Microbial Ecology: Introduction into Data Analysis

Nina Dombrowski

Institute for Biodiversity and Ecosystem Dynamics (IBED)
University of Amsterdam (UVA)

n.dombrowski@uva.nl

Table of contents

Introduction

Welcome to the *Microbial Ecology Command-Line Crash Course*!

In this short tutorial, you will learn how to use the command line and a High-Performance Computing (HPC) environment in order to move from raw 16S sequencing reads to a basic count table of microbial taxa.

This course is designed for students in microbial ecology with little or no prior experience using the command line. During this tutorial, you will learn how to:

1. **Install a Terminal**
    - How to access a Unix/Linux command line
    - How to run basic commands
2. **Document your Code**
    - How to use markdown, Quarto, and comments in scripts
    - How to track your work and make your analyses reproducible
3. **Navigate the Command Line**
    - Navigating the filesystem (`cd`, `ls`, `pwd`)
    - Working with files (`mkdir`, `cp`, `mv`, `rm`, `cat`, `head`, `less`)
    - Searching and filtering data (`grep`, `wc`, `cut`)
    - Viewing and understanding FASTQ files
4. **Work with an HPC**
    - Connecting to the cluster (SSH)
    - Understanding file systems and job schedulers (e.g., SLURM)
    - Running and interpreting command-line tools (e.g., `fastqc`, `seqkit`)
    - Setting up and activating conda environments
    - Running workflows with Snakemake

1 Setting up a terminal

1.1 Terminology

The **command-line interface (CLI)** is an alternative to a graphical user interface (GUI), with which you are likely more familiar. Both allow you to interact with your computer's operating system but in a slightly different way:

- In a **GUI**, you click buttons, open folders, and use menus
- In the **CLI**, you type text commands and see text output

The CLI is also commonly called the **shell**, **terminal**, **console**, or **prompt**. These terms are related but not identical:

- The **terminal** (or **console**) is the window or program that lets you type commands. It is the interface you open, for example, *Terminal* on macOS, *WSL* on Windows, or an HPC login session.
- The **shell** is the program that actually interprets the commands you type inside the terminal and tells the operating system what to do. The shell understands commands like `cd` that is used to change directories.
- **Prompt**: the text displayed by the shell that indicates it is ready to accept a command. The prompt often shows useful information, like your username, machine name, and current directory. Example of a prompt: `user@machine:~$`

There are several types of shells — for example, **bash** or **zsh** (that use slightly different languages to issue commands). This tutorial was written on a computer that uses **bash**, which stands for *Bourne Again Shell*.

1.2 Installation guides

1.2.1 Linux

If you're using Linux, you already have everything you need — no installation required. All Linux systems come with a terminal and a shell, and the default shell is usually Bash.

You can open a terminal from your applications menu or by searching for Gnome Terminal, KDE Konsole, or xterm, depending on your desktop environment.

To confirm which shell you're using, type:

```
echo $SHELL
```

If the output does not end in `/bash`, you can start Bash manually by typing:

```
bash
```

You should then see a new prompt (often ending in `$`) indicating that Bash is running.

1.2.2 Mac

All Mac computers also come with a built-in terminal and shell. To open the terminal:

- In Finder, go to Go → Utilities, then open Terminal.
- Or use Spotlight Search (+ Space), type Terminal, and press Return.

The default shell depends on your macOS version:

- macOS Mojave (10.14) or earlier → Bash
- macOS Catalina (10.15) or later → Zsh

Check which shell you're currently using:

```
echo $SHELL
```

If the output does not end in `/bash`, you can start Bash manually by typing:

```
bash
```

Then check again with echo `$SHELL`. If you have trouble switching, don't worry — nearly all commands in this tutorial will also work in Zsh.

