like any program: it accepts a variable-length list of arguments and does something with them. Scripts can access those arguments, so that they themselves act like an ordinary program. These are accessible via the variables `$1, $2, ...`;[10] missing variables are passed as an empty string `''`. There's also `$0`, which is the name of the script currently being executed. Let's see an example.

```
b0036119@caldew:~/demo/scripts$ cat num_files_general.sh
#! /bin/bash
echo "Running $0 with argument '$1'"
echo "Number of files in $(pwd) is $(find $1 -type f | wc -l)."
echo "There are also $(find $1 -type d -not -name . | wc -l) directories".

b0036119@caldew:~/demo/scripts$ ./num_files_general.sh
Running ./num_files_general.sh with argument ''
Number of files in /home/b0036119/demo/scripts is 5.
There are also 0 directories.

b0036119@caldew:~/demo/scripts$ ./num_files_general.sh ../
Running ./num_files_general.sh with argument '../'
Number of files in /home/b0036119/demo/scripts is 50.
There are also 19 directories.
```

This worked out quite nicely. If I don't pass an argument to the script then `$1` is the empty string. In this situation, the subcommands `find` do *not* get passed an empty string as an argument; it's as if I never wrote `$1` in the first place.

> **Exercise.** Write a shell script that takes the name of two files F_1 and F_2 and prints out the number total number of lines in both files.

4.2.3 Conditionals

The syntax for `if` statements takes the general form:

```
if [ <condition> ]
then
```

[10] For indices greater than 9 enclose them in braces: `${10}, ${11}, ...`

```
        <statement 1>
        ...
        <statement n>
fi
```

The spaces after the `[` and before the `]` are important. The `[` is effectively a *command* that absorbs all arguments up to the next `]` and determines if they evaluate to True or False. It's equivalent to the `test` command, except `test` doesn't need a closing square bracket. In fact, `test` came first and the alternate name `[` came later as a convenience.

A `<condition>` can be built out of a number of boolean expressions combined together with various operators. See `man test` for a full list. We'll just mention a few commonly used examples.

`[ -f FILE ]` tests to see if the given FILE exists.

`[ -d DIR ]` tests to see if the given DIRectory exists.

`[` I_1 `-eq` I_2 `]` tests to see if the integers I_1 I_2 are equal, i.e. `-eq` stands for $=$. Similarly you can use `-ne`, `-gt`, `-ge`, `-lt` and `-le` for $\neq$, $>$, $\geq$, $<$ and $\leq$.

`[` A `-a` B `]` tests if both A and B are true.

`[` A `-o` B `]` tests if A or B or both are true.

`[ !` A `]` tests if A is false.

`[` S `==` T `]` tests if the strings S and T are equal.

`[` S `!=` T `]` tests if the strings S and T are not equal.

> **Exercise.** Why can't we use $>$, $\geq$, $<$ and $\leq$ for mathematical comparisons?

Like with many programming languages, there are `elif` (*else if*) and `else` statements that let you build more complicated chains of condition handling. For instance, you can say:

```
if [ -d build ]
then
        rm -r build
elif [ $1 == "party" ]
then
        echo "Let's have a PARTAY"
else
        #nothing to delete
        echo "Build directory doesn't exist"
fi
```

Note that hashes `#` denote comments.

> **Exercise.** Write a shell script that accepts one argument. If that argument is the name of a directory D in the working directory, select a file from D at random and print its `head`. If not, and there is a file called D in the working directory, print its `head`. Otherwise, or if no argument is provided, print a suitable error message.

4.2.4 Loops

Just like your favourite programming language, shell scripts support `for` loops,`while` loops and `until` loops; we'll just cover the former. The syntax is

```
for <variable> in <list>
do
        <statement_1>
        ...
        <statement_n>
done
```

The `<list>` is a space separated list of values you want the `<variable>` to loop through. To access the value of `<variable>` in the `<statement>`s, prefix it with a dollar `$`. For instance:

```
b0036119@caldew:~/demo/scripts$ cat counting_loop.sh
#!/bin/bash
for i in 1 2 3 4 5
do
        echo "The value of i is $i"
done
b0036119@caldew:~/demo/scripts$ ./counting_loop.sh
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
```

