Introduction to Unix/Linux

Víctor Sojo (vsojo@amnh.org)

Contents

I. A few words on formatting before we start

1. Follow the instructions in these blocks closely.

You can ignore most black & white text, but **do not** ignore these red blocks or the grey ones!

And with that, let's get started.

II. What is Unix, what is Linux, and why do geeks think it's so cool? (*optional*)

The following sections give a bit of historical background, but you can just skip to section III if you're on Windows, or section IV if you're on a Mac, on Linux, or on Windows with access to Linux (see section III).

II.1. What does GNU stand for? (*optional*)

GNU stands for **G**NU is **N**ot **U**nix… yes, the 'G' in 'GNU' stands for 'GNU'. You'll get the joke if you're a computer geek. If you don't get it, don't worry about it, the important thing here is that Linux is pretty much the same thing as Unix, but generally built with open code, so that anyone can inspect it, use it, hack it, change it, and so on. You've probably heard the term "free software". The emphasis here is in the *freedom* part, I.e. the ability to do whatever you want with the source code (as opposed to Windows or macOS where you just have to use what you get from Microsoft or Apple). The other meaning of free (not having to pay for it) is also typically included, but the emphasis is on the accessibility of the code. Some *flavours* of Unix-like systems are free and open (e.g. Linux), while some are neither (e.g. macOS).

II.2. What's with this "*flavour*" and "*distro*" thing? (*optional*)

Flavour is a loosely used term that may mean different things to different people. For simplicity, you can consider it an implementation of the Unix paradigm into a full-fledged operating system. For example, you could say Solaris, Linux and macOS are all "flavours" of Unix.

Distro is short for "distribution", and it is used specifically within the Linux community. Because Linux is GNU (it's actually called GNU/Linux by some people)—i.e. its code is open to all, many people create new versions to suit their needs. Because Apple is only one company with only one board of directors and one CEO, they can decide that their one and only current operating system will be macOS X, and nothing else; they may have a lot more options developed, but they publish only one. Similarly, Microsoft currently has two—Windows 10 Home and Windows 10 Pro—as their latest operating systems. Now, Linux is not a company, it is an open community with open code that is open to modification without requiring anybody's approval or permission. You can just go and download the source code right now and change it in whichever way you want. Understandably, there are hundreds if not thousands of different Linux options, or *distros* (from *distribution*).

The most popular Linux distro for regular end users is probably Ubuntu, which is built to be visually appealing and user friendly, like macOS and Windows, and it looks pretty much like a mishmash of those two. Calculation clusters tend to run very different distros meant to handle the load efficiently, not look nice. Internet servers may have yet another distro that prioritises security and defence against constant attacks. The one thing they all have in common is the core of Linux, the *kernel*, which behaves very much like the Unix one—because it was built to do so. If you don't have any special plans or reasons to choose differently, and you don't have much experience with Linux, then you'll probably be fine with Ubuntu.

II.3. Hang on, so if I have a Mac then I already have Linux? (*optional*)

Yes, kind of. For most intents and purposes, you do, or more precisely, you have a Unix-like system. macOS is a (somewhat loose) implementation of the Unix paradigm, and it works pretty much the same for many if not most purposes (unfortunately not all… but then again no two Unix "flavours" behave the same, not even two Linux "distros").

A major source of frustration for Mac users is that some commands don't translate directly. A famous one is `wget`, which is used very frequently to download stuff from the web in Unix/Linux, such as in `wget https://url.of/the.file`, but does not exist by default on Mac. If you want to download a file on a Mac, use `curl -O https://url.of/the.file` instead (that's a big letter 'o', *not* a zero `0`).

II.4. I'm on Windows, actually, what do I do?

Selber schuld. Read on.

III. Installing Linux (Windows users)

Mac/Linux users can safely skip ahead to section IV.

There are several ways to get a Linux installation running in your Windows PC:

1. **Putty + WinSCP:** This gives you a link to a remote Linux server, e.g. a calculation cluster at your institution. It is a very minimalistic solution, but it works well **as long as you have internet access to the remote Unix server**. You may also need a running VPN to access your institution's network (that's the case at the AMNH, whose servers cannot be accessed without VPN validation).
   Putty is used to access the remote server from Windows, whereas **WinSCP** lets you share files between your local Windows machine and the remote Linux server (you can just drag and drop files between the two, create folders, and so on, very nice). In combination with Putty, this works very well, but it is no replacement for a proper Unix installation.

   This only works if you have internet access to a remote Unix server! Otherwise read on.
2. **Ubuntu shell:** Microsoft seem to have finally heard the wind of change and allowed Canonical, the people behind Ubuntu (the most popular desktop Linux distro), to develop a Linux shell for the Windows 10 operating system. The installation takes a few steps, but ultimately you end up with an Ubuntu terminal app that should run just like any other app (i.e. like firing up Photoshop or Excel). **Note:** you must be on Windows 10 for this to work. Go to https://ubuntu.com/tutorials/ubuntu-on-windows for instructions.
3. **Dual boot:** You could install a Linux distro such as Ubuntu in parallel to Windows, so that at the beginning when you power up your computer you're given the option to choose which of the two operating systems you want to load. This is a complex task that should not be undertaken lightly, but it can be a good option if you truly want a local Linux version (many Windows-based geeks don't feel they need one).
4. **VirtualBox:** You could install VirtualBox on your Windows machine and put a virtual installation of Linux inside it. VirtualBox is just a program that runs inside Windows, like Microsoft Word, but instead of loading documents, it loads an entire operating system. Sounds great, and it often is. The problem is that it can be very heavy, since you need to hold both operating systems in memory at the same time.
5. **Docker:** Somewhat similar to the above, but much lighter. However, it is not trivial to set up.
6. **Cygwin:** Heavens, no.

There are more options that we're not covering here.

All in all, Putty+WinSCP is a good solution, but it does require you be constantly connected to a remote server. The **Ubuntu shell** is a pretty good solution too, and I would recommend it for most cases.

Make very sure you have a happily running Unix/Linux/macOS installation long before you go to a workshop that requires it. Don't expect a Linux installation to be the kind of thing that can be solved quickly on the spot. It isn't.

Ok, let's get started!

IV. Getting started

IV.1. Let's make a working directory

I promise I'll explain the below briefly, but before we do anything, let's create a directory where we can do what we want without worries of messing something else up:

2. Open the terminal.
3. **Optional:** go into your `Documents` directory by typing `cd Documents` and pressing enter (you can pick any directory you want, or your can stay where you were when you opened the terminal).
4. Create a new directory by typing `mkdir learnunix` and pressing enter.
5. Now go into this new directory by typing `cd learnunix` and pressing enter.

That's it, you should be inside the new `learnunix` directory. You can:

6. Confirm that you're in the new directory by issuing (i.e. typing and pressing enter) `pwd`. This *prints the working directory* (i.e., the directory or folder where you presently are).

Now we can start learning Unix a little more safely.

V. Navigating the Unix file system using `ls`, `cd`, and `pwd`

You're probably used to examining and moving around the file systems of Windows and/or Mac by clicking, double clicking, and so on. Although it's possible to do that in many Linux environments (e.g. Ubuntu), for most day-to-day data tasks such as those performed in bioinformatics, you'll probably be working on the **console** or **terminal**. So, let's explore that.

7. Open a Terminal or Console (the black screen that geeks use to talk to machines).

(on Windows, that would probably be the Ubuntu console, see section III above).

You should see a prompt, i.e. either a whiteish box or a blinking cursor or something that indicates that you can type into it. Just go ahead and try to type. This is what's called a ***shell***, the main way of direct communication between humans and Unix machines.

The shell is thus where you execute Unix programs, and it is itself a program. There are many different types of shells. Most Linux distros use `bash`, whereas macOS started using `zsh` by default in 2019 (they were using `bash` before then). They are all very similar, but not identical, just like two versions of Windows or macOS.

If you're ever trying to follow instructions from the internet and they don't work, there can be many reasons, but one of them is that your shell may be different from the one used on the website you're following. This is especially likely for Mac users, since modern Macs use `zsh` by default, whereas Linux typically uses `bash`.

V.1. Use `ls` to *list* the contents of a directory

8. In the terminal, type `ls` and press enter.

You should see a list of files in the directory where you are located. These are the contents of that directory. But that's not necessarily *all* the contents. There can also be hidden files and directories in any Unix directory. These typically have a name that starts with a dot. You can show them by asking `ls` explicitly to show you the hidden files and directories.

V.1.1. You can often use a dash – to give *options* to most Unix commands

Most Unix commands take *options* or *flags* via a simple dash preceding one or more letters. What exactly the letters do depends on the command, but for `ls`, we can use `-a` to show hidden files:

9. In the terminal, type `ls -a` and press enter. This will show hidden files too.

This will now show all files, hidden or not. At the very top of the list, you should see a single dot and a double dot that appeared as new files. These are two very special directories that are present in all Unix directories. The single dot `.` is *this directory* (i.e. the one you're in), and the double dot `..` is the *parent directory*, i.e. the one above—the one that contains the directory you're in. You can use these names to navigate, as we will see soon, but let's first explore a few more `ls` options.

10. Use `ls -l` to show details of each file or directory in the present directory.

A great Unix feature is that you can combine these flags, not only by chaining them (`ls -l -a`), but you can actually squash them together:

11. Use `ls -la` to show details of each file or directory in the present directory.

Here, it doesn't matter which order you throw the flags in, so `ls -al` will do the same as above. A nice combination is `-lAtrh`.

12. Use `ls -lAtrh` to show details (`l`) of each file or directory inside the present directory, including hidden ones other than the `.` and `..` (`A`), sort them by time (`t`), but in reverse order (`r`) so that the most recently modified file is shown at the end… oh, and show the byte-sizes of each file in human-readable format (`h`), so that it doesn't come up as e.g. `102347051` but `98M` instead.

Pretty neat, isn't it?

P.S.: Actually, in Unix, a directory is a file, and so is a program, and so is pretty much everything else, but try not to think too much about it—at least for this workshop.

V.2. Find out where you are using `pwd`

Just as you are familiar with in the file explorers of Windows or Mac, you can navigate the different directories (folders) of the Unix filesystem through the terminal. First, let's find out where we are:

13. Use `pwd` to print the working directory (i.e., your current location in the filesystem).

V.3. Making directories with `mkdir`

14. Run `mkdir dir1` to make a new directory. Then run `ls` to make sure it worked.

Make another directory:

15. Run `mkdir dir2` to make another new directory. Check with `ls`.

You can actually make a new a directory inside another:

16. Run `mkdir dir1/dir11` to make an internal directory.

If you try to do the same using an outer directory that does not exist, you will get an error:

17. Run `mkdir dir3/dir31` to get an error because `dir3` does not exist.

But you can force the console to make any necessary internal directories:

18. Run `mkdir -p dir3/dir31` to force the creation of both `dir3` and `dir31` inside it.
19. Run `mkdir -p dir3/dir32` create `dir32` inside `dir3`.

If you run `ls` now, you won't see any difference, because the new directories `dir11`, `dir31` and `dir32` are inside `dir1` and `dir3`, so you cannot see them from here. However, you can list the contents of any directory you want—not just the present one—you just have to give the path:

20. Run `ls dir1` to see the contents of `dir1`. Try the same with `dir3`.

V.4. Moving around the filesystem with `cd` (changing directories)

21. Go into the `dir1` directory by issuing `cd dir1`. Run `ls` once you're there.

You should see the internal `dir11` that we created before.

22. Go into the further internal `dir11` directory by issuing `cd dir11`. Find out where you are with `pwd`.

Hopefully the path that you see makes sense.

V.5. Moving *up* the filesystem with `cd ..` (change directory to the one above)

You should now be inside the `dir11` directory, which itself is inside `dir1`. Let's get back to our main working directory `learnunix`:

23. Issue `cd ..` to go up from `dir11` to `dir1`.
24. Issue `cd ..` again to go up from `dir11` back to `learnunix`.
25. Confirm that you are indeed inside `learnunix` by issuing `pwd`.

Actually, we could have done the whole thing in one go.

V.6. Moving into further directories with `/`

26. Get back into the innermost directory with `cd dir1/dir11`

As you see, we can use the `/`s to move into directories that are inside other directories.
It works backwards too:

27. Get back to `learnunix` in one go with `cd ../..`

V.7. `/Ab/so/lute` and `re/la/tive` paths

So far we have been using **relative paths**. This means we have been pointing to all directories by reference to where we presently are. So, `cd dir1` means "there's a directory in here called `dir1`, let's go in there". But you can actually go anywhere you want in your file system in one go. There are two important ways to remember. One is with **absolute paths** (we will explore the other one later).

28. Issue `pwd` to print the current working directory (i.e., where you are).

`pwd` prints **absolute paths**. You can tell because:

absolute paths always start with a `/`, whereas relative paths do not.

That very first `/` is the "root" of your file system. All your files and programs, i.e., pretty much everything in your computer, is inside it.
In my case, when I typed `pwd`, I saw:

```
/Users/vsojo/Documents/learnunix
```

I can now use this to get anywhere I want within that directory without having to provide relative references:

29. Go to an absolute path with:
    `cd /Users/youruser/Documents/learnunix/dir3/dir32`

Change `youruser` to whatever your username is. Also, if you didn't put your `learnunix` directory inside `Documents`, just use whatever path `pwd` printed.

Important: Use the **tab** key to complete the name of files, directories and programmes! Don't type everything letter by letter. This not only saves you time, it also significantly reduces errors.

V.8. Going home with *nothing*, `$HOME` , or `~`

30. Type `cd` (just like that, on its own) to go home (i.e. `/Users/youruser`).
31. **Use the up arrow in the keyboard to recover old commands**. If you click it twice you should recover `cd /Users/youruser/Documents/learnunix/dir3/dir32`. Run it.
32. Enter `pwd` to make sure you're in the right place.
33. Type `cd $HOME` to go home again. This uses the **environment variable** `$HOME`. Test that it worked with `pwd`.
34. Once again, use the up arrow twice to go back to:
    `/Users/youruser/Documents/learnunix/dir3/dir32`
35. A third way of going to your home directory is with `cd ~`.

V.9. Using `~` to navigate anywhere inside your home directory

Above we saw that one way to navigate anywhere we like is by providing absolute paths that start with `/`. The other main way is with `~`. Because ~ points to your home directory, you can use it to get anywhere inside your home directory in one go:

36. If you didn't in the last instruction, run `cd ~` to go home.
37. Run `cd ~/Documents/learnunix/dir3/dir32` to go into a directory that's inside the home directory.

This is super useful and super common. Note that you could use `$HOME/Docu…` instead, but `~/Docu…` is much shorter and preferred.

There are some cases in which `$HOME` may work better than `~`, chiefly because the shell only recognises `~` as home if there's no other character before it.

V.10. The `*` means "any combination of characters, of any length"

38. Run `cd ~/Documents/learnunix/dir2`
39. Make a file in there with `touch file1`

Note: `touch` makes an empty file, or updates the date of an existing one.

40. Make a couple more empty files: `touch file2 filx33` **(note the `x` in the last one!)**

Now there should be three files inside `dir2`: `file1`, `file2`, and `filx33`. If you run `ls`, you should see all three of them.

41. Run `ls *`

This will simply show everything (`*` means "any characters", or "everything").

42. Run `ls file*`

Now this changes things. Only those files or directories that start with `file` are shown. The one that starts with `filx` is ignored. A typical use of this behaviour is to show all files of a given extension, e.g.: `ls *.jpg`

V.11. The `?` means "any *one* character"

43. Run `ls fil?3*`

This will show any file or directory that has a name that begins with `fil`, is followed by any one character, then ends with any combination of characters of any length.
Note: These `?` and `*` tricks work with most other Unix commands.

44. Return to our main working directory: `cd ~/Documents/learnunix`

VI. Writing text to screen and to files

VI.1. Writing to screen with `echo`

45. Run `echo "Hello, World!"` to print to screen.

VI.2. Writing and appending to files with `>`, `>>` and `nano`

Let's send some text to a new file with `>`:

46. Run `echo "Hello, World!" > file.txt` to send text to a **new** file.
47. Check the contents of the file with `cat file.txt`

Now, let's **append** more text to the existing file with `>>`:

48. Run `echo "Greetings!" >> file.txt` to **append** text to a file.

If you use `>>` with a file that *does not* exist, it just gets created (no problem). However, if you use the single `>` with a file that *does* exist, you destroy it and replace it with whatever you sent last!

VI.3. Send the output of other commands to files using `>` and `>>`

49. Run `ls -l > myfiles.list` to send the output of `ls` to a new file.

Note that above we gave the text file an extension of `.txt`, and now we're using `.list`. **Unix does not care about extensions at all!** To a Unix system, it is up to the user to keep track of which file is supposed to contain what. Extensions have absolutely no meaning in Unix.

We can of course send this to an existing file too:

50. Run `ls -l >> myfiles.list` to send text the output to an existing file.

(Now the file has the list twice)

VI.4. Use `nano` to edit text files

51. Run `nano myfiles.list` to open a basic terminal-based text-file editor.
52. Write a bunch of text in different lines (whatever nonsense you like). Make sure you end up with at least 15 lines (they don't have to be long or meaningful, "`asfsadsf`" will suffice).
53. There are instructions in the lower part that tell you how to save and get out (the ^ means the Control key, not the actual ^ in the keyboard).

A very popular alternative to `nano` is `vim` (or `vi`). This however comes with a bit of a learning curve, but it is very much worth learning if you'll be spending a significant amount of time on the terminal.

VII. Reading the contents of files with `cat`, `head`, `tail` and `less`

54. Run `cat myfiles.list` to print the entire contents of a text file to screen.

Actually, `cat` is short for "concatenate", which means "to chain things together". This means we **can send multiple files to** `cat`:

55. Run `cat myfiles.list file.txt` to print the contents of multiple files together.

These files are small, so no problem. But if the files were big, we would not want to send it all to screen. For this, we can explore the first or last 10 lines:

56. Use `head myfiles.list` to print the **first** 10 lines of the file to screen.
57. Use `tail myfiles.list` to print the **last** 10 lines of the file to screen.

We can request a specific number of lines from both `head` and `tail`, using the desired number as a flag:

58. Use `head -3 myfiles.list` to print the **first 3** lines of the file.
59. Use `tail -2 myfiles.list` to print the **last 2** lines of the file.

VIII. Copying files and directories with `cp`

60. Issue `cp myfiles.list copied.file` to make a copy of the file.

Now run `ls -lrt`. You should see that both files are there, and they have the same size.
You can also make copies of a file to put them in a different folder, in which case you do not have to specify the name:

61. Copy a file into a directory with `cp myfiles.list dir1/.`

The `.` means "in that directory" is not necessary, but it is highly recommended.

62. Copy a file into a directory with `cp myfiles.list dir1/.`

It works the other way around too:

63. Go into one of the directories: `cd dir2`
64. Copy a file into *this* directory with `cp ../myfiles.list .`

In this case, the `.` on its own means "here".

VIII.1. Copying directories with `cp -r`

To copy a directory, you need to use the `-r` flag:

65. Copy a directory with: `cp -r dir1 dir4`

If you don't use the `-r` flag, `cp` will refuse to copy a directory.

IX. Hacking your `.bashrc` or `.zshrc.` shell profile

Running a customised version of `ls` (e.g. `ls -ltrhA` in my case) is so common that most people like to create themselves an **alias**, i.e. some sort of shorthand to speed up the command so that they don't have to type the entire thing every single time.
You could create a temporary `alias` directly on the command line:

```
alias l='ls -ltrhA'
```

But please don't do this.

The problem is that this will work fine during this active session, but the next time you open a new terminal session, the computer will have forgotten that alias.
You can get the computer to remember it forever by hacking your **profile file**, which is in your home directory (~, or `$HOME`) and is typically called `.bashrc` and/or `.bash_profile` if you're on bash, or `.zshrc` if you're on zsh.

66. Go to your home directory by issuing a simple `cd`, without any target.
67. Find out if you're on a **bash** shell or a **zsh** shell by typing `echo $SHELL`
68. Do `ls -la` to see the hidden files. You should see `.bashrc` or `.zshrc` there, depending of which of the two shells you're running.
69. Create a backup of this file, just in case you mess things up:

```
cp .zshrc .zshrc.bkup.20210607
```

Creating backup files is an extremely good idea, particularly when you're messing with high-level files such as `.zshrc`/`.bashrc`… or generally with files you don't understand much.

Let's open the file to edit it. I would typically use `vi`/`vim` as my editor of choice, but there's no time to introduce it properly here (just note that it's awesome). Let's just use `nano` instead:

70. Open the file with `nano .zshrc` (or `nano .bashrc` if you're on bash).

You'll probably see a few things in there already.

71. At the very top, enter the following lines:

```
# ALIASES
alias l='ls -ltrhA'
alias la='ls -lah'
alias ll='ls -lh'
```

What these do should be obvious: we're creating quick pseudo-commands to access versions of the `ls` command quickly.

72. **If you're on zsh** (i.e. on most modern Macs; you can find out by entering `echo $SHELL` into your terminal), please also add the following:

```
alias scp='noglob scp'
```

This is important for Mac users. I'll explain why later. Do not do that if you're on bash.

73. We're done here. Save and exit the file.

Now, if you type `l` or `la` you'll get an error. That's because we haven't actually loaded the new profile file, we just changed it. To load it, you could simply **open a new terminal tab (⌘T on Mac)**. Your profile gets loaded every time you start a terminal session (including new tabs/windows).
Alternatively:

74. Load the newly modified profile file with `source .zshrc` (or `source .bashrc`).

Now you can try issuing `l`.

X. Getting help with `man`, `whatis` and `apropos`

How do you actually find out about all the options of a command? Well, you could google for the command and see what you find. For something such as `ls`, there will probably be hundreds of thousands of tutorials online. StackOverflow is a great resource for all your coding needs. It's essentially just a web forum, but it's a vast one, where most coders go when in need and to help each other out. Think of it as a watering hole for coders of all skill levels.
But Unix systems come with their own help embedded, and in several forms. Let's explore the main one:

X.1. Use `man` to show the *manual* of a command

75. Type `man ls` to open the manual page for the command.

This will show you a very complete description of the command, including the full list of flags (options) and a lot more.

76. You'll see a colon `:` at the bottom left. This means there is more information if you scroll down. In a `man` page use space bar to scroll one screen ahead, or the up and down arrow keys to scroll line by line.

When you reach the end, you'll see `(END)`.
The following is important:

77. To get out of a `man` page, just hit the `Q` key in your keyboard.

Q: *"Help! How the hell do I get out of a `man` page?! I tried escape, space bar, nothing works!"*

A: Just hit `Q` on your keyboard. This is common to other similar Unix tools. Please remember it!

Alternatives to get help in Unix-like systems are `whatis` and `apropos` but, to be honest, you will only need them if you don't have access to the Internet... StackOverflow is your friend!

X.2. Try giving the –h flag to your external software command

It is a convention in software developed for Unix systems to have `–h`, `–help`, and/or `––help` be a signal for the programme to print out its help text. So just try running:

```
yourdesiredprogram –h
```

…and hopefully the developers of `yourdesiredprogram` were nice enough to include a help and did so in the standard way. On that note:

When you make your own scripts (including in Python or R or wherever), make sure to include a help text, which should be found by issuing the `–h` flag.

XI. Types of files and naming conventions in Unix

XI.1. Characters allowed in Unix names

Names in Unix systems can be created using pretty much any character you can think of, but this can get truly annoying depending on your keyboard and your skills finding hidden characters within it. In general, it is best if you avoid using non-English characters such as `ü`, `ñ`, `å` or `ç`, or `í`. Other "special characters", such as `$`, `/`, `&` or `<>`, `()` or `|` can cause trouble because they have a special meaning for Unix (we will cover some of these below), so, in general:

Don't use special characters in your variable or file names.

XI.2. Avoid spaces in Unix names! Like the plague!

Unlike in programming languages like C or Python, you *can* use space names in your Unix names, but please don't:

In the name of everything good, do not use spaces in your Unix names!

XI.3. Uppercase and lowercase are different!

In Unix, `apple`, `Apple`, `apPle` and `appLE` are all different things. Needless to say, if you're defining your own variables or naming your own files and directories, just stick to one variant and don't use more than one casing of the same word, it's a very easy way to make silly mistakes.

Remember that `b` and `B` are two entirely different things for Unix systems!

XI.4. The five main ways to separate multiple words in Unix names

- `camelCasingCapitalisesEveryWordExceptTheFirst`
- `PascalCasingLooksJustLikeCamelButStartsUppercase`
- `snake_casing_uses_underscores`
- `pincho-casing-or-skewer-casing-or-kebab-casing-looks-like-this`
- `hansel.and.gretel.casing.uses.dots`

All five are valid and very common in Unix systems, and you will encounter them all regularly, as well as combinations thereof within the same name.

In general, `snake_casing` and `hansel.gretel.casing` are favoured by many. Although it's very common, I'm not a fan of `pincho-casing` because the `-` can be confused with an algebraic minus in some instances (e.g., in scripting). `CONSTANTS_AND_ENVIRONMENT_VARIABLES` should be named using `UPPER_SNAKE_CASE` where possible.

Again, whatever you do, **do not use spaces or "special" characters** in your Unix names!

XII. Downloading files from the web with `wget` (Linux) or `curl -O` (Mac)

Note: Some of the following sections are inspired by the marvellous Linux tutorial by Adeel Malik & Muhammad Farjan Sjaugi, which is chapter 2 of the *Bioinformatics Practical Handbook* edited by Lloyd Low, ISBN 9789813144743. You are encouraged to acquire that book—it's very good!

We will download a FASTQ file from the internet, so that we can use it in our further analyses.

78. Make sure you're back in our working directory:

```
cd ~/Documents/learnunix
```

79. Now let's download a file from the internet with `wget` (Linux) or `curl -O` (Mac):

On Linux/WSL:

```
wget https://github.com/vsojo/Miscellaneous/raw/master/eDNA_AC_VOI26.fastq.gz
```

Important: `wget` will not work on Mac! You'll need to use `curl -O` instead (that's an upper-case letter 'o', not a zero, which looks like this `0` in the font I'm using):

On Mac:

```
curl -O https://github.com/vsojo/Miscellaneous/raw/master/eDNA_AC_VOI26.fastq.gz
```

This will get a file from the internet and put it in whichever directory you happen to be in. There are ways to get the file to end up somewhere else, but typically, it's best to just go to the place where you want the file to be using `mkdir` and `cd`, as we did above, and download from there.

80. Check that the download was successful with `ls -lh`

It's a good habit to check that your downloads have worked correctly. There is a program called `md5sum` that does this, but we won't explore it here. The file should be around `19 MB`.

XIII. Moving and renaming files with `mv`

81. Make a directory to store the file we just downloaded: `mkdir sequences`
82. Move the downloaded file into the new directory:
    `mv eDNA_AC_VOI26.fastq.gz sequences/.`

The `.` is not necessary, but it is very helpful and increases safety.

Warning: `mv` is very dangerous! It will replace anything that is named the same way as the destination file, without confirming.

83. Go into the new directory with `cd sequences` and confirm the file is there with `ls`.
84. Rename the file: `mv eDNA_AC_VOI26.fastq.gz eDNA.fastq.gz`

`mv` can be used to move a file to a different ***location***, or to a different ***name*** (i.e. to rename it).

XIV. Compressing and uncompressing files with `gzip`, `gunzip`, and `tar`

XIV.1. Using `gzip` and `gunzip` for compression and decompression

You will have noticed that the file we downloaded has a `.gz` extension. Do remember that **extensions mean absolutely nothing to Unix**, but `.gz` typically denotes a "gzipped" compressed file. You can very easily de-compress it (or "unzip" it):

85. Decompress the file with `gunzip eDNA.fastq.gz`

If you run `ls -lh`, you'll see that the .gz extension disappeared and the uncompressed file is much larger at approximately `109 MB`. You can easily regenerate the compressed file:

86. Compress the file again with `gzip eDNA.fastq`

This will take a few seconds. You'll see that the file is `19 MB` again. Note also that the uncompressed file disappeared and now only the compressed one remains.

gzip and **gunzip** destroy the original file, leaving only the resulting (de)compressed one.

Leave it compressed for now.

XIV.2. Using `tar` for packaging and compression

Another very common tool to compress files is `tar`, which works roughly as follows:

```
tar -zcvf name_of_the_new_package.tar.gz what*.ever you_want/to put/in* the/package
```

`tar` is a package maker—it bundles groups of files into a single "tarred" file (a package) for easier distribution and storage—you'll be using it very often to move stuff around or store it for posterity. The options (flags) above are:

- `c` : `c`reate a package (i.e. I want to make a new package, not e`x`tract things out of an existing one, which we'll do later using the `x` option instead of `c`).
- `z` : compress it with `gzip` (or decompress it if we are e`x`tracting).
- `v` : `v`erbose" (show me everything you're doing. This is optional and you can leave it out for quieter output. Definitely leave it out if you have thousands of files!).
- `f` : the following is the name of the packaged `f`ile that I want to create.

Try to get familiar with `tar`, you'll be using it quite a bit.

We could now go on and package the file with:

```
tar -zcvf package_name.tar.gz file_or_dir_to_package
```

But **please don't compress the downloaded file**. We will instead compress the entire directory that holds it.

87. Go up one directory with `cd ..`
88. Make sure you're in the right place with `pwd` and `ls`.

`ls` should show you the `sequences` directory that we created earlier, which contains the downloaded file.

89. **Package and compress** the entire directory with `tar`:

```
tar -zcvf seqs.tar.gz sequences/
```

Remember to use TAB to fill in the file names (won't work for the compressed file, since it doesn't exist yet and the computer cannot yet read our minds; but it should work for the directory).

You should now have a compressed file called `seqs.tar.gz`. But notice that we also have the original directory here, so:

Unlike, `gzip` and `gunzip`, `tar` does not destroy the original file or directory when it packages and compresses it.

90. Check the size of the new compressed file with `l` (or `ls -lh`).

You'll see that the size is the same as before. We tried to compress the directory with `gzip` by passing the `-z` flag to `tar`. But the directory contains a file that was already compressed by `gzip`. So, not much happens with the size when we tell `gzip` to try to compress something that it has already compressed—if anything it could get larger. There is no endless *compressinception* here.

XIV.3. Unpackaging/decompressing tar files

To unpackage and decompress a tar file, use:

```
tar -zxvf my_compressed_file.tar.gz
```

But please **don't decompress** the file that we created above. We will do it later.

XV. Removing (deleting) files and directories with `rm`

You will have noticed that now we have both the compressed directory and the original one with the file in it. Let's remove (delete) the downloaded file:

91. Delete the downloaded file with: `rm sequences/eDNA.fastq.gz`

Now if you check the contents of the directory with `rm sequences`, you will see that it is empty. Let's try to remove the empty directory with `rm`:

92. Try to delete the directory with: `rm sequences`

This fails because rm refuses to delete directories unless you insist by adding the flag `-r`. Let's also add the flag `-f`, which would make sure of deleting any files inside the directory:

93. Delete the directory with: `rm -rf sequences`

SUPER IMPORTANT: deleting files with `rm` is irreversible! If you tell Unix to remove a file, it will indeed *remove* the file. No recycling bin, no undo, nothing. Gone!

If you now run `ls`, you should see only our package file. The old directory and the downloaded file should be gone.

XVI. Matching and extracting text with `grep`

94. Let's unpackage the directory with: `tar -zxvf seqs.tar.gz`

Good, but if you remember, inside the directory we actually had a file that was itself already compressed.

95. Go into the directory with: `cd sequences`
96. Decompress the file with `gunzip eDNA.fastq.gz`

We now have a `.fastq` file. Let's explore it:

97. Run `head -16 eDNA.fastq` to show the first 16 lines

You'll see that there's a pattern repeating every 4 lines. If you don't know what these are:

1. The first line has a unique identifier for each sequence read, followed by a bunch of other things.
2. The second line has the actual sequence.
3. Don't ask me why the hell there's a `+` in the third line, nobody seems to know for sure other than "it's always been there".
4. And the fourth line is the "quality score" of the read, which depends on which machine was used to acquire the read.

Don't worry too much about the molecular genetics or bioinformatics of any of this right now—it's a good file to hack text in any case.

We can use `grep` to extract the lines of a file that match a desired pattern:

```
grep "regex" the_file_in_which_I_am_search.ing
```

Where `regex` is a "**regular expression**". No time to cover those here, but they are one of the most powerful things invented by humankind ever. Consider yourself emphatically encouraged to learn about them if you are planning to be a bioinformatician of any sort.

XVI.1. Finding lines that contain simple strings using `grep`

The simplest regular expression is just simple text. Actually, I always wanted to know whether the string "GATTACA" from that cool 1997 film (watch it if you haven't. Really!) appears in my sequences. Let's check:

98. Find all lines that contain a desired string:

```
grep GATTACA eDNA.fastq
```

So, you don't really need the quotes if you're only looking for a simple string.
I got a bunch of lines that actually do have the string. How many?

99. Count the number of lines that contain a desired string with `grep -c`

```
grep -c GATTACA eDNA.fastq
```

And which lines are those?

100. Show the line numbers for all lines that contain a desired string with `grep -n`

```
grep -n GATTACA eDNA.fastq
```

XVI.2. Unleashing the true power of `grep` with regular expressions

`grep` is far more powerful than a simple string finder.
I now want to find all lines that contain "GATTACA", but *only* if the line starts with at least two As, which are followed immediately by something that is not a T, then come any number of characters followed by the GATTACA, then any other characters, and then the line ends in T.

101. Learn and use regular expressions to get seriously advanced pattern matching:

```
grep -E "^A{2}[^T].*GATTACA.*[GT]$" eDNA.fastq
```

Don't forget the quotes; we do need them this time.
The `-E` is to call the "extended", more powerful version of `grep` (you could also call `egrep` directly, which is a shorthand for the same thing).
As you can surely guess, this gets extremely powerful really quickly. We're not even scratching the surface here. The message bears repeating a third time:

Learn as much as you can about **regular expressions**!