1.2.3 Windows

Unlike macOS or Linux, Windows doesn't include a Unix-style terminal by default, so you'll need to install one of the following:

1. MobaXterm (recommended)
    - Provides a terminal with Linux-like commands, built-in
    - Installation guide: MobaXterm setup instructions
    - Easiest option for beginners and lightweight to install.
2. Windows Subsystem for Linux (WSL2)
    - Gives you a full Linux environment directly on Windows.
    - Recommended if you're comfortable installing software or already have some command-line experience.
    - Uses Ubuntu by default, which includes Bash and all standard Linux tools.
    - Installation guide:Microsoft WSL install instructions

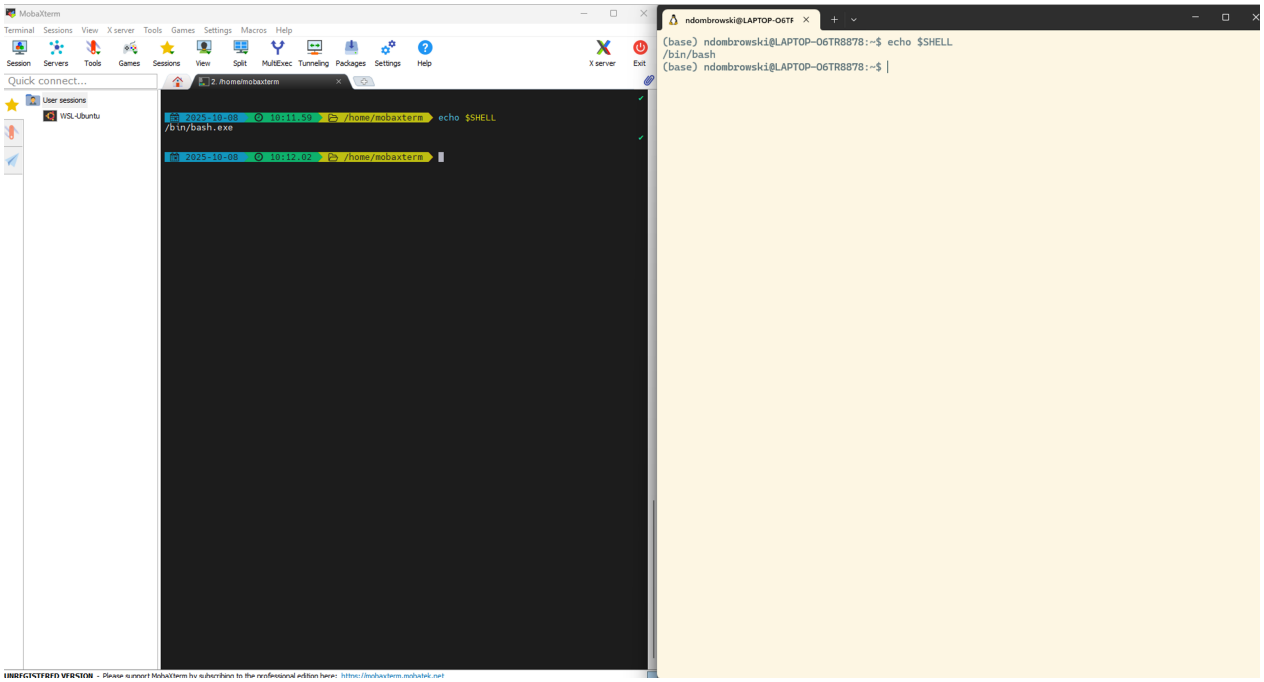
Once installed, open your terminal (MobaXterm or Ubuntu via WSL2) and verify that Bash is available:

```
echo $SHELL
```

1.3 Sanity check

After setup, open your terminal. You should see something like this:

Mobaxterm terminal **WSL terminal**



If you see a prompt ending in `$` (for example `user@machine:~$`), your shell is ready — you're all set to follow along with the tutorial!

2 Documenting your code

Documenting your code is crucial for both your future self and anyone else who might work with your code. Good documentation helps others (and your future self) understand the purpose, functionality, and usage of your scripts, much like a detailed lab notebook.

For more in-depth guidance, see A Guide to Reproducible Code in Ecology and Evolution. While examples are mainly in R, the principles are general and apply across programming languages.

2.1 Choosing your editor

2.1.1 Plain text editor

Avoid visual editors like Word, as they are not designed for code and can inadvertently change syntax (e.g., replacing backticks \`` `with apostrophes`' ').

Start simple with a **plain text editor**, such as:

- **TextEdit** (Mac)
- **Notepad** (Windows)

These allow you to write and save code safely, though they lack advanced features like syntax highlighting or integrated code execution.

2.1.2 Rmarkdown in RStudio

RMarkdown combines plain text, code, and documentation in one document. You can write your analysis and explanatory text together, then "knit" the document to HTML, PDF, or Word.

