MMB8052 - Practical 01 / MMB8052 - Practical 1

# MMB8052 Practical 01 - Introduction to the Linux Command Line

## Introduction to the Module

### Module Organisation

Welcome to *MMB8052 - Bioinformatics for Biomedical Scientists*. This module is mostly practical in nature - 10 computer lab sessions will introduce you to the fundamental computing tools used throughout much of modern bioinformatics, and will also provide case studies of the "read world" use of these tools. The lectures in the module will be delivered by a range of scientists from across the Faculty of Medical Sciences, and will highlight the *application* of bioinformatics in modern biomedical research.

## Module Assessment

Due to the practical emphasis of the module, the assessment will also focus on these practical aspects. The module is assessed through two exercises - one short answer quiz in which the solutions to the posed problems should be derived computationally (more on this later). The second assessment will be a lab report style write-up of the application of some of the tools you will learn how to use in the practicals. Assessment 1 is worth 25% of the module mark. Assessment 2 makes up the remaining 75%.

In addition to these assessed, summative exercises, there will also be a range of formative tests throughout the module. These will mostly take the form of multiple choice quizzes, integrated into the practical sessions or provided separately on Canvas. While these components are not assessed, or compulsory to complete, they will aid and reinforce your understanding of the material presented.

## About This Handbook

The practical sessions will each be accompanied by a handbook like this one. These handbooks will provide you with a lot of background information relevant to the practical at hand, and will provide you with walk-through instructions for what you are supposed to do in each session. Computer code (usually intended to be typed in to the appropriate computational interface - we'll get to what this means later) will be presented in chunks styled like this:

```
# these are some command line instructions:
$ pwd
/home/student
$ ls -l -h ~
```

The `$` sign is used to represent the beginning of each command, but ***should not be typed***.

Exercises, which will direct you to accomplish some computational task, will be presented like this:

### Example Exercise

Estimated time: 0 mins

The instructions to follow will be in this block of text.

- First instruction
- Second instruction
- etc.

## Linux

Linux is an umbrella term which describes a family of open-source, "Unix-like" computer operating systems which are based on the Linux kernel. The first Linux operating systems were released in the mid-90s and today they are used throughout computing, particularly in the infrastructure which runs the World-Wide Web, and in scientific computing and virtually all supercomputers. The Android smartphone operating system is a Linux system. Popular Linux distributions include:

- Ubuntu
- Fedora
- Debian
- MX Linux

## The Kernel

A kernel is the computer program at the heart of an operating system (OS), and is responsible for the control of everything in the system. The kernel facilitates interactions between hardware and software (via *drivers*) and optimises the use of system resources such as CPU time and RAM usage. The main kernels in use in modern computing are the Linux kernel, the Windows NT kernel and the MacOS kernel. All of the Linux distributions listed above use the same kernel at their core.

## Unix

The Unix operating system is ancient, in computing terms. It was conceived and implemented in 1969 at AT&T's Bell Labs. It is modular by design, with a number of robust tools each designed to perform a limited, well-defined function. A program, known as the Unix *shell* provides a text-based interface to these tools, and allows the user to combine them in order to perform complex workflows. Thanks to its efficient and robust nature, this computing paradigm persists today in the modern, Unix-like operating systems, Linux and MacOS.

> This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.
>
> -- Doug McIlory (Bell Labs)

## The Shell

A Unix shell is a command-line user interface for Unix-like operating systems. It provides a programmable environment for controlling the OS, and running executable programs. It is typical for the user of a Unix-like OS to interact with the shell via a *terminal emulator* - a program which simulates the features of a hardware video terminal interface. Feature-rich terminal emulators are available for all modern operating systems.

There are many different shell programs, all of which have different features. The shells you will encounter most often are bash (the "Bourne-Again Shell") which is the default in most major Linux distributions, and zsh (Z Shell) which is the default in MacOS. Zsh is backwards compatible with bash (so any code written for bash will work in zsh, but not necessarily vice versa).

## Logging in to a Linux Server

Rather than install Linux directly onto your desktops, we will use a terminal emulation program to log into a virtual cloud server which has been configured for these practicals. It should be noted that you'll be using your computer as a dumb terminal (client) that will display the information generated by programs running on the cloud VM (this is an example of a *client-server* model).

You should have received an email from Microsoft Azure, inviting you to register for the lab - click the 'Register for the lab' button in the email, and log in to Azure Labs with your University credentials.

Once logged in, you should see a heading 'My virtual machines', with a single entry underneath. Click the slider in the bottom-left of the VM box to power the machine on. Once it's started up (this will take a couple of minutes), click the icon in the bottom-right of the VM box which looks like a small grey monitor. Pick 'Connect via SSH' from the menu which appears, and set your password when prompted. Once the password has been set (again, this takes a couple of minutes) select 'Connect via SSH' again, copy the text in the popup box which appears - this is your SSH invocation to log in to your VM. For example:

```
ssh -p 65432 student@ml-lab-77568ef7-c936-416a-a10
```

## Logging in from Windows

### Exercise 1.1(a)

Estimated time: 2 mins

- Open the **Windows PowerShell** app (press Windows+S to search, then type 'powershell')
- At the prompt (`>`), paste in the connection string shown when you click 'Connect via SSH' on the Azure Lab (above). As per the example above:

```
ssh -p 65432 student@ml-lab-77568ef7-c936-416a-a10
```

- To paste into PowerShell, press the right mouse button at the prompt (Ctrl-C/Ctrl-V do not always work in the Terminal environment as they mean something different to copy/paste)

You will then be prompted for a password. Enter the password you set when starting your VM for the first time.

## Logging in from MacOS

### Exercise 1.1(b)

Estimated time: 2 mins

- Open 'Terminal.app' (located in /Applications/Utilities/Terminal.app).
- Paste in the connection string shown when you click 'Connect via SSH' on the Azure Lab (above). As per the example above:

```
ssh -p 65432 student@ml-lab-77568ef7-c936-416a-a10
```

- Enter your password when prompted.

## Running Commands

Once you have logged in, you should see what's known as a "command prompt" - it is here that you can type commands to be interpreted by the active Unix shell (on our VMs, the default shell is bash). You should see something like this in your terminal:

```
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 6.8.0-103

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

 System information as of Thu Sep 11 09:18:04 UTC 

  System load:  0.13               Processes:
  Usage of /:   2.1% of 123.87GB   Users logged in
  Memory usage: 8%                 IPv4 address fo
  Swap usage:   0%


Expanded Security Maintenance for Applications is 

0 updates can be applied immediately.

Enable ESM Apps to receive additional future secur
See https://ubuntu.com/esm or run: sudo pro status


The list of available updates is more than a week 
To check for new updates run: sudo apt update

Last login: Wed Aug 13 10:42:21 2025 from 128.240.
student@labTWBI48:~$
```

The bulk of this is information about the state and health of the system, produced by the OS on login. The final line contains 3 key elements:

1. `student@labTWBI4` is your username (`student`) at the name of the computer (`labTWBI4`)
2. `~` denotes your *current working directory* - more on this below - this tilde character is a widely used shorthand for the *home* directory.
3. The `$` symbol is the prompt, and will be used throughout these practicals to indicate the beginning of a bash command (you shouldn't type the `$`).

## Exercise 1.2

Estimated time: 2 mins

Try typing the following commands:

```
$ ls
$ touch emptyfile
$ ls
```

- What do you see after each command?
- Can you work out what the `ls` and `touch` commands are doing?

## Command Anatomy

The commands you can issue at this prompt are many and varied, but all share a common anatomy. This anatomy is illustrated in figure 1. They begin with the name of the executable command, then are followed by options (or flags), then finally come the arguments (or parameters). Options and arguments change the behaviour of the command in certain prescribed ways.

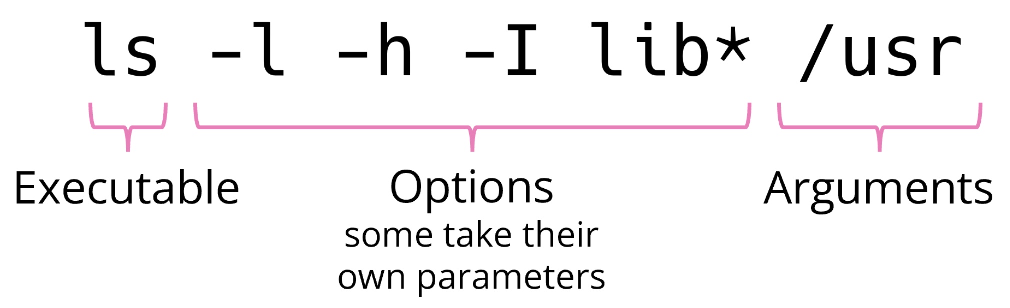

**Figure 1: The common anatomy of a Linux command**

The whitespace in the command can be just as important as the rest of the text - it is itself interpreted by the shell in specific ways, and the presence (or absence) of a space at a particular place in the command can completely change how it is interpreted. It's also worth mentioning at this point that bash is **case sensitive**.

We'll look in more detail at what the commands in exercise 2 (and many others) are for later on in the practical. For now, we have another important feature of Linux to introduce.

# The Linux File System

Modern operating systems appeal to a broad user base by layering abstractions on top of hidden complexity. The graphical interface to the file system ("Explorer" in Windows and "Finder" in MacOS) is one such abstraction. By making it convenient to search and browse for files, and providing easy access to commonly-used areas of the file system (Documents, Downloads etc) the user doesn't need to put much thought in to where files are actually stored.

Command line Linux does not offer these abstractions. Consequently, the onus is on the user to understand the topology of the file system, since otherwise things can get in a real mess - particularly because tracking down files if you don't know where they have gone can be difficult.

## File System Layout

For convenience, the Linux file system is usually thought of as having a tree-like structure (or a *hierarchy*). Everything in this hierarchy is a file, although some have special properties - for example a *directory* is a file which acts as a container for other files (some of which may themselves be directories). This tree has a *root* - indicated by a forward slash (`/`). This is a directory which contains all of the other directories and files in the file system.

A *path* describes a file system location as a string of characters, with each path component separated by a delimiting character. In Linux, the delimiting character is the forward slash (`/`). In the example in figure 2, the highlighted path is `/home/user1/Project1/results/mapping.bam`. This path uniquely identifies the location of the `mapping.bam` file in the file system. We can have other files named `mapping.bam`, but they will have their own path.

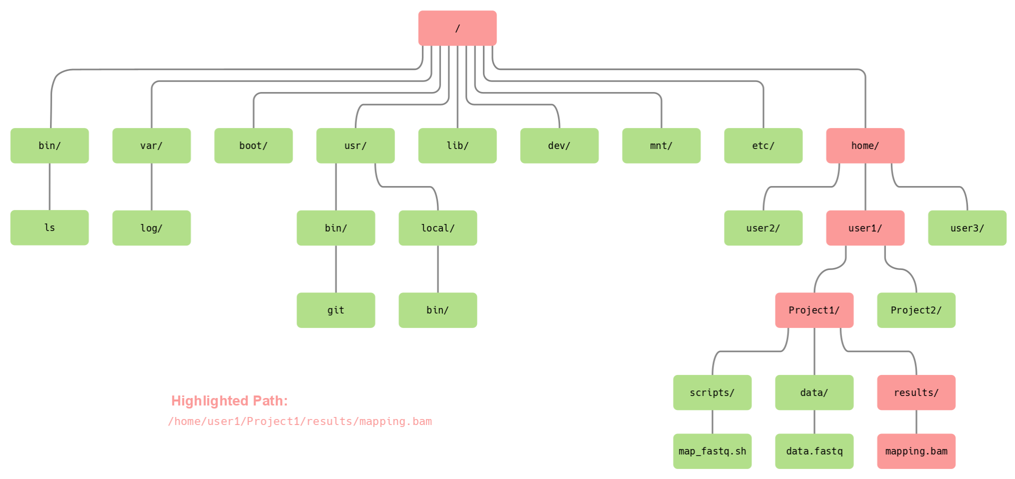

**Figure 2: A typical Linux file system hierarchy**

## Navigating the file system

When you log in to a Linux system (as in exercise 1, above), your shell places you in a particular directory, from which the commands you issue will be run. By default this location is your individual *home* directory - `/home/username` (where `username` is replaced by your username on the system you're using). Knowing this location (also known as your *current working directory*) is important because the commands you issue will run relative to this location. To explain further, consider the following:

```
# list the contents of the current working directo
$ ls
# list the contents of your home directory
$ ls /home/username
```

At login, these two commands are effectively equivalent, as your current working directory *is* your home directory. As soon as you change your current working directory, which you can do by using the command `cd`, the results of these two commands will diverge.

On the VM you are using, your current working directory is shown before the prompt (when you login you will see a tilde (`~`) before the `$` - this is a shorthand for home). If you want to find it explicitly, the command `pwd` prints your current working directory.

## Exercise 1.3

Estimated time: 5 mins

Run the commands listed below:

```
$ pwd
$ cd ..
$ pwd
$ cd /
$ pwd
$ ls
$ cd
$ pwd
```

Some things to consider:

- Does the output of `pwd` match the text in front of the prompt?

- Can you work out what `cd ..` does?
- How does the output of `ls` compare to the directories shown in figure 2?
- What does the third `cd` command do?

## Absolute and relative paths

File system locations in Linux can be described in two ways:

1. As a full path all the way to the root of the file system - this is known as an **absolute file path**
2. In relation to the current working directory - this is known as a **relative file path**

Absolute (or fully qualified) file paths always begin with a `/` forward slash. Relative file paths do not.

Consider the file hierarchy shown in figure 2 again. For `user1` at login (so therefore with `/home/user1` as the current working directory), the following are true:

- The *relative* path to the highlighted `mapping.bam` file is `Project1/results/mapping.bam`. Note the lack of a forward slash at the beginning of the path.
- The *absolute* path to this file is `/home/user1/Project1/results/mapping.bam`. This time there is a forward slash at the beginning - showing that the path takes us back to the root.
- The absolute path of the `ls` executable is `/bin/ls`. Try typing this fully qualified path at the command line.
- The relative path to this file is more complicated, since we need to navigate "up" the tree before going back down: `../../bin/ls`. The double dot notation means to look in the parent directory, so `../..` is the parent of the parent (in this case, the root directory -

the parent directory of `/home/user1` is `/home`, and the parent of `/home` is `/`).

Absolute file paths are always unambiguous addresses of a single file on the file system. Relative file paths have some inherent ambiguity, as the file they point to can differ depending on your current working directory. For example, `mapping.bam` may be a common output of running a command, so therefore you may have a `mapping.bam` file in lots of different projects, where the distinguishing feature is not the name of the file, but the name of a parent directory that defines that project. Running a command which points to `results/mapping.bam` (a relative path) will have a different effect depending on whether you run it from `~/Project1` or `~/Project2`.

| Command | Summary | Behaviour with no arguments |
|---|---|---|
| `ls` | List directory contents | List the contents of the current working directory |
| `cd` | Change working directory | Change to the home directory |
| `pwd` | Print working directory | No common arguments |

Table 1: Commands for navigating the Linux file system.

## File manipulation

Now we understand something of the layout of the file system, and how to navigate it we can begin to learn how to manipulate files. For this we will introduce 6 new commands:

- `echo` - "repeats" what follows the command in the terminal.

- `nano` - a simple command-line text editor.
- `cp` - copies a file.
- `mv` - moves a file.
- `mkdir` - makes (mk) a new directory (dir).
- `rm` - removes a file.

Before we start to look at these commands, we have to learn something about output streams and redirection.

## Output Streams

Many command line programs produce output in the terminal. We've already seen a couple of examples of this - `ls` prints its list of files in the terminal, for instance. There are actually two streams of output which a program can use to produce this effect - these are known as `STDOUT` (standard output) and `STDERR` (standard error). Conventionally, `STDOUT` is used for the results of the program, and `STDERR` is reserved for error messages.

We can take advantage of a feature of bash known as *output redirection* to manipulate the destination of this output - making it possible to store it in a file instead of printing it to the screen. It is even possible to use the output of one command as the *input* for a subsequent command.

Useful redirection commands:

```
# redirect the output of **command** to **file** -
$ command > file
# append the output of **command** to **file** - i
$ command >> file
# the input of **command** will be read from **fil
$ command < file
# the output of **command1** will be used as the i
$ command1 | command2
```

All of the redirection commands above work with `STDOUT`. `STDERR` will still end up being printed in the terminal. It is also possible to redirect `STDERR`, though not necessary for anything we're attempting to accomplish on this course.

We can make use of output redirection to create a file, since the `>` and `>>` operators will create a file if it does not already exist.

## Exercise 1.4

Estimated time: 5 - 10 mins

Run the commands listed below:

```
$ echo "Hello, World"
$ echo "Hello, World" > ~/myfile.txt
$ ls ~
$ nano ~/myfile.txt
```

Take some time to investigate how `nano` works - there are some instructions on screen when it opens.

Can you figure out how to change the file contents, how to save, and how to exit?

Some other things to consider:

- What does the `echo` command do?
- How does the addition of `> ~/myfile.txt` change the behaviour of the `echo` command?
- Is `~/myfile.txt` a relative or absolute path?
- What does `^` mean in the on-screen help for `nano`?

- Can you figure out how to use `echo` and `>>` to add extra content to the file?

## Copying and Moving files

`cp` and `mv` are the Unix commands for copying and moving files, respectively. In both cases they take two compulsory arguments - (1) the name of the file to act on, and (2) the destination location that you want it to end up. As the names suggest, after using `cp` you will have 2 copies of the file (the original and a copy at the destination), after using `mv` you will still only have one copy (the one at the destination location - the original will be removed). The destination file location can be in the same directory as the original file, but must have a different name, or it could be in a different directory - in this case the filename can remain the same.

```
$ ls
emptyfile myfile.txt
# copy myfile.txt (.bak is a Unix convention for a
$ cp myfile.txt myfile.bak
$ ls
emptyfile myfile.bak  myfile.txt
# rename the new file
$ mv myfile.bak mf.b
$ ls
emptyfile mf.b  myfile.txt
```

## Working with Directories

Directories are directly analogous to *folders* in Windows and MacOS. As described previously, they are special files which are capable of containing other files (and indeed other directories). Sensible and well-planned use of directories can impose a logical organisational structure on your filesystem. The command for creating new directories is `mkdir` (for **mak**e **dir**ectory) - like many other tools we've looked at so far, the simplest mode of operation for `mkdir` is to provide it with a single argument - the name of the directory you want to create.

```
# make a new directory called **mydir** in the cur
$ mkdir mydir
$ ls
emptyfile  mf.b  mydir/  myfile.txt
```

### Note

The syntax highlighting I use here cannot replicate the colours you will see in your terminal (where directories will be coloured in blue). I've addressed this by listing directories with a trailing slash. You can replicate this by adding the `-p` option to `ls` - the exact results seen in the block above can be obtained with: `ls --color=never -p`

Other commands already introduced work with directories too - `mv` will move a directory (and its contents) just like any other file. It is possible to copy whole directories with `cp` as well, although you do have to add the `-r` option for this to work correctly:

```
# move file in to new directory
$ mv mf.b mydir
# move the directory and its contents
$ mv mydir mynewdir
$ ls
emptyfile myfile.txt  mynewdir/
$ ls mynewdir
mf.b
# copy the directory and its contents
```

```
$ cp -r mynewdir mycpdir
$ ls
emptyfile mycpdir/  myfile.txt  mynewdir/
$ ls mycpdir
mf.b
```

## Deleting Files

All the commands we've looked at so far create files (and directories), however, sometimes it is necessary to delete things too. The command for removing files is `rm` and again, the simplest way of using it is with a single argument - the file to be deleted.

Unlike the operating systems you may be used to, there is no request for confirmation when you issue the `rm` command - the file will be immediately removed from the file system (it also doesn't go into any sort of recoverable "Trash" folder). For this reason, `rm` should be treated with caution - especially when used with the `-r` option (which like `cp` is used to "recursively" act on all the files in a directory, including the directory itself). If you do want to be prompted for confirmation before deleting a file, you can add the `-i` option to your `rm` command.

```
# remove a file
$ rm emptyfile
# remove a directory (and contents)
$ rm -r mycpdir
$ ls
myfile.txt  mynewdir/
# remove a file and prompt for confirmation
$ rm -i myfile.txt
rm: remove regular file 'myfile.txt'? n
$ ls
myfile.txt  mynewdir/
```

## Exercise 1.5

Estimated time: 5 - 10 mins

Run the commands listed below, use `ls` to keep track of your files throughout:

```
$ mkdir ~/mydir
$ cp ~/myfile.txt ~/mydir
$ cat ~/mydir/myfile.txt
$ mkdir -p ~/level1/level2
$ touch ~/level1/level2/myfile.txt
$ mv ~/level1/level2/myfile.txt ~/mydir
$ cat ~/mydir/myfile.txt
$ rm -r ~/level1
```

Quick quiz:

- There's a new command in here (`cat`) - can you work out what it does?
- What happens to the file `~/mydir/myfile.txt` between the two times you run `cat` on it?
- What does the `-p` option do to the behaviour of `mkdir`?
- What happens if you run command 4 without this option?
- What is the effect of the final command on `~/level1/level2`?

## Viewing files

Linux offers several commands for viewing the contents of a text file on the screen.

You've already met `cat` - this command will print the contents of a file to `STDOUT`. If the file is long, the text will scroll up the screen until the end of the file is reached.

The `less` command lets you view text one page (or screen) at a time. The space bar moves forward one page, pressing "B" moves back one page and pressing "Q" quits.

`head` will show you the first lines of a text file and `tail` will show you the last lines. Both show 10 lines by default, but this can be changed with the `-n` option.

Note that running `head`, `tail` or `cat` without specifying a filename will cause the command prompt to "hang" (become unresponsive to input). Pressing Ctrl+C will cancel the command by sending a *kill* signal to your terminal.

Pressing Ctrl+C will abort most processes in Linux on the command line.

## Getting Help

You will have realised by now that the catalogue of Unix commands is very large, and each individual command comes with an array of options and arguments to modify the default behaviour. This means that, especially for the beginner, command line computing can quickly become overwhelming. Fortunately, it is not necessary (nor is it possible) to remember all of the commands and their available options. Linux has an extensive built-in help system to assist you with navigating this bewildering complexity, `man` (short for Manual).

```
# get help on the ls command
$ man ls
```

This command will display the `man` page for the `ls` command. Linux manual pages follow a common structure, which should make it easy to track down the information you require, once you know how to navigate them.

| Section | Description |
|---|---|
| NAME | Name of the command |
| SYNOPSIS | General usage parameters of the command |
| DESCRIPTION | Describes what the command does, and lists all of the options of the command |
| SEE ALSO | Lists other commands that are directly related to the command in the man page or closely resemble its functionality |
| REPORTING BUGS | Explains how/where to report bugs with the tools |
| EXAMPLES | Common usage examples that give the reader an idea of how the command can be used |
| AUTHOR | The author(s) of the man page/command |

Table 2: The common sections of a `man` page. Other sections may be present, but these are not well standardized across `man` pages.

The SYNOPSIS, in particular, is useful for giving the general form of the command including any required arguments. The SYNOPSIS has its own standard format, for example let's look at the SYNOPSIS for `ls`:

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

Anything in square brackets (`[]`) is *optional* and the `...` means it is possible to provide one or more of that thing (so `ls` can be given multiple options, and multiple file paths). The full list of available options, and their functionality is given in the `DESCRIPTION`.

### Exercise 1.6

Estimated time: 5 - 10 mins

Read the `man` page for the command `cat`. Use the page to figure out the following:

- What does the command `cat` do?
- Can you work out how to get `cat` to print the file `myfile.txt` with and without line numbers?
- What do you think the command listed in `SEE ALSO` does?

**NOTE** - `man` pages open in the system "pager" program, which on most modern Linux systems is `less`. This means that to close an open `man` page, you will need to press "Q".

## Downloading Data at the Command Line

`wget` (web get) is described as a "non-interactive network downloader" and is a command-line utility which will download the content at a given URL to your Linux file system. This makes `wget` a very useful command for downloading software and data to your command-line driven machine. It is particularly powerful in combination with particular kinds of *web service*.

## RESTful API

An API is an **A**pplication **P**rogramming **I**nterface, and provides a way for computer programs to communicate with one another.

REST stands for **Re**presentational **S**tate **T**ransfer, which is a set of guidelines for developing web services in a particular style. REST is popular due to its relative simplicity.

A RESTful API is a web API which provides stable addresses for resources on the web. These endpoints can be used to provide access to these resources, and to allow interaction with them. Many popular biological databases provide a RESTful API for working with their data.

## UniProt API

The Universal Protein Resource (UniProt) is a comprehensive resource for protein sequence and annotation data. The database is a collaboration between the European Bioinformatics Institute, the Swiss Institute of Bioinformatics and the Protein Information Resource. The UniProt Knowledgebase contains high-quality, annotated protein sequence records in two sections, UniProtKB/Swiss-Prot, a manually annotated, non-redundant database; and UniProtKB/TrEMBL, a larger, computationally annotated database. UniProt provides programmatic access to the database via a RESTful API.

Every protein in the UniProt database has a unique *accession* which identifies the entry. These accessions provide stable API endpoints for each protein in the UniProt database. For example, the human PAX6 protein has the accession P26367, for this protein:

- The plain text database entry: https://rest.uniprot.org/uniprotkb/P26367.txt
- The protein sequence (FASTA format): https://rest.uniprot.org/uniprotkb/P26367.fasta

- Sequence features (GFF format): [https://rest.uniprot.org/uniprotkb/P26367.gff](https://rest.uniprot.org/uniprotkb/P26367.gff)

The UniProt API allows us to construct these endpoints for any protein in the database, by replacing *P26367* in the addresses above with the accession for our protein of interest.

## Exercise 1.7

Estimated time: 5 - 10 mins

The human Retinoblastoma protein (RB1) has the accession **P06400**, use `wget` to download the plain text database entry for this protein.

You can view the file you've downloaded using the *pager* program `less` (open the file using `less filename`, then use the space bar to page through the file. Press 'Q' to quit).

Use the entry you've downloaded to answer the following questions:

- How many amino acids are in the RB1 protein sequence?
- List 3 proteins that RB1 interacts with.
- What are the last (C-terminal) 10 amino acids of the primary sequence?
- Can you find the publication and PDB ID associated with an X-Ray crystallography structure of the protein which has a resolution of 2.3 Angstroms?

**HINT** - with a file open in `less`, you can use forward slash (`/`) followed by a search term to find specific text in the file (try `/pdb`, for example). You can read more about this functionality in the `man` page for `less`.

## Searching UniProt with the API

The UniProt API, as well as giving straight-forward access to individual proteins, as explored above, also allows more complicated interactions with the UniProt database. The `stream` API endpoint allows the user to conduct programmatic searches of the database according to the parameters provided. For example, the following address:

https://rest.uniprot.org/uniprotkb/stream?query=reviewed:true+AND+organism_id:9606&format=fasta

Will return all of the sequences of UniProtKB/Swiss-Prot entries (`reviewed:true`) for human (`organism_id:9606`) proteins, formatted according to the FASTA specification (`format=fasta`).

It is possible to construct these queries using the 'Advanced Search' on the UniProt website.

Using `wget` with this address will download the full set of 20,386 manually curated human proteins:

```
$ wget -O human.fa 'https://rest.uniprot.org/unipr
```

The address has to be enclosed in quotes, as it contains characters with special meaning in `bash`, which would alter the behaviour of the command otherwise. Additionally, this address does not end in a neat file name like the simple API requests in the previous section, so we tell `wget` the name of the output file using the `-O` option - in this case, our results will be stored in a file called `human.fa` in the current working directory.

### Exercise 1.8

Estimated time: 5 - 10 mins

Using the UniProt `stream` API, as described above, download the sequences of proteins which match the following criteria:

- protein_name contains 'Interferon'
- organism_id = 9606 (human)
- reviewed = true

We will look at some tools which will allow us to examine these results more carefully in the next practical.

## Summary

In this first practical we have encountered the Linux command line for the first time and introduced commands for navigating the filesystem, creating and editing files and directories and for downloading data from the web. Due to the maturity of these tools, and the consistent way in which they are developed, the fundamentals we have learned here should allow us to pick up and use virtually any command line tool. This is particularly true if these tools have a `man` page or a well-written help file.

In future practicals, we will investigate more of these tools, and different ways of using them.