Introduction to the unix command line

1 Introduction to Unix

1.1 What is Unix?

Nearly all working Astronomers use a Unix-based operating system such as Linux or Mac OS. What defines "Unix" is a collection of very powerful command-line tools, a certain layout of files and directories, and some additional aspects like permissions on files (the latter is the main reason why you don't get viruses in Linux or Mac OS). It is highly recommended that you familiarise yourself with the Unix environment if you are at all interested in astrophysics.

In this tutorial we will cover the basics of what you need for scientific computing. We will learn how to use the Unix shell, write a simple 'shell script' and write a 'hello world' program in the three main scientific programming languages.

1.2 The terminal

The unix command prompt or 'shell' is what you get when you open a 'terminal'. In Mac OS this is an application called 'terminal'. In Ubuntu you can find the terminal application from the main menu. Open this, and you should find something like this:

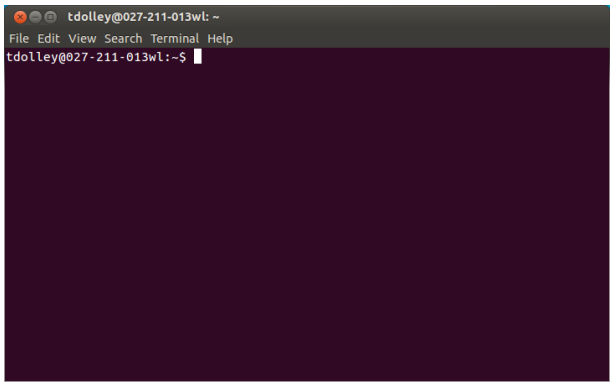

Figure 1: The terminal

In the example above the $ is the *command prompt*. This is where you type commands.

1.3 Simple commands and the Unix shell

Type a simple command:

```
$ ls
```

this should give you a list of files in the current directory. Check what directory you're in using `pwd`:

```
$ pwd
/Users/dprice
```

Next, let's work out what shell we are using. First, we need the `echo` command, which just echoes back anything you type:

```
$ echo hello world
hello world
```

The useful thing is that we can define variables. Try the following:

```
$ NAME=Daniel
$ echo hello $NAME
hello Daniel
```

Certain variable are already set and used by the operating system. These are called *environment variables*. For example, type the following:

```
$ echo $USER
dprice
$ echo $HOME
/Users/dprice
```

You can get a full list of all the system-set variables by typing `printenv`.

Work out what 'unix shell' you are using by echo-ing the `SHELL` variable:

```
$ echo $SHELL
/bin/bash
```

Most modern Linux distributions use the "bash" shell interpreter by default (it stands for "Bourne Again Shell", replacing an earlier version which was just the "Bourne Shell"). `/bin/bash` is just the location of the executable file that runs when you open the terminal. Certain commands are the same across all shell environments, but there are important differences also (the "c-shell" is another popular shell with quite different syntax). I strongly recommend working through any of the numerous Bash tutorials on the web in your own time. We will only cover the basics here.

1.4 Basic file and directory operations

1.4.1 Command options

We have already discovered the `ls` command. You can also give options to commands, for example: *(Note: use minus-the-letter-l, not minus one)*

```
$ ls -l
```

This lists in 'long form' details about the files in the current directory. For any command you can find the full list of options with the `man` command (for 'manual'):

```
$ man ls
```

Experiment with a few of the options for the ls command listed in the manual.

IMPORTANT: You can retrieve previous commands in your history by using the up arrow key. This avoids the need to re-type commands you have are only modifying slightly.

1.4.2 Making, navigating and deleting directories

Next we need to make directories and navigate through them. Make a new directory called unix_tutorial:

```
$ mkdir unix_tutorial
```

Check that this succeeded by typing `ls` — you should see your new directory in the list. Use the `cd` command to change directory and `pwd` to confirm the directory we are in:

```
$ cd unix_tutorial
$ pwd
```

Importantly, most Unix shells have a feature called *tab completion*. So you don't need to type long directory names in full. Let's go back a directory and try again:

```
$ cd ..
$ pwd
$ cd unix[tab]
```

where `[tab]` means *press the tab key*. **You should get used to using tab completion everywhere**. Note the special directory names — a double dot `..` means "one directory back" while a single dot `.` means "the current directory". Hence the following command should have no effect:

```
$ cd .
```

(type `pwd` to confirm this). You can also change back to your home directory at any time by supplying no argument at all:

```
$ cd
$ pwd
/Users/dprice
```

Deleting directories is with the `rmdir` command. However, the directory *must* be empty first. In our case this should work:

```
$ rmdir uni[tab]
```

Make a new directory called `unix_tutorial` and change into it.

1.4.3 Working with simple text files

In Unix, nearly everything can be treated as a 'file'. Programs are just 'executable files', external devices are just 'files that receive input', and so on. The concept of 'text files' barely even exists in Windows (except as something you might open in Notepad) but these are the basic building blocks of Unix. As an example, let's create a file called `output` that contains the output of the `echo` command:

```
$ echo hello world > output
```

The `>` means 'put the output of the command into...'. **Check that a new file called 'output' exists using the `ls` command**. You can look at the contents of a file using the `cat`, `more` or `less` commands:

```
$ cat output
hello world
$ more output
hello world
$ less output
hello world
```

`more` or `less` mostly do the same thing, since `less` was written to replace and update the `more` command; the difference with `cat` is that `cat` will blindly spit out the whole file, whereas with `more` and `less` you can scroll backwards and forwards if the file is long.

1.4.4 Copying, moving and deleting files

Use the `cp` command to copy the file you just made:

```
$ cp output input
```

As previously, use the `more` command to look at the contents of `input`. Change the name of a file or directory using the `mv` command:

```
$ mv input input1
```

(use `ls` to confirm what just happened). Delete a file using `rm`:

```
$ rm input1
```

The most dangerous unix command is `rm -r` which recursively deletes a file or entire directory full of files.

You can also use wildcards when specifying files. For example, copy the file again (`cp output input`) and try the following commands:

```
$ ls *put
$ ls out*
$ ls in???
$ ls -l ??p??
```

1.4.5 Editing files

Importantly, you can edit text files in a variety of ways. There are two classes of editor: "in-terminal" editors and gui editors. A simple and intuitive gui editor is `gedit`. Open the file in `gedit` as follows:

```
$ gedit output &
```

The `&` at the end means 'run this process in the background', giving you back the prompt. Otherwise you cannot type in the terminal until gedit has finished running. **Type the following lines into the file, save, and quit gedit:**

```
hello world
The quick
brown fox
jumps over
the lazy
dog
```

Now, once you have returned to the terminal, **use the `more` command to look at the file contents**:

```
$ more output
```

The "in-terminal editors" are mainly useful for editing files when logged into a remote machine (because you do not have to pop open a window). The main ones are `vi`, `emacs` or `nano`. Of these, `nano` is probably the easiest for a beginner, whereas `emacs` is the most powerful and `vi` is just, well, weird, but some people like it. **You should teach yourself to use at least one of these**. At the very least you need to know how to *exit* these programs.

vi Open the file in `vi` as follows:

```
$ vi output
```

To exit from vi, type `:wq` or `:q` or under desperate circumstances `:q!`.

Emacs comes in both in-terminal and gui varieties. Which one you get by default depends on your Linux distribution. Open the file in emacs as follows:

```
$ emacs output
```

You can force emacs to run in the terminal by using the `-nw` (no window) option:

```
$ emacs -nw output
```

Commands in emacs use the `Ctrl` key. To exit emacs type `Ctrl-x Ctrl-c`. Emacs has a *lot* of features, including an in-built Tetris game (`Esc-x tetris`) and a house psychologist (`Esc-x doctor`). When you become un-distracted, let's continue...

1.4.6 Making head or tail of files

Look at the first `n` lines in a file using `head`:

```
$ head -3 output
hello world
The quick
brown fox
```

and the last `n` lines using (you guessed it) `tail`:

```
$ tail -3 output
jumps over
the lazy
dog
```

A useful option when running simulations is `tail -f`, which 'follows' the output of a file as content gets added to it (e.g. if your simulation code is writing to the file continuously).

1.4.7 Searching the content of files with grep, sed and awk

Some of the most powerful unix tools are `grep`, `sed` and `awk`. The `grep` command allows you to search for matching strings in a file or series of files. Try the following:

```
$ grep brown output
$ grep lazy *
$ grep the *
$ grep -i the output
```

`sed` stands for 'stream editor' and is a simple way to do search-and-replace. Try the following:

```
$ sed 's/brown/red/' output
```

By default the result is just output to the screen. To actually replace the text in the file(s), use the `-i` option.

```
$ sed -i.bak 's/brown/red/' output
```

Typing `ls` and using `more` you should see a backup of the old file in `output.bak`, while the new file has the word 'brown' replaced by 'red'.

The `awk` command can do more powerful things, but is almost a whole language in itself. As an example, try the following command:

```
$ awk "/The/,/fox/ { print }" output
```

You should find that this prints everything between the words "The" and "fox". These days it is better to use a fully featured scripting language like `perl` or `python` (see §2.3.4, below) instead of `awk` if you need this kind of text processing.

1.5 Putting commands together

Perhaps the most powerful feature of Unix is the ability to string a series of commands together. To do this, you use the `|` symbol, known as the *pipe*. Try the following examples:

```
$ cat output | grep fox | sed 's/fox/bird/'
red bird
```

You can also use a file as *input* to a command by using the `<` symbol

```
$ grep fox < output
$ grep fox < output | sed 's/fox/bird/'
```

and put the results into a new file using the `>` symbol:

```
$ grep fox < output | sed 's/fox/bird/' > birds.txt
```

Alternatively append to an existing file:

```
$ echo 'owl' >> birds.txt
```

There are also 'special files' that can be used to receive output. The most useful is the *null device*, `/dev/null`, which can be used to silence the output of a command. Try the following:

```
$ echo hello > /dev/null
```

Do you understand what is going on here? If not, ask your tutor.

1.6 Jobs and processes

Another Unix concept is to know what processes are running, and whether they are running in the *foreground* or *background*. The ampersand puts a job in the background:

```
$ gedit birds.txt &
[1] 6401
```

Pay attention to the `[1]` and the number following. These are the *Job identifier* and the *process identifier* (*pid*). You can get a list of running jobs with the `jobs` command:

```
$ jobs
[1]+  Running                 gedit birds.txt &
```

Try killing a job running in the background using the `kill` command. You can either use the job id with a `%` sign, or the pid:

```
$ kill %1
[1]+  Terminated: 15          gedit birds.txt
```

The default kill signal (15) asks for a graceful exit. A more forceful command is to send the (9) signal (this is the same as "force quit" on Mac OS). For example

```
$ gedit birds.txt &
[1] 6432
$ kill -9 6432
[1]+  Killed: 9               gedit birds.txt
```

The `jobs` list only shows you jobs launched from the current terminal. Type `top` to see all jobs running on the machine. You can kill any running job on the machine by using the `kill` command with the appropriate pid.

You can move a job launched accidentally in the foreground to the background using `Ctrl-z` to suspend the job, and `bg` to send it to the background:

```
$ gedit birds.txt
^Z
[1]+  Stopped                 gedit birds.txt
$ bg
[1]+ gedit birds.txt &
```

Use `Ctrl-c` to interrupt (kill) any job you are currently running in the foreground:

```
$ sleep 10
^C
```

1.7 Files and permissions

Type the following into a text file (called `test`) using your favourite editor:

```
SUBJECT=ASPxxxx
echo Assignment for $SUBJECT
echo '-- file list --'
ls -l
```

You can then execute this as a sequence of commands by typing '`source` *file*'.

```
$ source ./test
Assignment for ASPxxxx
-- file list --
total 40
-rw-r--r--  1 dprice  staff    20B 16 Jul 14:48 birds.txt
-rw-r--r--  1 dprice  staff    57B 16 Jul 12:43 input
-rw-r--r--  1 dprice  staff    55B 16 Jul 13:11 output
-rw-r--r--  1 dprice  staff    55B 16 Jul 13:11 output.bak
-rw-r--r--  1 dprice  staff    74B 16 Jul 15:29 test
```

Note that we need to specify the full path to the file `./test` (that is, the file `test` in the current directory). A better way is to make our file *executable*. We can do this simply by changing the *permissions* on the file:

```
$ chmod +x test
$ ./test
Assignment for ASPxxxx
-- file list --
total 40
-rw-r--r--  1 dprice  staff  20 16 Jul 14:48 birds.txt
-rw-r--r--  1 dprice  staff  57 16 Jul 12:43 input
-rw-r--r--  1 dprice  staff  55 16 Jul 13:11 output
-rw-r--r--  1 dprice  staff  55 16 Jul 13:11 output.bak
-rwxr-xr-x  1 dprice  staff  74 16 Jul 15:29 test
```

Notice the `x`'s that now appear in the left column next to the test file. This column indicates the file *permissions*. The letters indicate permission to read (`r`) write (`w`) and execute (`x`) the file. Directories also need the directory permission (`d`).

There are 3 sets of permissions, applying (from left to right) to the *user* (you), a *group of users*, and to *all users*. For information on how to set each of these, have a look at the `man` page for the `chmod` command (i.e. type `man chmod`). For example, try deleting the read permission from the output.bak file:

```
$ chmod -r output.bak
$ more output.bak
output.bak: Permission denied
```

Control over file permissions is why viruses are pretty much non-existent in Unix environments. In general each user only has permission to change files in their home directories, and only the 'super user' has permission to change system files (you usually only become the super user temporarily by using the `sudo` command, e.g. to install software).

1.8 Shell scripts

The file we just typed is an example of a *script*, that is a text file consisting of a series of commands. By default the commands will be interpreted by whatever the default shell is (in our case, bash) and is called a *shell script*. You can also explicitly indicate which program should be used to interpret the commands in a file. To do this, open the file `test` in your favoured editor and add the line to the top so that the file is as follows:

```
#!/bin/bash
SUBJECT=ASPxxxx
echo Assignment for $SUBJECT
echo '-- file list --'
ls -l
```

The first line is called the *shebang*. To understand what this does, create a new file called `pointless` with contents as follows:

```
#!/bin/rm
echo 'hello world'
```

then give the file the executable permission (`chmod +x pointless`), run the script (`./pointless`), and list the files in the directory (`ls`). What did this do? If you understand why, then you understand what the shebang is doing.

Shell scripts are very powerful, but the shell is not a fully-fledged programming language (e.g. one limitation of the shell is that it can only do integer arithmetic). We will not have time to use them very much, but if you plan on further study in astrophysics I recommend taking yourself through an online bash tutorial to appreciate the full range of things you can do. We will use a few more examples below.

1.9 Further reading

Cooper, Mendel, *The advanced Bash Scripting Guide*:

http://www.tldp.org/LDP/abs/html/

An in-depth exploration of the art of shell scripting.

2 Getting started with scientific programming

2.1 Plotting

For scientific data we often have data in text files that we want to plot. The simplest plotting tool is `gnuplot`. Let's create a file with some simple data in it, and create a plot. First, use a bash loop to create some data and put it in a file (called `datafile`)

```
$ for (( i=1; i <= 100; i++ )); do echo $i $(( i*i )); done > datafile
```

Check the file contents using the `more` command:

```
$ more datafile
1 1
2 4
3 9
4 16
5 25
...
```

Then launch `gnuplot`:

```
$ gnuplot

 G N U P L O T
 Version 4.6 patchlevel 4    last modified 2013-10-06
 Build System: Darwin x86_64

 Copyright (C) 1986-1993, 1998, 2004, 2007-2013
 Thomas Williams, Colin Kelley and many others

 gnuplot home:     http://www.gnuplot.info
 faq, bugs, etc:   type "help FAQ"
 immediate help:   type "help"  (plot window: hit 'h')

Terminal type set to 'aqua'
gnuplot>
```

At the gnuplot prompt, use the following command to get a basic plot:

```
gnuplot> plot 'datafile' using 1:2 with lines
```

labelling axes just means typing a few more commands before plotting:

```
gnuplot> set xlabel 'x'
gnuplot> set ylabel 'y'
gnuplot> plot 'datafile' u 1:2 w lines
```

Use the up arrow key to retrieve your command history to avoid re-typing the same thing. You can set the axes limits as follows:

```
gnuplot> set xrange [0:10]
gnuplot> set yrange [0:100]
gnuplot> plot 'datafile' u 1:2 w lines
```

To print the plot, you need to change the 'device' and specify a filename:

```
gnuplot>set terminal pdf
gnuplot>set output "myplot.pdf"
gnuplot>plot 'datafile' u 1:2 w lines
```

To exit gnuplot and return to the Unix shell, type `quit`:

```
gnuplot>quit
```

You should find a file called `myplot.pdf` in the current directory. You can open this in the default pdf viewer using:

```
$ xdg-open myplot.pdf
```

(on Mac OS the command is just `open`). The plot should look something like the following:

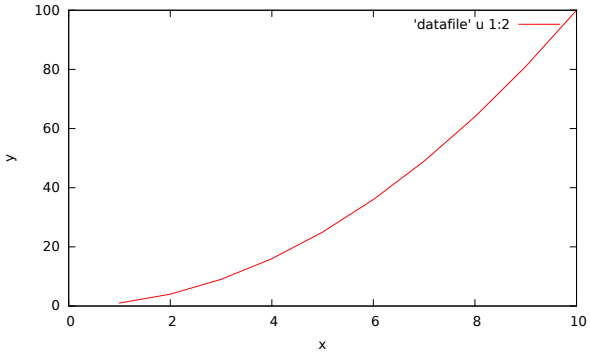

Figure 2: Example plot using gnuplot

For more advanced features, read some of the online `gnuplot` tutorials, or browse the help pages.

2.2 Manipulating data files

There are some useful shell commands for extracting data from files. These include `cut` and `paste`. Extract the first column as follows:

```
$ cut -d' ' -f 1 datafile > col1
```

and the second column:

```
$ cut -d' ' -f 2 datafile > col2
```

then use the paste command to put them back together with the column order changed:

```
$ paste col2 col1
```

To decode the commands above, type `man cut`.

2.3 Scientific programming

'Real' programming is fairly straightforward once you know how to use the Unix shell. It is just a matter of learning the particular programming language you want to use. Here we will briefly introduce three of the main languages used in scientific computing. From there you should be able to teach yourself by following online tutorials.

2.3.1 Fortran

Fortran is a modern version of an old language (pre-1990 versions are referred to as FORTRAN). It is designed specifically for scientific computing (as in *For*mula *trans*lation) and hence used widely for that purpose but rarely outside this. To type and run a basic program in Fortran, create a text file with the extension `.f90`. For example type `gedit main.f90 &` and enter contents as follows:

```
program hello
 print*,'hello world from Fortran'
end program hello
```

Fortran, like C, is a *compiled* language, so you need to run a *compiler* to create the executable program. We will use the *Gnu Fortran compiler*, `gfortran` (since it is free), so the relevant command is as follows (having made sure you've saved the file to disk):

```
$ gfortran -o bob main.f90
```

Assuming this did not throw up any errors, this creates an executable file with the name `bob`, that you can run as usual:

```
$ ./bob
 hello world from Fortran
```

Fortran's forté is in solving mathematical equations. Variables are declared explicitly (e.g. `real` or `integer`), and a simple mathematical calculation in Fortran might look like the following (e.g. in a file called `maths.f90`):

```
program maths
 implicit none
 real :: r, area
 real, parameter :: pi = 4.*atan(1.)

 r = 2.0
 area = pi*r**2
 print*,' pi is ',pi
 print*,' the area of a circle of radius ',r,' is ',area

end program maths
```

with output as follows:

```
$ gfortran -o maths maths.f90
$ ./maths
  pi is    3.14159274
  the area of a circle of radius    2.00000000      is    12.5663710
```

2.3.2 Further reading

To learn more about programming in Fortran, see:

- Rosemary Mardling and Daniel Price *A simple Fortran primer* (2nd Ed.), Monash University, 2016
- Metcalf M., Reid J., & Cohen, M., *Modern Fortran Explained.* The Fortran Bible. (This is the 4th edition. The 3rd edition is titled *Fortran 95/2003 explained*). Available from the Hargrave/Andrew library (3rd edition)

2.3.3 C

Unix itself is written in C. One of the key concepts in C is that the program is called "`main`", accepts arguments from the command line, and has a return value (0 or 1 to indicate success or failure). Create a file called `main.c` with contents as follows:

```
#include <stdio.h>
int main(int argc, char **argv)
{
 printf("The program is called %s\n",argv[0]);
 if (argc > 2) {
    printf("argument 1 is %s\n",argv[1]);
    printf("argument 2 is %s\n",argv[2]);
 }
 return 0;
}
```

Like Fortran, C programs must be compiled. Use the *Gnu C Compiler* (`gcc`), as follows:

```
$ gcc -o jane main.c
```

and run, both with and without command-line arguments:

```
$ ./jane
The program is called ./jane
$ ./jane hello world
The program is called ./jane
argument 1 is hello
argument 2 is world
```

C++ is an object-oriented extension to C that is widely used for scientific computing.

2.3.4 Python

Python is a scripting language, so is very much like writing a shell script. A basic program starts with a *shebang* specifying the python interpreter and then is like writing a shell script, but following the python syntax. Type the following into a file called `main.py`:

```
#!/usr/bin/python
print "hello world"
```

As it is a scripting language, you don't need to compile the programs, just make the file executable (`chmod +x main.py`) and run it directly:

```
$ ./main.py
hello world
```

Python contains extremely useful modules for plotting (the `matplotlib` module), Mathematica-like symbolic mathematics (`sympy`) and numerical calculations (`numpy`). It is a worthy and free replacement for MATLAB and such like and is now used extensively for scientific computing.

3 Exercises

Test your unix knowledge with the following exercises.

1. (a) Change into another directory that is not your home directory
   (b) Demonstrate using the `pwd` command that `cd`, `cd ~/`, and `cd $HOME` are all equivalent (in between each command you should change into a another directory that is not your home directory)
   (c) Create an empty directory called `empty`
   (d) Show, using the `ls` command, that this directory appears in the list of files
   (e) Remove this (empty) directory and show that it was successfully removed with the `ls` command
2. (a) Type `printenv` and parse the output to print only lines containing the word 'SHELL' (hint: use the `grep` command)
   (b) Print the value of the SHELL variable to the screen using the echo command and check this agrees with your previous finding
   (c) Parse the output of the `printenv` command so that every time the letter 'e' appears it is replaced by 'EEK'
3. (a) Change to your home directory
   (b) Use the `man` command to find the appropriate option to the `ls` command to print one file per line
   (c) Make a list of all of the files and directories in your home directory, one per line
   (d) Place the results into a new file called 'myfiles.txt'
   (e) Print the first 3 lines of this file to the terminal
   (f) Print the last 3 lines of this file to the terminal
   (g) Count how many files are in your home directory using a combination of the `cat` and `wc -l` commands
4. (a) Open `myfiles.txt` in `gedit`. Ensure that you launch the editor in the background by appending an apersand (i.e. `gedit myfiles.txt &`).
   (b) Returning to the command line, find the job id of your 'gedit' process
   (c) Kill your gedit session using the `kill` command
   (d) In your terminal, type 'yes Daniel is a legend'
   (e) Open a new terminal (in a separate tab or window) and use the `ps` command to find the process id of the previous command (use `man ps` and `grep` if necessary)
   (f) Kill the 'yes' process in the other terminal using the kill command

(g) Type 'yes I love astrophysics' and show that ctrl-c kills the job in the current terminal without needing to close the terminal

5. (a) Copy your myfiles.txt file into a new directory called `words_of_the_bard`

   (b) Change into your new directory and rename `myfiles.txt` to `shakespeare.txt`

   (c) Using the `echo` command, *append* some quotes from Shakespeare (one per line) to the end of the file (*make sure you append rather than overwrite the file — copy the file again from the other directory if you accidentally overwrite it*)

   (d) Using only command line tools, extract only the part of the file that shows your file list and output this into a new file called `newfiles.txt`

   (e) Use the `diff` command (`man diff` for help) to show that `newfiles.txt` and your `myfiles.txt` in the parent directory are identical. Also use `diff` to demonstrate that `shakespeare.txt` is different

   (f) Similarly, extract only the part of the file that contains words from the Bard and put these into a new file called `quotes.txt`

   (g) Delete the entire `words_of_the_bard` directory (hint: first you have to delete it's contents)

6. *Optional (but fun)*

   (a) Browse the unix dictionary by using the `more` command on the file `/usr/share/dict/words`

   (b) Count the number of words in the dictionary using the `wc` command

   (c) Use `echo $RANDOM` to print a random integer (try this a few times)

   (d) Use a combination of the `head` and `tail` commands to print a single random word from the dictionary (use the `$RANDOM` variable to select the line number)

   (e) Use backticks to set a variable called `MYWORD` equal to the result of your command, e.g. `MYWORD=‘head -$RANDOM ...‘`

   (f) Print an intelligible sentence including your random word using the `echo` command, e.g. `echo "my favourite word is $MYWORD"`

   (g) Use the backticks to insert the result of your random word generation command directly into your sentence (i.e. replace `$MYWORD` with `‘head -$RANDOM /usr/share...‘`).

   (h) Use the up arrow key to repeatedly generate new random sentences