To create an RMarkdown file in RStudio:

1. Go to **File → New File → R Markdown**
2. Choose a **title**, **author**, and **output format**
3. Write your code and text
4. Click **Knit** to render the document

More info: RMarkdown tutorial

2.1.3 Quarto in Rstudio

Quarto is a next-generation alternative to RMarkdown. It supports R, Python, and other languages, and offers more output formats and customization options.

To create a Quarto document:

1. Go to **File → New File → Quarto Document**
2. Choose a **title**, **author**, and **output format**
3. Click **Render** to generate your document

More info: Quarto documentation

2.2 Markdown for Documentation

Markdown is a lightweight language for formatting text. You can easily add headers, lists, links, code, images, and tables.

Headers:

Use `#` to add a header and separate different sections of your documentation. The more `#` symbols you use after each other, the smaller the header will be. When writing a header make sure to always put a space between the `#` and the header name.

```
# Main Header
## Subheader
```

Lists:

Use `-` or `*` for unordered lists and numbers for ordered lists.

Ordered lists are created by using numbers followed by periods. The numbers do not have to be in numerical order, but the list should start with the number one.

```
1. First item
2. Second item
3. Third item
4. Fourth item
```

```
1. First item
2. Second item
3. Third item
    1. Indented item
    2. Indented item
4. Fourth item
```

Unordered lists are created using dashes (`-`), asterisks (`*`), or plus signs (`+`) in front of line items. Indent one or more items to create a nested list.

```
- First item
- Second item
- Third item
- Fourth item
```

```
 - First item
- Second item
- Third item
    - Indented item
    - Indented item
- Fourth item
```

You can also combine ordered with unordered lists:

```
1. First item
2. Second item
3. Third item
    - Indented item
    - Indented item
4. Fourth item
```

Code Blocks:

Enclose code snippets in triple backticks followed by the computational language, i.e. bash or python, used.

````
```bash
grep "control" downloads/Experiment1.txt
```
````

Links:

You can easily add links to external resources or within your documentation as follows:

```
[Link Text](https://www.example.com)
```

Emphasis:

You can use * or _ to write italic and ** or __ for bold text.

```
*italic*
**bold**
```

Pictures

You can also add images to your documentation as follows:

```
![Alt Text](path/to/your/image.jpg)
```

Here, replace `Alt Text` with a descriptive alternative text for your image, and `path/to/your/image.jpg` with the actual path or URL of your image.

Tables

Tables can be useful for organizing information. Here's a simple table:

```
| Header 1 | Header 2 |
| ---------| ---------|
| Content 1| Content 2|
| Content 3| Content 4|
```

3 Navigating the command line

3.1 `pwd`: Find out where we are

Once your terminal is open, let's get oriented by typing your first command:

```
pwd
```

The command `pwd` stands for **print working directory**. It tells you where you currently are in the file system — that is, which folder (directory) your shell is "looking at" right now. You should see something like:

```
/Users/YourUserName
```

> Tip: Finding the Desktop on Different Systems
>
> Your home directory path varies slightly across operating systems. Here's how to locate yourself and connect to familiar locations like the Desktop:
> **macOS**
>
> - Your home directory is `/Users/YourUserName`
> - To open the folder you are currently in Finder:`open .`
> - Your desktop is at `/Users/YourUserName/Desktop`
>
> **MobaXterm (Windows)**
>
> - Your home directory is `/home/mobaxterm`
> - By default, this is temporary and is deleted when you close MobaXterm. To make it permanent:
>   - Go to Settings –> Configuration –> General
>   - Under Persistent home directory, choose a folder of your choice
> - To open the folder you are currently in the Windows File explorer: `explorer.exe .`
> - Your Desktop is usually at: `/mnt/c/Users/YourUserName/Desktop` or `/mnt/c/Users/YourUserName/OneDrive/Desktop` (when using OneDrive)
>
> **WSL2 (Windows)**
>
> - Your home directory is`/home/YourUserName`
> - To open the folder you are currently in the Windows File explorer: `explorer.exe .`
> - Your Desktop is usually at: `/mnt/c/Users/YourUserName/Desktop` or `/mnt/c/Users/YourUserName/OneDrive/Desktop` (when using OneDrive)