They are not exclusively a Unix thing. They are implemented in every language you're ever likely to come across (Python, C, Perl, Java, R, JavaScript, MatLab, Julia, Ruby… and many more), as well as in text editors such as BBEdit (Mac), Notepad++ (Win), or vim.

XVII. Modifying text with `sed`

If you take a look at all the #1 lines of the file, you'll see that they all start with "@J00113:238:HFWCLBBXX:4:". Let's say we decided that this is not informative and just want to replace most of that long string with a single letter just for reference:

102. Pipe text to `sed` to make replacements.

```
cat eDNA.fastq | sed 's/J00113:238:HFWCLBBXX:4/J/' > eDNA.cleaned.fastq
```

Make sure the text above is all on the same line. I just couldn't fit it in this document.
The first `s` before the `/` means "substitute". The two blocks separated by the `/`s are what you're substituting, and what with.
Note that we are creating a new file with `>` so that we don't alter the old one. Be warned that sometimes that's not feasible or convenient in bioinformatics (namely when you're working with gigantic data, which will probably be often). If you're changing a source file, make sure that you really want to do that.

103. Run `head -16` again to take a look at the changes.

Good!
Actually, I don't like that second block of text on the identifier line, the one that starts `2:N`…; let's get rid of it. But this time, we can do the replacement directly on our new file:

104. Use `sed -i` to replace text directly within a file:

```
sed -i 's/ 2:N.*//' eDNA.cleaned.fastq
```

Note the space after the first `/` and that we're using the new `cleaned` file, not the original one! Here we're taking all that text e.g. " 2:N:0:NGATCAGT+NGAGGATA" and replacing it with nothing. You'll see that we're using regular expressions again (the `.*` means any characters following). Did I mention how great and powerful RegExes are?

On Mac, for some silly reason, you need to add an extra `''` (empty single quotes) after the `-i`:

```
sed -i '' 's/ 2:N.*//' eDNA.cleaned.fastq
```

Cool. Now that we're at it, I also don't like the colons `:`, I want to change them to underscores `_`

105. Try with `sed -i 's/:/_/' eDNA.cleaned.fastq`

(remember to add the empty `''` if you're on Mac).

That worked, but only partly. You'll see that only the first `:` got replaced. To replace them all, you need to…

106. Use the greedy version of `sed`:

```
sed -i 's/:/_/g' eDNA.cleaned.fastq
```

Nice, so now you know that `sed` will do only one replacement per line unless you tell it to be greedy. There's of course a lot more to `sed`, but we'll stop here.

XVIII. Use `paste` to put multiple lines in one line

FASTQ files have 4 lines for each entry. It would be cool to have all that info in one line so that the information is a bit more manageable. We can do that easily with `paste`:

107. Use paste to put multiple lines in one line side by side:

```
cat eDNA.cleaned.fastq | paste - - - - > eDNA.tsv
```

…where the four dashes mean "put four lines side by side".

To be honest, if you find yourself doing a lot of `paste` and `cut`, you should probably be writing a Python script.

XIX. Use `awk` to extract information from a text file and do operations on it

Let's turn that tab separated file into a good old FASTA file using awk.

108. Do the following:

```
head eDNA.tsv | awk '{print $1 "\t" $2}'
```

Pretty neat, aye? `awk` is way way way more powerful than this. It can do mathematical operations such as calculating sums and averages by column, and a lot more, but we don't have time to go over it here.

109. OK, let's turn the entire thing into (almost) a FASTA file:

```
cat eDNA.tsv | awk '{print $1 "\n" $2}' > eDNA.fasta
```

Look at the head. We're nearly there. All we need to do is…

110. Use `sed` to change that `@` to a `>`

```
sed -i 's/@/>/g' eDNA.fasta
```

Et voilà, we have a FASTA file!

Note: In most Windows US keyboards the pipe `|` is found above the right Shift and left of Enter, as the shifted key of the backslash `\`. On US Macs, same thing but above the Return key.

XX. Connecting to other computers (remote servers) with `ssh` and `scp`

XX.1. Sending files to a remote server using `scp`

Note: `scp` does not work in zsh the same way as in bash. That's why those of us on macOS added that alias above into our `.zshrc` file: `alias scp='noglob scp'`. If you didn't, do it now.

An extremely important activity that you will probably need to do on a regular basis is sending files between Unix systems, e.g. between your local machine and the remote server. The most typical way to do this is with `scp`, which works almost exactly like `cp`, but between your local machine and a server, and in either direction. So, just as `cp` does it like:

```
cp the_existing.file the/new/place/.
```

`scp` works almost identically:

```
scp the_local.file yourusername@remote.server:.
```

The dot means "put it in that directory". By default `scp` connects directly to your `$HOME` directory, so `:.` means the file will end up there. If you want it somewhere else:

```
scp the_local.file yourusername@remote.server:where/in/there/you/want/it/.
```

Of course, the directory where you're sending the file needs to exist, otherwise you'll get an error.

Because of that, many people just put their files in `$HOME` and then move them from there.

It works exactly the same the other way around:

```
scp yourusername@remote.server:wherever/is/the.file .
```

That last dot means "put it here" (i.e. in whichever directory you happen to be in when you run the command). Needless to say, if you want it somewhere else:

```
scp yourusername@remote.server:wherever/is/the.file ~/where/you/want/it/.
```

Where `~/where/you/`… means "from my `$HOME` directory, go up to `where/you/`…

Just like with `cp`, you can also rename the file as you copy it:

```
scp yourusername@remote.server:remote_name.txt local_new_name.txt
```

XX.2. Connecting to a session on a remote server using `ssh`

This is very simple to do with `ssh`:

```
ssh yourusername@remote.server
```

Note: `ssh` stands for "secure shell", which is a way of saying "starting a shell session in a remote computer somewhere else, securely".

The complications with `ssh` are the same as with `scp`:

1. You need to make sure you know the address of the remote server.
2. You need to have your proper username on that server.
3. You need to know your password on that server.
4. You *may* need to have a VPN activated for that server, otherwise your connection may be refused (in fact, you probably won't even be able to see the remote server).
5. You'll need a stable connection to the internet. If your internet drops for more than a few seconds (including if your computer goes to sleep), you will have to connect again.

XXI. HELP! Getting out of the black screens of death

EMERGENCY INSTRUCTIONS TO REGAIN CONTROL OF YOUR ROGUE UNIX CONSOLE

If you ever get stuck in a strange place that does not look like the normal shell prompt, and nothing you do seems to bring you back—trust me, this *will* happen—try the following from your keyboard, in increasing order of despair (i.e. try the first one first, if that doesn't do the trick, try the second, and so on):

1. Press the `q` key.
2. Press `Escape`.
3. Press the `q` key, then hit Enter… but first make sure there's nothing typed on the last line that can be deleted with Backspace. In that case, delete everything, then `q`+Enter.
4. Press `:` then `q`, then hit Enter… but first make sure there's nothing on the last line.
5. Press `Ctrl` + `c`.
6. Press `Ctrl` + `d`.
7. Press `Ctrl` + `z`.

XXII. Configuring your `.ssh` profile

Assuming you do not yet have a .ssh id, you can create one with:

111. Go to your home directory: `cd`
112. Create public and private ssh ids by issuing: `ssh-keygen`

You will be prompted for details. For now, you can just hit Enter/Return to leave the details empty.

There will now be a folder called `.ssh` on your home directory. Note that this folder is hidden, so you will need to do `ls -la` to see it.

XXII.1. Using .ssh/config to access a remote server quickly

You probably realise that it's a pain to have to write your username and the name of the server every single time you want to connect to it or copy something to/from there.

113. Go into the `.ssh` directory: `cd .ssh`
114. If it doesn't exist, create and edit a new configuration file: `nano config`
115. Add the following:

```
Host SERV1
     Hostname your.servers.address.edu
     User yourusername

Host SERV2
     Hostname another.servers.address.edu
     User yourusername
```

Replace SERV1 and SERV2 with whichever name you like as a nickname for your server. For example, I use one called Huxley, so my nickname is HUX. Of course, also replace the address and username with whatever is appropriate for your case.

Save and exit.

Now, if you restart your terminal, you should be able to use these newly created connections to quickly and easily `ssh` and `scp` to your servers:

116. Try `ssh SERV1`. It should get you to your server easily.
117. To copy files *to* the server, use `scp mylocal.file SERV1:where/I/want/it/.`
118. And to copy files *from* the server: `scp SERV1:where/it/is/remote.file .`

Remember that you can use `?` and `*` to copy many files of similar names.

You will of course have to be connected to the VPN and will be asked for your password. You will also potentially have to go through two-factor authentication. Not fun, but it is what it is. (actually, there are ways to go into a remote server without having to specify your password every time. Remember when we ran `ssh-keygen`? This created an `id_rsa.pub` file. This file can be used as a key to tell the remote server that this user on this computer is trusted, so it should just let it in whenever it comes calling. Search online for "ssh authorized_keys". Note that some admins don't like this, so you didn't hear it from me!)

XXIII. A very timid introduction to `shell` scripting

119. Create a new directory with `mkdir scripts`
120. Create a new file with `nano sayhello.sh` and make its contents:

```
#!/bin/bash
echo "Hello, World!"
```

Save and exit. We won't explore shell scripting any further here. Know that it's a big universe, and that you can use any combination of the commands that we have been learning here inside your shell scripts, plus many more.

XXIII.1. Use `chmod` to change file permissions

If you now list the files with `ls -l`, you will see that on the left there is a section that reads `-rw-r--r--`. These are the **permissions** of the file.

- The first character indicates whether this is a directory (a `d` would be present in that case).
- The next 9 are 3 blocks of 3, and they indicate what the owner, group members, and everyone else can do with this file/directory. `r` means read, `w` means "write" (i.e. modify), and `x` means "execute" (i.e. run). In the example above, the owner can read+write, the rest can only read. Nobody can execute so, we have to change that because this is a script:

121. Add execution permissions with `chmod 754 sayhello.sh`

What those numbers mean is not trivial to explain here, but, briefly, they are binary code, one number for each of the three triplets:

r	w	x
$2^2 = 4$	$2^1 = 2$	$2^0 = 1$

So, if you want the owner to be able to read, write and execute, you turn all the bits on:

$$1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 7$$

Then members of my group should be able to read and execute, but not write:

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$$

And everyone else should only be able to see the file contents, but not change or execute it:

$$1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 4$$

…and that's where the 754 came from. The most typical permissions for executable files are 755.

122. You can now run this script with `./sayhello.sh`

The `./` is necessary to let the shell know where the script is to be found.

XXIII.2. Add a directory to the `$PATH` so that it can be seen from anywhere

123. Check where you are with `pwd`.

I have `/Users/vsojo/learnunix/scripts`, but you may have something else.

124. Add the directory into the path: `PATH=$PATH:/Users/vsojo/learnunix/scripts`

Now you should be able to access the `sayhello.sh` script from anywhere. Just enter `sayhello.sh` in any directory you like, and it should work.

This alteration to the PATH is only valid during this terminal session.

If you want the contents of the directory to always be available whenever you start a terminal session, you need to add the path modifier in the last instruction into your `.zshrc`, `.bashrc`, or `.bash_profile`, as appropriate.