Unix Tutorial Documentation

J-Donald Tournier

Apr 08, 2020

Contents

CHAPTER 1

Introduction

The purpose of this introductory guide is to provide a gentle explanation of concepts fundamental to using the Unix command-line, in a manner that will hopefully be accessible to beginners. It relies heavily on small examples to illustrate these ideas with hopefully relevant situations. It does *not* provide a comprehensive description of everything that can be done with the Unix command-line, and does not even mention most of the many useful commands available on any Unix system. The hope is that by explaining the concepts behind how things actually work, readers will quickly be able to figure out the rest by themselves, and find out anything else they might need from the enormous amount of excellent documentation available online. It's only a Google search away – but hopefully this document will help you to figure out what to ask in the first place.

1.1 What is the command-line?

The command line is simply a way of passing instructions to the computer. It is a text-only interface, and commands are supplied to the shell by typing them in and pressing the return key. Any output produced by these commands is printed out on the same text interface. Although daunting at first, the command line provides a very rich interface to the available functionality, and it is worth learning how to use it.

The command line interface will print out a prompt, after which commands can be typed. After the command has completed, the prompt will be printed out again, indicating that the shell is ready to accept new commands.

An important point to note is that Unix is case-sensitive: a file called `Data` is not the same as a file called `data`.

The command line prompt can vary from system to system, but usually consists of some text followed by the `$` character. In many systems, it might look like this:

```
username@machine:path $
```

where `username` corresponds to your own username, `machine` corresponds to the name of the system you are currently logged onto, and `path` indicates your current *working directory*. Note that this can be configured differently, do not be surprised or alarmed if it looks different on your system.

Warning: If the prompt ends with a #, this most likely means that you are logged in as root (the super-user or Administrator). If this is the case, please log out before doing anything irreversible - unless you know what you are doing.

Note: A session will typically involve two separate programs:

- the **shell**: the program that actually interprets the commands that are typed in, and takes the appropriate action. There are a number of different shells available, each with their own syntax rules. The default shell in most Linux distributions is the Bourne Again Shell (bash). This is also the default on macOS and MSYS2 (used in Windows installations), and for this reason is the default assumed in this guide.
- the **terminal**: the program responsible for displaying the output of the shell, and what the user types in. There are a number of different terminals available, each with different levels of sophistication.

This leads to a bewildering array of combinations between the different shells and terminal programs available. Nonetheless, the general principles are fairly universal.

1.2 How to access the command line

There are many ways to start a command line session, depending on your OS and desktop environment.

- **GNU/Linux**

  Different Linux distributions include different desktop environments. For instance Ubuntu installations will typically be running Unity, Red Hat and derivatives will typically be running GNOME, while SuSE would typically come with KDE.

  Given the many different possible configurations, it is impossible to give specific instructions for each case. Nonetheless, in general you ought to be able to identify an application called 'Terminal' or similar in the desktop's main menu.

- **macOS**

  You will find the 'Terminal' application within the *Utilities* folder in the *Applications* folder in the Finder.

- **Windows (MSYS2)**

  You will need to use the MSYS2 *MinGW-w64 Win64* shell, either by double-clicking on the shortcut (on the Desktop or in the Start menu), or by searching for it (hit the Windows key and start typing 'MSYS2' – it should show up in the list of applications displayed).

1.3 Basic Structure of a command

In the simplest form, a command consists of a line of text, which is first broken up by splitting it using spaces as boundaries. The first 'word' must be the command name, corresponding to a real program to be executed. All other 'words' will eventually be passed to the program as arguments that should provide it with enough information to perform what is intended.

1.3.1 Command line arguments

To perform whatever action is required, the program that is being run may need some additional information. This information can be supplied to the program via the arguments that follow it in the command. Essentially, arguments are a series of 'words' specified in the right order so that the program can understand what is required of it.

For example, you need to supply two arguments to copy a file: the original file name, and the name of the file to be created. The program to do this, *cp*, expects the first argument to be the name of the file to be copied, and the second argument to be the name of the duplicate file to be created. If it does not find two arguments, it will produce an error message.

Note that some programs may accept a number of arguments, but use default values if they are omitted (example of these are *cd* and *ls*). Other programs may accept variable numbers of arguments, and process each argument in turn.

1.3.2 Command line options

There is a special type of argument that you might encounter, often referred to as a command line option or switch. The purpose of these optional arguments is to modify the behaviour of the program in some way. Command line options always start with a minus symbol to distinguish them from normal arguments. For example, passing the appropriate option (`-l`) to the *ls* command when listing the files in the current folder will produce a longer listing, including information such as file size and modification time as well as the file names normally output.

Command line options can also require additional arguments. In this case, these additional arguments should be entered immediately after the option itself – see the examples below.

1.3.3 Examples

Below are some typical command examples. (the `$` symbol indicates the prompt):

- To list the contents of the current working directory:

```
$ ls
```

- To list the contents of the current working directory, along with the file permissions, owner, size and modification date:

```
$ ls -l
```

- To copy the file `source`, creating the file `dest`:

```
$ cp source dest
```

- To convert image `source.mif` (*MRtrix* format) into image `dest.nii` (NIfTI format):

```
$ mrconvert source.mif dest.nii
```

- To convert image `source.mif` into image `dest.nii`, changing the voxel size to 1.25 x 1 x 1 mm and changing the datatype to 32-bit floating-point:

```
$ mrconvert source.mif -vox 1.25,1,1 -datatype float32 dest.nii
```

1.4 Dealing with spaces in arguments

As previously mentioned, the command actually typed in will first be split up into *tokens* using spaces as delimiters. In certain cases, it may be necessary to provide arguments that contain spaces within them. A common example of this is when file names contain spaces (note that this should be avoided, especially since other programs and scripts often have issues dealing with these). This is obviously a problem, since an argument with a space in it will be interpreted as two separate arguments. To supply an argument with a space in it, use the following syntax.

As an example, if we need to supply the argument "argument with spaces" to some command, we can use any of the following:

- `argument\ with\ spaces`
- `'argument with spaces'`
- `"argument with spaces"`

In the first example, the backslash character tells the shell to ignore the subsequent space character and treat it as a normal character.

1.5 Escaping special characters

We have already seen that spaces are treated differently from other characters and need to be encapsulated by quotes `', "` or escaped by a preceding `\,` to prevent them being interpreted by the shell as token delimiters. You will most likely also encounter other special characters such as `!#$^&*?[](){}<>~;|` in more advanced usages; these come in handy for instance for processing multiple files using wildcard characters.

One can influence the way the shell interprets these special characters by quoting and escaping the input. For instance, the string `'argument with spaces'` uses single-quotes (*strong quoting*): this means that everything between the two `'` symbols is treated as literal characters without special meaning. In the command below the series of special characters are treated as a simple string and printed to the terminal via the command *echo*.

```
$ echo look how ordinary these characters are: '!#$^&*?[](){}<>~;|\'
```

Unless encapsulated in single quotes, individual special characters can also be marked to lose their special meaning using the backslash. For instance, `\'argument with spaces\'` would expand to three arguments `'argument`, `with`, and `spaces'`. The only exception to this rule is the newline character, which allows commands to span across multiple lines:

```
$ echo look how ordinary these characters are: '!#$^&*?[](){}<>~;|\' \
and \'
```

Double quotes `"` are used for *weak quoting*, which escapes all characters expect for `\`, `$` and itself. This disables some of the shell's interpretations (spaces, single-quotes, pattern matching, pathname expansions) while others remain active (such as parameter expansion `$`).

```
$ ls -l "$HOME/folder with spaces"
```

Note that special characters' meaning can be shell- and context-dependent. For example, in the Bourne Again Shell (bash), the string `filename[].mif` is not interpreted but in the Z shell (zsh, the default shell for new user accounts since macOS version 10.15), the opening `[` needs to be quoted `"filename[].mif"` or escaped using a backslash `filename\[].mif`.

For more information, consult your shell's man page or this overview post,

CHAPTER 2

Specifying filenames: paths

Filenames are often supplied to programs as arguments. For this reason, it is essential to have a good understanding of how files are specified on the command line. In Unix, a *path* is a term commonly used almost interchangeably with *filename*, for reasons that will hopefully become clear in this section.

2.1 Files and folders

Files and folders are stored on computers using a folder or directory structure. For example, on a Windows computer, you might find a folder called `MyDocuments`, within which there might be a `data` folder, and within that some more folders, etc. Specifying a file or folder is simply a matter of providing enough information to uniquely identify it.

The easiest way to visualise the directory structure is to think of it as a *tree*. If you listed the contents of the root folder (the root of the tree), you would find a number of other folders (the main branches). For example, one of these folders would be the `home` folder, where user accounts are kept. These folders might contain more folders (smaller branches) and/or files (leaves), as illustrated below:

```
/
├── bin/
├── boot/
├── home/
│   └── donald/
│       ├── data/
│       │   ├── pilot/
│       │   │   ├── subject1/
│       │   │   │   └── dwi.mif
│       │   │   ├── subject2/
│       │   │   │   └── dwi.mif
│       │   │   └── subject3/
│       │   │       └── dwi.mif
│       │   └── project/
│       │       ├── analysis_script.sh
│       │       ├── control1/
│       │       │   ├── anat.nii
```

(continues on next page)

(continued from previous page)

```
        │       │   └── dwi.mif
        │       ├── control2/
        │       │   ├── anat.nii
        │       │   └── dwi.mif
        │       ├── control3/
        │       │   ├── anat.nii
        │       │   └── dwi.mif
        │       ├── patient1/
        │       │   ├── anat.nii
        │       │   └── dwi.mif
        │       └── patient2/
        │           ├── anat.nii
        │           └── dwi.mif
        ├── Desktop/
        └── Documents/
├── usr/
└── var/
```

Here, the folder called `project` can be uniquely identified by starting from the root folder, going into `home`, then `donald`, `data`, and finally `project`. This process can be thought of as specifying the *path* to the file or folder of interest. In fact, this is the exact term used in Unix jargon, essentially meaning 'an unambiguous file name'. Thus, specifying a filename boils down to providing a unique, unambiguous *path* to the file.

Note: In this context, *directory* and *folder* are synonymous.

2.2 Absolute paths

On Unix, the root of the tree is always referred to using a simple forward slash. Folders are referred to using their names, and are delimited using a forward slash. For example, the full, absolute path to the `project` folder in the figure above is:

```
/home/donald/data/project/
```

This simply means: starting from the root folder (`/`), go into folder `home`, then `donald`, then `data`, to find `project`. This is an example of an *absolute path*, because the start point of the path (the root folder) has been specified within the filename. Thus, an absolute path must start with a forward slash – if it does not, it becomes a relative path, explained below.

2.3 The working directory

When using the command line, you will often find that many of the files you are manipulating reside in the same folder. It therefore makes sense to specify this folder as your *working directory*, and then simply refer to each file by its name, rather than its absolute path. This is exactly what the working directory is, and it can save you a lot of unnecessary typing.

You can also think of it as your current location on the directory tree. For example, if your current working directory is `/home/donald/data/project`, you can imagine that you have climbed up branch `home`, then up branch `donald`, then up branch `data`, then up branch `project`, and since you're sitting there, you have direct access to all the files and folders that spring from that branch.

Your working directory can be specified with the command *cd*, and queried using the command *pwd* (both described in *Basic commands*).

2.4 Relative paths

Relative paths are so called because their starting point is the current working directory, rather than the root folder. Thus, they are relative to the current working directory, and only make sense if the working directory is also known.

For example, the working directory might currently be `/home/donald/data/project/`. In this folder there may be a number of other files and folders. Since the file `analysis_script.sh` is in the current working directory, it can be referred to unambiguously using the relative path `analysis_script.sh`, rather than its full absolute path `/home/donald/data/project/analysis_script.sh` – that's a lot less typing.

When you specify a relative path, it will actually be converted to an absolute path, simply by taking the current working directory (an absolute path), appending a forward slash, and appending the relative path you supplied after that. For example, if you supply the relative path `analysis_script.sh`, the system will (internally) add up the current working directory `/home/donald/data/project` + `/` + `analysis_script.sh` to give the absolute path.

Since the system simply adds the relative path to the working directory, you can see that files and folders further along the directory tree can also be accessed easily. For example, the `project` folder contains other folders, `patient1`, `patient2`, etc. The file `anat.nii` within one of these folders can be specified using the relative path `patient1/anat.nii` (assuming your current working directory is `/home/donald/data/project`).

Of course, if you changed your current working directory, the relative path would need to change accordingly. Using the same example as previously, if `/home/donald/data/project/patient1` was now your current working directory, you could use the simpler relative path `anat.nii` to refer to the same file.

2.5 Special filenames

A few shortcuts have special significance, and you should learn to use them, or at least know of them. These are:

- `~` (tilde):

  shorthand for your home folder. For example, I could refer to the `project` folder as `~/data/project`, since my home folder is `/home/donald`.

- `.` (single full stop):

  the current directory. For example, if my current working directory is `/home/donald`, I can refer to the `project` folder by specifying `./data/project`, or even `data/./project`. Although this may not look very useful, there are occasions when it becomes important (see examples below).

- `..` (double full stop):

  the parent folder of the current directory. For example, if my current working directory is `/home/donald/Desktop`, I can still refer to the `data` folder using the relative path `../data`. This shortcut essentially means "drop the previous folder name from the path", or "go back down to the previous branch". Here are some alternative, less useful ways of referring to that same `data` folder, just to illustrate the idea:

```
../../donald/data
../Documents/../data
~/Desktop/../data
```

2.6 Using wildcards

There are a number of characters that have special meaning to the shell. Some of these characters are referred to as *wildcards*, and their purpose is to ask the shell to find all filenames that match the wildcard, and expand them on the command line. Although there are a number of wildcards, the only one that will be detailed here is the * character.

The * character essentially means 'any number or any characters'. When the shell encounters this character in an argument, it will look for any files that match that pattern, and append them one after the other where the original pattern used to be. This can be better understood using some examples.

Imagine that within the current working directory, we have the files `file1.txt`, `file2.txt`, `file3.txt`, `info.txt`, `image1.dat`, and `image2.dat`. If we simply list the files (using the *ls* command), we would see:

```
$ ls
file1.txt   file2.txt   file3.txt
image1.dat  image2.dat  info.txt
```

If we only wanted to list the text files, we could use a wildcard, and specify that we are only interested in files that end with `.txt`:

```
$ ls *.txt
file1.txt   file2.txt   file3.txt   info.txt
```

We might only be interested in those text files that start with `file`. In this case, we could type:

```
$ ls file*.txt
file1.txt   file2.txt   file3.txt
```

This use of wildcards becomes very useful when dealing with folders containing large numbers of similar files, and only a subgroup of them is of interest.

Note: It will be important later on to understand exactly what is going on here. Typing a command such as:

```
$ ls *.txt
```

does *not* instruct the *ls* command to find all files that match the wildcard. The wildcard matching is actually performed by the *shell*, before the *ls* command is itself invoked. What this means is that the *shell* takes the command you typed, modifies it by *expanding* the arguments, and invokes the corresponding command on your behalf. In the case above, this means that the command actually invoked will be:

```
$ ls file1.txt file2.txt file3.txt info.txt
```

In other words, your *single* argument containing a wildcard is expanded into multiple matching arguments by the *shell*.

As another example, a command like:

```
$ cp *.dat
```

will be expanded to:

```
$ cp image1.dat image2.dat
```

which will cause `image2.dat` to be *overwritten* with the contents of `image1.txt` – presumably causing irretrievable loss of data. In other words: think carefully about what you're typing...

CHAPTER 3

Basic commands

Below is a list of commands that are very commonly used. Of these, *cd* and *ls* are essential to using the command line, and you should be familiar with them.

Each command described below has a syntax line, which gives a brief description of how it should be used. The first item on the line is always the name of the command, followed by a number of arguments. Arguments given in square brackets are optional and can be omitted. If an argument is followed by three dots ('...'), it means that any number of that type of argument can be provided, and each will be processed in turn. In addition, any options that are of particular interest are listed in the corresponding section for each command.

3.1 `pwd`: print working directory

```
$ pwd
```

Print the current working directory onto the screen.

3.2 `cd`: change working directory

```
$ cd [folder]
```

Change the current working directory. If no folder is specified, the current working directory will be set to the home folder. Otherwise, it will be set to `folder`, assuming that it is a valid path.

Note that in this command, the token `-` has special meaning when passed instead of `folder`: it refers to the previous working directory. This can be useful to rapidly change back and forth between folders, e.g.:

```
# our current working directory is /home/donald/Documents

$ cd ../data/project/patient2
...          # current working directory is now /home/donald/data/project/patient2
```

(continues on next page)

(continued from previous page)

```
...          # do something useful

$ cd -       # switch back to /home/donald/Documents when you're done
```

3.3 `ls`: list files

```
$ ls [option]... [target]...
```

List files. If no target is specified, the contents of the current working directory are listed. Otherwise, the files specified are listed in the order provided, assuming that they are valid paths. If a target is a folder, its contents will be listed.

Options:

- `-a`

  list all files, including hidden files (hidden files are those that start with a full stop '.').

- `-l`

  provide a long listing, including file size, ownership, permissions, and modification date.

3.4 `cp`: copy files or folders

```
$ cp [option]... source target
$ cp [option]... source... folder
```

In its first form, copy the file `source` to create the file `target`, assuming both are valid paths. You should be aware that in general, if the file `target` already exists, it will be overwritten (although this behaviour can be modified through an alias).

The second form is used to copy one or more `source` files into the `target` folder. In this case, target must be a pre-exising folder. Each newly created file will have the same base name as the original.

Options:

- `-i`

  ask for confirmation before overwriting any files.

- `-r`

  recursive copy: use this option to copy an entire folder and its contents.

3.5 `mv`: move/rename files or folders

```
$ mv [option]... source target
$ mv [option]... source... folder
```

In its first form, move or rename the file (or folder) `source` to `target`, assuming both are valid paths. Note that renaming is essentially equivalent to moving the file to a different location, if `source` and `target` reside in different folders.

The second form is used to move one or more `source` files into the `target` folder. In this case, `target` must be a pre-existing folder.

Options:

- `-i`

  ask for confirmation before overwriting any files.

3.6 `echo`: output strings to the terminal

```
$ echo [option]... [string]...
```

The echo command writes any specified arguments, separated by single space characters and followed by a newline (`\n`) character to the standard output.

3.7 Examples of typical command use

Below are some examples of commands in typical use, illustrating some of the concepts explained in this document. To fully understand the examples, you may need to refer back to the sections on *Specifying filenames: paths*, using special filenames, or using wildcards.

- To change your current working directory to its parent folder (move one branch down the directory tree):

  ```
  $ cd ..
  ```

- To change your current working directory from whatever it was to the `data` folder in your home directory:

  ```
  $ cd ~/data
  ```

- To list all images (with the `.png` suffix) whose filename start with `ns` from the `controls` folder:

  ```
  $ ls controls/ns*.png
  ```

- To move the file `data.mat`, residing in the current working directory, into the parent folder of that directory:

  ```
  $ mv data.mat ..
  ```

- To copy the file `info.txt` from the folder `important` into the current working directory:

  ```
  $ cp important/info.txt .
  ```

- To copy all shell script files from the `data` folder in your home directory into the `scripts` folder in the current working directory:

  ```
  $ cp ~/data/*.sh scripts/
  ```

- To copy all images for study 3 of patient *Joe Bloggs* from the `/data` folder into the current working directory (assuming the images are named according to the convention `ID-study-slice.ima`):

  ```
  $ cp /data/bloggsj_010203_123/*-3-*.ima .
  ```

CHAPTER 4

Useful tips

The commands that you might type in can get quite long, and using the command line can quickly become laborious. You are strongly encouraged to read the tips below, which can not only save you a lot of unnecessary typing, but also help you avoid making mistakes and typos. Some of these features may not be immediately obvious, so you are advised to spend a little time familiarising yourself with them.

4.1 Previous command history - the up arrow key

You will often find yourself typing a whole series of similar commands, that differ only by a few characters each time. It would be tedious if you had to type each one of them in individually. Similarly, you may find that the long command you just typed in has failed because of a single little typo. To save yourself the time and the hassle, use of the command history.

The shell remembers all the commands that you recently typed in. At any time, you can set the text of your command to any of these previous commands, as if you had just typed it in. It is then trivial to go back and edit those sections that need to be amended, using the left and right arrow keys, etc.

To access previous commands, simply press the up arrow. This will give you the last command you typed in. Pressing it again will produce the command before that, and so on. You can also press the down arrow key to find more recent commands if you have gone too far back in the command history.

4.2 Command completion - the tab key

You will often need to supply filenames as arguments to commands. Some of these may be long and hard to remember, for example, if they contain patient IDs and dates. However, the computer knows everything about which files are in which folder, and can therefore help the user to type in the right filename. You will find that often, it is only necessary to type in the first few letters of the file and the shell will fill in the rest for you.

You can ask the computer to attempt to complete the filename by pressing the `TAB` key. At this point, the shell will look at the fragment that you have already typed in, and compare it to the list of files in the corresponding folder. For example, if you type:

```
$ ls /data/3T_scanner/b
```

and press the TAB key, the shell will look at the contents of the /data/3T_scanner folder, and see if any of these files or folders begin with a b. At this point, one of three things can happen:

1. there is only one file that begins with the fragment you supplied. In this case, the shell will complete the filename for you. Using the example above, if the only file or folder in /data/3T_scanner that started with a b was bloggsj_010203_123, then the command would be changed to:

   ```
   $ ls /data/3T_scanner/bloggsj_010203_123/
   ```

2. there are more than one file that begin with the fragment you supplied. In this case, the shell will emit a beep or some other visual feedback (unless configured not to) to notify the user.

   Pressing the TAB key a second time will cause the shell to output a list of all the potential candidates, so that the user can select the right one. You should then just type in more of the filename and press the TAB key again once you think that fragment is unambiguous.

   Using the example above, there might have been another folder in /data/3T_scanner that started with a b, say brownj_030201_789. In this case, the first TAB key press would have produced a beep, and the second would have produced a list like the following:

   ```
   $ ls /data/3T_scanner/b
   bloggsj_010203_123        brownj_030201_789
   ```

   You then simply need to type in an extra l and press TAB again for the shell to change the command as above.

3. there are no files that start with the fragment you supplied. In this case, the shell will also emit a beep to notify the user. This time, pressing the TAB key again will only cause repeated beeping. You should amend what you have typed in, and make sure that it is correct up to that point.

CHAPTER 5

What's going on under the hood?

While the description given so far is sufficient to get started, you will rapidly encounter situations where things don't work as you might expect. In these cases, it really helps to understand how what you type translates into action, so that you can usefully reason about what's going on. The main thing to get to grips with is what the various components are and what their specific roles are.

5.1 The terminal

The terminal's role is purely one of user *input and output*: it displays the output produced by the shell or the command current executing within it, and receives input from the user via the keyboard (and potentially the mouse too) to relay it to the shell or the command currently executing. In both cases, the input and ouput consists merely of a stream of *bytes*. Some of these characters correspond to printable characters, others to non-printable characters (e.g. carriage return, newline, ...), and yet others to *sequences* that might carry special meaning (e.g. the VT100 Terminal Control sequences). You might see some of these garbled sequences of characters when output intended for the terminal is instead written to file.

5.2 The shell

The role of the shell is to *interpret* the input provided by the user, and provide output in response to be displayed by the terminal. The most obvious output produced by the shell is the *prompt*, which typically looks like:

```
username@machine:path $
```

This is simply a string of characters produced by the shell, instructing the terminal to display this to the user, in order to indicate to the user that it is ready to receive input. The convention on modern systems to display the username, machine and path is entirely optional, but you'll quickly find it very useful as you navigate around the folders on your system, or if you start using the terminal to remotely access other systems (e.g. a high performance computing cluster).

The input that you type at this point is then passed to the shell, which decides what to do with it. In most cases, it will simply *echo* the characters you've entered so you can see what you've typed on the terminal. But some of your input might already start to be interpreted, for example `Ctrl-A` will bring the cursor to the start of the line, `Ctrl-E` back

to the end (along with the Home & End keys, on a well-configured system). The Left & Right arrow keys will allow you to move the cursor along to edit your command. The Up & Down arrow keys will bring up previous commands that you've typed. These are all examples of the shell *interpreting* your input and responding accordingly.

Note: The shell's handling of your input is typically managed by the readline library. It is highly configurable, and the actions tied to each of these keys can be modified in many ways. This tutorial describes the actions you'd expect on most Unix systems by default, but don't be surprised if things behave slightly differently on other systems.

Once you've typed in your command, you will typically just hit the Return key to execute it. In the simplest case, when the shell receives this instruction, your input will simply be broken up into a list of distinct 'entries', each separated by spaces (although it entirely possible to have entries with spaces, as *detailed earlier*). In other cases, it will also perform any additional interpretation required (e.g. for wildcard expansion, variable substitution, etc). These entries are typically called the command-line *arguments*.

The first (and possibly only) entry in this list is expected to be the command name, and the shell will try to locate the corresponding executable (unless it's a special shell built-in command, such as `cd`, `pwd`, `echo`, `export`, ...). Executables are just regular files that happen to have a special flag set to signal the fact that they can be executed (you can see this using `ls -l` – look for the `x` in the permissions field). Some of these executables might be human-readable text, in which case they tend to be called *scripts*. Other executables consist of machine code: the raw instructions executed by the CPU – these are often referred to as *binaries*. Regardless, the shell will need to locate this file so that it can execute the code it contains.

To locate the executable, the system will typically look through the list of folders contained in the `PATH` environment variable. You can query this list yourself by typing `echo $PATH` (it's a colon-separated list of folder paths). This is why it is often necessary to modify the `PATH` when installing non-standard software: this is how the shell 'knows' where to find these new commands. It is also possible to provide the exact location of the command, by invoking the command via the *absolute path* to its executable (e.g. `/usr/bin/whoami`, rather than just `whoami`), or a *relative path* to it (e.g. `./bin/myexecutable`); this can come in handy if you just need to execute your script from a location not in your `PATH`, without having to modify the `PATH` explicitly.

5.3 The command

At this point, the shell is now ready to actually run your command. It will place your list of command-line arguments in a special location, and launch the executable itself. The executable now takes over the terminal: any input you provide will be send to it, and any output it produces will be displayed in the terminal. It also has access to your list of command-line arguments, and will perform its actions based on what it finds there.

It is important to note that how the command interprets the arguments it's been given is *entirely* up to the command itself – or rather, up to the command's developer. This means that different commands will adopt slightly different conventions as to how their arguments are expected to be provided. Some commands expect all *command-line options* to be provided before any of the main arguments, and others don't. Some commands expect options to be provided in short form (e.g. `ls -l`), other in long form (e.g. `git --help`), others will accept a mixture of both (some short, some long), and yet others may accept both forms for the *same* option (e.g. `cp -f` is equivalent to `cp --force`). There are accepted conventions, but this by no means implies that all developers will abide by them.

Another potential source of information that the executable will have access to is via *environment variables* (the `PATH` is one such variable). This allows the user to set a variable, which the command can query as it is executing. This might be to specify the location of important configuration files, or the number of threads that the application is expected to run, etc. In general, this is reserved for information that is unlikely to change very often, allowing the user to set this information once within the shell's startup file (typically `~/.bashrc`), and no longer have to worry about it. For example, adding this line in `~/.bashrc` means that applications can query this variable at runtime:

```
export MYAPP_DIR=/usr/local/myapp/configfiles
```

5.4 Implications

Once you appreciate the way these components fit together, various aspects of the system may start to make more sense. For instance, there are many terminal programs available, from raw VT100 terminals, to various graphical ones, all with various levels of functionality (e.g. multiple tabs, split display, transparent background, unlimited scrollback, etc.). But within these various terminals, you will generally be running the same *shell*, and it'll behave the same way no matter which terminal it is running in. However, if you log into a different system, you may find subtle differences in the way it behaves (different prompts, some keyboard shortcuts that work differently, etc.). You may also find that the default shell you are logging in with is different on different system: some HPC systems are configured with the C shell as the default, and its syntax can be quite different. But at heart, the concepts outlined here will be the same.

It's also important to understand what the shell will do to your command as it's interpreting it, and what arguments this will translate to once passed to the executable itself. A useful trick here is to prefix your intended command with `echo`: the shell will perform all the variable substitute and wildcard expansion that it normally would, but because `echo` is now the command, all this will now simply be printed on the terminal. For example:

```
$ echo cp files*.txt destination/
cp file.txt file_1.txt file_2.txt file_3.txt destination/
```

Note that the command itself is not executed; it is merely displayed as it would have been interpreted by the shell. This might come in handy to see that the `files.txt` file will also be copied, when that might not have been intended. This trick is particularly useful when performing more advanced substitutions, as you'll see in the *Advanced usage* page.

CHAPTER 6

Advanced usage

So far, we have only covered the simplest aspects of what the shell can do. But it can be used for far more than this, and can even be used as a full-blown scripting language capable of running complex applications. While this level of mastery is probably unnecessary for most users, there are a few advanced topics that are very useful and worth covering in more detail, even in an introductory document such as this. The interested reader is referred to more complete guides, such as this one.

6.1 Redirection

The *standard output* of commands, normally intended to be displayed on the terminal, can be *redirected* to a file if needed, using the > symbol.

For example, assuming that:

```
$ ls
docs  html  LICENSE  README.md
```

The file listing can be *redictered* to a file, called `listing.txt` in the example below:

```
$ ls > listing.txt
```

This creates the file specified, and the output normally shown by `ls` is not visible on the terminal. It has however been stored in the `listing.txt` file, as we can verify with `cat`:

```
$ cat listing.txt
docs
html
LICENSE
README.md
```

This can also work in *append* mode, where the output of the command is appended to the file, rather than overwriting its entire contents.

For example:

```
$ app1 input output -options > log.txt
$ app2 arg1 arg2 >> log.txt
```

will create the `log.txt` file in the first line, and record any output from the `app1` command. The second line will then *append* its output to the log file.

Likewise, we can redirect the *standard input* to feed in the contents of a file as input, rather than typing it in, using the < symbol.

For example:

```
$ sort < myfile.txt
```

will feed the contents of `myfile.txt` to the `sort` command's *standard input*.

6.2 Pipes

This is a special type of redirection, where the *standard output* of one command can be fed directly into the *standard input* of another, using the `|` symbol. Both commands run concurrently, with the second command able to process the output of the first as soon as it is provided. This can be incredibly useful to build compound commands.

For example:

```
$ grep ERROR log.txt | sort | uniq
ERROR: error type one
ERROR: input file not found
ERROR: something bad happened
```

uses the `grep` command to find all lines in `log.txt` that contain the character string `ERROR`, then feeds those lines (which would normally be displayed on the terminal) via the pipe as input for the `sort` command. This sorts the lines in alphabetical order, and feed its output to the `uniq` command, which remove duplicates. The outcome of the full pipeline is a list of all unique error messages logged in the `log.txt` file.

Another particularly useful example is to capture the output from a command expected to produce a lot of output, and browse through it at a more suitable pace rather than seeing it fly past on the terminal. This can be done using the command `less` (a paginator):

```
$ complex_process -verbose | less
```

This ability to quickly implememt otherwise non-trivial functionality is one of the great strengths of the command-line. Unix is full of little tools like `grep`, `sort` and `uniq` that are designed to operate on text and to be daisy-chained in this manner.

6.3 Conditional execution

While BASH provide its own `if` statement for more complex situations, it also offers a simple construct to allow execution of one command based on the success or failure of another, using the `&&` and `||` operators respectively.

For example:

```
$ myapp args -options || echo "myapp failed to run!" >> log.txt
```

will record the fact that the `myapp` command has failed to the `log.txt` file.

On the other hand:

```
$ stage1 -options inputdata/ tmpdata/ && stage2 tmpdata/ outputdata/
```

will only run the `stage2` command if the `stage1` executable has completed successfully (useful if the data produced by the first command is to be processed by the next one).

6.4 Variables

It is often useful to store information in variables. For instance, you might want to use a long and complicated filename often, and rather than typing it in every time you need it, you could use a variable. Variables are assigned using the = symbol (beware: no spaces around it), and retrieved (dereferenced) using the `$` symbol.

For example:

```
$ logfile=/some/complicated/location/myapp/logs/run1.txt
$ myapp input intermediate > $logfile
$ otherapp intermediate output >> $logfile
...
```

The variable `logfile` is set to the filename of the logfile, and the output of all subsequent commands is then redirected to that file (see above).

6.5 Iterating with for loops

It is often required to perform the same command for a number of files. This can be achieved simply and effectively with a `for` loop, like this:

```
$ for item in logs/run*.txt; do grep OUTPUT $item; done
```

This will find all lines that contain the token `OUTPUT` in the logfiles stored in the `logs/` folder that match the filename `run*.txt`, and print them on the terminal.

What actually happens here is that a variable `item` is used to store each token listed after the `in` keywords (until the end of line or `;` symbol), and the command(s) between the `do` and `done` keywords are then executed for each token. The current value of the token can then be retrieved within the loop by dereferencing it like any other variable, using the `$` symbol.

Note that the above does not need to be all on the same line. In practice, lines can be broken wherever the `;` was used in the example above:

```
$ for item in logs/run*.txt
> do
>   grep OUTPUT $item
> done
```

6.6 Parameter substitution

There are certain operations that can be performed on variables at the point where they are being dereferenced. Of these, the most useful are probably the ability to strip a suffix or prefix. This is done using a syntax like `${var#prefix}` or `${var%suffix}`. This is most useful in scripts and when combined with `for` loops.

For example:

```
$ for data in *.dat
> do
>   process $data ${data%.dat}.out > ${data%.dat}.log
> done
```

will run the `process` command on all files in the current folder that end with the `.dat` suffix, and pass as second argument the same filename with the `.dat` suffix stripped and replaced with the `.out` suffix. The output of each command will individually be stored in log files, each with the `.log` suffix. If the current folder contained the files:

```
$ ls
backup/ final.dat original.dat parameters.txt trial2.dat
```

Then the commands actually run will be:

```
$ process final.dat final.out > final.log
$ process original.dat original.out > original.log
$ process trial2.dat trial2.out > trial2.log
```

There are many other types of parameter substitutions possible, see the relevant documentation for details.

CHAPTER 7

Customising the shell

As noted in previous sections, the shell can be configured in various way to make it easier to use or more powerful. Many of the more useful modifications can be made via the readline library, since this is the part of the shell that allows commands to be edited. There are also various environment variables that can be set, for instance to modify the prompt, or store more commands in the history.

7.1 Startup files

All of these modifications are typically implemented by adding the relevant commands to the shell's various startup files. Part of the difficulty in getting these modifications to work lie in figuring which file gets executed under what circumstances, and what settings this file affects.

When the shell starts, it will typically be invoked as a *login* or *non-login* shell, depending on whether you have just logged into the system directly into this shell. Generally, when you log into a desktop session, the session will be started from the shell, and this shell is the login shell. This means any terminal sessions you start subsequently from this graphical session are *non-login* shells (although not on macOs, by default). The reason this matters is because different startup files are used depending on whether it's a login shell or not. This means that settings placed in one startup file (e.g. `~/.bashrc`) may have no effect when logging in via a remote SSH connection, for instance, since this session would start a *login* shell. On top of that, different distributions have come up with subtly different conventions as to what files are used, making it very difficult to make recommendations guaranteed to work in all cases.

The information below relates to the Bourne Again shell (BASH), and will hopefully be representative of most systems (see the official BASH documentation for full details). The relevant files would be different for different shells:

- *login* shells will typically read the system-wide `/etc/profile` if it exists, then the user-specific `~/.profile` or `~/.bash_profile` if it exists.
- *non-login* shells will typically read the `~/.bashrc` file if it exists.
- on many distributions, the system-wide `/etc/profile` will contain instructions to read the system-wde `/etc/bash.bashrc` file if it exists.
- likewise, on many distributions, the user `~/.profile` will contain instructions to run the user `~/.bashrc` file if found.

- Settings that affect the readline library will normally go in the `~/.inputrc` file (see below).

As you can appreciate, things can rapidly become complicated. In most cases, you should be able to add your settings to the `~/.bashrc` file, and on a properly configured system, that should be sufficient.

7.2 Useful ~/.bashrc customisations

These suggestions are things that I've found useful to add to `~/.bashrc`, you may find some of them to your taste:

- Append to the history, do not overwrite it:

```
shopt -s histappend
```

- Keep a lot more commands in the history than the default:

```
export HISTSIZE=10000 HISTFILESIZE=100000
```

- Don't keep duplicate entries in the command history:

```
export HISTCONTROL=ignoreboth
```

- On colour-capable terminals, use colours in file listings:

```
alias ls='ls --color=auto'
```

- Make common operations prompt you if they are about to overwrite files:

```
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
```

7.3 Useful readline customisations

Most distributions will come with the following already set, but just in case, the following is useful to put into your `~/.inputrc` if your Home, End, or Del keys don't work, and if you want to be able to skip over words with Ctrl+Left/Right:

```
# mappings for Home/End:
"\e[1~": beginning-of-line
"\e[4~": end-of-line
"\e[7~": beginning-of-line

# Del key:
"\e[3~": delete-char

# Ctrl+arrows to skip words:
"\e[5C": forward-word
"\e[5D": backward-word
"\e\e[C": forward-word
"\e\e[D": backward-word
"\e[1;5C": forward-word
"\e[1;5D": backward-word
```

Generally, most distribution set up the Up & Down arrows to go through the history, with PgUp & PgDn going to the oldest and most recent entry respectively. I find a more useful use for the Up & Down arrows is to perform a *search* through the history. If nothing has been typed yet, this just goes through the history as is normally the case. But as soon as a few characters have been entered, only those commands in the history that start with the same fragment will come up when you press Up. This allows you to quickly retrieve a command you might have typed quite some time ago, as long as you know how it started:

```
# alternate mappings for "up" and "down" to search the history
"\e[A": history-search-backward
"\e[B": history-search-forward
```

For example, if you set a complicated environment variable at the beginning of your session, but now need to modify its value slightly, you could just type `exp` followed by the Up arrow key, and the chances are the first match will be the `export COMPLICATED_VARIABLE=some_other_complicated_value` line that you wanted to edit – no need to type it all in again...