If you want to access the Uva OneDrive folder:
If your OneDrive folder name includes spaces (like OneDrive - UvA), use quotes around the path:

```
cd "/mnt/c/Users/YourUserName/OneDrive - UvA"
```

3.2 `ls`: List the contents of a directory

Now that we know where we are, let's find out what is inside that location. The command `ls` (short for *list*) shows the files and folders in your current directory. Type the following and press enter:

```
ls
```

You should see something like this (your output will vary depending on what's in your directory):

```
Custom_scripts  LICENSE  README.md  _quarto.yml  docs  img  index.qmd  source  styles.css
```

The colors and formatting depend on your terminal settings, but typically:

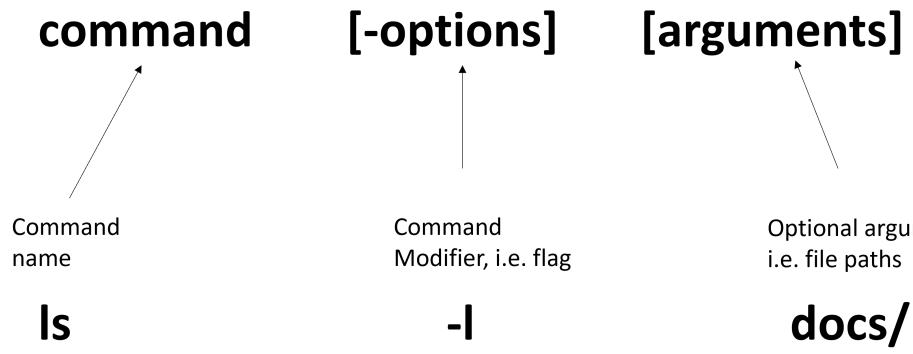
- Folders (directories) appear in one color (often green or blue)
- Files appear in another (often white or bold)

If your directory contains many items, the output can quickly become overwhelming. To make sense of it, we can use options and arguments to control how commands behave.

3.3 The structure of a command

A command generally has three parts:

- A command: The program you want to run, i.e. `ls`
- An option (or flag): A way to modify how the command behaves, i.e `-l` (long format)
- An optional argument: The input, i.e. a file or folder

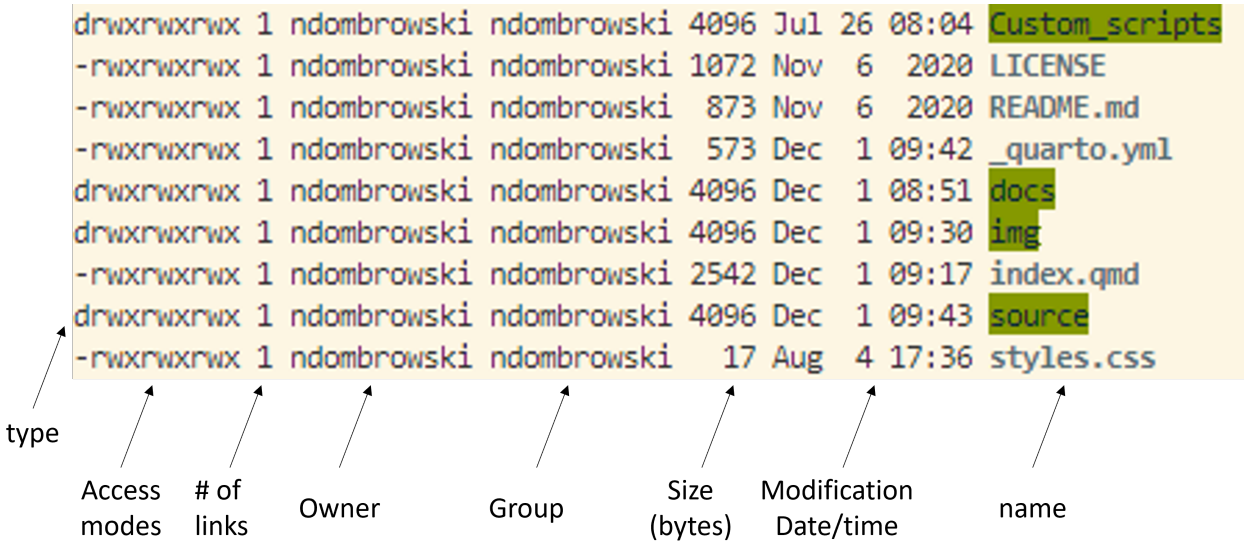


- Unix is case –sensitive
- Need to have a space between command, options and arguments
- Do not add a space between the `-` symbol and the option