For a more general sequence, I need to substitute the output of the `seq` (*sequence*) command.

```
b0036119@caldew:~/demo/scripts$ cat counting_loop_2.sh
#!/bin/bash
for i in $(seq 3 4 20)
do
        echo "The value of i is $i"
done
b0036119@caldew:~/demo/scripts$ ./counting_loop_2.sh
The value of i is 3
The value of i is 7
The value of i is 11
The value of i is 15
The value of i is 19
```

I don't just have to loop over integers either.

```
b0036119@caldew:~/demo/scripts$ cat string_loop.sh
#!/bin/bash
for name in world sailor stranger $USER "Barrack Obama" you
do
        echo "Hello, $name!"
done
b0036119@caldew:~/demo/scripts$ ./string_loop.sh
Hello, world!
```

```
Hello, sailor!
Hello, stranger!
Hello, b0036119!
Hello, Barrack Obama!
Hello, you!
```

This explains why I needed the `$()` in `counting_loop_2.sh`.

```
b0036119@caldew:~/demo/scripts$ cat counting_loop_2_error.sh
#!/bin/bash
for i in seq 3 4 20
do
        echo "The value of i is $i"
done
b0036119@caldew:~/demo/scripts$ ./counting_loop_2_error.sh
The value of i is seq
The value of i is 3
The value of i is 4
The value of i is 20
```

Top tip! Just like most programming languages, you can use the `break` and `continue` keywords to exit a loop or skip to the next iteration.

There is a special shorthand to loop over the script's arguments, namely `"$@"`.

```
b0036119@caldew:~/demo/scripts$ cat echo_arguments.sh
#! /bin/bash
echo "Here are the arguments I received:"
for arg in "$@"
do
        echo "\"$arg\""
done
b0036119@caldew:~/demo/scripts$ ./echo_arguments.sh a b c "d e f"
Here are the arguments I received:
"a"
"b"
"c"
"d e f"
b0036119@caldew:~/demo/scripts$ ./echo_arguments.sh
Here are the arguments I received:
```

The quotes are necessary to handle spaces in arguments correctly. If I leave the quotes out, as in `echo_arguments_error.sh` I get the following.

```
b0036119@caldew:~/demo/scripts$ ./echo_arguments_error.sh a b c "d e f"
Here are the arguments I received:
"a"
"b"
"c"
"d"
"e"
"f"
```

Exercise. Write a shell script that takes the names of *an arbitrary number* of files $F_1, F_2, \ldots$ and prints out the number total number of lines in all files. Include a suitable error message if an arguments isn't the name of a file!

I haven't explained how to use variables, so here's a quick intro. You can make a variable called `sum` with value `0` by writing `sum=0`; don't include spaces around the `=` or the equals sign will be interpreted as an argument to pass to a `sum` command. You can increment `sum` by assigning an arithmetic expression; for instance:

```
sum=$(($sum + 2))
```

4.2.5 Glob patterns

We've seen how the shell expands substrings starting with `$`; but it does more pattern-matching according to a `glob` pattern. There are four special characters in such a pattern:

`*` matches any string of characters, including the empty string.

`?` matches exactly one character.

`[...]` matches any character contained within the braces. You can use a hyphen to specify a range too.

This is similar to, but not the same as regular expression syntax. In particular `*` and `?` take on more general meanings in a regular expression.

Let's have some examples.

```
b0036119@caldew:~$ ls
archive  bin      Downloads      parallel  robertson  temp.pdf
Audio    demo     Dropbox        Pictures  scripts    thompsons_v
barker   Desktop  Metro_Map.pdf  R         teaching   website
b0036119@caldew:~$ echo d*
demo
b0036119@caldew:~$ echo D*
Desktop Downloads Dropbox
b0036119@caldew:~$ echo [dD]*
demo Desktop Downloads Dropbox
b0036119@caldew:~$ echo *.pdf
Metro_Map.pdf temp.pdf
b0036119@caldew:~$ echo ???????
archive Desktop Dropbox scripts website
b0036119@caldew:~$ echo ??
??
```

Notice that in the last example there was no match, so the pattern `??` was *not* expanded by the shell.

