Introduction to Bioinformatics for Malaria Molecular Surveillance

Fighting Malaria Across Borders

Pieter Moris

August 2, 2023

Table of contents

About

Welcome to the Bioinformatics Module of the Fighting Malaria Across Borders (FiMAB) created by the Institute of Tropical Medicine (ITM) in Antwerp (Belgium). This is an international training programme (a VLIR-UOS project) to support the implementation of targeted NGS AmpliSeq assays to strengthen malaria molecular surveillance and help guide national control programmes. In conjunction with laboratory training, this bioinformatics course is intended to allow young academics around the globe to become familiar with molecular surveillance as a key activity to monitor transmission, sources of epidemics and the emergence and spread of drug resistant mutations in the *Plasmodium* parasite.

Scope of the course

This course aims to provide an overview of key bioinformatics aspects related to performing population genetics and molecular epidemiological research in *Plasmodium*. It is divided into the following sections:

1. Introduction to the Unix shell (command line interface) and basic scripting
2. Introduction to R
3. Genomic files and tools
4. Population genetics and molecular epidemiology for *Plasmodium*

Section 1-3 are online self-paced modules, whereas section 4 will include classroom lectures and practical sessions. Evaluation exercises will be conducted on the ITM course page.

Acknowledgements

Development of this course was supported by VLIR-UOS and the Institute of Tropical Medicine.

License

Introduction

What is bioinformatics anyway?

Bioinformatics is a scientific discipline that develops and utilises computational methods to analyse large amounts of biological data, typically molecular sequence data such as DNA, RNA and amino acid sequences, as well as annotations describing those sequences. It spans the entire spectrum of collecting, storing and annotating data, to modelling, predicting and discovering new biological insights, from the level of individual molecules, to cells, organisms and populations. Bioinformatics is a highly interdisciplinary field that brings together many different types of researchers, including biologists, computer scientists, statisticians, clinicians and even chemists and physicists. A closely related term that you might see being used interchangeably with bioinformatics, is *computational biology*. However, there is no consensus on a clear distinction between the two[1]; like so many things in nature (e.g. the concept of a species), scientific disciplines are often hard to define and delineate.

A few examples of popular topics in bioinformatics are:

- Population genetics and molecular epidemiology (what we will be focusing on in this course)
- Analysis of gene/protein expression and regulation, e.g. in disease models (for RNA-seq spatial and single-cell approaches are become more prevalent)
- Structural bioinformatics, e.g. predicting the structure of proteins
- Network and systems biology: mapping and analysing the relationships between interacting biomolecules such as proteins, metabolites and their signal cascades, e.g. gene regulatory networks)

[1]

- https://biology.stackexchange.com/questions/3192/is-computational-biology-different-from-bioinformatics
- https://www.nature.com/subjects/computational-biology-and-bioinformatics
- https://www.ebi.ac.uk/training/online/courses/bioinformatics-terrified/what-bioinformatics/
- https://www.genome.gov/genetics-glossary/Bioinformatics

- Comparative genomics and phylogenetics: studying the ancestry of species, genes or entire genomes through time and space
- Genomic annotation and cataloguing genetic mutations and associated diseases in databases

Computational thinking

We hope that this course can teach you a few computational thinking and problem solving skills that will help you along your bioinformatics journey. The learning curve in computational biology can be quite steep at times and the path is littered with arcane commands and obtuse syntax, but as you practice the concepts introduced in this course on your own, your command-line efficiency will improve and you will start to spot similarities across different types of environments and languages. Through this course, we hope to arm you with the necessary skills to make tasks like running custom analysis scripts or installing bioinformatics software seem a little less daunting.

Part I

Introduction to the Unix shell

1 What is a CLI?

This section of the course will introduce you to the general concept of the Unix command line interface (CLI) - as opposed to the graphical user interface (GUI) that you are familiar with - and Bash, one of the most ubiquitous Unix shells.

1.1 Learning objectives

- Knowledge of what a Unix shell and the CLI are and why/when they can be useful.
- Setting up your own Unix environment.
- Familiarity with basic `bash` commands for e.g., navigation, moving/copying and creating/deleting/modifying files and directories.
- Introduction of a few more advanced commands and concepts like redirection, piping and loops.
- First look at scripts and how they can be used in the context of DNA sequencing pipelines for variant calling.

Resources

This section of the course draws inspiration from the following resources:

- Conor Meehan's UNIX shell tutorial (CC BY-NC-SA 4.0)
- Mike Lee's Unix Crash Course (https://doi.org/10.21105/jose.00053)

- [Data Carpentry's Introduction to the Command Line for Genomics](https://datacarpentry.org/shell-genomics/ (https://doi.org/10.5281/zenodo.3260560 CC-BY 4.0)
- Ronan Harrington's Bioinformatics Notebook (MIT)
- A Primer for Computational Biology by Shawn T. O'Neil (CC BY-NC-SA)

1.2 What is Unix?

Unix is a family of operating systems, with one of their defining features being the *Unix shell*, which is both a **command line interface** and **scripting language**.

In simpler terms, shells look like what you see in the figure below and they are used to *talk* to computers using a CLI - i.e., through written text commands - instead of via a **graphical user interface** (GUI) where you primarily use a mouse cursor.

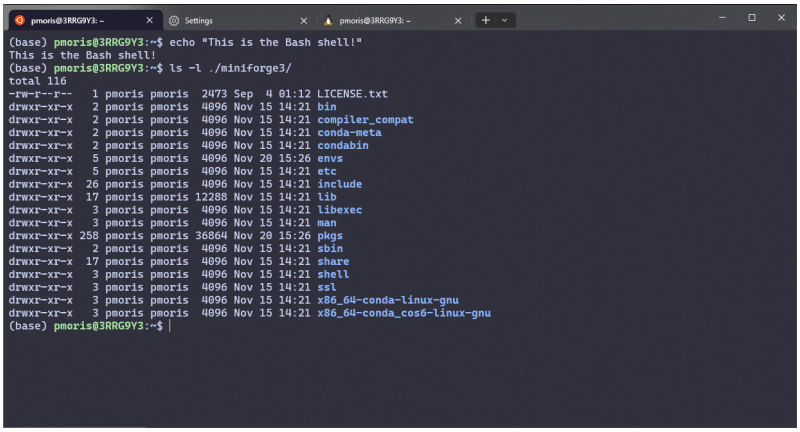

Figure 1.1: Bash shell in WSL

There exist many different flavours of Unix, collectively termed "*Unix-like*", but the ones you will most likely encounter yourself are Linux (which itself comes in many different varieties we call distributions, e.g. Debian, Ubuntu, Fedora, Arch, etc.) and

MacOS. These operating systems come with a built-in Unix shell. While Windows also comes with a command line interface (Command Prompt and PowerShell), it is not a Unix shell and thus uses different syntax and commands. We'll dig into how you can get your hands on a Unix shell on a Windows machine in a later section. The most ubiquitous Unix shell is **Bash**, which comes as the default on most Linux distributions.

1.3 Why bother learning the Unix shell as a bioinformatician?

Even if you are primarily a wet lab scientist, learning the basics of working with CLIs offers a number of advantages:

- **Automation**: CLIs and scripting excel at performing repetitive tasks, saving not only time, but also lowering the risk of mistakes. Have you ever tried manually renaming hundreds of files? Or adding an extra column to an Excel spreadsheet with millions of rows?
- **Reproducibility**: reproducibility is key in science and by using scripts (and other tools like `git`, package managers and workflow systems) you can ensure that your analyses can be repeated more readily. This is in stark contrast to the point-and-click nature of GUIs.
- Built-in **tools**: the Unix shell offers a plethora of tools for manipulating and inspecting large (text) files, which we often deal with in bioinformatics. E.g., DNA sequences are often stored as plain text files.
- Availability of **software**: many bioinformatics tools are exclusively built for Unix-like environments.
- Access to **remote servers**: Unix shells (usually bash) are the native language of most remote servers, High Performance Computing (HPC) clusters and cloud compute systems.
- **Programmatic access**: CLIs and scripts allows you to interact in various ways (e.g., via APIs) with data that is stored in large on-line databases, like those hosted by NCBI or EBI.

As a concrete example of what we will be using the shell for, consider the task of processing hundreds of *Plasmodium* DNA sequencing reads with the goal of determining the genetic variation in these samples (e.g., the presence of SNPs). Suppose we were to do this in a GUI program, where we would open each individual sample and subject it to a number of analyses steps. Even if each step were to only require a few seconds (in reality, minutes or even hours…), this would take quite a long time and be prone to errors (and quite boring!). With shell scripting, we can automate these repetitive steps and run the analysis without requiring human input at every step. Some of the key techniques we will use for this are:

- Navigating to directories and moving files around
- Looping over a set of files, calling a piece of software on each of them.
- Extracting information from a particular location in a text file.
- Compressing and extracting files.
- Chaining commands: passing the output of one tool to another one. E.g., after aligning reads to a reference genome, the resulting output can be fed to the next step of the pipeline, the variant caller.
- Etc.

1.4 Don't get discouraged

Learning to use the shell, or learning programming languages and bioinformatics skills in general, can be daunting if you have had little experience with these types of tasks in the past. Don't worry though, just take your time and things will become easier over time as you gain more experience.

We do not expect you to be able to memorize every single command and all of its option. Instead, it is more important to be aware of the existence of commands to perform particular tasks, and to be able to independently retrieve information on how to use them when the need arises.

Finally, the appendix (Section A.1) of the course also contains a bunch of tips and tricks to keep in mind while learning your way around the shell.

1.5 A note on terminology

You will often see the terms `command line (interface)`, `terminal`, `shell`, `bash`, `unix` (or `unix-like`) being thrown around more or less interchangeably (including in this course). Most of the time, it is not terribly important to know all the minute differences between them, but you can find an overview here if you are curious: https://astrobiomike.github.io/unix/unix-intro.

2 Setting up your own Unix shell

In order to get started, you need to get access to a Unix environment of your own. You can either work locally on your own computer or you can use the online environment that we have created. The latter comes with a `bash` shell and will immediately give you access to a bunch of files that we will use throughout the course and exercises.

There are also online playgrounds/simulations available to try out the Unix shell, for example https://sandbox.bio/playground. These are great to learn and we highly recommend checking out some of the tutorials on there, but the downside of course is that they are not true true environments and you cannot interact with your own files. You can use this site to follow along while learning some of the basic unix commands, but you will need to switch to a different option for the exercises eventually.

2.1 Online Unix environment

We have provided two different options for getting access to an online unix environment: through Binder (free, but less powerful) or through GitHub Codespaces (free for 60 hours per month).

Both options will launch an environment containing all relevant training files, based on this GitHub repository.

2.1.1 Binder

To access the remote `bash` shell, browse to https://mybinder.org/v2/gh/pmoris/FiMAB-bioinformatics/HEAD and wait for the launcher to start. This process can take quite a while, so be patient.

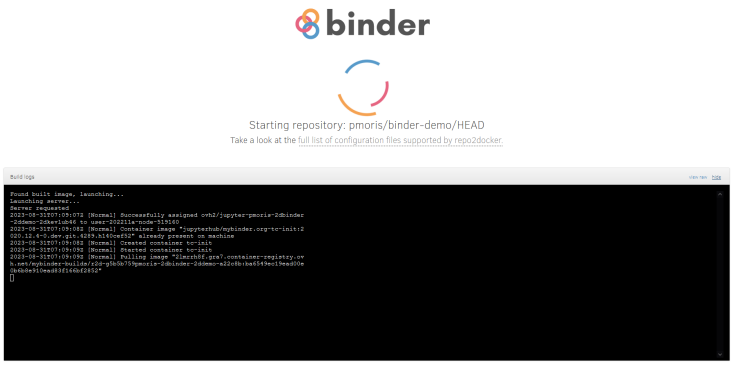

Figure 2.1: Launching the Binder environment

Eventually, you should be greeted by a screen (Jupyter Lab) with a number of launchers. Simply select the one labelled "Terminal" (a black square with a white $) and you should be all set.

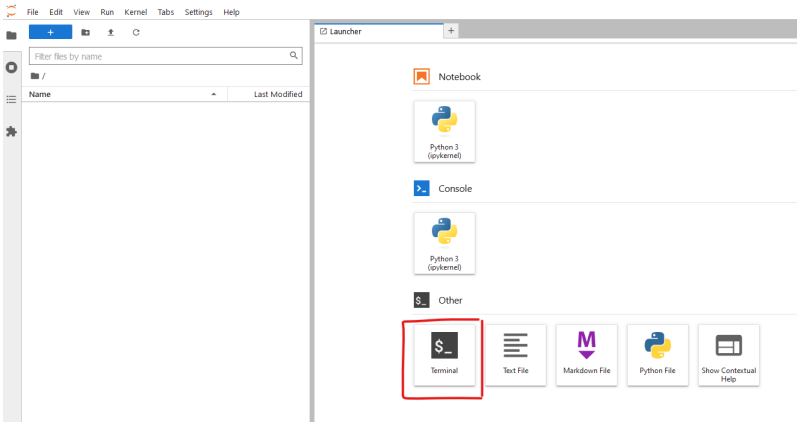

Figure 2.2: Starting a new Bash shell

> **What is Binder?**
>
> Binder is a service that allows people to share a customized compute environment based on a Git repository. It is mainly aimed at sharing Jupyter Notebooks (Python), but it also supports RStudio, Shiny and fortunately for us, a plain `bash` terminal too.
> You can find more info on the Binder website.

2.1.2 GitHub Codespaces

To access Codespaces, you will first need to create a GitHub account via https://github.com/signup. Just follow the instructions and be sure to enable one of the two-factor authentication options (via a TOTP app like Authy, Google Authenticator or Microsoft Authenticator, or via text messages), otherwise you might not receive access to Codespaces.

Afterwards, you can click this link, optionally change the region to the one closest to you under *change options* (but leave the machine type on 2-core to remain eligble for 60 free hours!), and then press the *Create codespace* button.

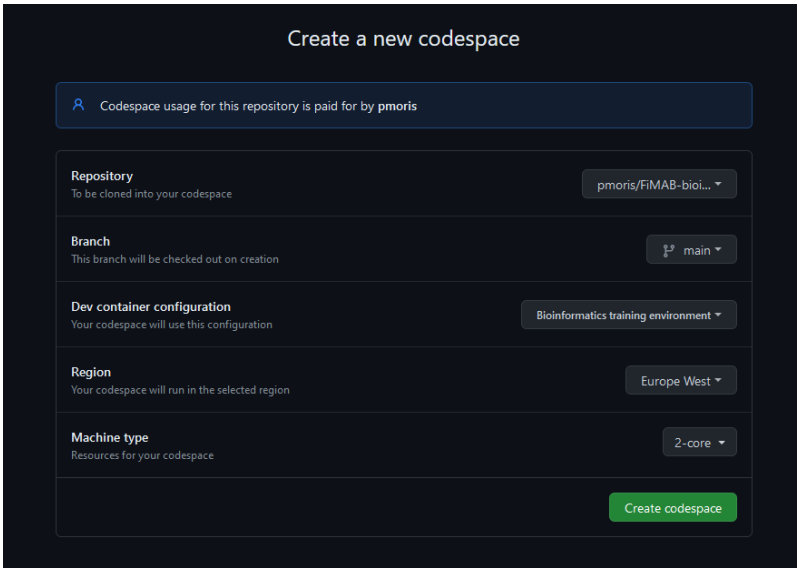

Figure 2.3: Creating a new codespace

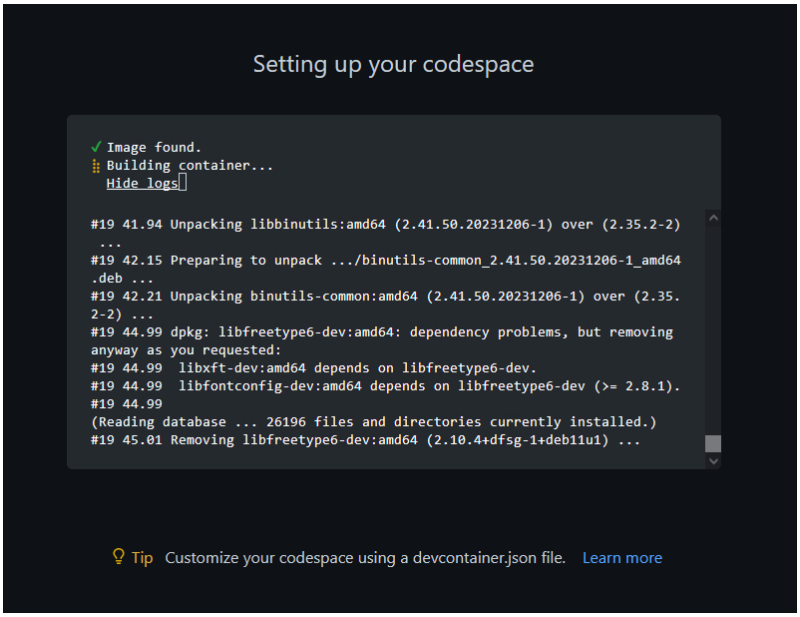

Figure 2.4: Launching a new codespace

Setting up the codespace can take a while, but eventually you will be greeted by a VSCode environment. The terminal is accessible at the bottom (or by pressing the hamburger icon in the top left and selecting "new terminal").

You will only receive 60 hours of free usage of Codespaces per month. This means you should manually shutdown your codespace whenever you are done with it. Otherwise it will keep running for 30 more minutes (by default). Just closing your browser will not shut down the workspace. Instead, you need to manually shut it down from within the codespace (by clicking the `><` button in the bottom left corner and selecting *stop current codespace*) or by browsing to https://github.com/codespaces and shutting it down from that page.

> **What are GitHub and `git`?**
>
> GitHub is a place to host code and software via a tool named `git`, which is a *version control system*. It allows you to keep track of the history of your code, easily revert

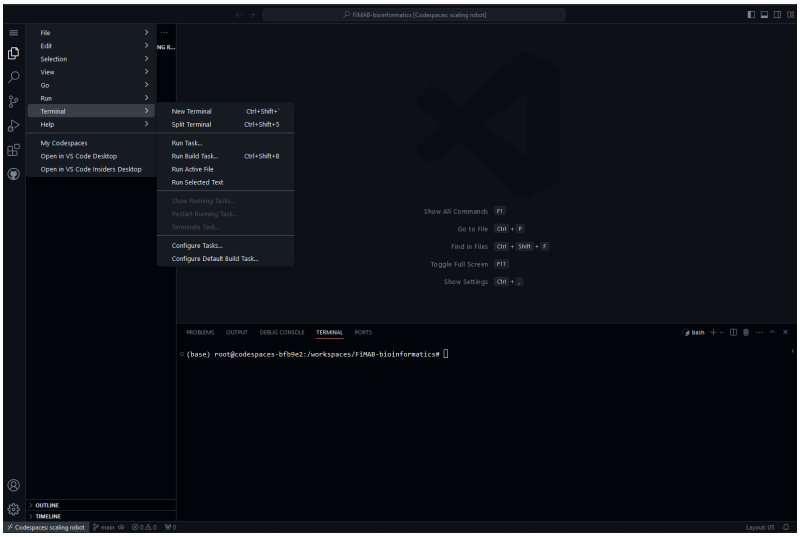

Figure 2.5: Terminal inside VSCode editor in GitHub Codespace

changes and allows for collaborating with multiple people on the same project. We will not go into further detail on using version control, but for now just remember that it can play an important role in scientific reproducibility.
If you want to learn more about `git` already, you can have a look at the following resources:

- https://happygitwithr.com/
- https://hwheeler01.github.io/CompBio/github/
- https://pmoris.github.io/git-workshop/ (self-promotion)

What is GitHub CodeSpaces

Similar to Binder, Codespaces are development environments that are hosted in the cloud. This is a paid service provided by GitHub/Microsoft, which offers 60 hours of free usage per individual per month. Instead of Jupyter notebooks, Codespaces use code editors, like VSCode and Jetbrains IDEs, which come bundled with a `bash` terminal too.

> You can find more info in the GitHub Codespaces docs.

2.2 Local Unix environment

If you are using MacOS or Linux, then you will already have access to a Unix shell (either `bash` or `zsh`, which will mostly behave identical for our purposes). To access it, simply search for a program called `Terminal` (or search for anything resembling "command", "prompt" or "shell").

In case you are using a Windows machine, things are slightly more complex and different methods exist, each with their own pros and cons. You could use a fully-fledged virtual machine like VirtualBox to emulate a Linux machine within Windows. Or you could rely on the minimal `bash` emulator that comes bundled with git for windows. However, nowadays we recommend that you use the Windows Subsystem for Linux (WSL), which was developed by Microsoft itself. In our opinion, it is one of the most polished methods to get access to a (nearly) full-featured Linux environment from within Windows, without the overhead of a full virtual machine or dual boot setup (dual boot means you install two different operating systems on your machine, and you switch between them when booting). For instructions on how to set it up, you can refer to this section.

2.2.1 Download the course files

Regardless of what type of local Unix environment you use, you will need to download the files that we will be using in our examples and exercises. You can do this directly on the command line or by manually downloading the files in the correct location.

1. Open your terminal and `cd` to a location where you want to place the training files.
2. Enter the command `git clone https://github.com/pmoris/FiMAB-bioinformatics.git`.
3. Afterwards, a new directory named `FiMAB-bioinformatics` will have been created.

It should look similar to this:

```
$ git clone https://github.com/pmoris/FiMAB-bioinformatics.git
Cloning into 'FiMAB-bioinformatics'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 7 (delta 0), reused 7 (delta 0), pack-reused 0
Receiving objects: 100% (7/7), done.
```

Alternatively,

1. Browse to https://codeload.github.com/pmoris/FiMAB-bioinformatics/zip/refs/heads/main
2. Save the `.zip` file in a directory accessible by your Unix environment. For Windows/WSL, the easiest option is to choose the Linux file system (e.g., `\\wsl.localhost\Ubuntu\home\pmoris`), which is accessible by clicking the Linux/WSL entry in your explorer.
3. Extract/unzip the file.

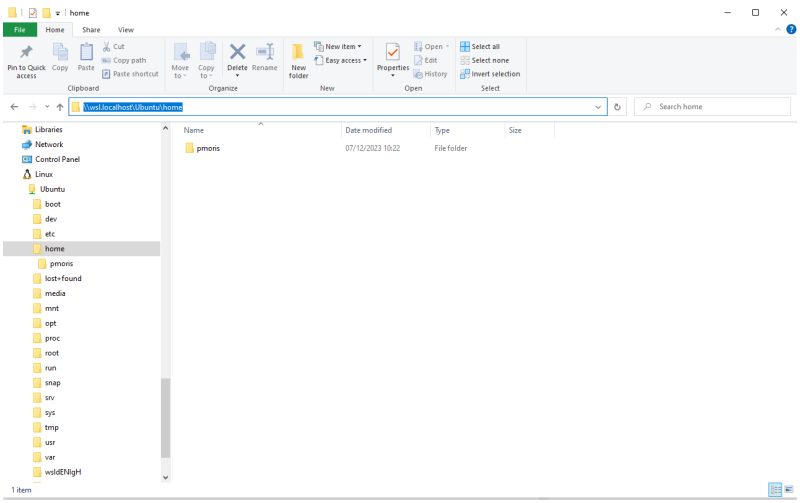

Figure 2.6: Linux file system inside Windows File Explorer

2.2.2 WSL installation

If you are using an updated version of Windows 10 (or 11), you should meet all the requirements and can simply follow the installation instructions listed here: https://learn.microsoft.com/en-us/windows/wsl/install. We recommend that you follow the instructions for WSL 2 (default), rather than the older WSL 1, and use the default Ubuntu 22.04 distribution (Linux comes in many different flavours, Ubuntu being one of the more popular ones).

Briefly:

- Open Windows PowerShell as administrator by right clicking your Windows Start Menu or searching for it in your list of applications.
- Type `wsl --install` and press enter.
- Afterwards, restart your PC.
- You can then launch WSL by searching for `wsl` or `Ubuntu` in your start menu.
- The first time you launch WSL, you will need to configure it.

If you use software like RStudio or VSCode, you can tell these programs to use WSL as their built-in terminal from now on, instead of Command Prompt.

> **i** WSL1 vs WSL2
>
> WSL 2 is the newer version of WSL 1. For most tasks, WSL 2 tends to be much faster, hence why we (and Microsoft) recommend using it in favour of the previous version. However, WSL 2 is only faster when you interact with files that are stored directly on the WSL file system, rather than working directly on the Windows file system. More info on the distinction between these file systems can be found further below and in Microsoft's WSL documentation.
> You can switch between WSL1 and WSL2 on the fly by just calling `wsl --set-version <distro_name> 2` (or 1) in PowerShell, so feel free to experiment for yourself.

> For a full overview of the differences, check out: https://docs.microsoft.com/en-us/windows/wsl/compare-versions.

2.2.2.1 Configuring WSL

Microsoft also provides an excellent tutorial on setting up your WLS environment, which you can find here.

After installing WSL and a Linux distribution, you will have access to it via its own built-in terminal emulator. It should be located in your Windows Start Menu with a name corresponding to the distribution that you installed, e.g. `Ubuntu 20.04 LTS`, or simply `wsl`.

The first time you run WSL, you will need to setup a Linux username and password. Note that while you are entering a password, nothing will appear on the screen, but this is intended (blind typing). The username will determine, among other things, the name of your home folder, whereas the password will grant you administrator rights (referred to as super users or admins in Linux land; the `sudo` command is used to invoke these rights).

You will also need to upgrade the packages by running the following command: `sudo apt update && sudo apt upgrade`, followed by your password.

For more information, check the docs.

2.2.2.2 Accessing files across the Windows and WSL file systems

> **Note**
>
> Some of the information below might be a bit confusing at this point, but things should become more clear after working your way through the Unix section of this course.

Newer versions of WSL will automatically add a shortcut to the WSL file system in your Windows File Explorer (look for Tux, Linux' penguin mascot). The file path will look similar to `\\wsl$\Ubuntu\home\<user name>\Project`, indicating that Windows treats the WSL file system as a sort of network drive. You can also open a file location in Windows File Explorer from within a WSL terminal (e.g. after you browse to a particular directory `cd ~/my-project`) by simply using the command `explorer.exe .` (don't forget the dot!).

Vice versa, you can also access the Windows file system from within WSL because it is mounted under `/mnt/c`. So, you could for example do something like `cp /mnt/c/Users/<user name>/Downloads/file-downloaded-via-webbrowser ~/projects/filename`.

More information can be found in the WSL documentation.

2.2.2.3 Windows Terminal

Even though WSL comes with its own terminal application, it is rather bare-bones and can make some operations like copying and pasting via `CTRL+C`/`CTRL+V` a bit tricky (you will need to use `CTRL+SHIFT+C` to copy and right mouse click to paste). Fortunately, Microsoft has also been working on a new terminal emulator that is much nicer to work with. Meet the Windows Terminal.

3 Using the shell

> **Prior experience**
>
> You can skip this section and proceed directly to the exercises if you are already familiar with the basic syntax of unix commands. We still recommend checking out the tips and hints in Section A.1 and the special syntax overview in Section A.2 though.

3.1 Interacting with the shell

When you launch your (Bash) shell, you will be greeted by what is called a shell prompt: a short snippet of text followed by a cursor, which indicates that the shell is waiting for input. The prompt can look different on different systems, but it often consists of your linux username followed by the name of your machine (like in the picture below) or sometimes just a single `$` symbol. When you see the prompt, you can enter commands interactively and execute them by pressing enter.

Figure 3.1: A bash shell prompt waiting for user input

Already note that you cannot use your mouse cursor to move around your terminal. You will need to use your arrow keys (or shortcuts) to move around while typing commands.

3.2 Command syntax

Unix commands generally follow the format:

```
command [OPTIONS] argument
```

where,

- `command` is the name of the (usually built-in) command that you want to execute.
- `[OPTIONS]` is a list of optional flags to modify the behaviour of the command. They are often preceded by a single (`-`) or double (`--`) dash.
- `argument` is a thing that your command can use. E.g., it can be a file name, a short piece of text (or *string*)

Try it yourself with the following command:

```
echo "Hello world!"
```

What did that do? (Click me to expand!)

`echo` is a command that simply prints a message to your screen (technically, to the *standard output stream* (stdout) of the terminal). `echo` is the command, teling the shell what we want to do. `"Hello world!"` is the target, in this case the message we want to print.
We place the message between quotes (`"`) because it contains spaces, and as you will see, spaces (and certain other special characters) can cause confusions. For now, just note that the message that gets printed, is whatever was written between the quotes, but not the quotes themselves.

```
$ echo "Hello world!"
Hello world!
$
```

3.3 Tips and hints

We have compiled a number of helpful tips in the appendix of this course (Section A.1), some of which will hopefully be

helpful on your journey towards mastering the unix shell. For now, we recommend at the very least checking out the section on tab-completion and your command history. In fact, you can give it a try already. Just press the up arrow and see if you can recall your previous command! Next, try to type out `ec`, and press `<tab>`, to see auto-complete in action.

4 Navigating the Unix file system

> **Prior experience**
>
> You can skip this section and proceed directly to the exercises if you are already familiar with the Unix directory structure and basic commands like `cd` and `ls`.

> **Tip**
>
> We have provided a list of helpful tips and hints in the appendix: Section A.1. Have a look already and refer back to it after you have worked your way through the next sections on navigation and basic commands.
> Additionally, there is an overview of some of the most common symbols that are used by the unix shell here: Section A.2.

4.1 Layout of the Unix file system

All files and directories (or folders) in Unix are stored in a hierarchical tree-like structure, similar to what you might be used to on Windows or Mac (cf. File Explorer). The base or foundation of the directory layout in Unix is the *root* (`/`) (like the root of a tree). All other files and directories are built on top of this root location. When navigating the file system, it is also important to be aware of your current location. This is called the *working directory*.

The address of a particular file or directory is provided by its *filepath*: this is a sequence of location names separated by a

forward slash (/), like `/home/user1`. Note that this differs from the convention in Windows, where backslashes (\) are used in file paths instead.

There are two types of file paths: *absolute* and *relative* paths.

- Absolute file path: this is the exact location of a file and is always built up from the root location. E.g., `/home/user1/projects/document.txt`.
- Relative file path: this is the relative address of a file compared to some other path. E.g., from the perspective of `/home/user1`, the file `document.txt` is located in `projects/document.txt`.

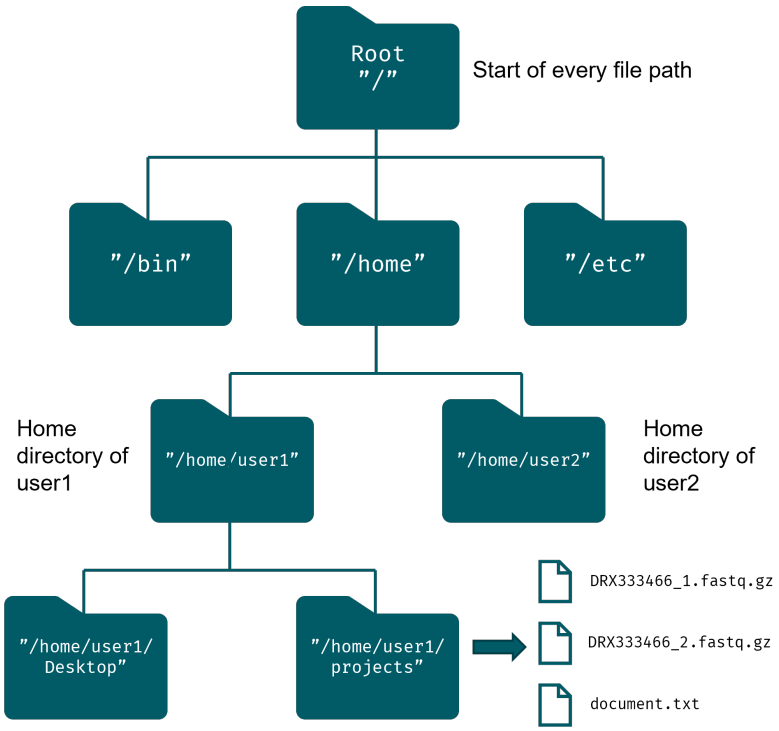

Figure 4.1: Overview of the Unix file system or directory layout

Another important location is the *home* directory. In general, every user has their own home directory, found in `/home/username`. A frequently used shortcut for this is the tilde symbol (`~`). Depending on the current user, this will refer to a particular directory under `/home/.`.

💡 How can `user1` write the file path to `document.txt` using the `~` shortcut?

```
~/projects/document.txt
```

Lastly, the dot (`.`) also has an important function in file paths:

- `.` represents the directory you are currently in, i.e. the working directory.
  - E.g., while inside the `projects` directory, any files inside can be accessed using either `filename` or `./filename`.
- `..` represents the parent directory of the working directory.
  - E.g., from `/home/user1/Desktop`, the relative path to file `document.txt` can be written as `../projects/document.txt`.
  - These expressions can be nested; while inside the `projects` directory, `../../user2` can be used to access the `user2` home directory.

4.2 Moving around the file system

In this section we will introduce a few essential commands that allow you to navigate the file system: `pwd`, `cd` and `ls`.

4.2.1 `pwd`: avoid getting lost

`pwd` stands for *print working directory* and it does exactly that: it allows you to figure out where you are in the file system. For example, in the figure above, `user1` would generally find themselves in their home directory upon login:

```
$ pwd
/home/user1
```

4.2.2 `cd`: on the move

Next, there is the `cd` command. This is used to move between directories (the name derives from *change directory*). Simply follow the command name by a file path to navigate there: `cd <filepath>`. To move from `user1`'s home directory to the `projects` directory:

```
cd projects
```

Note that you can use the special symbols we saw earlier as navigational shortcuts:

Command	Result
`cd ~`	Change to home directory (`/home/username`)
`cd ..`	Change to parent directory (e.g., go up 1 directory)
`cd /`	Change to the root location

4.2.3 `ls`: show me what you got

Finally, we have the `ls` command. Its name stands for *listing* and it will list the names of the files and directories in the current working directory. The basic structure of the command `ls [OPTIONS] <target>`, with `<target>` being an optional path to a directory.

To continue upon our previous example, from inside `/home/user1/projects` we would see:

```
$ ls
DRX333466_1.fastq.gz    DRX333466_2.fastq.gz    document.txt
```

Note that we did not specify a path, in which case `ls` will just list the contents of the current working directory. If we do specify a path, we will of course be shown the contents of that particular location:

```
$ ls /home
user1   user2
```

By default, the files and directories are listed in alphabetically order and depending on your terminal settings, files and directories might even be colour-coded differently.

`ls` also comes with a few handy optional flags to modify its behaviour:

Command	Result
`ls -l`	Show detailed list view
`ls -hl`	Show detailed list view and print file sizes in a human readable format
`ls -a`	List all files and directories, including *hidden* ones
`ls -lha`	Combine all options into one command
`ls --help`	Show more information on the ls command and its options

ls

> **i** What are hidden files?
>
> Earlier, we mentioned that `.` is used to refer to the current working directory, but it actually has a second function as well. Any file or directory name that starts with a dot (like `/home/user1/.ssh`) will be hidden and not displayed by default when using `ls`, hence the need for the `-a` flag. Linux often hides system or configuration files to avoid cluttering up your (home) directory. We will not deal with hidden files directly in this course, but one of the situations where you might encounter them are when modifying your `.bashrc` file (e.g., when creating custom functions, aliases or tweaking your `PATH` **?@sec-unix-path**) or when managing SSH keys for remote server access Section A.5).

The `ls -l` command is particularly useful, because it shows all types of additional information.

What do the different columns in the output of `ls -l` represent?

```
$ ls -l
total 83764
-rw-r--r-- 1 pmoris pmoris 14367565 Dec  7 09:39 3B207-2_S92_L001_R1_001.fastq.gz
-rw-r--r-- 1 pmoris pmoris 16622378 Dec  7 09:39 3B207-2_S92_L001_R2_001.fastq.gz
-rw-r--r-- 1 pmoris pmoris 13592342 Dec  7 09:39 MRA1242_S28_L001_R1_001.fastq.gz
-rw-r--r-- 1 pmoris pmoris 15821981 Dec  7 09:39 MRA1242_S28_L001_R2_001.fastq.gz
-rw-r--r-- 1 pmoris pmoris 12131772 Dec  7 09:39 NK6_S57_L001_R1_001.fastq.gz
-rw-r--r-- 1 pmoris pmoris 13226198 Dec  7 09:39 NK6_S57_L001_R2_001.fastq.gz
```

The first column represents the permissions of the files/folders. In a nutshell, these determine things like who can read or write (= modify, including deletion) particular files. There is a column for the owner, a group of users and everyone else. There is more info in the appendix (Section A.4). The next column showing a `1` for each entry, you can ignore for now (they represent hard links, a concept we will not dive into). The two names in the following columns are the *user* and the *group* owner of the file. Next is the size of the file in bytes. If we had used the `-h` flag, the size would have been shown in KB, MB or GB instead. Next we have the time of the last modification and finally the name of the file/directory.

4.3 Exercises

1. Navigate to your home directory and list all the files and folders there. Try typing the path with and without using the `~`. Rely on tab-completion to assist you and avoid typos (Section A.1).
2. Print the name of the current working directory to your screen.

3. List the contents of the `training/data/fastq/` directory of the course files, without first moving there. Experiment with absolute and relative paths.

4.4 Summary

Overview of concepts and commands

- *Absolute* versus *relative* file paths
- Root (`/`) and home directory (`~`)
- `.` represents the current working directory
- `..` represents the parent directory
- `pwd`: print the path of the current working directory
- `cd <path>`: navigate to the given directory
- `ls <path>`: list files and directories in the given location
- Hidden files contain a `.` at the start of their name and are not visible by default

5 Working with files and directories

> **Prior experience**
>
> You can skip this section and proceed directly to the exercises if you are already familiar with basic commands like `cp`, `mv`, `less` and `nano`.

> **Tip**
>
> Remember that we have provided a list of helpful tips and hints in the appendix: Section A.1.

5.1 Examining files

5.1.1 `cat`: viewing short files

The most basic command for viewing a file is the `cat <file>` command. It simply prints all of the contents of a file to the screen (= *standard output*).

```
$ cd training/unix-demo
$ cat short.txt
On the Origin of Species

BY MEANS OF NATURAL SELECTION,

OR THE PRESERVATION OF FAVOURED RACES IN THE STRUGGLE FOR LIFE.

By Charles Darwin, M.A., F.R.S.,
```

```
Author of "The Descent of Man," etc., etc.

Sixth London Edition, with all Additions and Corrections.
```

> Try using `cat` on the file named `long.txt` and see what happens (Click me to expand!)
>
> The entire file (in this case, the entirety of the Origin of Species by Charles Darwin) is printed to the screen. This works, but is not very easy to navigate. Especially if you consider the fact that this text is still just tiny compared to some of the files that we deal with in bioinformatics; it is only ~0.03% of the size of the (rather short) human Y chromosome (~60 Mbp) that we will look at next.

While `cat` is very useful, it is clearly not suitable for large text files. Since long files are very prevalent - and not only in bioinformatics - we need an alternative. Enter the `less` command.

5.1.2 `less`: viewing large files

This tool is suitable for streaming very large files which would otherwise crash a normal text editor or program like Excel. `less` will open the contents of the file in a dedicated viewer, i.e. your terminal and prompt will be replaced by a unique interface for the `less` tool. You can exit this interface by pressing `q`.

Using `less`, we can have a look at the (truncated) version of the human Y chromosome (in FASTA format):

```
$ less Homo_sapiens.GRCh38.dna.chromosome.Y.truncated.fa
```

> Navigating inside `less`
>
> - Use arrow keys to navigate. `space` and `b` can also be used to go forward and backwards, and `page`

Figure 5.1: Opening a FASTQ file in `less`

`up`/`page down` work as well.

- Press `g` to jump to the start of the file
- Press `G` (`shift + g`) to jump to the end of the file
- Type `/` followed by a string to search forward (`?[string]` for backwards search) and `n`/`N` for the previous/next match
- To exit, press `Q`
- Use the help command for more info: `less --help`

DNA sequence file formats: FASTA

The FASTA file format (usually denoted by a `.fa` or `.fasta` file extension) is very common in bioinformatics. As you can see, the FASTA files contain a long stretch of nucleotides, which in our case represent the sequence of the human Y chromosome (or at least the first ~6,000,000 basepairs). The sequence itself is usually broken up over multiple lines. At the very top of the file there is a header or identifier, which always starts with the `>` symbol, followed by a short description. FASTA files can store one or multiple sequences, each with their own header.
FASTA files are a type of text-based or *plain text files*, meaning that we can simply read them using a tool like `cat` or `less`. This seems obvious, but we will later en-

counter another file type, namely *binary files*, where this is not the case.
We will learn more about these DNA sequence file formats in a later chapter. For the time being, it is enough to know that both of these files are usually very large and that they are commonly used in genomics.

5.1.3 `head` and `tail`: viewing the start or end of files

Sometimes we are not interested in viewing the entire file, but just the first few or last lines. The commands `head` and `tail` were created for exactly this use case. The basic usage is simply `head <filepath>`, but there again are a few optional flags that can alter the default behaviour.

Command	Result
`head file`	Print the first 10 lines of a file
`tail file`	Print the last 10 lines of a file
`head -n # file`	Display the first # lines of a file
`tail -n # file`	Display the final # lines of a file
`tail -n +2 file`	Display all lines except for the first one (i.e., perform `tail`, but start at line 2)

Let us inspect the first lines of one of the (uncompressed) FASTQ files in the `unix-demo` directory:

```
$ head PF0512_S47_L001_R1_001.fastq
@M05795:43:000000000-CFLMP:1:1101:16134:1717 1:N:0:47
TTGGTCAAGATCTTTTACATTCCATGCACACAAAAGAATTCTTCTACTTGTCTGATCCTTTTTCATTATATTTATTATCTTTTTTTATTTTT
+
-8B<CGGFGGGDGGGGGEFGGGAFFEFCFGGDC888C,CF@@FFG,6FC,C,,,,<<,<<CCE,,<C,,,<6B,;B,<,<A@EEC+,:@,,9
@M05795:43:000000000-CFLMP:1:1101:20605:1731 1:N:0:47
GAAAAAGGAAGAGAATTGAACTTTTGGCAGCAAACTCAAACATTATAAGTGAAATTAAGATGCCCAAGTCTGTGCTCAATCTCATTTTTTGT
+
-@CCCGGGGGGGGCGFG@<EFGGGGAFFFEGGC8FEF9,,;,CE,C,,,<,,,,<C,,,,<,,6;,,,;;C,<,6;C,,<,,:,:@,5,8+,
@M05795:43:000000000-CFLMP:1:1101:9135:1768 1:N:0:47
CGTTAAAATCTTGCTCCTCATCACTACTAACCTTTTGTTCATTCTCATCACAAATATTATCCTTATCTTCATTATCTACTTCATCTACATTA
```

DNA sequence file formats: FASTQ

Aside from FASTA files, another typical DNA sequence format is FASTQ (extension `.fastq` or `.fq`), which is used to store the raw output of high-throughput sequencing in the form of short read fragments. Like FASTA, it is text-based format, but instead of just identifiers and sequences, it also contains quality scores associated with each nucleotide. Each read is described by four lines of text. A single read might look like this:

```
@SEQ_ID
ACTACTAGGATTGAGGACGTCCTCCCAACAGGGAGTTGGTTGGGCGCCCGGTGCCGTCATGTCCGATCGCTATCTACGTCTAGTACTAGAGAA
+
"H85<EI4A533D;E1A56C@@GHI=BFGIIH6;F=3::HGF8C;9/>;EI?E4I(F?FID<CBAFFD69E:BB>+#<58H:/<>IE;881&'
```

Line	Description
1	identifier: always starts with '@' and contains information about the read (e.g., instrument, lane, multiplex tag, coordinates, etc.
2	The sequence of nucleotides making up the read
3	Always begins with a '+' and sometimes repeats the identifier
4	Contains a string of ASCII characters that represent the quality score for each base (i.e., it has the exact same length as line 2)

Try inspecting the contents of one of the `.fastq.gz` files in the `training/data/fastq` directory (Click me to expand!)

The `less` command most likely behaves as we expect it to, but if we were to try `cat`, `head` or `tail`, you would see a lot of gibberish being printed to your screen.
The reason is that these FASTQ files are compressed using `gzip` (which is why the file extension ends in `.gz`). Because of this, they are no longer plain text files, but compressed binary versions. We will learn more about

compressed files and how to deal with them in a later section of this course. In a nutshell though, compressed files either need to be unpacked or they require a tool that was designed to handle them (e.g., `zcat`, `zgrep`, `zless`. In most linux distributions, `zless` is called automatically when you try to `less` a file with the `.gz` extension, which is why the text seemed normal.

5.1.4 `wc`: counting lines

Sometimes we're not interested in the specific contents of a file, but only in how long it is in terms of text (not file size). For this we can use the `wc` command: it can count the number of lines, words and characters in a text file. By default, it prints all of this information, but by providing the `-l` flag, you can tell the command to only return the number of lines. Taking the example of our FASTQ file again, we see:

```
# number of lines, words and characters
$ wc PF0512_S47_L001_R1_001.fastq
582940   728675 69598721 PF0512_S47_L001_R1_001.fastq

# number of lines only
$ wc -l PF0512_S47_L001_R1_001.fastq
582940 PF0512_S47_L001_R1_001.fastq
```

How many reads are there in this FASTQ file? (Click me to expand!)

Each read in a FASTQ file consists of four lines (see **?@sec-fastq**). Therefore, we can simply divide the output of `wc -l` by four to figure out the number of reads. In this case:

$$\frac{582,940}{4} = 145,735$$

reads.

5.2 Editing files

You can edit files directly on the command line, i.e. without opening them in a text editor like Notepad(++) or VSCode, by using the `nano` command. This can come in quite handy in a variety of situations, like fixing small errors in your code before running it or to editing configuration files. Similar to `less`, `nano` will open a special editor interface where you can edit text files.

Figure 5.2: The `nano` text editor

> **Navigating inside `nano`**
>
> - Your mouse pointer won't work. Use arrow keys to move instead.
> - To save, press `ctrl+o`, followed by return/enter.
> - To exit, press `ctrl+x`, followed by return/enter.

There exist many other editors, one of the most beloved, yet notorious ones, being `vim`. It is quite a bit more powerful, but also more complex. Even closing vim has become somewhat of a meme because it can be difficult to figure out (it's `<escape>` followed by `:q` and 'enter/return").

5.3 Moving things around

Now that we have spent some time on inspecting files, let us move on to moving them around.

5.3.1 `cp`: copying files and directories

`cp` stands for copy and it does exactly what it says on the tin. It can copy files, as well as directories to a new location. For files, the syntax is as follows:

```
cp path/to/source_file path/to/destination
```

Where `source` is the original file that you want to copy and `destination` is the new path where you want to place the copy. If the destination is a directory, the file will be placed inside of it with the same name as the original file. If the destination does not exist yet, it will be used as the new name for the copy.

When we want to move around directories instead of files, we need to add the `-r` flag (short for `--recursive`).

```
cp -r path/to/source_directory path/to/destination_file
```

You can even copy multiple files at the same time!

```
$ cp file_1 file_2 file_3 /path/to/destination

$ ls /path/to/destination
file_1 file_2 file_3
```

5.3.2 Intermezzo: globbing and wildcards

Now seems like a good time to introduce the concept of the globbing and wildcards. Globbing allows you to perform operations on multiple files. By providing specific patterns, the shell will be able to expand them into a list of matching file

names. The patterns are built using wildcards, one of the most common ones being the asterisk `*`.

How does this work? Well, `*` can represent any number of other characters. For example, the string `*.txt` can match all file names ending with `.txt` in your directory. Let's look at a concrete example, using the `ls` command we saw earlier:

```
$ ls
Homo_sapiens.GRCh38.dna.chromosome.Y.truncated.fa  PF0512_S47_L001_R1_001.fastq  files_to_co

$ ls *.txt
long.txt  short.txt
```

As you can see, we can make `ls` list only those files that match a particular pattern, instead of showing all the files in the directory. What happens behind the scenes is that `*.txt` is expanded to `long.txt short.txt`. This means that the command that the shell eventually sees is actually `ls long.txt short.txt`.

Similarly, we can combine wildcards with the new `cp` command.

```
$ cp *.txt ..

$ ls ..
```

What do you think this previous command will do? (Click me to expand!)

`*.txt` will be expanded to a list of all `.txt` files in the current working directory. The `cp` command will then try to copy each of those files to the destination, which is `..` in this case. As we saw before, `..` represents the parent directory of the current directory (see Section A.2).
This means that the command is equivalent to `cp long.txt short.txt /absolute/path/to/parent_directory` and will move all the `.txt` files in the current directory to its parent directory.

Another type of wildcard is `[...]`. This is used to supply a list of possible character matches. For example, the glob pattern `[bcr]at` would match `bat`, `cat` and `rat`.

There are a number of other wildcards, but even `*` alone will prove to be very useful. If you'd like to find out more, have a look at this resource. Also note that globbing looks similar to *regular expressions*, but while related, these two concepts behave slightly differently. We will not dive into regular expressions here though, but we will mention them again when we talk about the search tool `grep`.

To summarise, globbing is an extremely powerful tool that will allow you to more easily target multiple files. We will rely on the power of globbing a lot going forward.

5.3.3 `mv` Moving or renaming files and directories

The `mv` (move) command behaves very similar to the `cp` command, the main difference being that the former allows you to move rather than copy files and directories. Also note that `mv` is used to *rename* files as well.

```
# move around/rename a particular file
mv <source_file> <destination_file>

# move a directory
mv <path/to/source_directory> <path/to/destination_directory>
```

5.4 Creation and destruction

We will end this section by teaching you how to create and delete files or directories.

5.4.1 Creating files

There are several ways of creating new files in Unix, but one of them is the `nano` command that we already introduced earlier.

If you provide a file name that does not yet exist, `nano` will create the file for you.

```
$ ls

$ nano new_file.txt
# inside nano, use ctrl+x to save the file and then close the editor via ctrl+x

$ ls
new_file.txt
```

Another option is to use the `touch /path/to/file` command. This will just create a new empty file at the specified location.

5.4.2 `mkdir`: creating directories

`mkdir` stands for *make directory* and it does just that:

```
$ mkdir new_dir

$ ls
new_dir
```

One useful optional flag is `-p/--parents`: this allows you to create multiple nested (parent) directories in one go. For example, if we're inside an empty directory, we could call:

```
mkdir -p my/new/multi/level/directory
```

And all the intermediate directories would be automatically created.

5.4.3 `rm`: removing things

> ⚠ Watch out…
>
> Be careful while learning your way around the command-line. The Unix shell will do *exactly* what you tell it to, often without hesitation or asking for confirmation. This means that you might accidentally move, overwrite or delete files without intending to do so. For example, when creating, copying or moving files, they can overwrite existing ones if you give them the same name. Similarly, when a file is deleted, it will be removed completely, without first passing by a recycle bin.
> **No matter how much experience you have, it is a good idea to remain cautious when performing these types of operations.**
> For the purposes of learning, if you are using your own device instead of a cloud environment, we recommend that you work in a dedicated playground directory or even create a new user profile to be extra safe. And like always, backups of your important files are invaluable regardless of what you are doing.

The `rm` command (*remove*) is used to delete files and directories. Be warned though, once deleted, things are really gone. There is no recycle bin or trash folder where you can restore deleted items!

```
# for files:
rm <file path>

# for directories
rm -r <directory path>
```

For files, this works as expected, but for directories you need to provide the `-r` flag (or `--recursive`). This tells unix to remove the directory recursively, i.e. all of its contents need to be removed as well. If you don't use this option, you will see the following warning:

```
rm directory
rm: cannot remove 'test/': Is a directory
```

Protected files

Sometimes, files will be protected and you will get another warning message when you try to remove them. If you are really sure that you want to delete them, you can type `y` and press enter. Alternatively, you can cancel the operation (by entering `n` or by pressing `ctrl+c`) and try again, but this time providing the `-f/--force` option.

```
# create a new empty file
$ touch protected-file
# change its permissions so that it is protected against writing and deleting (see appendi
$ chmod a-w protected-file
# try to remove it
$ rm protected-file
rm: remove write-protected regular empty file 'protected-file'? n
# use the --force flag
$ rm -f protected-file
```

5.5 Exercises

1. Create a new directory named "my_dir" inside the `./training/unix-demo` directory. Next, without using `cd` first, create another directory named `my_sub_dir` inside of it. Finally, again without using `cd`, create a final directory named `my_sub_sub_dir` inside of that one.
2. Read the last 20 lines of the FASTA file in the `./training/unix-demo` directory.
3. Create a new text file named `lines` inside `my_subdir` using `nano`. Store the number of lines of the file `long.txt` inside. Then read it using `cat` and `less`.

4. Navigate to the `files_to_copy` directory and copy its contents to the `my_sub_dir` directory. What is the relative path of the destination to use?
5. Move the file under `files_to_move` to its parent directory.
6. Remove all the files under `files_to_delete` using a glob pattern.
7. Rename the directory `files_to_delete` to `empty_dir`.
8. List the contents of the `./training/unix-demo` directory.

5.6 Summary

Overview of concepts and commands

- FASTA file format is used to store DNA sequences
- FASTQ file format is used to store sequence reads and their quality scores
- Compressed files (e.g., `.gz`) are smaller in file size, but are no longer plain text files and require special tools.
- The permission of files can be set to prevent users from reading, writing or (re)moving them.

Command	Result
`cat <path/to/file>`	print the content of files
`less <path/to/file>`	read the contents of (large) files in a special viewer
`head/tail <path/to/file>`	view the first or last lines of a file
`wc <path/to/file`	display the line/word/character count of a file
`nano <path/to/file>`	open a file (or create a new file) in the `nano` text editor

`cp [-r] <source> <destination>`	copy a file/directory to a new location
`mv [-r] <source> <destination>`	move a file/directory to a new location (or rename it)
`rm [-r] <path/to/file_or_directory>`	permanently remove a file/directory
`mkdir <path/to/directory>`	create a new directory

6 More advanced commands

> **Prior experience**
>
> You can skip this section and proceed directly to the exercises if you are already familiar with commands like `grep`, `sort` and `cut`.

> **Tip**
>
> Remember that we have provided a list of helpful tips and hints in the appendix: Section A.1.

6.1 Searching in files: `grep`

Being able to search through (long) text files is incredibly useful in a wide range of scenarios. For this, we make use of the `grep` command. It is a very powerful and complex command, with many different options to tweak its behaviour, but even just the basic version can already be a lifesaver. The basic syntax is

```
grep "PATTERN" <path/to/target_file>
```

For example, we can look for the word *"evolution"* in the Origin of Species:

```
$ grep "evolution" long.txt
evolutionists that mammals are descended from a marsupial form; and if
At the present day almost all naturalists admit evolution under some
evolutionists; but there is no need, as it seems to me, to invoke any
Everyone who believes in slow and gradual evolution, will of course
```

```
of gradual evolution, through the preservation of a large number of
a strong disbeliever in evolution, but he appears to have been so much
historian will recognise as having produced a revolution in natural
the fact would be fatal to the theory of evolution through natural
the revolution in our palæontological knowledge effected by the
opposed to the admission of such prodigious geographical revolutions
has thus been arrived at; and the belief in the revolution of the earth
subject of evolution, and never once met with any sympathetic
agreement. It is probable that some did then believe in evolution, but
evolution. There are, however, some who still think that species have
we can dimly foresee that there will be a considerable revolution in
```

It might not be obvious from the above snippet, but inside your own terminal, the matching words in the search results will be highlighted. As you can see, `grep` will return each line that contains a match. Also note how partial matches like ***revolution*** are returned as well.

> Try searching for the string "species" instead. Do you think these are all the hits? (Click me to expand!)
>
> When you run `grep "species" long.txt`, you will indeed find many occurrences of this word. However, we are missing all the occurrences of "Species". Try running `grep "Species" long.txt` to compare the results. Lastly, try running the command `grep -i "species" long.txt`. This option will
> Remember, capitalization matters to `grep` (and to unix as a whole)!

Aside from the `-i` option explained in the box above, there exist several other flags to improve your search results.

Option	Effect
`-i`	Case insensitive (i.e., `ACTG` = `actg`)
`-n`	Also print the line number of the result
`-c`	Count the number of matches
`-5`	Print 5 lines surrounding each match

Option	Effect
`-e/-E/-P`	Use regular or extended regular expressions
`-r`	Recursive search through all files in a folder

The `-#` option is particularly useful to learn more about the context around you search result. You can supply any number of lines here, which will get printed both before and after each match. The example below, you can see how it helps us find the identifier of a DNA read that contains a particular sequence (`AACCGGGGT`):

```
$ grep -3 AACCGGGGT PF0512_S47_L001_R1_001.fastq
+
CCCCCGGCFGGGGFGGGGDGCEFGGGGGGGGGGGGGGGGGGGGGFGGGGGGGGFFGGGFGGGGGGGGGGGGGGFGGGGGGGGGGGGGGGG
@M05795:43:000000000-CFLMP:1:2106:16840:21815 1:N:0:47
AGCCATACCAAGACCACAATTCTGAAGAGGAAACAAAACAAAAAAAAAAAAAATAATTAAAAAAAAAAAAAATTTAAAATTAAAAAAAAAATTT
+
CCCCCGGFGGGFGGGGCFG9@@C<FGDG8EGGGFGGGGGGGGGGGGGGGGD,:C,96CFD,8>FG7FGG,,,,B<B9,9CFFGGCG+3,,8
@M05795:43:000000000-CFLMP:1:2106:14940:21820 1:N:0:47
```

The `-c` options makes `grep` return the total number of matches it found. This method of counting is useful to complement the `wc` command, in case you are presented with a file that does not have such an orderly structure as the FASTQ format we saw earlier. To demonstrate, consider the `penguins.csv` file, which contains morphological data on three different penguin species.[2] We can count the number of Adelie penguin records via:

```
grep -c "Adelie" penguins.csv
152
```

In some situations we want to search through multiple files simultaneously. This is where the `-r/--recursive` flag comes in. It allows us to target a directory and search through all of its contents (including subdirectories). Let us try searching for the same DNA sequence as before, but this time targeting all the files in the `unix-demo` directory:

[2] Gorman KB, Williams TD, Fraser WR (2014) Ecological Sexual Dimorphism and Environmental Variability within a Community of Antarctic Penguins (Genus Pygoscelis). PLoS ONE 9(3): e90081. doi:10.1371/journal.pone.0090081

```
$ grep -r "AACCGGGGT" .
./PF0512_S47_L001_R1_001.fastq:AGCCATACCAAGACCACAATTCTGAAGAGGAAACAAAACAAAAAAAAAAAAAATAATTAAAA
./Homo_sapiens.GRCh38.dna.chromosome.Y.truncated.fa:AATAAAACCGGGGTTGATACCACCACTTCCAGGTTCCCAC
./Homo_sapiens.GRCh38.dna.chromosome.Y.truncated.fa:CTGGAGTCAGGACGTGAGCCGACTTGCTTAAAAATAAATC
./Homo_sapiens.GRCh38.dna.chromosome.Y.truncated.fa:ACAGCTAACCGGGGTTTTAGTATATGTGCCACATCTCTGT
./Homo_sapiens.GRCh38.dna.chromosome.Y.truncated.fa:TATGATCGTGCCACTGCACTTCAACCGGGGTGACAAAGCG
```

How would you search through `.txt` files only? (Click me to expand!)

Instead of using the `-r` flag, we can also rely on globbing again (see Section 5.3.2). To search for the string "needle" in all `.txt` files in a particular folder, we can do the following:

```
grep "needle" path/to/directory_with_txt_files/*.txt
```

We already mentioned *regular expressions* in the previous section: they allow you to search for particular patterns that can match more than one exact string of text. This is tremendously useful, but we will not dive into how they work during this course. If you are interested, you can check out an excellent tutorial here.

We will see more elaborate use cases for `grep` when we introduce the unix concepts of piping and redirection.

6.2 Tabular data: `cut`

We already encountered tabular data (the penguin dataset in `.csv` format) when talking about `grep`. Tabular data files like `.csv` are a very common format, and not just in bioinformatics.

Tabular data and `.csv` files

Tabular data files are usually plain text files, where each row corresponds to a record (e.g., an individual penguin),

and each column represents a particular field (e.g., species, flipper length, body mass, etc.). The columns can be separated by different *field delimiters* or *separators*. In `.csv` files, these are usually commas (*comma separated values*), but they can also be TABS (`.tsv`) or semicolons (`;`).

A particularly useful unix tool for manipulating tabular data files, is `cut`. It allows us to extract particular columns from these files. The syntax is as follows:

```
cut [OPTIONS] target_file
```

Option	Effect
`-d ","/--delimiter ";"`	Change the default delimiter (TAB) to another character like `,`
`-f 1`	Select the first column
`-f 2,3`	Select the second and third column
`-f 1-3,6`	Select columns one through three and columns six
–complement -f 1	Select all columns *except* for the first one
-r	Recursive search through all files in a folder

6.3 File sizes: `du`

We already saw that the `ls -lh` can be used to figure out the file size of files in a particular directory. `du` is another tool to do this, but it operates on individual files or directories directly. Like `ls`, it also provides the `-h/--human-readable` option to return file sizes in KB/MB/GB, so it is generally recommended to always use this option. When used on a file, it will simply return its size, but when used on a directory, it will output information for all files, as well as the total file size of the entire directory (the final line of the output).

```
# targetting an individual file
$ du -h Homo_sapiens.GRCh38.dna.chromosome.Y.truncated.fa
5.9M    Homo_sapiens.GRCh38.dna.chromosome.Y.truncated.fa

# targetting a directory
$ du -h training/data/
284M    training/data/fastq
62M     training/data/reference
346M    training/data/
```

6.4 Compressed files: `gzip`

We already introduced the concept of file compression when talking about the FASTQ files in the `training/data/fastq` directory. As a reminder, compressed files are binary files (as opposed to human-readable plain text files) that are used to reduce the file size for more efficient storage. Many of the files that we use in bioinformatics tend to be compressed. Some of the tools we use, will not work on compressed files (e.g., try to `cat` a compressed file and see what happens), so we either need to 1) use specialized tools that expect compressed files as their input, or 2) decompress or extract the files first.

For gzipped (`.gz`) files specifically, we can do this via the `gzip` and `gunzip` commands. The former allows us to create a gzip-compressed version of the file, whereas the latter will extract one back to a plain text file. The basic syntax is `gzip/gunzip <path/to/file>`, but a very useful option is the `-k`/`--keep` flag. Without it, compressing a file would replace the uncompressed file with the new compressed one (vice versa: extracting would replace the compressed version with the extracted one), but when using the flag both files will be retained.

Try compressing the FASTQ file in the `unix-demo` directory. By how much does its file size change? (Click me to expand!)

```
$ du -h PF0512_S47_L001_R1_001.fastq
67M     PF0512_S47_L001_R1_001.fastq
$ gzip --keep PF0512_S47_L001_R1_001.fastq
$ du -h PF0512_S47_L001_R1_001.fastq.gz
17M     PF0512_S47_L001_R1_001.fastq.gz
```

After compressing the FASTQ file with `gzip`, it shrunk to less than a third of its original size.

Do note that there exist other types of file compression besides gzip, like `.zip`/`.7zip`. In unix we also often make use of `tar` (which technically is not a compression tool, but a file archiver). File compression and tar can even be combined, leading to files with suffixes like `.tar.gz`. This allows us to compress entire directories, instead of only individual files.

To extract these so called *tarballs*, we need to use the `tar` command:

```
# extract .tar.gz archive
$ tar -xzvf tar_archive.tar.gz
tar_archive/
tar_archive/3
tar_archive/2
tar_archive/1

$ ls tar_archive
1  2  3

# create .tar.gz archive
$ tar -czvf new_archive.tar.gz <path/to/target_directory>
```

This command is notorious for how arcane its option flags are, but you can either try to remember it using a mnemonic (*"e**X**tract/**C**ompress **Z**e **V**ucking **F**iles"*, pronounced like a B-movie vampire) or the meaning of the individual flags (`z` tells tar

that we are using gzip compression, `-v` stands for `--verbose` to make the command show more information and output, `-c/x` switches between compression and extraction mode, `-f` is the last option and points to the tar file) . And of course, the correct syntax is only a google/tldr search or `tar --help` call away.

6.5 Downloading files: `wget`

`wget` is a command that allows you to download files from a particular web address or URL and place them in your working directory. While there are several optional flags, in its most basic form the syntax is simply: `wget URL`. This command is not only useful when automating certain tasks, but also crucial if you ever find yourself in a unix environment that does not have a GUI at all (e.g., compute clusters or cloud servers).

Try downloading the *Plasmodium falciparum* 3D7 reference genome in FASTA format from PlasmoDB and store it in `./training/data/results`. (Click me to expand!)

- At the top of the page, click on *Data -> Download data files*.
- Search for *falciparum 3D7* and then narrow down your search by selecting the most recent release and the FASTA file format. Alternatively, you can click on the Download Archive link in the top and navigate the file directory to the current release.
- The file name of the 3D7 reference genome in FASTA format is `PlasmoDB-66_Pfalciparum3D7_Genome.fasta`.
- Right click the file and copy its URL to your clipboard: https://plasmodb.org/common/downloads/release-66/Pfalciparum3D7/fasta/data/PlasmoDB-66_Pfalciparum3D7_Genome.fasta.
- Create and navigate to the output directory (`mkdir -p ./training/data/results` and `cd ./training/data/results`)
- Download the file here using the command: `wget`

```
https://plasmodb.org/common/downloads/release-66/Pfalciparum3D7/fasta/data/PlasmoDB-66_P
```

Two optional flags that you might find useful are: 1) `-o` allows you to rename the download file, and 2) `-P <path/to/directory` saves the file in a directory of your choice instead of the current working directory. Of course, these are just small convenient timesavers, since you can always `cd` to a particular location and use `mv` to rename the file afterwards.

Lastly, an alternative to `wget` that you might encounter at some point is `curl`. On the whole, it acts quite similar to `wget` for the most part.

6.6 Retrieving file names: `basename`

`basename` is a rather simple command: if you give it a long file path, it will return the final section (i.e., the file name).

```
# starting in the `training` directory
$ pwd
/home/pmoris/itg/FiMAB-bioinformatics/training

# get the file name for the reference genome we just downloaded
$ basename data/reference/PlasmoDB-65_Pfalciparum3D7_Genome.fasta
PlasmoDB-65_Pfalciparum3D7_Genome.fasta
```

At this point in time, it might not seem particularly useful to be able to extract the file name of a file, but when we introduce the concept of for loops and bash scripting, it will become more clear why this can be so useful.

6.7 Sorting and removing duplicates: `sort` and `uniq`

The final two commands that we will introduce are yet again tools to manipulate plain text files. The first is `sort`, which

does exactly that you expect it to. It can sort all the rows in a text file. Its syntax is:

```
sort [OPTIONS] <./path/to/file>`
```

There are optional flags that allow you to choose the type of order to use (e.g., numerical `-n`/`--numeric-sort` instead of alphabetical), reverse the order of the sort (`-r`/`--reverse`) or ignore capitals (`-f`/`--ignore-case`).

The second command, `uniq`, is used to remove duplicate lines in a file.

These two commands are often used in conjunction, because `uniq` on its own is not capable of filtering out identical lines that are not adjacent. So to truly remove all duplicate lines in a file, we would first need to sort it. In the next section, we will introduce a method of combining commands in a more convenient way than running them one by one and without needing to create any intermediary files.

6.8 Other commands

Of course, there exist many more unix commands than the ones we introduced here. We will end this section by briefly mentioning two that you might run into at some point `awk` and `sed`. Both of them allow you to search and replace patterns in text files, and with `awk` you can even perform more complex operations including calculations. We will not dive into them here, but keep them in the back of your mind for the future.

6.9 Exercises

1. Create a new directory named "my_dir" inside the `./training/unix-demo` directory. Next, without using `cd` first, create another directory named `my_sub_dir`

inside of it. Finally, again without using `cd`, create a final directory named `my_sub_sub_dir` inside of that one.

2. Read the last 20 lines of the FASTA file in the `./training/unix-demo` directory.
3. Create a new text file named `lines` inside `my_subdir` using `nano`. Store the number of lines of the file `long.txt` inside. Then read it using `cat` and `less`.
4. Navigate to the `files_to_copy` directory and copy its contents to the `my_sub_dir` directory. What is the relative path of the destination to use?
5. Move the file under `files_to_move` to its parent directory.
6. Remove all the files under `files_to_delete` using a glob pattern.
7. Rename the directory `files_to_delete` to `empty_dir`.
8. List the contents of the `./training/unix-demo` directory.
9. Visit PlasmoDB again and download the *Plasmodium vivax* P01 reference genome sequence in FASTA format and store it in `./training/data/reference`.
10. Report the file size of this reference genome in MBs.
11. Find out how many lines of text the file contains.
12. Search through the file for the `>` character, which is used to denote every chromosome/contig.
13. Use a single command to count the number of chromosomes.
14. Compress the FASTQ file `PF0512_S47_L001_R1_001.fastq` in the `unix-demo` directory using `gzip`.
15. Navigate to the directory `./training/data/fastq` and in a single command, extract the forward (`PF0097_S43_L001_R1_001.fastq.gz`) and reverse (`PF0097_S43_L001_R2_001.fastq.gz`) of the `PF0097_S43`

sample, without removing the compressed files. Hint: use globbing!

16. Search both FASTQ files for the read fragment with identifier `@M05795:43:000000000-CFLMP:1:1101:21518:5740 2:N:0:43` using a single command.
17. Compare the file sizes of the two compressed and uncompressed FASTQ files.
18. Extract the columns containing the *island* and *flipper length* of each penguin from the `./training/unix-demo/penguins.csv` file.
19. Count how many times the sequence `CATCATCATCATCAT` occurs in the FASTA file `./training/unix-demo/Homo_sapiens.GRCh38.dna.chromosome.Y.truncated.fa`

6.10 Summary

Overview of concepts and commands

- Tabular data: `.csv`, `.tsv`

Command	Result
`grep <pattern> <path/to/file>`	Search through a (very large) file for the supplied pattern
`du <-h> <path/to/file_or_directory>`	Check how much space a file or directory occupies
`cut -f [--delimiter ","] <path/to/file>`	Extract columns from tabular data using the specified delimiter
`gzip / gunzip (-keep) <path/to/file>`	Compress or extract a gzip compressed file (`.gz`)
`wget <url>`	Downloads a file from the URL to the current directory

`basename <path/to/file>`	Returns the name of the file without the path prefix

Part II

R basics

7 R introduction

8 R chapter 1...

Part III

Bioinformatics file formats and tools

A Various Unix topics

A.1 Tips and hints

Naming conventions and cases

Never (never!) use spaces in your file or directory names. This will only lead to pain... Instead, use hyphens (`-`) or underscores (`_`) to separate words. E.g., `my_first_script` and `3B207-2_S92_L001_R1_001.fastq.gz`.
Additionally, unlike in Windows, in Unix everything is case-sensitive. Thus, `/home/documents` != `/home/Documents`. Be mindful of this when naming or pointing to files/directories.

Autocompletion and command history

Avoid unnecessary typing and just make things easy for yourself!
While typing commands in the shell, you can almost always use the `tab` key for **auto-completion**. This will automatically type out paths, file names or known commands. If there exist multiple matches, a single press of `tab` will not appear to do anything, but if you press the button twice, a list of possible options will appear on your screen. This is incredibly useful, not only for speeding things up, but also for avoiding typos when dealing with long or complex file names.
An equally useful tool is your **command history**. While on the shell prompt, pressing the up arrow (`↑`) will bring up your most recent previous command. Pressing it again will cycle through the entire history, in reverse order. You can also search through your history by pressing `ctrl+r`

allows you to search through your command history. Just start typing and you will see the search try to narrow down on the command that you are looking for. Once you find it, just press `enter` to run it directly or `tab` to copy it to your prompt (in case you still want to change it). The search form will look like this: `(reverse-i-search)`world': echo "Hello world!"`

Don't panic when you lose control of your shell!

If a command seems to hang or get stuck, your terminal becomes unresponsive, or if you tried to print a very large text file to your screen, you can use `CTRL+C` to interrupt almost any operation and regain control.
Similarly, CTRL+D is an often used shortcut for exiting/logging out (e.g., when dealing with remote servers of nested shells).
In some cases, like when using an interactive terminal program such as the text editors `nano` and `vim` or a text viewer like `less`, you will only be able to exit them using that particular program's shortcut keys (`CTRL+X`, `:` followed by `q` and `enter`, and `Q`, for these applications respectively).

Watch out...

Be careful while learning your way around the command-line. The Unix shell will do *exactly* what you tell it to, often without hesitation or asking for confirmation. This means that you might accidentally move, overwrite or delete files without intending to do so. For example, when creating, copying or moving files, they can overwrite existing ones if you give them the same name. Similarly, when a file is deleted, it will be removed completely, without first passing by a recycle bin.
No matter how much experience you have, it is a good idea to remain cautious when performing these types of operations.
For the purposes of learning, if you are using your own de-

vice instead of a cloud environment, we recommend that you work in a dedicated playground directory or even create a new user profile to be extra safe. And like always, backups of your important files are invaluable regardless of what you are doing.

Copying and pasting

Copying and pasting might work slightly different to what you are used to, depending on the terminal application that you are using. If `ctrl+c` and `ctrl+v` do not appear to work, you can try`ctrl+shift+c` and `ctrl+shift+v` instead. Often times, the mouse middle or right click can also be used for pasting.
For the native WSL terminal specifically, you can refer to this site for more info: https://devblogs.microsoft.com/commandline/copy-and-paste-arrives-for-linuxwsl-consoles/

Google and `--help` are your friends.

At the beginning things will be awkward, so don't worry about having to search for the same information multiple times. That's part of the learning process.
It can be a good idea though to keep a list of commands that you often use, but have a difficult time committing to memory. Moreover, for more complex scripts, it is a good idea to add some comments on how they work, because code that seems clear while you are writing it, has the unfortunate tendency of becoming confusing when you refer back to it at a later time.

Google and `--help` are your friends.

At the beginning things will be awkward, so don't worry about having to search for the same information multiple times. That's part of the learning process.
It can be a good idea though to keep a list of commands that you often use, but have a difficult time committing to

memory. Moreover, for more complex scripts, it is a good idea to add some comments on how they work, because code that seems clear while you are writing it, has the unfortunate tendency of becoming confusing when you refer back to it at a later time.

i Make your scripts easier to read by using comments and breaking up long lines

Remember that you can always write comments inside of your scripts by starting a line with `#`. That way you can add a short explainer or extra info to the different sections of a script.

```
#!/usr/bin/env bash

############################################################
# Script to map fastq files to reference genome with bwa #
############################################################

# make sure to run this script from within the directory where it is stored!

# move to the directory containing the fastq files
cd ../data/fastq

# create a directory to store the results and store the path as a variable
output_dir="../../results/bwa"
mkdir -p ${output_dir}

...
```

Additionally, you can break up long commands using a `\` to make them easier to read. You can do this both in scripts or on the command line. E.g.,

```
bwa mem \
  ../reference/PlasmoDB-65_Pfalciparum3D7_Genome.fasta \
  ${read_1} \
  ${sample_name}_R2_001.fastq.gz
```

https://explainshell.com/ & https://tldr.inbrowser.app

The first website is tremendously useful for figuring out what a command and all of its options mean. Whereas the second shows you a quick summary of the most command usages of a particular command.
Use both of these to your advantage! But do not forget that most commands also have a built-in help page that can be accessed using the `--help` flag (in some cases just typing the command without any arguments also shows some help information).

A.2 Overview of special syntax

The table below gives you an overview of some of the special characters that we will encounter. You do not need to memorize them, but you can always refer back to this section if you see a symbol later on and are not quite sure what its purpose is.

Symbol	Name	Uses
`/`	Forward slash	File path separator or root location/file path
`\`	Back slash	Split long command to a new line and escape special characters (+ file path separator in Windows)
`~`	Tilde	Shortcut for home directory in file paths
`|`	Pipe or vertical bar	Chains the output of one command to the input of another one (piping)
`#`	Hash	Part of the shebang at the top of scripts `#!` and used for comments in shell scripts
`$`	Dollar sign	Used to access variables in bash
`*`	Asterisk or wildcard	Globbing operator

Symbol	Name	Uses
`>`	Greater than symbol	Redirect output of a command (`>>` redirect and append instead of overwriting)
`<`	Less than symbol	Redirect input to a command
`.`	Dot	In the context of a path, it represents the current working directory
`..`	Double dot	In the context of a path, it represents the parent directory of the working directory

A.3 What is `$PATH`?

The `$PATH` is a way of letting your computer know where specific tools or other special locations are stored on your file system. Unless you tell it explicitly, it won't know where to find any new software you install. Fortunately, most methods of installing software automatically take care of this for you, but every now and then you will need to manually add things to your `$PATH`. If you don't, you will be greeted by messages like `Command 'python' not found, did you mean:`.

The `$PATH` is nothing more than a list of locations on your computer. Everything that is found in those locations, will become available to use directly on the CLI without having to type out its full location. Even the basic Unix commands, like `ls` and `cd` are only known to your shell because they are in a location that is indexed by your path.

In the following example, we will demonstrate how you can add a custom directory with scripts to your `$PATH`, making them callable from anywhere.

```
# show the contents of PATH
echo $PATH
/home/pmoris/miniforge3/bin:/home/pmoris/miniforge3/condabin:/usr/local/sbin:/usr/local/bin:
```

```
# temporarily add directory of scripts to PATH
$ ls ~/itg/FiMAB-bioinformatics/training/scripts
call_variants.sh  download_reference.sh  map.sh  remove_dups.sh  trim.sh
$ export PATH="$PATH:~/itg/FiMAB-bioinformatics/training/scripts"

# now these scripts can be invoked directly without having to type out their full location
# e.g., map.sh works just as well as ~/itg/FiMAB-bioinformatics/training/scripts/map.sh
```

To make these changes permanent, you'd have to add that `export` statement to your `.bashrc` file (stored in your home directory). This file is run every time you launch a new shell, so that will allow the `$PATH` to be modified every time during startup.

You can find more information on modifying the PATH here.

Lastly, be careful when modifying your PATH. If you mess it up, it can cause all kinds of havoc.

A.4 Dealing with file permissions

You can find an excellent explanation on file permissions here.

A.5 Working with remote machines via SSH

In some cases, you will need to work on a Linux machine that is physically located somewhere else, i.e. a remote server. Access to these is usually managed via a command-line tool called SSH (or a stand-alone GUI tool like Putty in Windows). The syntax of the `ssh` command is as follows:

```
ssh username@domain
```

Where username is a name given to you by the admin of the system and domain is the address of the server (can be a URL or an IP address).

The connection is secured via *SSH keys*: a pair of files used for authentication stored in `~/.ssh`.

1. Public key: e.g., `id_rsa.pub` or `id_ed25519.pub`, located on the remote server.
2. Private file: e.g., `id_rsa` or `id_ed25519.pub`, located on your own machine. *Never share this file with anyone else!**

Instructions to generate new SSH keys can be found https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent.

A common problem when connecting is that the file permissions of your keys or credential files are messed up. This can happen if you generate them in Windows and later move them to a Linux file system. To fix this, check https://superuser.com/a/215506.