Try the following command in your current directory to "List (ls) the contents of the current folder and show details in long format (-l)":

```
ls -l
```

After running this you should see a more detail list of the contents of your folder.In the example below we can see that we now print additional information about who owns the files (i.e. access modes), how large the files are, when they were last modified and of course the name:



3.4 Getting help

At some point, you'll want to know what options a command has or how it works. In this case, you can always check the **manual pages** (or *man pages*). Try this:

```
man ls
```

This opens the manual entry for the command ls. You can scroll through it using:

- ↑ / ↓ arrows or the space bar to move down,
- b to move back up,
- q to quit the manual.

Not all commands use man. Depending on the program, there are a few common patterns you can try

- `man ls`
- `ls --help`
- `ls -h`

For complex software, like bioinformatics tools, the most helpful documentation is often:

- The tool's official website or GitHub repository
- The –help output that lists all parameters and examples

3.5 `mkdir`: Make a new folder

Before we start moving around, let's first learn how to create new folders (also called directories).This is something we will do often, for example, to keep raw data, results, and scripts organized in separate places. The command we use for that is `mkdir`, which stands for *make directory‘.

For now, we will use it to create a shared working folder for this tutorial. Don't worry about how to move into the folder yet — we'll cover that next with the cd command.

```
# Move into the home directory (the starting point of your system)
cd ~

# Create a new folder called 'data_analysis'
mkdir data_analysis

# Check that the folder was created successfully
ls
```

You should see a new folder called data_analysis appear in the list. We will use this folder as our project space for all exercises in this tutorial.

```
# Move into the new folder
cd data_analysis

# Confirm where we are
pwd
```

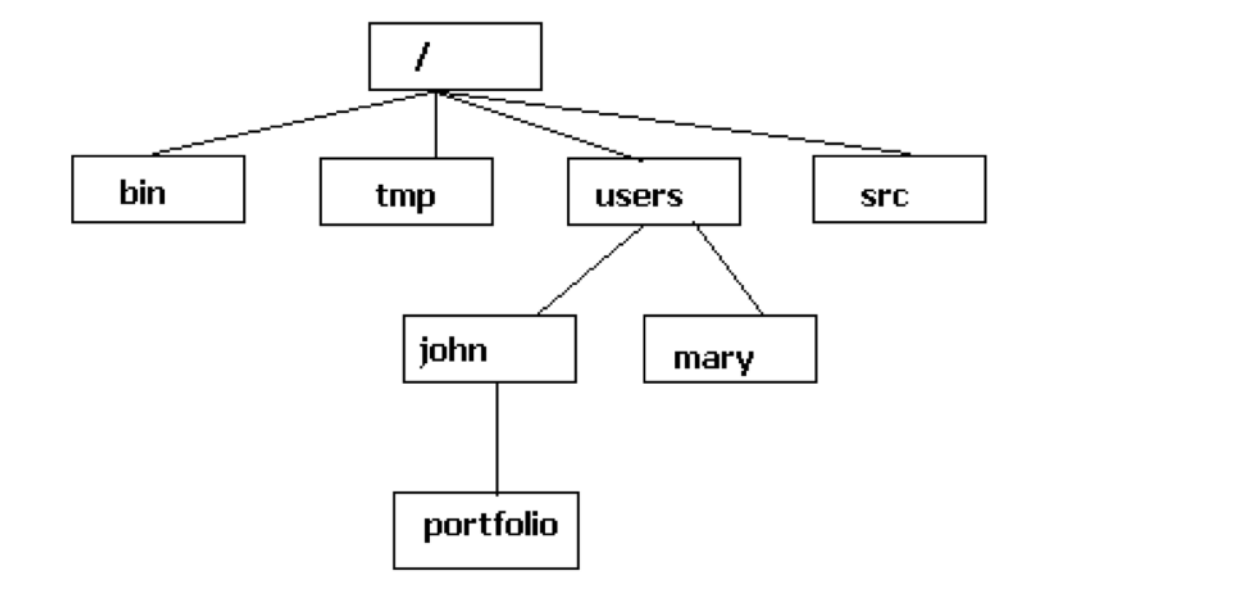
You should now see that your current working directory (pwd) ends with `/data_analysis`. This is where we'll keep all our files and run our commands for the next exercises.