The reason I'm including this section here is that they can be very handy when writing `for` loops. For instance, here's a script which compiles all LaTeX files within current directory. Note that `$?` contains the return code of the last-run command.

```
#! /bin/bash
for tex_file in *.tex
do
        echo -n "Compiling $tex_file... "
        pdflatex -interaction=batchmode -synctex=1 $tex_file > /dev/null
        if [ $? -eq 0 ]
        then
                echo "✓"
        else
```

```
                echo "✗"
        fi
done
```

4.3 Further reading

I haven't mentioned the use of 'environment' or ordinary variables in shell scripts; nor have I explained how to define and use functions. These can be very powerful tools to have at your disposal!

The `make` command is a more advanced system for automation, used by many software projects to automate the process of building from source code. It allows you to specify a hierarchy of named 'targets' as a list of shell commands. For instance, if I was preparing a `Makefile` to help compile a LaTeX document I might have three 'targets': `make draft` to quickly check that I haven't made an error; `make final` to compile the document for 'final' publication, and `make clean` to remove the auxiliary files created by the `latex` executable.

Take a look at the `make` documentation for more info, or at for a terse introduction.

5 Process control

This section gives a brief overview of how you'd monitor and interact with multiple processes that are being run.

5.1 Background and foreground

While running any command in a terminal, you can press `Control-Z` to pause it and return to the terminal prompt.[11]

```
b0036119@caldew:~/teaching/linux$ python3
Python 3.4.3 (default, Oct 14 2015, 20:28:29)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
^Z
[1]+  Stopped                 python3
```

The three arrows `>>>` are the default Python prompt, just like how the shell uses a dollar `$` as a prompt. The caret `^Z` is the terminal's notation to say that I pressed the `Control` key with `Z`. The `[1]` indicates that `python3` is the first in the list of processes running in or paused by this terminal. The word "stopped" is a little dramatic; I prefer to think of such processes as "suspended." we can restart `python3` by typing `fg` to bring the process to the *foreground*.

```
b0036119@caldew:~/teaching/linux$ fg
python3
print('hello')
print('hello')
hello
>>> x = 2
>>> x
2
>>> quit()
```

[11] This is achieved by sending a `SIGTSTP` signal to the process; see `https://en.wikipedia.org/wiki/Unix_signal`.

Alternatively if I stop a process, I can allow it to continue in the *background* by typing `bg`. This is the same as starting the process and putting an `&` after its name.

```
b0036119@caldew:~/~$ firefox
^Z
[1]+  Stopped                 firefox
b0036119@caldew:~/~$ bg
[1]+ firefox &
b0036119@caldew:~/~$ 1467381921128 addons.xpi
WARN        Bootstrap state is invalid (missing add-ons:
/home/b0036119/.mozilla/firefox/qryzz7h5.default/extensions/
firefox-hotfix@mozilla.org.xpi)
```

Note that processes running in the background will continue to print their `stdout` and `stderr` to the terminal. Despite the gibberish on screen, the shell is still happy to accept commands.

Let's try the `jobs` command.

```
b0036119@caldew:~/~$ jobs
[1]+  Running                 firefox &
```

This prints the list of programs you've started from or paused in the current terminal, together with a job number. If I start some more commands and pause them, the list gets longer.

```
b0036119@caldew:~/~$ python3 &
[2] 15666
b0036119@caldew:~/~$ Python 3.4.3 (default, Oct 14 2015, 20:28:29)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
jobs
[1]-  Running                 firefox &
[2]+  Stopped                 python3
```

I'm going to start a new dummy job which does nothing, suspend it, and then resume running it in the background.

```
b0036119@caldew:~/~$ sleep 100
^Z
[3]+  Stopped                 sleep 100
b0036119@caldew:~/~$ jobs
[1]   Running                 firefox &
[2]-  Stopped                 python3
[3]+  Stopped                 sleep 100
b0036119@caldew:~/~$ bg %3
[3]+ sleep 100 &
```

The `%` stands for 'job number', so I asked the shell to put job number 3 in the background. Commands that interact with other processes like `fg`, `bg` and `kill` know how to interpret the `%` notation.

After 100 seconds, the `sleep` command is done. The next time I run a command, the shell will notify me of this and it will no longer be available in the jobs list.

```
b0036119@caldew:~/~$ echo "blah blah blah"
blah blah blah
[3]-  Done                    sleep 100
b0036119@caldew:~/~$ jobs
[1]-  Running                 firefox &
[2]+  Stopped                 python3
```

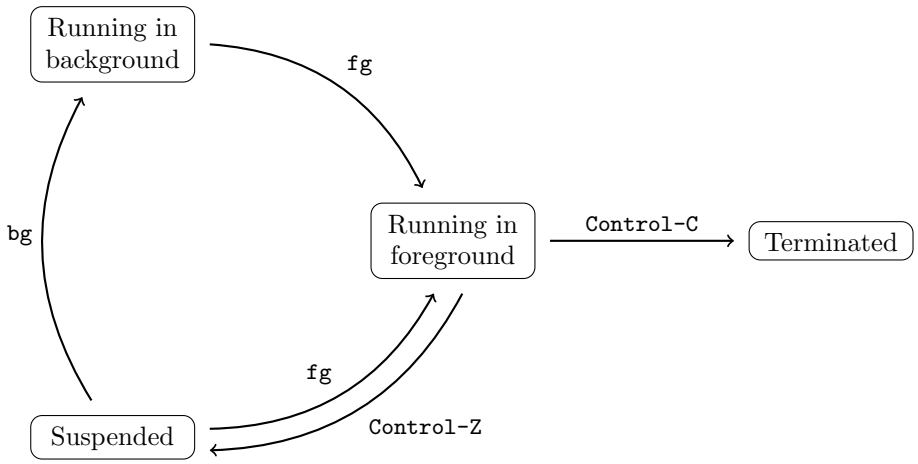

Figure 22: The different states that a job can be in, and how to navigate between them.

If I want to suspend job 1, I can bring it to the foreground and then suspend with `Control-Z`.

```
b0036119@caldew:~$ fg %1
firefox
^Z
[1]+  Stopped                 firefox
b0036119@caldew:~$ jobs
[1]+  Stopped                 firefox
[2]-  Stopped                 python3
b0036119@caldew:~$
```

What about if I want to permanently stop a process? I can bring it to the foreground with `Control-C`, or else use the `kill` command.

```
b0036119@caldew:~$ fg %1
firefox
^C
b0036119@caldew:~$ jobs
[2]+  Stopped                 python3
b0036119@caldew:~$ kill %2
b0036119@caldew:~$ jobs
[2]+  Terminated              python3
```

Top tip! When your shell closes, any processes it has started are terminated. This isn't always what you want! For instance, you might want to start a simulation running on a remote machine, close the connection and come back later to see the results.

To avoid the auto-termination, you can run your command through `nohup` (*no hangup*); this ignores the terminate signal that gets sent when the shell is closed. Consult a `man` page or WIKIPEDIA for more information.

5.2 Monitoring

The single most important command for keeping an eye on your programs is `top` (*table of processes*).

```
b0036119@caldew:~$ top
```

```
top - 17:24:06 up 17 days,  6:27,  6 users,  load average: 0.10, 0.18, 0.18
Tasks: 284 total,   1 running, 279 sleeping,   3 stopped,   1 zombie
%Cpu(s):  0.7 us,  0.4 sy,  1.3 ni, 97.6 id,  0.1 wa,  0.0 hi,  0.0 si,  0.0 st

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1611 root      20   0  724924 210292 179748 S   9.0  2.6   1986:13 Xorg
 5277 b0036119  32  12  587048  41880   9184 S   4.3  0.5   1288:05 gnome-system-mo
 9403 b0036119  32  12 1816160 248424  39264 S   4.3  3.1  47:58.30 texmaker
 5045 b0036119  32  12 1764392 222184  23964 S   1.7  2.8 691:27.52 compiz
 4773 b0036119  28   8  394124  68844   1700 S   0.7  0.9  36:30.36 ibus-daemon
 5193 b0036119  24   4  889168  38620  15956 S   0.3  0.5   7:27.20 gnome-terminal
20011 b0036119  35  15   27412   1984   1256 S   0.3  0.0  12:54.16 top
23096 b0036119  32  12 1696112 359212  60416 S   0.3  4.4   6:31.35 chromium-browse
    1 root      20   0   34192   2376    912 S   0.0  0.0   0:03.25 init
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    7 root      20   0       0      0      0 S   0.0  0.0   7:23.37 rcu_sched
```

By default, processes are sorted by CPU usage and the table is updated every 5 seconds. It's useful to get a quick overview of what programs are using the most resources.

Sometimes you want a static list of processes, e.g. for use in a log or to be consumed by another program. The `ps` (*process status*) is useful for this. By default it prints out a list of commands that you've launched or suspended in the current terminal.

```
b0036119@caldew:~$ ps
  PID TTY          TIME CMD
18309 pts/22   00:00:00 ps
27817 pts/22   00:00:02 bash
```

We can ask it to display more though: `ps u` shows you all commands you've started in any terminal.

```
b0036119@caldew:/tmp$ ps u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
b0036119  5260  0.0  0.0  24132   556 pts/5    Ss   Jun14   0:00 bash
b0036119  5383  0.0  0.0 216984  6708 pts/5    S+   Jun14   0:00 /usr/bin/python3
b0036119 13462  0.0  0.0  25216  6372 pts/8    SNs  Jun29   0:00 bash
b0036119 18346  0.1  0.0  24124  5016 pts/21   SNs  17:31   0:00 bash
b0036119 18377  0.0  0.0  48336  3892 pts/21   SN+  17:31   0:00 ssh hardy
b0036119 18383  0.2  0.0  24124  5100 pts/22   SNs+ 17:31   0:00 bash
b0036119 18401  0.2  0.0  32816  6788 pts/22   TN   17:32   0:00 python3
b0036119 18481  0.0  0.0  18692  1380 pts/8    RN+  17:32   0:00 ps u
b0036119 20011  0.2  0.0  27412  1984 tty1     SN+  Jun28  12:55 top
b0036119 29956  0.0  0.0  26356  4800 tty1     S    Jun17   0:00 -bash
```

If you're feeling really extreme, try `ps aux` to see every process running on your machine, including those which weren't spawned by you directly! Processes are allowed to spawn other child processes, so this last output will be very long; as I write this I have 281 processes running on my machine. You can see the parent/child hierarchy of processes by using `pstree`; unfortunately the output is too long to include here.

Finally let me mention the `free` command.

```
b0036119@caldew:/tmp$ free -h
             total       used       free     shared    buffers     cached
Mem:          7.7G       7.1G       657M       842M       711M       2.7G
-/+ buffers/cache:       3.7G       4.0G
Swap:          14G       383M        14G
```

It reports how much memory is being used by the applications on my machine. The key number is the second number under 'free': `4.0G` in this case. I have 4 Gigabytes of RAM to play with before things will start to get sluggish.

The 'swap' refers to an area of hard disk used as 'overflow RAM'. If I run out of memory, the kernel will start to swap unused bits of RAM onto the disk leaving more

memory for me to play with. Unfortunately, hard disks are significantly slower than RAM for data access and storage, so we don't want to use swap memory unless we have to. (Windows has a similar system called the 'pagefile'.)

> **Exercise.** Use `top`, `ps`, or `pstree` to investigate the processes running on your machine. Which do you recognise? Which do you not recognise but sound important? Which is using the most CPU power, and which is using the most memory? Which has been running the longest/shortest?

5.2.1 Terminating rogue processes

Every process has an ID number which allows it to be uniquely identified—this is the `PID` column in the output of `top` and `ps`. For instance, the first program started by the kernel is `init` and conventionally has process ID (PID) number 1. `init` is responsible for starting and stopping other processes, which is why it sits at the root of the `pstree`.

You can use the ominous-sounding `kill` command to send signals to processes using their process ID. (We've previously used `kill` with the `%`n notation to refer to job n.)

```
b0036119@caldew:~$ ps
  PID TTY          TIME CMD
13462 pts/8    00:00:00 bash
18852 pts/8    00:00:00 sleep
18862 pts/8    00:00:00 ps
b0036119@caldew:~$ jobs -l
[1]+ 18852 Running                 sleep 100000 &
b0036119@caldew:~$ kill 18852
b0036119@caldew:~$ jobs
[1]+  Terminated              sleep 100000
b0036119@caldew:~$ ps
  PID TTY          TIME CMD
13462 pts/8    00:00:00 bash
18868 pts/8    00:00:00 ps
```

The default signal that gets sent is `SIGTERM`, which politely asks the program to stop what it's doing. There are less polite ways to do this that can't be ignored by the target process, but I'll let you read `man` pages to find out more about that.

> **Exercise.** Run all the python scripts in the `demo/programs` folder. Terminate those which take longer than a minute to run.

I'll quickly mention the `killall` command, which allows you to `kill` commands by name (and other things).

6 Summary

Hopefully this document should have given you a rough idea of:

- What Linux is, a little bit of its history and why it's a useful tool;
- How to get around one of its graphical desktops;

- Using a command-line interface as a means to interact with a computer;
- Tips and tricks to use commands in a terminal more efficiently;
- How to automate the boring stuff that needs to be done repeatedly; and
- How to manage your multitasking with process control.

Any comments, complaints, corrections or queries are very welcome. Send them to `d.m.robertson@ncl.ac.uk`.

* * *

Unix/Linux Command Reference

FOSSwire.com

File Commands

ls – directory listing
ls -al – formatted listing with hidden files
cd *dir* - change directory to *dir*
cd – change to home
pwd – show current directory
mkdir *dir* – create a directory *dir*
rm *file* – delete *file*
rm -r *dir* – delete directory *dir*
rm -f *file* – force remove *file*
rm -rf *dir* – force remove directory *dir* *
cp *file1 file2* – copy *file1* to *file2*
cp -r *dir1 dir2* – copy *dir1* to *dir2*; create *dir2* if it doesn't exist
mv *file1 file2* – rename or move *file1* to *file2*
if *file2* is an existing directory, moves *file1* into directory *file2*
ln -s *file link* – create symbolic link *link* to *file*
touch *file* – create or update *file*
cat > *file* – places standard input into *file*
more *file* – output the contents of *file*
head *file* – output the first 10 lines of *file*
tail *file* – output the last 10 lines of *file*
tail -f *file* – output the contents of *file* as it grows, starting with the last 10 lines

Process Management

ps – display your currently active processes
top – display all running processes
kill *pid* – kill process id *pid*
killall *proc* – kill all processes named *proc* *
bg – lists stopped or background jobs; resume a stopped job in the background
fg – brings the most recent job to foreground
fg *n* – brings job *n* to the foreground

File Permissions

chmod *octal file* – change the permissions of *file* to *octal*, which can be found separately for user, group, and world by adding:

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

Examples:
chmod 777 – read, write, execute for all
chmod 755 – rwx for owner, rx for group and world
For more options, see **man chmod**.

SSH

ssh *user@host* – connect to *host* as *user*
ssh -p *port user@host* – connect to *host* on port *port* as *user*
ssh-copy-id *user@host* – add your key to *host* for *user* to enable a keyed or passwordless login

Searching

grep *pattern files* – search for *pattern* in *files*
grep -r *pattern dir* – search recursively for *pattern* in *dir*
command* | grep *pattern – search for *pattern* in the output of *command*
locate *file* – find all instances of *file*

System Info

date – show the current date and time
cal – show this month's calendar
uptime – show current uptime
w – display who is online
whoami – who you are logged in as
finger *user* – display information about *user*
uname -a – show kernel information
cat /proc/cpuinfo – cpu information
cat /proc/meminfo – memory information
man *command* – show the manual for *command*
df – show disk usage
du – show directory space usage
free – show memory and swap usage
whereis *app* – show possible locations of *app*
which *app* – show which *app* will be run by default

Compression

tar cf *file.tar files* – create a tar named *file.tar* containing *files*
tar xf *file.tar* – extract the files from *file.tar*
tar czf *file.tar.gz files* – create a tar with Gzip compression
tar xzf *file.tar.gz* – extract a tar using Gzip
tar cjf *file.tar.bz2* – create a tar with Bzip2 compression
tar xjf *file.tar.bz2* – extract a tar using Bzip2
gzip *file* – compresses *file* and renames it to *file.gz*
gzip -d *file.gz* – decompresses *file.gz* back to *file*

Network

ping *host* – ping *host* and output results
whois *domain* – get whois information for *domain*
dig *domain* – get DNS information for *domain*
dig -x *host* – reverse lookup *host*
wget *file* – download *file*
wget -c *file* – continue a stopped download

Installation

Install from source:
./configure
make
make install
dpkg -i *pkg.deb* – install a package (Debian)
rpm -Uvh *pkg.rpm* – install a package (RPM)

Shortcuts

Ctrl+C – halts the current command
Ctrl+Z – stops the current command, resume with **fg** in the foreground or **bg** in the background
Ctrl+D – log out of current session, similar to **exit**
Ctrl+W – erases one word in the current line
Ctrl+U – erases the whole line
Ctrl+R – type to bring up a recent command
!! - repeats the last command
exit – log out of current session

* use with extreme caution.