> Tip: Commenting your code
>
> Notice how we added # and some notes above each command?
> Anything written after # in Bash is a comment — it won't be executed, but it helps you (and others) understand what the command does.
> In your own work, add short, meaningful comments above key steps. Avoid restating the obvious — instead, explain why you're doing something or what it achieves.

3.6 `cd`: Move around folders

Now that we have our own project folder, let's learn how to move around the file system.

The file system is structured like a tree, starting from a single "root" directory `/`. All other folders branch out from it.



There are two ways to specify a path:

- Absolute path: starts from the root (e.g. `/Users/Name/Documents`)
- Relative path: starts from your current location (e.g. `Documents` if you're already in `/Users/Name`)

Let's practice moving between folders (at each step, use `pwd` in case you feel that you get lost):

```
# Make a new folder in data_analysis
mkdir raw_reads

# Move into a new folder (by using the relative path)
cd raw_reads
```

```
# Move one level up
cd ..

# Move multiple levels at once
cd data_analysis/raw_reads

# Quickly go back home
cd ~
```

In the code above, the tilde symbol (`~`) is a shortcut for your home directory. It's equivalent to typing the full absolute path to your home (e.g. `/Users/YourName`) but it is much faster to type.

Tip: Command-line completion

Some tips for faster navigation:

- Use Tab for autocompletion — type the first few letters of a folder name and press Tab.
- If there's more than one match, press Tab twice to see all options.
- Use ↑ / ↓ arrows to scroll through previously entered commands

Exercise

- Create a new folder inside your data_analysis directory called results.
- Move into that folder and confirm your location with pwd.
- Move back to data_analysis using cd ..
- Use ls to confirm both results and raw_reads are there.

3.7 `wget`: Download data

Next, let's download a sequencing file for this exercise. We'll use wget to fetch a file `barcode01_merged.fastq.gz` from the web:

```
# Download the example fastq into the current directory
wget https://github.com/ndombrowski/MicEco2025/raw/refs/heads/main/data/barcode01_merged.fastq.
```

After you download data, it is always a good idea to do some sanity check to see if the file is present and check how large it is:

```
# list files in long and human-readable format
ls -lh barcode01_merged.fastq.gz

# alternative: disk usage of that file
du -h barcode01_merged.fastq.gz
```

3.8 `cp`: Copy files

`cp` duplicates files or directories. Let's use it to copy the downloaded file into `raw_reads`:

```
# Copy file into raw_reads
cp barcode01_merged.fastq.gz raw_reads/

# Show content of both locations
ls -l
ls -l raw_reads
```

When running the two `ls` commands, we see that we now have two copies of `seq_project.tar.gz`, one file is in our working directory and the other one is in our data folder. Copying large amounts of data is not ideal since we will use unneccassary space. However, we can use another command to move the tar folder into our data folder.

3.9 `mv`: Move (or rename) files

`mv` moves or renames files without creating a second copy:

```
# Move the file into raw_reads
mv barcode01_merged.fastq.gz raw_reads/

# Verify
ls -l
ls -l raw_reads
```

Notice that `mv` will move the tar file and, without asking, overwrite the existing tar file we had in the data folder when we ran `cp`. This means for you that if you run move its best to make sure that you do not overwrite files by mistake.

3.10 `rm`: Remove files and directories

To remove files and folders, we use the `rm` command. We want to remove the `barcode01_merged.fastq.gz` file since we do not need two copies:

```
# Remove a file
rm barcode01_merged.fastq.gz

# Check if that worked
ls -l
```

If we want to remove a folder, we need to tell `rm` that we want to remove folders using an option. To do this, we use `-r` , which allows us to remove directories and their contents recursively.

> ❗ Important
>
> **Unix does not have an undelete command.**
> This means that if you delete something with rm, it's gone. Therefore, use `rm` with care and check what you write twice before pressing enter! Also, NEVER run `rm -rf` on the root `/` folder or any important path. Always double check the path or file name before pressing enter.

3.11 Exploring file contents

Before working with large compressed sequencing files, let's get comfortable with some basic tools to inspect text files. These commands are useful for quickly checking what a file contains without opening it in a text editor.

```
# Download Alice Adventures in Wonderland
wget https://www.gutenberg.org/files/11/11-0.txt

# Verify
ls -lh 11-0.txt
```

3.11.1 `cat`: Print the full file

We can use `cat` print the entire file to the screen — fine for short files, but overwhelming for long ones.

```
cat 11-0.txt
```

3.11.2 `head` and `tail`: View parts of a file

To only print the first few or last few lines we can use the `head` and `tail` command, respectively:

```
# Show the first 10 lines (default)
head 11-0.txt

# Show the last 5 lines
tail -n 5 11-0.txt
```

3.11.3 `less`: View the full file

`less` let's you view a file's contents one screen at a time. This is useful when dealing with a large text file (such as a sequence data file) because it doesn't load the entire file but accesses it page by page, resulting in fast loading speeds.

```
less -S 11-0.txt
```

- You can use the arrow Up and Page arrow keys to move through the text file
- To exit less, type `q`

> Tip: Editing text files
>
> You can also edit the content of a text file and there are different programs available to do this on the command line, the most commonly used tool is `nano`, which should come with most command line interpreters. You can open any file as follows:
>
> ```
> nano 11-0.txt
> ```
>
> Once the document is open you can edit it however you want and then
>
> - Close the document with `control + X`
> - Type `y` to save changes and press enter

3.12 `wc`: Count things

Another useful tool is the `wc` (= wordcount) command that allows us to count the number of lines via `-l` in a file. It is an useful tool for sanity checking and here allows us to count how many lines of text are found in the text document.

```
wc -l 11-0.txt
```

3.13 Pipes

So far, we've run one command at a time — for example, using wc -l to count lines or head to look at the first few lines. But often, you'll want to combine commands so that the output of one becomes the input of another. That's what the pipe (|) does.

```
# Example: show only the first 5 lines of the line count output
cat 11-0.txt | head -n 5
```

Here:

- `cat 11-0.txt`prints the full text.
- The pipe (`|`) sends that output directly to head.
- `head -n 5` only shows the first 5 lines of the result.

3.14 `zcat` and `head`: Working with compressed files

Now that you know how to combine commands, let's try the same idea with a compressed file.

The `.gz`ending we have in the sequence data file means that the file is compressed with gzip. To view its contents without uncompressing it, we use zcat. Because these files can be very large, we'll use a pipe again to only view the first few line

```
zcat raw_reads/barcode01_merged.fastq.gz | head
```

This command:

- Decompresses the file temporarily (zcat),
- Sends the decompressed text through a pipe (|),
- Displays only the first few lines (head).

For macOS users, the equivalent command is often gzcat:

```
gzcat raw_reads/barcode01_merged.fastq.gz | head
```

3.15 Working with multiple files

So far, we've worked with single files. In practice, sequencing data often comes as multiple files — for example, one FASTQ file per barcode. Let's practice handling several files at once.

3.15.1 `touch`: Make example files

To try this safely, we'll create a few empty "dummy" files in our raw_reads folder using the touch command:

```
cd raw_reads

# Create a few dummy FASTQ files
touch barcode02_01.fastq barcode02_02.fastq barcode03_01.fastq

# Check what was done
ls -lh
```

3.15.2 Wildcards (*): Match multiple files

Typing every filename can get tedious. Wildcards help you work with groups of files using pattern matching.

```
# List all files that start with "barcode02"
ls barcode02_*

# List all files ending with ".fastq"
ls *.fastq
```

3.15.3 `cat`: Combining files

The cat command doesn't just print files — it can also concatenate (join) multiple files into one. For example, if you have several sequencing runs from the same barcode and want to merge them:

```
# Combine the two barcode01 files into one merged file
cat barcode02_*.fastq > barcode02_all.fastq
```

Here:

- `barcode01_*.fastq`selects all files that start with barcode01_
- `>` tells the shell to write the combined output into a new file called barcode01_merged.fastq

> Important: Be careful with >
>
> The `>` operator overwrites files without asking. If you want to add (append) to an existing file instead of replacing it, use `>>`: