Introduction to UNIX for Biologists

Jakob Willforss

November 13, 2016
v1.6

Contents

1 Introduction to the course **6**
- 1.1 Acknowledgements . . . 6
- 1.2 Welcome . . . 6
- 1.3 The structure of this material . . . 7
  - 1.3.1 Chapters . . . 7
  - 1.3.2 Chapter structure . . . 7
- 1.4 How to find help . . . 9
  - 1.4.1 The man command . . . 9
- 1.5 Accessing data . . . 10
  - 1.5.1 ssh . . . 10
- 1.6 Exercises . . . 12
  - 1.6.1 Connect to a remote computer . . . 12
  - 1.6.2 Explore the man-pages . . . 12
  - 1.6.3 Finding help on the internet (*) . . . 13
- 1.7 Checkpoint . . . 14
  - 1.7.1 UNIX commands . . . 14
- 1.8 Further reading . . . 15
  - 1.8.1 Make command run even if you close your terminal . . . 15

2 Introduction to the File System **16**
- 2.1 The UNIX file system . . . 16
  - 2.1.1 A brief primer on paths . . . 17
- 2.2 Important file system commands . . . 17
  - 2.2.1 pwd . . . 17
  - 2.2.2 ls . . . 18
  - 2.2.3 cd . . . 19
  - 2.2.4 Demonstration . . . 20
- 2.3 Understanding paths . . . 20
- 2.4 Use tab completion . . . 21
- 2.5 Special directories . . . 23
  - 2.5.1 The root directory (/) . . . 23
  - 2.5.2 The home directory (~) . . . 23
  - 2.5.3 Current directory (.) . . . 24
  - 2.5.4 Parent directory (..) . . . 24
- 2.6 Exercises . . . 26

2.6.1 Download the files to your home directory . . . 26
2.6.2 Trying out the file system commands . . . 27
2.6.3 Investigating file system . . . 28
2.7 Checkpoint . . . 29
2.7.1 UNIX commands . . . 29
2.8 Further reading . . . 30
2.8.1 Explore the system directories . . . 30
2.8.2 Hidden files . . . 30

3 Working with files in UNIX 31
3.1 Files and file formats in UNIX . . . 31
3.1.1 Regular text files . . . 31
3.1.2 Binary files . . . 31
3.1.3 Compressed files . . . 32
3.2 File commands . . . 32
3.2.1 mv . . . 32
3.2.2 cp . . . 34
3.2.3 rm . . . 35
3.3 Folder commands . . . 35
3.3.1 mkdir . . . 36
3.3.2 rmdir . . . 36
3.3.3 rm -r . . . 36
3.4 Looking inside files . . . 37
3.4.1 cat . . . 38
3.4.2 head . . . 38
3.4.3 tail . . . 39
3.4.4 less . . . 39
3.5 Editing text . . . 39
3.5.1 nano . . . 40
3.6 Exercises . . . 41
3.6.1 Make and manage your own file . . . 41
3.6.2 FASTA management (*) . . . 42
3.7 Checkpoint . . . 44
3.7.1 UNIX commands . . . 44
3.8 Further reading . . . 45
3.8.1 Text editors . . . 45

4 Working with file content 46
4.1 Important bioinformatic file formats . . . 46
4.1.1 The FASTA file format . . . 46
4.1.2 FASTQ . . . 46
4.1.3 GFF . . . 49
4.2 File content commands . . . 49
4.2.1 wc . . . 51
4.2.2 diff . . . 51

4.2.3 grep . . . 52
4.2.4 cut . . . 53
4.2.5 sort . . . 55
4.2.6 uniq . . . 57
4.3 Exercises . . . 59
4.3.1 Introduction to the exercise . . . 59
4.3.2 Exploring the FASTA file . . . 60
4.3.3 Exploring the FASTQ file . . . 60
4.3.4 Exploring the GFF file (*) . . . 61
4.3.5 Working with the annotation - Case study (**) . . . 63
4.4 Checkpoint . . . 65
4.4.1 UNIX commands . . . 65
4.5 Further reading . . . 66
4.5.1 Converting multi-line fasta to single-line fasta . . . 66
4.5.2 Useful tool: seqtk . . . 66

5 File permissions, organizing files and UNIX hygiene 67
5.1 File permissions . . . 67
5.1.1 chmod . . . 68
5.1.2 Using file permissions . . . 69
5.2 gzip and tar archives . . . 70
5.2.1 gzip . . . 71
5.2.2 gunzip . . . 71
5.2.3 tar archives . . . 71
5.3 Downloading files . . . 73
5.3.1 wget . . . 73
5.4 Symbolic file links . . . 74
5.4.1 ln -s . . . 74
5.5 UNIX hygiene . . . 76
5.5.1 Backup your data . . . 76
5.5.2 Proper organization of your files . . . 76
5.5.3 Document your analyses . . . 77
5.5.4 Name your files properly . . . 77
5.6 Exercises . . . 78
5.6.1 Download the files to your home directory . . . 78
5.6.2 chmod . . . 78
5.6.3 Symbolic links . . . 79
5.6.4 Further work (*) . . . 79
5.7 Checkpoint . . . 81
5.7.1 UNIX commands . . . 81
5.8 Further reading . . . 82
5.8.1 Backup tool: rsync . . . 82
5.8.2 Organizing projects . . . 82

6 Working with file streams **83**
6.1 What are file streams? . . . 83
6.1.1 echo . . . 84
6.2 Redirecting input and output . . . 84
6.2.1 Standard output . . . 84
6.2.2 Standard error . . . 85
6.2.3 Standard input . . . 86
6.3 The pipe . . . 86
6.4 Filters . . . 87
6.4.1 tr . . . 87
6.4.2 sed . . . 87
6.5 Exercises . . . 90
6.5.1 Extracting information from GFF . . . 90
6.5.2 Stream editing . . . 90
6.5.3 Useful pipes . . . 92
6.5.4 The gene-count mystery (**) . . . 94
6.6 Checkpoint . . . 95
6.6.1 UNIX commands . . . 95
6.7 Further reading . . . 96
6.7.1 The tee command . . . 96
6.7.2 awk . . . 96

7 Pattern matching, variables, subshells and loops **97**
7.1 Pattern matching . . . 97
7.1.1 Pattern matching UNIX paths . . . 97
7.1.2 Using pattern matching with grep . . . 99
7.2 Variables . . . 101
7.2.1 What is a 'variable'? . . . 101
7.2.2 Variables in UNIX . . . 101
7.3 Subshells . . . 104
7.4 Loops . . . 104
7.4.1 Looping over a set of files . . . 105
7.5 Exercises . . . 108
7.5.1 Variables and subshells . . . 108
7.5.2 Loops and pattern matching . . . 109
7.5.3 Working with multiple files at once . . . 110
7.5.4 Processing multiple files (**) . . . 111
7.6 Checkpoint . . . 113
7.6.1 UNIX commands . . . 113
7.7 Further reading . . . 114
7.7.1 Variable expansion . . . 114
7.7.2 Regular expressions . . . 114

8 Introduction to scripting **115**
8.1 Bash and scripting . . . 115
8.2 Building a simple Bash script . . . 115
8.2.1 Comments in Bash scripts . . . 116
8.3 Gathering processing steps in a script . . . 117
8.4 Providing input to a script . . . 118
8.5 Exercises . . . 121
8.5.1 Hello, world! . . . 121
8.5.2 Retrieving information from a FASTA . . . 122
8.5.3 Building a useful script on your own (*) . . . 122
8.6 Checkpoint . . . 123
8.6.1 UNIX commands . . . 123
8.7 Further reading . . . 124
8.7.1 The PATH variable . . . 124
8.7.2 Further learning materials . . . 124

Chapter 1

Introduction to the course

1.1 Acknowledgements

This course material has been written for the course "Introduction to UNIX for Biologists" given by the National Bioinformatics Infrastructure in Sweden in collaboration with the organizations PlantLink and Geneco.

Dag Ahrén has contributed with the exercises for the chapter on file permissions and UNIX hygiene.

Thank you Ellen Sunström, Jonatan Leo, Johan Philipsson and Dag Ahrén for providing comments, thoughts and proof-reading of the material. You have made it better!

1.2 Welcome

Welcome to this introduction to using the UNIX terminal. The UNIX terminal is a powerful tool, and it can be especially useful when working with raw text - which commonly is the case when working with biological data. This material has the aim to:

- Provide an insight into what is possible with the UNIX terminal.
- Give enough hands-on practice to make you feel comfortable working with biological data in UNIX.
- Provide a foundation for further study of more advanced UNIX-based bioinformatic tools.

The UNIX terminal is found on many computers. It is tightly integrated with the Linux operating systems, and it is readily available on OSX (Mac) - based systems.

Recently even Microsoft has started natively supporting the UNIX terminal as a developer tools. If you are using the latest version of Windows 10 you can activate this feature by following the steps outlined in the link:
www.howtogeek.com/249966/how-to-install-and-use-the-linux-bash-shell-on-windows-10/

Large-scale computational clusters (like UPPMAX) are commonly based on Linux servers and accessed through the UNIX terminal, making it an invaluable tool for researchers performing any type of data processing.

Many important bioinformatic tools are designed to be used in the UNIX terminal as the terminal provides a flexibility which is hard to reproduce in a graphical user interface. It is a central tool for the field, and it seems like that isn't going to change any time soon.

1.3 The structure of this material

1.3.1 Chapters

This material is divided into eight chapters.

1 - Introduction to the course Overview of this course, how to get help, and how to connect to other computers.

2 - Introduction to the file system Understand and navigate the UNIX file system.

3 - Working with files in UNIX Move, copy, read and edit files and compressed files.

4 - Working with file content Bioinformatic file formats, and how to extract information from them.

5 - File permissions, organizing files and UNIX hygiene Tools and best practices for managing files and projects in a UNIX environment.

6 - Working with file streams Reading and writing files, and combining commands.

7 - Pattern matching, variables, subshells and loops Automate your analysis and get the computer to do more of the work.

8 - Introduction to Scripting Make your analyses reproducible by allowing later re-processing, and develop your own tools.

We will initially go through important UNIX fundamentals (chapters 1, 2 and 3). Then, we will introduce tools for and ways of processing biological data (chapter 4, 5 and 6). Finally, we will introduce some UNIX concepts used to automate your analysis which reduces the effort needed to process your data while increasing reproducibility (chapter 7 and 8).

1.3.2 Chapter structure

Each chapter contains the following parts:

- Presentation of concepts and commands, demonstrated by examples.
- Exercises. More challenging exercises are marked with asterisks (*) and can be skipped if in a hurry. They provide deeper learning for those who have the time.
- A recap page for evaluating your understanding of the introduced concepts and commands.
- Further reading material for the interested reader. This reading material extends outside the scope of this course.

Introducing commands

An example of how a command could be introduced is shown in figure 1.1. The syntax used here is:

- `<input_file>` Angular brackets means that the input is required for the option or the command.
- `-l` Required input arguments (flags) are shown without brackets.
- `[-l]` Optional flags or inputs are shown in square brackets. When running the command, type the flag without the square brackets.

Optional arguments are not required to run the command, but can often change the behaviour of the command in useful ways.

> ***ls****t directory contents* Command usage: `ls [-l] [-lh] [<directory_to_list>]`
>
> `ls` lists the files in a directory. If no directory is provided, lists the files in the present working directory. If you want more extensive information about the files, you can add the `-l` **flag** to the command. If you want the file sizes in a more readable format, you can add the `-h` flag to the `-l` flag.

Figure 1.1: Example command description.

Each command is introduced together with the following information:

- Mame of the command.
- Meaning of the command (cd - ***c****hange* ***d****irectory*).
- Usage examples for the command.

Colored boxes

You will encounter red, green and blue boxes in the text. The purpose of those are:

red Important note or advice.

green Example file used to demonstrate commands.

blue Technical information about command options, file formats or similar.

1.4 How to find help

When encountering problems with the UNIX commands, you can usually found more information in the following ways:

- Use the `man` command on your command to learn more about its meaning and usage (this is shown later in this chapter).
- Use a search engine (like Google) to find discussions about similar problems. Good search queries includes the command, specifics about the issue (example: part of the error message) and potentially the word "unix" or "linux" if your search matches unrelated topics.
- Ask someone with more experience about the problem. In many cases they have encountered and solved the same problem.

Make sure to do a bit of research using the man-pages and a search engine before handing over the question to someone else. This helps training debugging skills, provides a deeper insight into the system and lets you better understand the problem when you receive the solution.

If your problem is related to processing biological information, there are some web sites dedicated to help:

Seqanswers.com Large traditional forum with an extensive number of existing help threads, as well as a lively community.

BioStars.org More direct question / answers site which is used extensively for bioinformatic problems.

BioSupport.se Site run by NBIS which makes sure that you'll get a response and that it comes from an expert in the field.

1.4.1 The man command

*an interface to the online reference **man**uals*
Command usage: `man <target_command>`

The `man` command retrieves information about the target command that you are interested in. When running it, it opens up the manual within the terminal. The manual pages both contains descriptions about the commands, and information about the options that are available for the command.

In the gray box below you see an example of running the `man` command in a terminal prompt. This is seen in the line starting with the `$`. The line starting with a `#` is not part of the command or the output. It is a comment describing what is going on for you, dear reader.


```
$ man ls
# This opens up a manual page for the ls command
```

Note

The man-pages is opened in the text reader `less`. You will learn more about `less` in chapter 3.

- You can exit the man-pages by pressing "q".
- You can search the man-pages by typing "/", followed by the text you want to search for and then return. You can then use "n" and "N" to jump forward and backwards between matches.

1.5 Accessing data

We are using the SSH network protocol (a network protocol is way of communicating between computers) to connect to the server computer during this course.

If working on a Linux- or OSX-based machine, you can connect to the server computer using the `ssh` command. If you are working on a Windows-based machine and haven't activated the Bash feature on Windows 10, you will need to download an SSH-client for Windows like MobaXTerm (`mobaxterm.mobatek.net`) or PuTTy (`putty.org`).

1.5.1 ssh

*OpenSSH **SSH** client (remote login program)*
Command usage: `ssh [-v] <user>@<remote_address>`

The `ssh` command uses the SSH protocol to connect to a remote computer. The *remote* is a computer you are logging in to from your current computer. The address is built from your user name and the host address separated by a `@` (username@hostaddress).

If someone with the username "jakob" wants to connect to a particular computer with the IP address `130.232.46.00` (IP addresses - *Internet Protocol addresses* are used by computers to find each other over the internet), he could do the following:

```
$ ssh jakob@130.232.46.00
jakob@130.232.46.00's password:
Login successful, welcome to the computer!
[jakob@130.232.46.00]$
# We are now logged in to the remote computer

[jakob@130.232.46.00]$ exit
logout
Connection to 130.232.46.00 closed.
$
# We are now back on our local computer
```

When entering passwords in UNIX nothing shows up in the terminal. The letters are still registered by the computer. Just type the password and then hit "Enter". You can close a connection to a remote computer by typing "exit".

Note

Some information about the code examples:

- Text coming after #-signs in the code examples are comments. This is a common way of showing that what comes after the sign will not be run by the computer.
- Lines starting with `$` shows the commands we run in the examples. In this example we ran the command: `ssh jakob@130.232.46.00`.
- Other lines shows the resulting output received from the command.
- Commands are colored in blue, comments in red.

If you have trouble connecting using `ssh` you can try adding the `-v` flag (*v* for *verbose*) which provides detailed information about what the `ssh` command is doing. In this example, the command would have been: `ssh -v jakob@130.232.46.00`

1.6 Exercises

The goal with the exercises for this chapter is to get your connection to the remote computer up and running, and for you to get acquainted with how to find help.

The exercises with asterisks (*) are more challenging, and can be skipped if you are in a hurry.

A useful hint when working in the terminal:

- You can go back to and edit commands which you previously have been running by using the up / down arrows in the terminal.

1.6.1 Connect to a remote computer

1. Run the SSH command from your computer to connect to the remote computer. If you are using a Windows machine, this will require you to download MobaXTerm (mobaxterm.mobatek.net) or PuTTy (putty.org). Use the `ssh` command to connect to the server. You need to have an account on a Linux-server to do this step.

   In the examples below 'username' should be your user on the server, and 'host_ip' either the server's IP address (similar to "130.235.00.00") or the server's domain (similar to a web address, not always available).

   ```
   $ ssh username@host_ip
   ```

   This will prompt you for a password. No asterisks show up when you enter the password - type it out and press enter. After logging in, you are able to close the connection by running:

   ```
   $ exit
   ```

2. * Try running the SSH command with the `-v` *flag argument*. This argument often provides valuable information if you have trouble logging in.

   ```
   $ ssh -v username@host_ip
   ```

1.6.2 Explore the man-pages

Hint: You exit the man-pages by pressing "q". You can search the man pages by typing "/" followed by the search word. Then "n" or "N" can be used to get next or previous hit.

1. Try using the `man` command on the `man` command and the `ssh` command.

```
$ man ssh
```

1.6.3 Finding help on the internet (*)

You will frequently come to a situation where you wonder if something is possible, but where you don't know exactly how to do it. Sometimes the answer will be a particular option in the man-pages. Other times, you will have to search a bit more.

Say for example that you want to know how to copy files between two Linux computers. There are many ways to approach this. Perhaps the best place to start is to go straight for Google and type a search query similar to the following:

```
how to copy files between linux computers
```

Check the first few hits. Did you find any good ways? There will definitely be some good approaches among them. A habit of searching online for answers and solutions is a great asset for anyone working with computers.

1.7 Checkpoint

Before you continue, make sure that you can answer the following:

- How will you approach future problems and errors in UNIX?
- How do you connect to a remote computer from your laptop?

1.7.1 UNIX commands

Consider each command for a second. Make sure that you have an idea about what they do before moving on. We will build a collection of commands in the coming chapters.

Chapter 1 - Introduction to UNIX

```
man
ssh [-v]
```

1.8 Further reading

This material reaches outside the scope of this course. If you are interested and have the time this material might give you useful tools and insights. If not, save it for another time.

1.8.1 Make command run even if you close your terminal

When working in UNIX, you sometimes want your command to stay running even after you have left the computer and shut down the terminal. This can be achieved by: using the `nohop` command, together ending the command with the `&` - sign. This allows you to close the connection to a server without shutting down the commands that you are running.

```
$ nohup yourcommand &
```

The command shown above tells UNIX to both continue running your command after you closed your terminal (`nohup`) and that the terminal should let go of the command so that you can continue running other commands (`&`). Bioinformatic programs can often run for hours or days. In these cases, `nohup` definitely comes in handy.

http://www.computerhope.com/unix/unohup.htm

Chapter 2

Introduction to the File System

2.1 The UNIX file system

When navigating the UNIX system, you always have a **present working directory**. This is the directory that you currently are looking into. This can be compared to having a certain folder open in for example Windows or Mac.

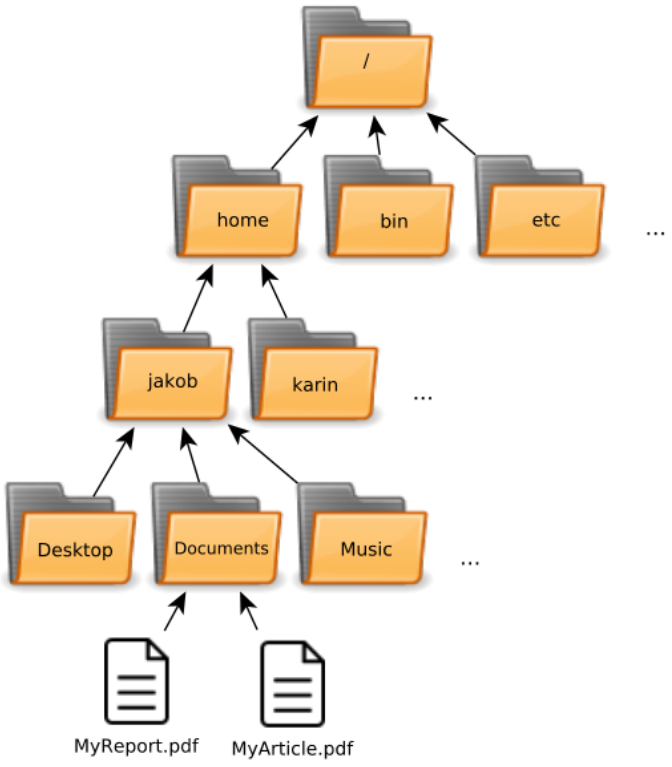

Figure 2.1: Subset of UNIX file system

All the files and directories in UNIX are organized in a tree-like structure. At the root of the tree, there is the **root**-directory (shown as "/" in the figures). The root directory is

the only directory which doesn't have a parent directory. All other directories have a parent which is shown with arrows in the figures. Each directory can contain an arbitrary number of files and other directories. Each file and directory is located within a directory (except the root directory).

2.1.1 A brief primer on paths

You navigate and interact with files using **paths**. A path is a way of specifying the location of a file or directory.

A path to a file in the file system shown in figure 2.1:

```
/home/jakob/Documents/MyReport.pdf
```

In this case, we started out in the root, traversed the `home`, `jakob` and `Documents` directories and finally found the file we were looking for. Take a look at the figure and make sure that you can trace the path from the root to `MyReport.pdf`

A path to a directory:

```
/home/jakob/Documents
```

In this case we traversed the `home` and `jakob` directories before stopping at our desired directory.

A path can traverse zero or more directories, and ends in either a directory or a file. Each directory/file is separated by a slash. The paths can either start in the root (as in the examples above) or in the present working directory.

We will dig further into different types of paths and how to use them in section 2.3 after introducing some file system commands.

2.2 Important file system commands

This is a core set of commands used to investigate and navigate the file system. We will be investigating the file system shown in figure 2.1. Make sure to understand the commands, you will be using them a lot.

2.2.1 pwd

***p**rint name of current/**w**orking **d**irectory*
Command usage: `pwd`

`pwd` prints your current position in the file tree.

Example:

```
$ pwd
/home/jakob
```

Note

The `pwd` command only shows your current location. It doesn't change the working directory. It is only used to keep track of where you currently are.

2.2.2 ls

***ls**t directory contents*
Command usage: `ls [-l], [-lh] [<directory_to_list>]`

`ls` lists the files in directory. If no directory is specified, it lists the files in the present working directory. If you want more extensive information about the file permissions, ownership, size and age, you can add the `-l` flag to the command. If you want the file sizes in a more readable format, you can add the `-h` flag to the `-l` **flag**.

Note

A **flag** argument is an extra option which can be provided to a command, telling the command to perform a specific task. Flag arguments are added after the command and (for most UNIX commands) consists of a dash and a letter. In the command `ls -l` we run the command `ls` and gives it the flag argument `-l`.

In the gray box below you see some usage examples for the `ls` command. The lines starting with `$` contain the commands. The following lines are the responses from the program. Lines starting with `#` are not run - They only contain descriptive information.

```
$ ls
Desktop Documents   Music

$ ls Documents
MyArticle.pdf   MyReport.pdf

$ ls -l Documents
-rw-r----- 1 user user 687438  sep 16 19:02 MyArticle.pdf
-rw-r----- 1 user user 2481860 may 13 14:05 MyReport.pdf

$ ls -lh Documents
-rw-r----- 1 user user 672K sep 16 19:02 MyArticle.pdf
-rw-r----- 1 user user 2.4M may 13 14:05 MyReport.pdf
# Note that the file sizes are written differently here
```

The output from `ls -l` contains information about:

1. File permissions

Note

The l in the `-l` flag in the `ls -l` command is a lower case instance of the letter L, not the digit one (`1`). The font used in the code examples caused some confusion here. Keep this in mind for coming chapters too.

2. Number of files in directories (1 for files)
3. To which user and group the file belongs
4. The file size
5. The last time the file was changed
6. The name of the file

We will revisit file permissions in a later chapter.

Wild-card matching

Another important concept is wild card matching using the `*` symbol. Many UNIX commands interpret this symbol as one or more characters of any kind (except line breaks). This can be used with `ls` to list a subset of files. In the following example, `D*` matches `Desktop` and `Documents`, but not `Music` which doesn't match the initial 'D'.

```
$ ls D*
Desktop Documents
```

There are other patterns available for more detailed control of what files you match. We will go through those in more detail in a later chapter.

2.2.3 cd

***c**hange **d**irectory*
Command usage: `cd <target_directory>`

The `cd`-command changes your present working directory. You tell it what location you want to change to by providing it the path to that location: `cd <path to location>`.

```
$ pwd
/home/jakob
$ cd Documents
$ pwd
/home/jakob/Documents
```

In this example, you started out in the home directory for the user `jakob`. After changing directory, your present working directory is `Documents`, a directory residing in the `jakob` directory. See figure 2.2 and figure 2.3 for visualization.

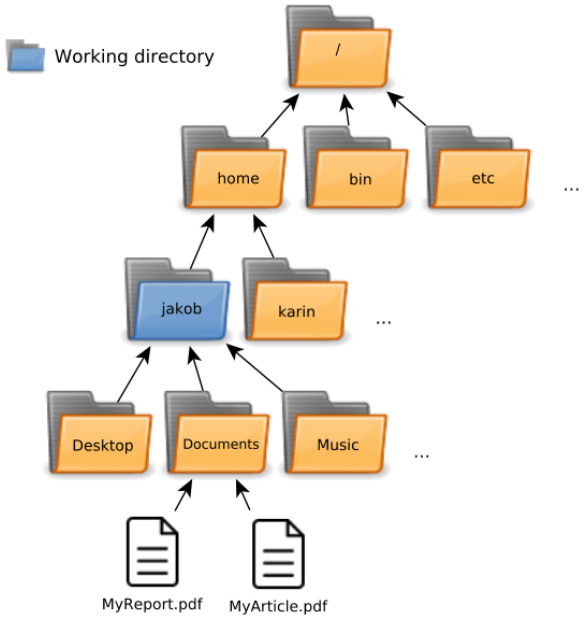

Figure 2.2: Present working directory **before** running "cd Documents"

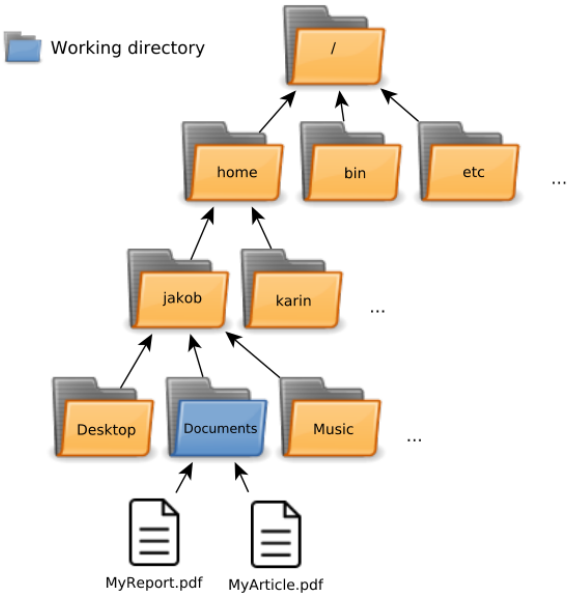

Figure 2.3: Present working directory **after** running "cd Documents"

2.2.4 Demonstration

Here, some investigation is done of the home directory. Look at figure 2.2 and 2.3, and make sure that you can follow how the command and their output relates to the figures.

```
$ pwd
/home/jakob
$ ls
Desktop          Documents        Music
$ ls Documents
MyReport.pdf     MyArticle.pdf
$ cd Documents
$ pwd
/home/jakob/Documents
$ ls
MyReport.pdf     MyArticle.pdf
```

2.3 Understanding paths

When navigating the UNIX file system, you always have a present working directory which you find by running `pwd`. You are able to reach other files on the computer in two ways.

- Using its `absolute path`, the exact location in the computer - starting from the root directory

- Using the `relative path` between the file and your present working directory

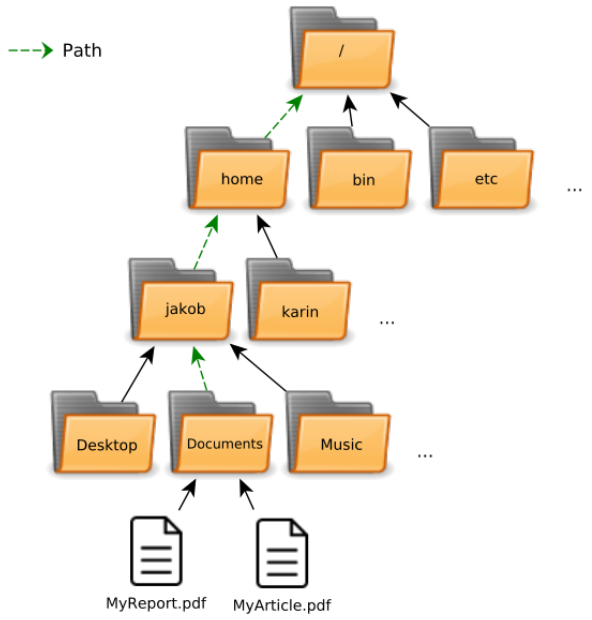

Figure 2.4: Accessing "Documents" by its **absolute path**

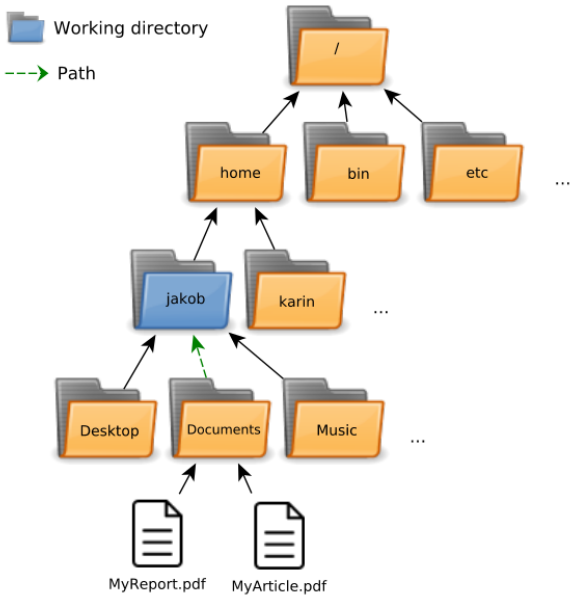

Figure 2.5: Accessing "Documents" from /home/jakob by its **relative path**

In the example below, the documents directory is first listed using its absolute path. Then, with the present working directory as `/home/jakob`, it is listed through its relative path. This is illustrated in figure 2.4 and figure 2.5.

```
# Listing Documents using its absolute path
$ ls /home/jakob/Documents
MyReport.pdf    MyArticle.pdf

# Listing Documents using its relative path from /home/jakob
$ pwd
/home/jakob
$ ls Documents
MyReport.pdf    MyArticle.pdf
```

You can always reach a directory either using its absolute path or the relative path from your present working directory. Which is most convenient depends on the situation.

Paths can traverse the file tree in both directions, which can be especially useful for relative paths. You will learn more about this later in this chapter.

2.4 Use tab completion

Using the terminal means that you will do a lot of typing. Often, you will be typing long file names and long paths to files. There is a feature in UNIX called **tab completion** which

Note

Remember - The arrows in the figures shows the parent folder for each folder or file. Paths usually go the other direction, from parent to child.

helps you do this more efficiently and with less typing errors, This allows you to write part of the file names and then let the computer expand it for you by pressing the tab-key.

You use tab completion by single- and double-tapping the tab key.

- After a **single tap**, the computer will attempt to figure out the entire file name which you have started writing. If there only is one possible match, it will write out the entire file name. If not, it will write out the part of the file name that is common for the possible matches.
- After a **double tap**, the computer will write out all possible files matching the letters you have already written. Often, you can follow up by typing a single unique letter before pressing the tab key again, expanding the rest of the file name.

```
$ pwd
/home/jakob
$ ls
Desktop Documents   Music

# Typing an M followed by tab will expand the word to Music,
# as it is the only file/directory starting with "M"
$ M <tab>
$ Music

# If there are multiple options, nothing will happen on the
# first press
$ D <tab>
$ D

# On the second press, all alternatives will be listed
$ D <tab><tab>
Desktop Documents

# This tells you what to write next, and you can complete
# the tab completion
$ De <tab>
$ Desktop
```

You will likely want to tab complete most of the files and commands you are typing. This will save you a lot of time (and sanity). So, make sure to tab complete. Tell all your friends to do it too.

Once more. **Use tab completion**.

2.5 Special directories

In the UNIX system, there are some directories that are special. Those directories have special ways to access them.

2.5.1 The root directory (/)

We have already discussed the root directory. It is represented by a forward slash (/). This is the highest parental directory for every directory and file in the computer. We have shown how it can be used to access any file based on the file's relation to the root.

```
$ pwd
/home/jakob
$ ls /home
jakob   karin
```

Here, we investigated the content of the `home` directory, using its absolute path.

2.5.2 The home directory (~)

Each user on a UNIX system has its own home directory. It is located at `/home/<username>`, where `<username>` is the username of the current user. In the previous examples, the user name has been `jakob`. When logging into UNIX using SSH, you will always start out in your home directory.

The home directory also has its own sign - tilde (~). If you are unsure about how to type this sign on your computer, check figure 2.6.

On a Swedish PC keyboard, tilde can be typed by pressing the following:

- Press AltGr + The key left of the return button (you can see a small tilde on it, together with a hat, and two dots)
- Press space

On a Swedish Mac keyboard, tilde is typed by pressing Alt + ^

Figure 2.6: Typing the character tilde

You can access the home directory similarly to how we previously accessed files through their absolute paths. You can think of it as a shorter way of writing out the path to your home directory: `/home/jakob`. The following three `ls` commands all access the same directory.

```
$ pwd
/home/jakob

# Using a relative path
$ ls Documents
MyReport.pdf    MyArticle.pdf

# Using the absolute path
$ ls /home/jakob/Documents
MyReport.pdf    MyArticle.pdf

# Starting from home path
$ ls ~/Documents
MyReport.pdf    MyArticle.pdf
```

2.5.3 Current directory (.)

Each directory also can refer to itself using the single-dot notation (.) This allows you to specify the working directory as input argument to commands. When running the `ls` command without input arguments, it per default refers to the current directory.

```
# If we provide . to ls, we list our current directory
$ ls
Desktop Documents   Music
$ ls .
Desktop Documents   Music

# If we provide . to cd, we change to our current directory
$ pwd
/home/jakob
$ cd .
$ pwd
/home/jakob
```

We will revisit this notation in more a useful context in the chapter about writing scripts.

2.5.4 Parent directory (..)

Each directory (except the root) has a parent directory. This directory can be accessed by typing two dots: `..` The ..-notation lets you access the parent directory to the present working directory.

```
$ pwd
/home/jakob
$ ls ..
jakob   karin
$ cd ..
$ pwd
/home
```

The parent directory to the home directory (in this case `jakob`) is the directory `home`. We can change directory to the parent directory by the command `cd ..` We can list the files in the current parent directory by running `ls ..` We can access other directories with the same parent by starting the path with `..`, before navigating down a parallel directory. Take a look at figure 2.7 and make sure that you can follow the example.

```
$ pwd
/home/jakob
$ cd ../karin
$ pwd
/home/karin
$ ls ../jakob/Documents
MyReport.pdf    MyArticle.pdf
```

Here, we first changed directory to the home directory of the user `karin`, before we used a relative path to list the Documents content. This is visualized in figure 2.7

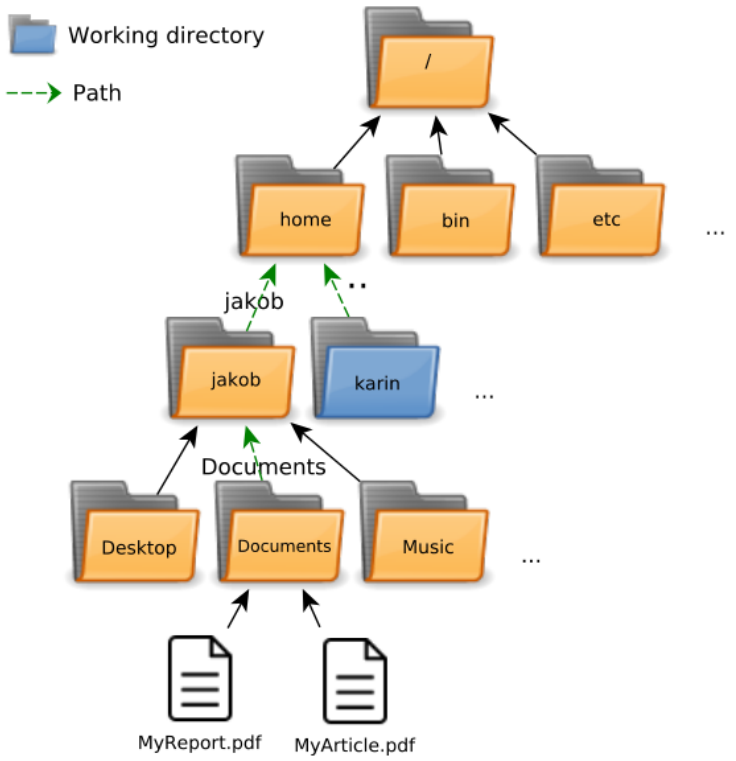

Figure 2.7: Accessing Documents by relative path, passing through parent

2.6 Exercises

The purpose of this exercise is to make you comfortable with navigating between directories and files in the terminal. You will also learn how to retrieve and unpack the exercise files from a server computer.

2.6.1 Download the files to your home directory

The files used for the exercises are stored in *tar archives* on a Linux server. We will go through the commands used here in more depth in a later chapter. For now, just follow the steps to get the files (if you are very curious, take a look in the man pages!).

We will use the `wget` command to download the FASTQ file from the Linux server on which they are stored. Run the following command from your home directory:

```
$ wget http://130.235.244.56/unix/tarballs/unix_course_chapter2.tar.gz
```

Be aware that if you copy this path directly from the PDF you might end up with additional spaces within the path. The command will not work unless you remove those spaces.

The path to the exercise can also be found by going to the web page 130.235.244.56/unix, right-clicking on "Exercise files chapter 2" and choosing "Copy link address". This can then be pasted into the terminal by right-clicking and choosing "paste".

A copy of the tar archive should now be located in your current directory. Check by using `ls`:

```
$ ls
```

In order to extract the directory from the tar archive, we use the `tar` command.

```
tar -xf unix_course_chapter2.tar.gz
```

Run `ls` to check that you have a directory named `unix_course_chapter2` in your home directory. This directory contains the exercise files used in this chapter.

2.6.2 Trying out the file system commands

There are three central commands introduced in this chapter: `pwd`, `ls` and `cd`. Let's get acquainted with them.

1. Run the command `pwd`. This is your present working directory. You can run this command at any time to keep track of where you are.

2. Now run the command `ls` in the same directory. This will show you what files and folders there are in your directory.

3. Next, let's change directory to the exercise directory using `cd`.

   ```
   $ cd unix_course_chapter2
   ```

   After changing directory, check what your current working directory is, and what files there are in this directory.

4. You can always go back by using `cd ..` This moves you up one step to the parent directory. Go back up one step.

5. You can list the content of the exercise directory while your working directory still is your home directory by running the following:

   ```
   $ ls unix_course_chapter2
   ```

Experiment with these commands until you feel comfortable using them. You will be using them frequently during this entire course. Start practicing now.

2.6.3 Investigating file system

Inside the directory containing the exercises, there is a directory called `file_tree`. This directory contains a number of directories and files. Your job is to locate the file `print_this_with_cat.txt`, and print its content. Its content can be printed using the `cat` command (when your present working directory is the same as the file):

```
cat print_this_with_cat.txt
```

This command will be introduced further in the next chapter. Also, don't forget to use tab completion.

1. First, navigate down the directories one by one. Keep track of your working directory with `pwd`, list the available directories with `ls` and change directory to the correct one using `cd` The names of the directories guides you to the correct directories. Are you able to find the file?

2. After finding the file, back up to the starting directory. Can you do this with a single command?

3. Now, let's try this again this without using `pwd`, `ls` or `cd`. Start by typing `cat` followed by a space, and then see if you can type out the entire path to the file with the help of tab completion.

4. * We used relative paths in exercise 1 and 3 to reach the file. Now, let's print it using its *absolute path*. Use either the root (`/`) or you home directory (`~`) as base. Can you see when this approach could be more useful?

2.7 Checkpoint

Before you continue, go through the following questions (even better, grab a classmate and discuss):

- What are the differences between absolute and relative paths?
- What is the root directory and what is the home directory?
- How can you change directory to or list a parent directory?
- What can the tilde sign (∼) be used for?

2.7.1 UNIX commands

Consider each command for a second. Make sure that you have an idea about what they do before moving on.

Chapter 1 - Introduction to UNIX

```
man
ssh [-v]
```

Chapter 2 - Introduction to the file system

```
cd
ls [-l] [-lh]
pwd
```

2.8 Further reading

Note that this material is outside the scope of this course. Only do it if you have the time and the interest.

2.8.1 Explore the system directories

Lets take a look into what resides beneath your home directory. Your home directory is yours, and contains your files. There is a lot more things going on in a UNIX computer, which can be found in the directories directly beneath the root. As a regular user you are in most cases not able to edit the system files, but you can in many cases explore and read configuration files and log files. This is often useful for trouble shooting, especially if you are running a UNIX system on your own computer.

1. * Change directory to the root. You can do this either by going up two steps from your home directory (relative path) or change directly to the root by typing `cd /` (absolute path).
2. * Change directory to `/var/log`. Here is the default location for so called log files, which contains status information from different programs running on the computer.
3. * List the content of `/home`. Your entire home directory is located on a branch of the entire file tree as one of the directories in `home`.
4. * Now change directory to `/bin`. Do you recognize any of the files here?

The following link gives an overview of the purpose of the many system directories.

http://www.thegeekstuff.com/2010/09/linux-file-system-structure/

2.8.2 Hidden files

Something which we haven't been discussing in this chapter is *hidden files*. In UNIX, files for which the name starts with a dot (`.`) are hidden files. To list all files including the hidden ones, run the `ls` command with its `-a` flag (***a**ll*). If you do this in your home directory, you will see that there is a lot more files present there compared to how it appears when running the regular `ls`.

Those hidden files commonly contains configuration information specific for your user. Some of them (`.bashrc` and `.bash_profile`) also contains commands which are run when logging in to your user or starting a new terminal. You can add commands here, and customize the behaviour of your terminal. See the following link for some discussion on useful changes to those two files. You could for example use *aliases* to add custom commands (for example: `alias ..="cd .."` would allow you to change directory to the parent directory by simply typing "`..`"). But, be a bit careful here, and only add commands which you know what they do.

https://www.quora.com/What-are-some-useful-bash_profile-and-bashrc-tips

Chapter 3

Working with files in UNIX

3.1 Files and file formats in UNIX

In Linux, everything is a file. We are mainly working with text files and directories in this course, but you will also encounter *binary* files and *compressed* files.

3.1.1 Regular text files

Regular text files only contain text characters. This is a common way of storing bioinformatic data. The files can be opened with a text editor and the content can be seen and understood directly, without any use of other tools.

3.1.2 Binary files

Binary files are files written in "binary code" - the computer's language. When you are typing UNIX commands into the terminal, you are telling the computer to run a particular binary file. If you accidentally open a binary file with a tool designed for plain text, you get output similar to what is shown in figure 3.1.

Figure 3.1: Inspecting the ls binary

If you encounter similar output when investigating files, you have likely encountered a

non human-readable file. This could be a binary file, a compressed file, an image file or something else.

3.1.3 Compressed files

Another frequently used format in UNIX is compressed files. This is a common way of storing bioinformatic data. Bioinformatic data can easily use tens or hundreds of gigabytes of hard drive space, which makes it essential to store the data in compressed format.

Many of the current bioinformatic tools are able to read and output compressed formats directly, removing the need for ever storing decompressed files on the hard drive.

On UNIX, two common way of storing compressed data is as *Gzipped files* and as *tar-archives*.

3.2 File commands

We will now go through some of the central file commands used to move, copy, read or remove.

3.2.1 mv

mov*e (rename) file*
Command usage: `mv (-i) <target_file> <end_location>`

The command `mv` moves a desired file to a target location. This command can be used for two different purposes:

- Move a file to a different location
- Rename a file

You can both move and rename a file in one command. If the file is directed to a directory, it will be moved into that directory retaining its initial name. If you write out a new file name for the end location, the file will gain that new name.

```
$ mv my_file.txt my_renamed_file.txt
```

See figure 3.2.

```
$ mv my_file.txt Docs/
```

See figure 3.3.

```
$ mv my_file.txt Docs/my_renamed_file.txt
```

See figure 3.4.

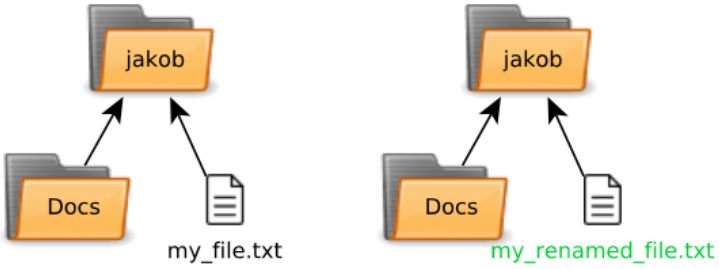

Figure 3.2: File tree before and after renaming my_file.txt

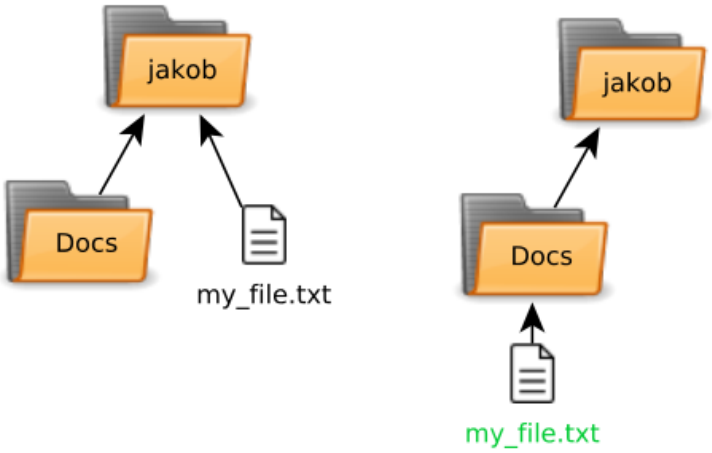

Figure 3.3: File tree before and after moving my_file.txt to the Docs directory

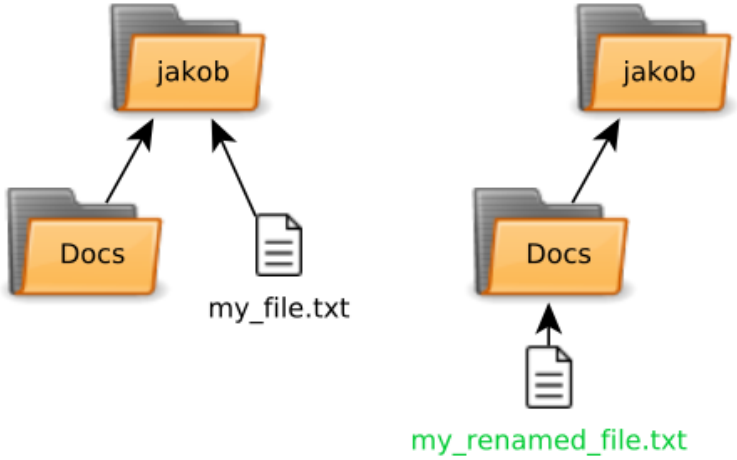

Figure 3.4: File tree before and after moving my_file.txt to Docs/my_renamed_file.txt

One useful flag argument when moving, copying or removing files is the `-i` (interactive) flag. If this flag is used, you will be asked if you want to remove files before they are permanently erased or overwritten.

In the following example, we have another file called `another_file.txt` which we move

Note

The UNIX terminal lacks many of the safety mechanisms we are used to when using GUI-based interfaces for Windows, Mac or Linux. Notably, when running commands from the terminal, files are *not* stored in a trash bin when removed. You are generally not given any warning that you are about to overwrite or remove valuable data.

into the file `my_renamed_file.txt` already residing in the directory.

```
$ mv -i another_file.txt Docs/my_renamed_file.txt
mv: overwrite 'Docs/my_renamed_file.txt'?
# You can reply to this by typing 'y' or 'n', and then return

$ mv another_file.txt Docs/my_renamed_file.txt
# Note: No warning here - The file gets overwritten
```

When prompted whether you want to overwrite the file, you need to type 'y' or 'yes' before pressing return to proceed (if you want to overwrite the file).

3.2.2 cp

***cop**y files and directories*
Command usage: `cp [-i] <target_file> <end_location>`

The `cp` command works similarly to the `mv` command, but retains the `target_file` copy of the file. If you specify a directory as `end_location`, a file with the file name of `target_file` will be created in the directory. If an entire file path is given as argument, the copy will get the new file name.

Similarly to `mv`, the `cp` command quietly overwrites existing files if not the `-i` flag is provided.

```
# This creates a new copy of "my_file.txt" in Docs
$ cp my_file.txt Docs/
```

See illustration in figure 3.5.

```
$ cp my_file.txt Docs/another_file.txt
```

See illustration in figure 3.6.

```
$ cp my_file.txt -i Docs/another_file.txt
cp: overwrite 'Docs/another_file.txt'?
```

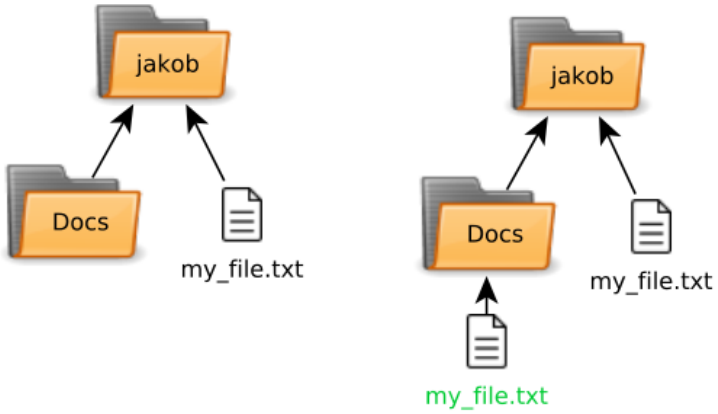

Figure 3.5: File tree before and after copying my_file.txt to the Docs directory

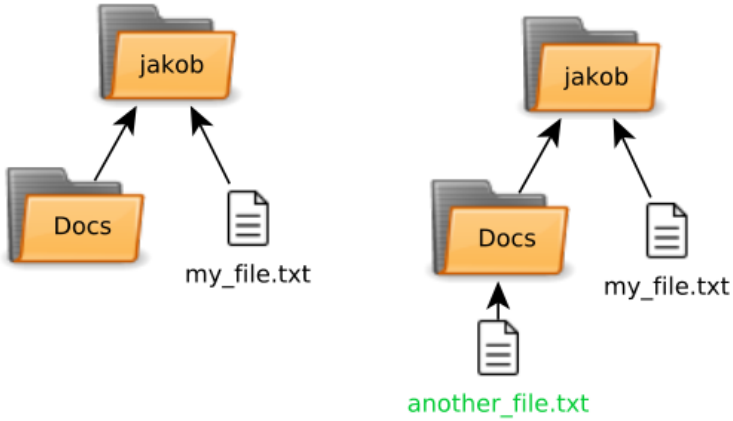

Figure 3.6: File tree before and after copying my_file.txt to Docs/another_file.txt

3.2.3 rm

***rem**ove files or directories*
Command usage: `rm [-i] <target_file>`

`rm` removes files from the file system. Be very careful when using the `rm` command, as UNIX gives no warning before you remove highly important files (for example, your valuable raw data from your sequencing experiments). To get some more protection, you can use the `-i` flag to prompt yes or no for each file. Another way of providing some protection is to properly set the file permissions for the files. This will be discussed in a later chapter.

3.3 Folder commands

Up until now we have worked with single files. You are familiar with directories (also called folders) from the operating system you commonly use. Folders shouldn't be underestimated (or underutilized). They serve an important purpose being the main tool for organizing your

files.

3.3.1 mkdir

***mk**e **dir**ectory*
Command usage: `mkdir <folder_name>`

`mkdir` simply creates a new empty folder.

```
$ mkdir my_directory
```

3.3.2 rmdir

***rm**ove **dir**ectory*
Command usage: `rmdir <folder_name>`

`rmdir` lets you remove an empty folder. It is one of the few 'safe' UNIX commands, as it will refuse to remove any directories containing other files.

```
$ rmdir my_directory

$ rmdir Docs
rmdir: failed to remove 'Docs': Directory not empty
```

3.3.3 rm -r

***rm**ove file or directory **r**ecursively*
Command usage: `rm -r <folder_name>`

This is likely the most dangerous command you will learn during this course. If you use this command unwisely, you are able to remove some or all of your analyses forcing you to re-load it from the latest backup (if you don't have a backup of your UNIX files, you are living dangerously). If you have administration rights on the computer, you can even remove some or all of the system files, crippling your system, destroying your data, and forcing you to perform a full re-installation.

The `rm -r` command is used to remove a directory and all contained directories and files. It can also be used with the `-i` flag to ask you for removal of each single file. This might not always be feasible if you have many files. Take care and think first before running `rm -r`.

```
$ rm -r Docs
```

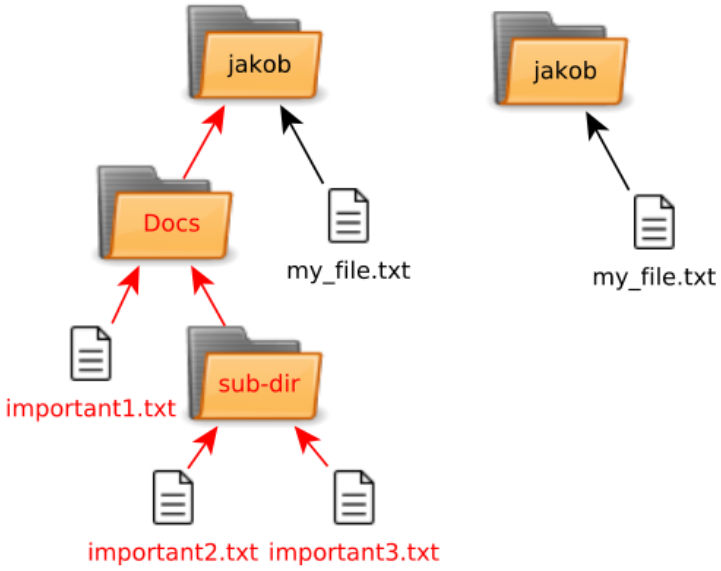

Figure 3.7: File tree before and after recursively removing the Docs directory (removed files are marked red)

3.4 Looking inside files

Up until now we have discussed how to handle files and directories, looking at them from the outside. Now, we will learn some commands used to investigate the content of the files.

In this section, we will work with two small files `my_file.txt` and `my_other_file.txt`. The content of the files are shown in figure 3.8 and figure 3.9.

my_file.txt

```
This is the first line of my_file.txt
This is the second line
This is the last line
```

Figure 3.8: The content of the file "my_file.txt"

my_other_file.txt

```
This line comes from my_other_file.txt
This is the last line of my_other_file.txt
```

Figure 3.9: The content of the file "my_other_file.txt"

Note

If you accidentally run `cat` for a very large file (example: 10 gigabytes of sequencing data) it will flood your terminal with its output. You can abort `cat` (and any other commands) by pressing the key combination `<Control> + <C>`.

3.4.1 cat

*con**cat**enate files and print on the standard output*
Command usage: `cat <target_file(s)>`

The `cat` command allows you to take a look inside files by printing the entire file to the terminal. It also allows you to concatenate (add together) multiple files.

```
$ cat my_file.txt
This is the first line of my_file.txt
This is the second line
This is the last line

$ cat my_file.txt my_other_file.txt
This is the first line of my_file.txt
This is the second line
This is the last line
This line comes from my_other_file.txt
This is the last line of my_other_file.txt
```

3.4.2 head

*output the **head**-part (the first part) of the file*
Command usage: `head [-number] <target_file>`

Often, you are not interested in all the file content, but only want to check the first part of the file to get an idea of its content. The `head` command does exactly this, printing the first ten lines of the file to the terminal. The exact number of printed lines can be adjusted by adding a flag with the number of lines you want to print. For example: `head -20 <target_file>` will print the first 20 lines of the file.

```
$ head -2 my_file.txt
This is the first line of my_file.txt
This is the second line
```

3.4.3 tail

*output the **tail**-part (the last part) of the file*
Command usage: `tail [-number] <target_file>`

The command `tail` works similarly to `head`, but prints the end of the file instead of the top. Per default, it outputs the last ten lines. This can be adjusted in the same way as for the `head` command: `tail -20 <target_file>`. This command is especially useful when looking into files to which content is continuously added to the end. One example of this is so called log files, to which information about ongoing programs are added continuously.

```
$ tail -2 my_file.txt
This is the second line
This is the last line
```

3.4.4 less

Command usage: `less <filename>`

The command `less` can be used to open and explore a file. It can be used to quickly navigate up and down the file content, and can also be used to search for particular words in the file. Compared to `cat`, `less` lets you explore a file without flooding the terminal. We have encountered `less` before when running the `man` command, which opened information about the command in `less`.

`less` has its own set of internal commands run from within the program. Some of the most useful are shown in figure 3.10.

q Quit less and return to the terminal

/ By typing / followed by string of text and pressing return, less will search for the next instance of that word.

space Jump down a half page.

Figure 3.10: less commands

3.5 Editing text

Up until this point we have only looked into the present file content - We haven't made any changes to the files. When processing files, you generally want to avoid making manual changes which might be hard to trace and reproduce. But in some cases, you will of course need to be able to create or edit files manually.

There are a number of different editors available for editing raw text. Examples of graphical editors are Notepad (Windows), Gedit (Linux) or TextEdit (Mac).

Note

Editors like Microsoft Office's Word is strongly **not** recommended to use for raw text editing as it doesn't store its files in plain text, but instead wraps it inside its own file format. This format is not possible to work with using the tools we present here.

If you are working on a remote computer through SSH, it is often useful to be able to open the file using a purely terminal based editor. We will use the text editor `nano` here due to its simplicity, but there are many other alternatives. `vim` and `emacs` are two editors which are popular among developers. They contain many of powerful features, but takes some effort to learn to use properly.

3.5.1 nano

***n**ano's **ano**ther editor, an enhanced free Pico clone*
Command usage: `nano <file_name>`

`nano` allows editing of a text file directly, similarly to most editors. There is no menu bar though. Instead, a number of options are listed at the bottom. The `nano` interface is shown in figure 3.11. To options are used by pressing the `<Control>` button (or corresponding key on a Mac) together with the shown letter.

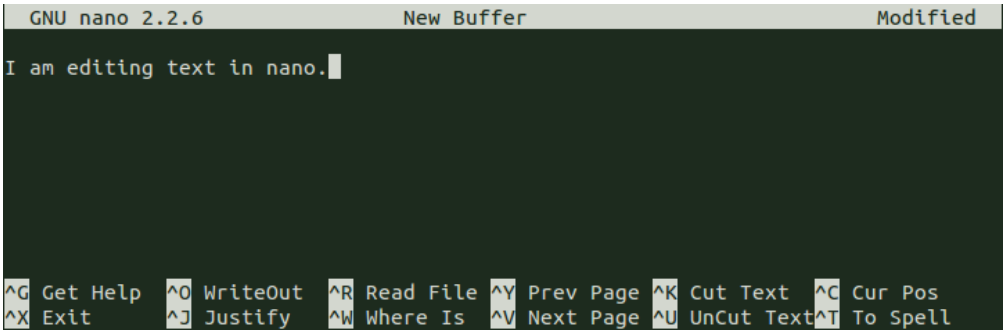

Figure 3.11: Editing a file in nano

Must-know `nano` commands are shown in figure 3.12.

Save the file `<Control> + O` (followed by Enter to confirm the filename)

Exit the file `<Control> + X`

Figure 3.12: nano commands

3.6 Exercises

The purpose with these exercises is to give you experience with the basic operations for moving, copying and removing files and folders, and with commands used to investigate text files.

3.6.1 Make and manage your own file

First, let's create a file from scratch, and try out the different operations on it. The purpose here is just to run through the commands. You are encouraged to experiment with the different commands - The goal is for you to feel comfortable using them.

1. Create a new directory for this exercise, and change working directory to it.

   ```
   $ mkdir exercise3
   $ cd exercise3
   ```

   Check your present working directory to see that it is inside the `exercise3` directory you changed to.

2. Create your own file using `nano`. In the example below, a file named `yourfile.txt` is created. In this file, enter at least five lines of text of any kind. Save the file and exit `nano`.

   ```
   $ nano yourfile.txt
   ```

3. Check the file size of your new file using `ls -l`. Files containing sequencing data can contain gigabytes of raw text data. This means that they easily can contain many millions times more data than your file.

4. Print the content of your file to the terminal using `cat`, `head` and `tail`. Adjust the `head` and the `tail` commands to just see the first and last 2-3 lines of your file. In the example below we look at the two first lines of `yourfile.txt`

   ```
   $ head -2 yourfile.txt
   ```

5. Investigate your file with `less`. `less` is often more convenient than `cat` as it doesn't print all of the (often huge) file content directly to the terminal. (Remember that you can exit `less` by pressing `q`).

   ```
   $ less yourfile.txt
   ```

6. Create a directory using the `mkdir` command.

7. Create a copy of your file using the `cp` command within the directory. This can be done in the following way (in this case the directory is named `yourdir`).

   ```
   $ cp yourfile.txt yourdir
   ```

   Can you also create another copy of your file in the directory while simultaneously assigning it a new name?

8. Rename one of the files within the directory using the `mv` command. A file can be renamed within the directory without you changing working directory:

   ```
   $ mv yourdir/yourfile.txt yourdir/yourrenamedcopy.txt
   ```

   This would rename the file `yourfilecopy.txt` within the directory `yourdir`. Check the current state of your files using the `ls` command.

9. Attempt to remove the directory (with at least one copy of your file in it) using the `rmdir` command. What happens?

10. Now, remove the directory using the `rm -r` command. Do it with the `-i` flag to see what files you are removing. Answer 'y' to remove the files.

    ```
    $ rm -r -i yourdir
    ```

There were a lot of commands here. If you are unsure about some of them, run them again until you feel comfortable creating, moving and removing files.

These commands are (together with the commands introduced in the previous chapter) the foundation for most UNIX-based work. When you have worked these commands into your muscle-memory you will be able to put your full attention to your analysis.

3.6.2 FASTA management (*)

This exercise is more challenging. Do it if you have the time.

Let's use the commands we have learned on some real data. After changing your working directory to your home directory, download and extract the tarball for this exercise:

```
$ wget http://130.235.244.56/unix/tarballs/unix_course_chapter3.tar.gz
$ tar -xf unix_course_chapter3.tar.gz
```

Make sure with `ls` that you now have a directory named `"unix_course_chapter3"` containing four files.

If you want to save the output from a command in a file, you can use the `>` operator. For example, if you want to save the output from `ls` to a file named `saved_output.txt` you would run:

```
ls > saved_output.txt
```

After running this command you will have a file named `saved_output.txt` containing the output from the `ls` command. This will be explained in more detail in the chapter about file streams.

1. * Do some investigation of the two types of FASTA files using the same commands that you used in the previous exercise. How are they similar/different?
2. * Concatenate the two FASTA-A files into a single FASTA named `combined_A.fa` and the two FASTA-B files into a single FASTA named `combined_B.fa`
3. * Investigate the two newly created files to make sure that everything looks good.
4. * Calculate the number of lines in your combined FASTAs by running the commands:

   ```
   $ wc -l combined-A.fa
   $ wc -l combined-B.fa
   ```

   The `wc` command will be introduced in the next chapter. How many lines did you get? You should get 4000 lines for the combined A-version and 34051 for the combined B-version. Does this support your previous hypothesis about the files' similarities / differences?
5. * Try removing one of the four initial FASTA files (not the ones that you created). What happens? Can you guess why?

Can you see why `less`, `head` and `tail` sometimes are more useful than `cat`?

3.7 Checkpoint

Before you continue, answer the following questions:

- When are plain text files, binary files and compressed files used?
- Why are compressed files especially important in bioinformatics?
- What makes the UNIX terminal especially dangerous when moving, copying and removing files?

3.7.1 UNIX commands

Consider each command for a second. Make sure that you have an idea about what they do before moving on.

Introduction to UNIX

```
man
ssh [-v]
```

Introduction to the file system

```
cd
ls [-l] [-lh]
pwd
```

Working with files in UNIX

```
cat
cp [-i]
file
head [-number]
less
mkdir
mv [-i]
nano
rm [-i] [-r]
rmdir
tail [-number]
```

3.8 Further reading

3.8.1 Text editors

Text editors is a never ending source of heated debate (in some circles). In this course we are using `nano` for editing raw text as it is a relatively simple editor. One of the more powerful editors is vim. It is also an extremely popular editor, and its predecessor vi is installed on virtually all UNIX machines.

Vim has two different modes - One for inserting text and another for running commands. It allows for powerful text editing, but can be somewhat unintuitive to pick up. If you are interested in learning vim, you could take a look at the following cheat sheet/tutorial:

`http://www.viemu.com/a_vi_vim_graphical_cheat_sheet_tutorial.html`

A friendly hint - You can (at most times) exit vim by typing ":wq".

Chapter 4

Working with file content

4.1 Important bioinformatic file formats

```
"A file format is a standard way that information is encoded for
 storage in a computer file"
                    - Wikipedia, 4th of April, 2016
```

A file format requires the content of a file to follow certain guidelines. The content varies between files, but the formatting of that content must be similar. This allow us to do assumptions about the files, which in turn allow us to run certain commands on them to retrieve desired information.

File formats is an important part of bioinformatics. You need to understand your files before you can know what to expect when processing them with UNIX commands.

4.1.1 The FASTA file format

A popular way of storing sequence data is the FASTA format. Each sequence is represented by a header-part and a sequence-part. The header contains information about the sequence (for example its ID) followed by the sequence itself. The structure of a FASTA file is shown in figure 4.1.

Sometimes the sequence of the FASTA file is divided into blocks of fixed length for easier viewing. This is called **multi-line fasta** (see figure 4.2). When processing the FASTA-file, it is usually easier to keep the sequence on a single line. This is called **single-line fasta** (see figure 4.3).

Sometimes you need to reformat the FASTA file from multi-line format to single-line format. See section 4.5.1 in the "further reading" materials if you want to learn how this can be done.

4.1.2 FASTQ

The FASTQ file format also contains sequence information, but has an additional line for each sequence containing quality information. The quality index is a measure used by the sequencer, and indicates how certain it is about each letter of the sequence. With a lower

File format specification

1. The content is divided into entries, each consisting of a header and a sequence
2. The header line stars with a >-sign, followed by information about the entry
3. The header is followed by one or more lines containing sequence data

Example:

```
>description line 1
AGTGTGATCGTAGCTAGC
>description line 2
ATTTAGATGATGAGAAGATGA
ATTGTGTACA
```

Figure 4.1: FASTA specification

Multi-line FASTA

```
>first_entry further_information
AGTAGCTGACTGACTGATCGATCGTAGCTA
GCTAGCTAGCTAGCTAGCTAGCTAGCTAGC
TGTAGCTAGC
>second_entry further_information
TGACGTGGCGCTAGGCATTATATACGGACG
GCGGCTACGATTATGCATCGTAGCAGATAT
TATTAGCTAGCA
```

Figure 4.2: Example of multi-line FASTA

Single-line FASTA

```
>first_entry further_information
AGTAGCTGACTGACTGATCGATCGTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTGTAGCTAGC
>second_entry further_information
TGACGTGGCGCTAGGCATTATATACGGACGGCGGCTACGATTATGCATCGTAGCAGATATTATTAGCTAGCA
```

Figure 4.3: Example of single-line FASTA

quality index, wrongly assigned letters are more common. The specification of the FASTQ-format is shown in figure 4.4.

The exact maximum quality value for the nucleotides depends on the sequencing machine, but is around 40. The numbers are each represented by different characters in order to be able to capture all 40 values in a single letter. Figure 4.5 shows an example of the quality

1. Each entry starts with a header line, starting with a `@`-sign
2. The header is followed by one or more lines containing sequence data
3. Then another header line, usually starting with a `+`-sign which either has the same content as line 1, or is left empty
4. Finally, quality indices for each nucleotide in the sequence data

Example:

```
@description line 1
AGTGTGATCGTAGCTAGC
+
%++)(%%%%).1***-22
@description line 2
+
ATTTAGATGATGAGAAGATGA
%++)(%%%%).1***-886((
```

Figure 4.4: FASTQ specification

encoding.

character	quality	character	quality
!	1	+	11
"	2	,	12
#	3	-	13
$	4	.	14
%	5	/	15
&	6	0	16
’	7	1	17
(	8	2	18
)	9	3	19
*	10	4	20
...			

Figure 4.5: Example of quality indices for the first twenty values in the FASTQ format

An example of a FASTQ-file is shown in figure 4.6. There are two entries, each represented by four lines. The first header lines start with a `@` sign, and the second header lines are left empty here (starting with a `+` sign). Each base pair is matched by a quality index sign, similar to the ones seen in figure 4.5.

Example FASTQ file

```
@first_entry
AGTAGCTGACTGACTGATCGATCGTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTGTAGCTAGC
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65%%).1*%%))
@second_entry
TGACGTGGCGCTAGGCATTATATACGGACGGCGGCTACGATTATGCATCGTAGCAGATATTATTAG
+
%%%++)(%%%%)55CCF>>>>>%++)(%%%%).1%%++)(%%55CCF>>>>>*-+*''))**5CCC
```

Figure 4.6: Example FASTQ file

4.1.3 GFF

The GFF format stands for *General Feature Format*, and is a common way of describing genes and other features in DNA, RNA and protein sequences. Currently, there are two versions of the GFF format in use - GFF2 by Sanger Institute and GFF3 by the Sequence Ontology Project. In this course we will only be using the GFF3-format.

The GFF file consists of rows containing information about different features. The exception is comment lines which similarly to the UNIX terminal language (Bash) starts with `#`. A dot '.' in a column means that there is no information present for that particular field of the entry. The nine columns of a GFF file are described in figure 4.7.

An example of a GFF-file is shown in figure 4.8. Note the two first comment lines. If those are present, they need to be taken into account when processing the file.

4.2 File content commands

You have already seen some commands for investigating file content:

- `cat`
- `head`
- `tail`
- `less`
- `nano`

Now, we are ready to expand our toolbox with a number of useful commands. We will start out by introducing the commands and their usages one by one. Soon, we will start learning how to combine those commands using `pipes`, linking together chains of processing steps. But that is for the next chapter.

1. **sequence** ID for sequence where feature is located.
2. **source** Keyword with information about what program or database that were used to identify the feature. Will not be discussed in this course.
3. **feature** The type of feature. Can for example be 'gene', 'exon', 'cds' (coding sequence).
4. **start** Start position of the feature.
5. **end** End position of the feature.
6. **score** Can be used to put a value for the confidence of the feature. Will not be discussed in this course.
7. **strand** If the feature is located in the plus- or minus-strand of genome.
8. **frame** For coding sequences, if there is a phase shift. Can have values 0, 1 and 2. Will not be discussed in this course.
9. **attributes** Other information related to the feature. Often contains information about its ID, and to which other features it is related (for example: To which gene exons belong).

Figure 4.7: GFF3 file format specification

Example GFF file

```
##gff-version 3.2.1
##sequence-region
ctg123 . gene  1000  9000  .  +  .  ID=gene00001;Name=EDEN
ctg123 . mRNA  1050  9000  .  +  .  ID=mRNA00001;Parent=gene00001
ctg123 . exon  1300  1500  .  +  .  Parent=mRNA00003
ctg123 . exon  5000  5500  .  +  .  Parent=mRNA00001,mRNA00002,mRNA00003
ctg123 . exon  7000  9000  .  +  .  Parent=mRNA00001,mRNA00002,mRNA00003
ctg123 . CDS   1201  1500  .  +  0  ID=cds00001;Parent=mRNA00001
ctg123 . CDS   3000  3902  .  +  0  ID=cds00002;Parent=mRNA00001
ctg123 . CDS   5000  5500  .  +  0  ID=cds00003;Parent=mRNA00001
ctg123 . CDS   7000  7600  .  +  0  ID=cds00004;Parent=mRNA00001
ctg123 . CDS   1201  1500  .  +  0  ID=cds00002;Parent=mRNA00002
```

Figure 4.8: Example of an GFF file in GFF3-file format

Note

It happens that files with biological information diverge from the file formats which they are said to have. In best case those errors cause the programs or the commands processing them to crash. In worst case, the errors silently slip through the processing, effecting the final numbers in a subtle and invisible way. Be careful and double check that your files actually follow the claimed format.

4.2.1 wc

Command usage: `wc [-l] [-w] [-m] <filename>`

The purpose of the `wc` command is to investigate the number of lines, words and characters in a file. By default, it will give you three numbers representing those three features. By providing flag arguments, you can limit the output to the feature of the file that you are interested in.

```
$ wc my_report.txt
   96  1125 11910 my_report.txt
# The file contains 96 lines, 1125 words, 11910 characters

$ wc -l my_report.txt
96 my_report.txt
# There are 96 lines in my_report.txt

$ wc -w my_report.txt
1125 my_report.txt
# There are 1125 words in my_report.txt

$ wc -m my_report.txt
11910 my_report.txt
# There are 11910 characters in my_report.txt
```

4.2.2 diff

Command usage: `diff <first_file> <second_file>`

The `diff` command compares two files and evaluates if there are any differences between the files. If it finds differences, it prints information about where the differences were found together with the diverging lines. If we want to examine a FASTA file and check whether it is identical to another copy, `diff` is the command to use.

Here, we compare the two FASTA files showed in figure 4.9 and figure 4.10.

raw_fasta.fs

```
>first_entry
AGTAGCTGACTGACTGATCGATCGTAGCTA
GCTAGCTAGCTAGCTAGCTAGCTAGCTAGC
TGTAGCTAGC
>second_entry
TGACGTGGCGCTAGGCATTATATACGGACG
GCGGCTACGATTATGCATCGTAGCAGATAT
TATTAGCTAGCATAGCTAG
```

Figure 4.9: Raw fasta file *raw_fasta.fs*

edited_fasta.fs

```
>first_entry
AGTAGCTGACTGACTGATCGATCGTAGCTA
GCTAGCTAGCTAGCTAGCTAGCTAGCTAGC
TGTAGCTAGC
>second_entry additional_information
TGACGTGGCGCTAGGCATTATATACGGACG
GCGGCTACGATTATGCATCGTAGCAGATAT
TATTAGCTAGCATAGCTAG
```

Figure 4.10: Edited fasta file *edited_fasta.fs*

```
$ diff raw_fasta.txt edited_fasta.fs
5c5
< >second_entry
---
> >second_entry additional_information
```

In this case, the `diff` command shows that one of the headers had additional information in one of the FASTA files.

4.2.3 grep

Command usage: `grep [-c] [-A] [-v] [-f] [-i] <pattern> <file_name>`

`grep` is a highly useful and versatile command. Its purpose is to extract certain lines from a file based on whether they match a particular pattern or not.

If we want to extract the FASTA headers from the file `raw_fasta.fs` (shown in figure 4.9), we could run the following:

-c Instead of returning the matches themselves, return the number of matches.

-A <number> Extract given number of lines trailing the matches.

-v Invert the match so that non-matching lines are return.

-f <filename> Obtain patterns from provided file, one per line.

-i Ignore upper/lower case when matching.

Figure 4.11: Useful grep flags

```
$ grep "^>" raw_fasta.fs
>first_entry
>second_entry
```

This command will extract all lines starting with the `>` sign. The `^` means that `grep` only looks for lines starting with the `>` sign. We will come back to this and other patterns in a later chapter. The quotation marks must be included for the `grep` command to interpret `^>` as a string.

There is a variety of useful flags for the `grep` command, which we will use in the coming exercises (see figure 4.11 for some useful examples). Remember, you can always check the man pages for more information - `man grep`.

Note

Make sure to keep track of which of the flags that require input arguments and which don't. For instance, when running `grep` with the `-c` flag or the `-v` flag no further extra input is required - the rest of the command is used as usual. When running with the `-A` flag, you need to provide a number to indicate how many trailing lines you want to retain. When running with the `-f` flag, you need to provide a file - but no regular pattern.

4.2.4 cut

cut *out sections from each line in a file*
Command usage: `cut [-d] -f/-c <fields> <file_name>`

The purpose of the command `cut` is to extract particular columns from lines. The most straight-forward case is to extract columns of data from tab-delimited files (like GFF files), but it can also be used to parse headers or extract particular stretches of nucleotides from sequences.

It is required to use one of the field flags `-f` or `-c`. Further information about the `cut` flags is seen in figure 4.12.

-d Delimitor - Specify character to use as separator between different fields

-f Fields - Per default text segments separated by white space (spaces, tabs)

-c Characters - Can cut out segments of text in particular ranges of characters

Figure 4.12: Central flags for the cut command

The field can be specified in three different ways - similarly to when selecting custom printer pages for most printers, if you are familiar with those. Those methods are shown in figure 4.13.

Ways of specifying fields to the cut command.

- A single number: `-f 3` (will cut field 3)
- A range of numbers: `-c 3-6` (will cut characters 3, 4, 5 and 6)
- Several picked numbers: `-f 3,4,7` (will cut fields 3, 4 and 7)

Figure 4.13: Central flags for the cut command

Usage example

example_annotation.gff

```
ctg123 . gene  1000  9000  .  +  .  ID=gene00001;Name=EDEN
ctg123 . mRNA  1050  9000  .  +  .  ID=mRNA00001;Parent=gene00001
ctg123 . exon  1300  1500  .  +  .  Parent=mRNA00003
ctg123 . exon  5000  5500  .  +  .  Parent=mRNA00001,mRNA00002,mRNA00003
ctg123 . exon  7000  9000  .  +  .  Parent=mRNA00001,mRNA00002,mRNA00003
ctg123 . CDS   1201  1500  .  +  0  ID=cds00001;Parent=mRNA00001
ctg123 . CDS   3000  3902  .  +  0  ID=cds00001;Parent=mRNA00001
ctg123 . CDS   5000  5500  .  +  0  ID=cds00001;Parent=mRNA00001
ctg123 . CDS   7000  7600  .  +  0  ID=cds00001;Parent=mRNA00001
ctg123 . CDS   1201  1500  .  +  0  ID=cds00002;Parent=mRNA00002
```

Figure 4.14: Example GFF file

If we want to extract only the column containing the feature information from the file `example_annotation.gff`, we could run the following:

```
$ cut -f 3 example_annotations.gff
gene
mRNA
exon
exon
exon
CDS
CDS
CDS
CDS
CDS
```

The default delimiter is tabs. If the data is separated - *delimited* - by another character, for examples commas (,), this can be specified using the `-d` flag. We could also retrieve parts of the lines characters directly using the `-c` flag. This is useful for extracting stretches from nucleotide- or amino acid-sequence.

```
$ cut -c 5-15 example_annotations.gff
23   .    gene     1
23   .    mRNA     1
23   .    exon     1
23   .    exon     5
23   .    exon     7
23   .    CDS      12
23   .    CDS      30
23   .    CDS      50
23   .    CDS      70
23   .    CDS      12
```

4.2.5 sort

Command usage: `sort [-n] [-r] <file_name>`

`sort` orders lines alphanumerically. It can also sort numerically if the `-n` flag is used. The sorting order can be reversed using the `-r` flag.

Usage example

If we for example have stored a feature column from a GFF-file in a file called `example_feature_column.txt` (seen in figure 4.15), we can run the following in order to sort it:

example_feature_column.txt

```
gene
mRNA
exon
CDS
exon
gene
CDS
exon
exon
CDS
CDS
CDS
```

Figure 4.15: GFF feature column

```
$ sort example_feature_column.txt
CDS
CDS
CDS
CDS
CDS
exon
exon
exon
exon
gene
gene
mRNA
```

-r Get the sorted results in reverse order

-n Sort on natural number instead of strings. This means that 9 would be sorted before 11. For regular sorting, the 11 would come first based on that its first letter is '1'.

Figure 4.16: Useful flags for the sort command

4.2.6 uniq

Command usage: `uniq [-c] <file_name>`

Finally, we have the command `uniq` which can be used to remove duplicate *adjacent* lines. This means that you have to sort the file before running the `uniq` command if you want to remove all duplicates, as it is unable to remove duplicates that are not adjacent.

-c Get the number of lines represented by each unique entry together with the entry

Figure 4.17: Useful flag for the uniq command

example_sorted_feature_column.txt

```
CDS
CDS
CDS
CDS
CDS
exon
exon
exon
exon
gene
gene
mRNA
```

Figure 4.18: Sorted GFF feature column

First, we try `uniq` on the sorted GFF column we received in previous exercise, stored in the file `example_sorted_feature_column.txt`. We include the `-c` flag to see the number of times the unique entries were encountered.

```
$ uniq -c example_sorted_feature_column.txt
      5 CDS
      4 exon
      2 gene
      1 mRNA
```

Now, we try out what happens if we run `uniq` for a non-sorted file. We use the file `example_feature_column.txt` seen in figure 4.15.


```
$ uniq -c example_feature_column.txt
      1 gene
      1 mRNA
      1 exon
      1 CDS
      1 exon
      1 gene
      1 CDS
      2 exon
      3 CDS
```

Can you see why this happened?

4.3 Exercises

4.3.1 Introduction to the exercise

Exercise files

Create a new directory in your home directory for this exercise. Keep the files for this exercise within this directory.

For this exercise (and later exercises) we will be using files from the potato reference genome (http://solanaceae.plantbiology.msu.edu/pgsc_download.shtml). We will also be using a FASTQ file taken from a real RNA-seq dataset. Those two datasets can be downloaded from the server:

```
$ wget http://130.235.244.56/unix/tarballs/genome_files.tar.gz
$ wget http://130.235.244.56/unix/tarballs/MhaptRNASeq.fastq.gz
```

Note that we use the `gunzip` command (`gunzip <gz-file>`) to extract the second file as it isn't a tar-archive. It is a single gzipped FASTQ-file.

The genome files uses IDs for genes, coding sequences, peptides and transcripts. The table file amount the genome files can be used to map between different types of IDs, as well as their annotation. The IDs looks like the following (the letter after "DM" shows if the ID belongs to a **c**oding sequence, a **g**ene, a **p**eptide or a **t**ranscript - in this case a coding sequence):

```
PGSC0003DMC400017652
```

When working with biological data it is often useful to first try the commands for smaller subsets before moving on to the whole files. This makes it easier to see what the commands are doing. Subsets of the large FASTA-, FASTQ- and GFF-files are provided in the tarball for this exercise.

```
$ wget http://130.235.244.56/unix/tarballs/unix_course_chapter4.tar.gz
```

Extract it and make sure that you have three files in the extracted folder with the file endings `.gff`, `.fasta` and `.fastq` Take a look inside the files using commands such as `head` or `less`.

Saving output from a command

You can save the output from a command to a file using the `>` operator:

```
head myfile.txt > my_saved_output.txt
```

This will be explained in depth in a coming chapter. Be careful when running this command so that you not print the output into an existing file. The following command would remove the file `my_valuable_data.fa` and replace it with the output from the `head` command:

```
head myfile.txt > my_valuable_data.fa
```

4.3.2 Exploring the FASTA file

We will start with a subset of the FASTA-file before moving on to the full file.

1. Take a look at the content of `cds_subset.fasta`. Use `grep` with the `-c` flag to calculate the number of entries.

   ```
   $ grep -c "^>" cds_subset.fasta
   ```

   The `-c` flag tells `grep` to count the number of matched entries instead of printing them. The `"^>"` pattern matches a `>` sign at the start of the line - The header lines. If we want to look directly at the headers, we can run the command without the `-c` flag.

   ```
   $ grep "^>" cds_subset.fasta
   ```

2. Use the `wc` command to count the number of lines and number of letters in the file. There should be 3891 letters in this file (including line breaks).

3. Make a copy of one of your FASTA file, and use `nano` to do some edits in the headers of the copy. Then compare it to the original FASTA file. using the `diff` command. Do you understand the output?

   ```
   $ cp cds_subset.fasta cds_subset_copy.fasta
   $ nano cds_subset_copy.fasta # Add information to a header
   $ diff cds_subset.fasta cds_subset_copy.fasta
   ```

   The `diff` command can be very useful to see whether a file has been changed compared to the raw data.

4. Let's say we want to extract only the sequence with the coding sequence with ID `PGSC0003DMC400024095`. Use the `grep` command with the `-A` flag to get the headers together with trailing sequence line. (If you are unable to copy the ID from this document, you could use a smaller part of the ID here).

4.3.3 Exploring the FASTQ file

Here, we start out with the subset of the FASTQ file: `RNAseq_subset.fastq`

1. Take a look into the file. How many sequences are represented in the file? Remember that each sequence is represented by multiple lines. See if you can understand the different lines. Go back to the introduction of the FASTQ format in this material if you don't remember how the format is structured.

2. Now, let's count the number of lines using a UNIX command. The naive way would be to approach it similarly to how we approach counting entries in the FASTA file - Counting the number of lines starting with the `@`-sign.

```
$ grep -c "^@" RNAseq_subset.fastq
```

   This is not the way to do it! What number did you get from this? How many sequences are really represented in the file? Take a look inside the file and see if you can figure out why this command gives the wrong number.

3. A better way is to count the total number of lines in the file, and divide the number by four. Count the total number of lines. If you divide this number by four, do you get the correct count?

4. * Download the full FASTQ-dataset and decompress it using the `gunzip` command (take a look into the man pages - or the next chapter - if you are unsure how to run it). How many lines are there in total in this dataset? This dataset is actually only a small part of the original dataset. This kind of sequencing data can result in huge files.

4.3.4 Exploring the GFF file (*)

Our goal is to find out how many genes, exons and other features we have in the full GFF-file. We will first use the subset of the GFF file named `annotation_subset.gff` found in the tar-archive for this exercise, before moving on to the full file. Make sure that you understand what is happening for each of the steps.

1. First, lets explore the gff-file. Investigate the content visually using the commands you learned in the previous chapter. How many genes does this subset contain? Do you see how the different features are linked together through the IDs in the attributes column?

2. We have some comment lines at the start of the file. Those can often contain useful information, but must be removed before we do further analysis. Use `grep` with an appropriate flag to remove those lines, and redirect this output to another file using the `>` operator.

   To print only the lines starting with the comment sign `#`, we could do the following:

```
$ grep "^#" annotation_subset.gff
```

   The `"^#"` is the pattern we are matching for. The hat (`^`) tells us to only look for the `#` sign at the start of the lines. `annotation_subset.gff` is the file which we are running `grep` on. Take a look at the description of `grep` in this chapter if this is unclear.

Now we got the comment lines. We want to flip the pattern so that we get all lines *except* the comment lines. We can do this using the `-v` flag. Let's retrieve all non-comment lines, and put them into another file named `subset_nocomments.gff`

```
$ grep -v "^#" annotation_subset.gff \
    > subset_nocomments.gff
```

The `\` means that the line is too long to fit on one line, and continues on the second line. It is possible to divide lines like this in UNIX. **Do not type out this character if you aren't going to divide your command into multiple lines**.

As mentioned, be careful so that you not point the `>` arrow into an existing file containing valuable data.

3. Double check that you now have your GFF-file without the comment lines by investigating the `subset_nocomments.gff` file.

4. Use the command `cut` to extract the column with the features (gene, mRNA, exon, CDS) and make another file with this output. Print the output to the terminal first to make sure that it looks like expected before redirecting it to another file.

```
$ cut -f 3 subset_nocomments.gff  # Check that output looks OK
$ cut -f 3 subset_nocomments.gff > annotation_column.txt
```

Make sure that you have the right content in the `annotation_column.txt` file.

5. We now want to use the `uniq` command to get the number of each type of entry, but first we need to sort the file. (You could try and see what you get if you run `uniq -c` on the `annotation_column.txt`). Create the sorted file using `sort`.

```
$ sort annotation_column.txt > sorted_features.txt
```

6. Almost there! Let's count the number of the different features we have in `sorted_features.txt` using `uniq`.

```
$ uniq -c sorted_features.txt
```

Does the number correspond to the numbers in the subset of the file? (You can count them manually).

7. * Now you are ready to count the number of the different features in the full GFF file (found in the genome files directory). How many of each feature do you get? If you have done it correctly, you should get counts for four features, and you should find 141037 exons.

This process will be greatly simplified when pipes are introduced in a coming chapter, as it will remove the need for creating the intermediary files.

4.3.5 Working with the annotation - Case study (**)

Introduction

This exercise is designed to provide some extra challenge for those who might have some background with UNIX, or those who particularly likes a challenge. This type of task is often encountered when working with bioinformatic data. If you are short on time - Save this exercise for the future.

We have a GFF file with the genes (`representative_genes.gff`), together with the sequences in FASTA format (`representative_cds.fasta`).

We also have a table-file (`mapping_table.tsv`) linking the different types of IDs to each other and their annotations. The four different columns contains IDs for genes, transcripts, coding sequences and peptides.

Hint: You can convert a multi-line FASTA to single-line format in many ways. One of the simpler ways is to use the program `seqtk` and run the following command:

```
$ seqtk seq -l 0 multilinefile.fa > singlelinefile.fa
```

The task

You have been in contact with a biologist who is particularly interested in investigating genes containing the zinc finger motif. You task is to retrieve all gene sequences which have been given annotations related to "zinc finger". All genes with annotations that include "zinc finger" in their text should be extracted, and their sequences should be gathered into a separate file.

To extract this information you will need to use some or all of the three genome files. Make sure to think through the problem before trying to solve it. Make an outline. What information do we need? What information do we have? Are there particular challenges with the file formats we need to take into account - for example, which versions of the IDs do we have in which file?

Outline of possible approach

Feel free to approach this exercise in any way you want, using what you have learned this far in this course. An outline of a potential approach is described below.

1. * Start by extracting the coding sequence IDs that have an annotation related to "Zinc finger" from the mapping file. Ignore upper/lower case. The goal here is to get a file only containing coding sequence IDs which are related to the zinc finger. You will likely need to make an intermediate file to extract this information. At this point you should have 620 IDs (if you have 529 - did you ignore upper/lower case?).

2. * Extract the corresponding sequences from the FASTA containing all coding sequences. `grep` is likely useful here, together with a flag argument mentioned in this chapter.

3. * Count the number of entries and lines. Do you have 395 FASTA entries? Check with the `wc` command that there is 423803 characters in the final file. (If you got 424961, make sure to double check your output. Does the FASTA file look as you expected?)

If you made it through this exercise - Well done! If you didn't - no problem. You can revisit it at a later time.

4.4 Checkpoint

Before you continue, make sure that you can answer the following:

- What is the difference between single-line and multi-line FASTA files, and why do you need to keep track of it when processing them in UNIX?
- What is the purpose of the FASTQ format, when we already have the FASTA format?
- What can the GFF-format be used for?

4.4.1 UNIX commands

Consider each command for a second. Make sure that you have an idea about what they do before moving on.

Introduction to UNIX

```
man
ssh [-v]
```

Introduction to the file system

```
cd
ls [-l] [-lh]
pwd
```

Working with files in UNIX

```
cat
cp [-i]
file
head [-number]
less
mkdir
mv [-i]
nano
rm [-i] [-r]
rmdir
tail [-number]
```

Working with bioinformatic data

```
cut [-d] -f/-c
diff
grep [-c] [-A] [-v] [-f] [-i]
sort [-n] [-r]
uniq [-c]
wc [-l] [-w] [-m]
```

4.5 Further reading

4.5.1 Converting multi-line fasta to single-line fasta

Make sure to keep track of what format you are using. There are many ways of converting multi-line FASTA files to single-line format. One straight-forward way of doing it is by using the seqtk-software suite in the following way:

```
$ seqtk seq -l 0 my_multi_line_fasta.fs > \
    my_single_line_fasta.fs
```

The "-l 0" tells `seqtk` to put all based on a single line. It can be set to a positive number if you instead want the file in multi-line format. Note that you in order to run this command needs to have `seqtk` installed on your computer.

Other ways of solving this problem are discussed in the following BioStars thread:

https://www.biostars.org/p/9262/.

4.5.2 Useful tool: seqtk

`seqtk` is a versatile tool able to work with both compressed and uncompressed FASTA and FASTQ in many useful ways. It is not a built in UNIX command, but a separate command-line tool (similar to many other more powerful bioinformatic software). This means that it isn't preinstalled on all UNIX-computers, but needs to be installed separately (like many other software). If you are further interested in how `seqtk` could be useful and how to install it, take a look here:

https://github.com/lh3/seqtk

There are a lot of useful bioinformatic tools for a wide variety of purposes. Sometimes, you need to make tools yourself, but if you look around you will often find that others already have solved your problem.

Chapter 5

File permissions, organizing files and UNIX hygiene

5.1 File permissions

File permissions are used in UNIX to control which users that are allowed to use files and directories in different ways. They can also be used to protect files from accidental editing or removal. The effects of file permissions for files and directories are listed in figure 5.1.

For files, the file permissions control who is allowed to:

- Read the file content
- Edit/re-write the file content
- If the file is a program or script - Who can run/execute the program.

For directories, the file permissions control who is allowed to:

- Read the content of the directory.
- Create/write new files in the directory.
- Change the working directory to that directory.

Figure 5.1: Purpose of file permissions

In chapter 2, we ran the following command:

```
$ ls -l Documents
-rw-r----- 1 user user 687438  sep 16 19:02 MyArticle.pdf
-rw-r----- 1 user user 2481860 may 13 14:05 MyReport.pdf
```

An overview of the meaning of all the different fields is shown in figure 5.2.

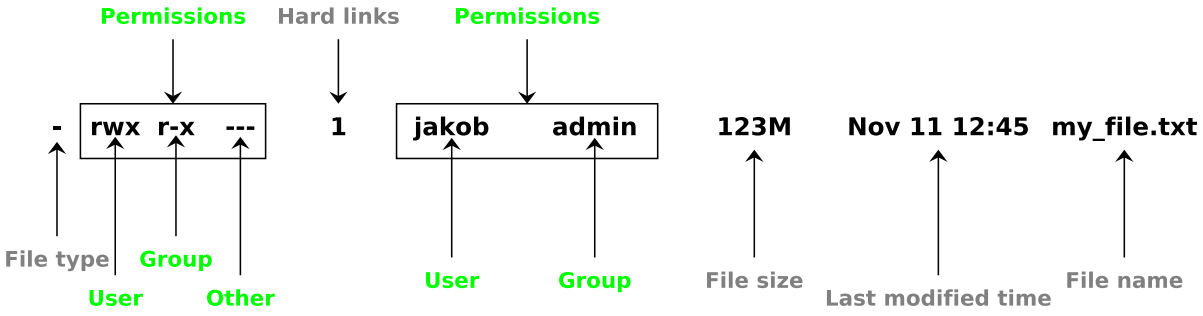

Figure 5.2: Explanation of the different parts of the ls -l output. Parts related to permissions are marked in green.

The first column here (the `-rw-r-----`) is the *file permissions* of that particular file. The third and fourth column (`user` and `user`) tells us to which *user* and to which *user group* a file belong. Each user can be part of several different user groups.

Here you see another example of running `ls -l` for a directory containing two files - A directory and a binary file:

```
$ ls -l
drwxr-xr-x 2 jakob jakob  4096  jan 25 08:58 a_directory
-rwxr-xr-x 1 jakob jakob  10803 mar 24 08:27 a_binary
```

For the directory, we have an initial 'd' before the rest of the permissions, telling us that it is a directory. Its permissions tell us that everyone can read and access the directory, but only the user `jakob` is able to create new files in it.

The permissions for the binary tells us that everyone can read and run the program, but only the user `jakob` can edit it.

5.1.1 chmod

ch*ange file* ***mod****e*
Command usage: `chmod <new_permission> <target_file>`

The `chmod` command is used to change the existing permissions of a file. If you are not an administrator, you are only able to change permissions for files for which you are the owner.

Examples of changing the permissions of a file are shown below:

```
$ ls -l
-rw-rw-r-- 1 jakob jakob 182 apr  7 14:45 raw_data.fa
$ chmod +x raw_data.fa
$ ls -l
-rwxrwxr-x 1 jakob jakob 182 apr  7 14:45 raw_data.fa
$ chmod -w raw_data.fa
$ ls -l
-r-xr-xr-x 1 jakob jakob 182 apr  7 14:45 raw_data.fa
```

If we want to add execution permission to a file, we can do that with the `chmod +x` command. If we on the other hand want to remove a type of permission (in this case write permission) we can do that with the `chmod -w` command. So, we can add/remove the target permission using a plus/minus sign followed by the letter representing the permission.

It is also possible to set the exact setup of permissions for a file in a single command. That is left for the further reading part.

5.1.2 Using file permissions

File permissions are important for security reason, as they control who can access your files. But, they can also play another important role in your projects by protecting your data from accidentally being edited or overwritten. We have previously mentioned the use of the `-i` flag for the commands `rm`, `mv` and `cp`. Another way of providing 'soft' protection of your files is to remove the write permission of files that you don't want changed. This is recommended for files like raw data and analysis output files which you don't plan to edit.

You can remove the write permissions from a file by running the command `chmod -w <target>`.

```
$ ls -l
-rw-rw-r-- 1 jakob jakob 9 apr  7 14:06 raw_data.fa

$ chmod -w raw_data.fa

$ ls -l
-r--r--r-- 1 jakob jakob 9 apr  7 14:26 raw_data.fa

$ rm raw_data.fa
rm: remove write-protected regular file 'raw_data'?
# Type n and enter to not remove it

$ mv other_file.fa raw_data.fa
mv: try to overwrite 'other_file.fa',
overriding mode 0444 (r--r--r--)?
```

As long as you are the owner of the file, you are still able to remove it, but it will ask you first. If a program attempts to overwrite the file, it will usually shut down with an error message.

See in figure 5.3 and figure 5.4 how `nano` reacts if you attempt to edit `raw_file.fa` after removing the write permission.

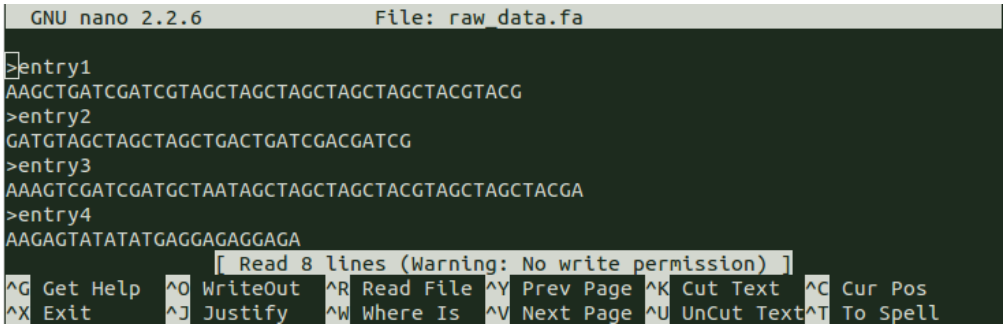

Figure 5.3: Opening a file without write permission in nano

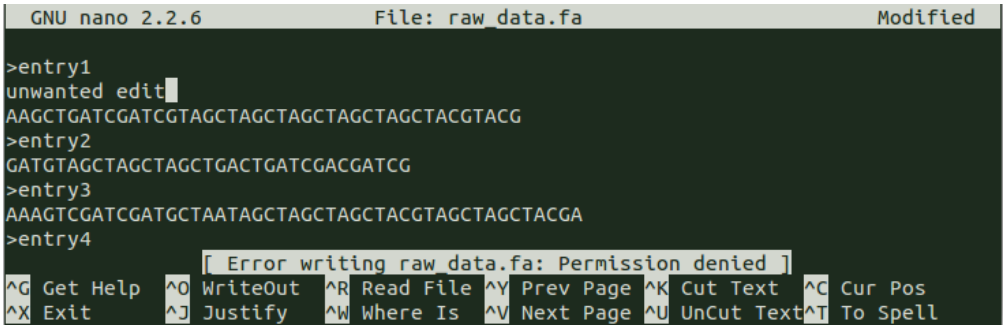

Figure 5.4: Attempting to save changes to a file without file permission in nano

5.2 gzip and tar archives

In other operating systems, the zip-format is often used both for compression of files, and for packaging a directory of files together in a single file.

On many UNIX-systems, two other file formats are frequently used to compress and package the files.

- The standard way of compressing files is by using the Gzip format. This is commonly seen as files with a `.gz`-extension.
- The standard way of packaging files together is by using the tar-format. This is commonly seen as files with a `.tar`-extension.

When files are packaged in the tar-format, they are usually also gzipped, which is seen as files with a `.tar.gz`-extension. Those packages are often called *tar archives* or *tar balls*.

5.2.1 gzip

gzip - *compress files*
Command usage: `gzip <file>`

The `gzip` command is used to reduce the size of the file by compressing its data. After compression, the `.gz` suffix will per default be added to the existing file name.

```
$ ls -lh
-rw-r--r-- 1 jakob jakob  49M maj  6  2011 nucleotides.fs
$ gzip nucleotides.fa
$ ls -lh
-rw-r--r-- 1 jakob jakob  13M maj  6  2011 nucleotides.fs.gz
```

Notice that the compression reduced the file size to almost a quarter of its original size.

5.2.2 gunzip

gunzip - *uncompress Gzipped files*
Command usage: `gunzip <gzipped_file>`

The `gunzip` command acts in the reverse compared to `gzip`. It restores the gzipped file to its original format (and size).

```
$ ls -lh
-rw-r--r-- 1 jakob jakob  13M maj  6  2011 nucleotides.fs.gz
$ gunzip sequence_data.fa.gz
$ ls -lh
-rw-r--r-- 1 jakob jakob  49M maj  6  2011 nucleotides.fs
```

If you want to investigate the content of a gzipped file without unzipping it, you can either use `less` which is able to read the files directly, or `zcat` which prints the content similarly to the `cat` command.

5.2.3 tar archives

The `tar` command has historically been used to store files on a disk- or tape archive. An example of an collection of tape archives can be seen in figure 5.5.

The flags used for the `tar` command to create and extract tar archives are shown in figure 5.6.

The tar format is commonly used for storing bioinformatic data. If you work with bioinformatics, chances are high that you will encounter tar archives and gzipped files.

Create gzipped tar archive

*Create archive of files (previously **ta**pe a**r**chive, now file archive)*
Command usage: `tar -czf <tar_archive> <my_directory>`

Figure 5.5: Old tape archive. Source: U.S. Department of Agriculture

-x Extract the archive

-c Compress the archive

-z Compress/decompress gzipped archive

-f Create the following tar archive

Figure 5.6: Tar archive flags

The `tar` command is used together with the `-c` flag, the `-z` flag and the `-f` flag to create a new tar archive.

```
$ ls -l
drwxrwxr-x 2 jakob jakob 4096 jan  7 11:10 directory
$ ls directory
-rw-rw-r-- 1 jakob jakob 0 apr  7 12:01 nucleotides1.fa
-rw-rw-r-- 1 jakob jakob 0 apr  7 12:01 nucleotides2.fa
$ tar -czf directory
$ ls -l
drwxrwxr-x 2 jakob jakob  4096 apr  7 11:10 directory
-rw-rw-r-- 1 jakob jakob   173 apr  7 12:58 directory.tar.gz
```

Note that when using the `tar` command you retain the initial directory you are archiving

or the initial tar archive you are extracting. When using the `gzip`/`gunzip` commands your original file is not retained if not specifying elsewise.

Extract tar archive

*Extract archive of files (previously **ta**pe a**r**chive, now file archive)*
Command usage: `tar -xf <tar_archive>`

The `tar` command is used together with the `-x` flag and the `-f` flag to extract an existing tar archive. The extracted archive is found as a directory without the `.tar.gz` suffix.

```
$ ls -l
-rw-rw-r-- 1 jakob jakob   173 apr  7 12:58 directory.tar.gz
$ tar -xf directory.tar.gz
$ ls -l
drwxrwxr-x 2 jakob jakob  4096 apr  7 11:10 directory
-rw-rw-r-- 1 jakob jakob   173 apr  7 12:58 directory.tar.gz
$ ls directory
nucleotides1.fa nucleotides2.fa
```

5.3 Downloading files

Two commonly used commands for downloading files from the internet using the terminal are `wget` and `curl`. The `wget` command is the default command in Linux distributions. On Mac on the other hand, `curl` is the default command. Here, we will present the `wget` command, but if you don't have it available - explore the `curl` command.

5.3.1 wget

*The non-interactive net**w**ork downloader/**get**ter*
Command usage: `wget <internet_address>`

`wget` can be used both to download data from the internet, or to download the HTML-text for the web pages themselves.

When running the command, you get information about whether the connection to the target computer hosting the data succeeded, followed by a progress bar with information about the ongoing download.

```
# This link does not actually exist
$ wget http://dummy.edu/data/genes.gff.gz
--2016-04-07 13:18:28--  http://dummy.edu/data/genes.gff.gz
Resolving dummy.edu (dummy.edu)... 35.8.196.182
Connecting to dummy.edu (dummy.edu)
|35.8.196.182|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4888883 (4,7M) [application/gz]
Saving to: 'genes.gff.gz'

100%[======================================>] 4 888 883
2,80MB/s   in 1,7s

2016-04-07 13:18:30 (2,80 MB/s) - 'genes.gff.gz'
saved [4888883/4888883]

$ ls
genes.gff.gz  # Now on your computer!
```

5.4 Symbolic file links

When working with bioinformatic data, it is not uncommon for some files of interest to be re-used in many different parts of the project, or even in different projects. To avoid making copies of the data to all those different locations, or writing extensive file paths to reach the data, you can use a file link as a 'shortcut' to the data from a more convenient location.

5.4.1 ln -s

*make **ln**ks between files*
Command usage: `ln -s <path_to_target> [<link_location>]`

The `ln -s` command is used to create so called 'soft links' to files. Those can be thought of as shortcuts to a file using a relative path or absolute path. If the command is used without the `-s` flag, you get a hard link to the physical memory location of the computer that the target file is using. In this course we will only be using the soft links.

> *Hardlinks* refer to the physical location of the computer. If the file is moved, the link will follow. *Softlinks* points to a certain location in your file structure and is managed by the operating system.

To demonstrate, we will use the file structure shown in figure 5.7. Here, the user 'jakob' has two projects, where he wants to reuse the file `RawData.fa.gz` used in the first project.

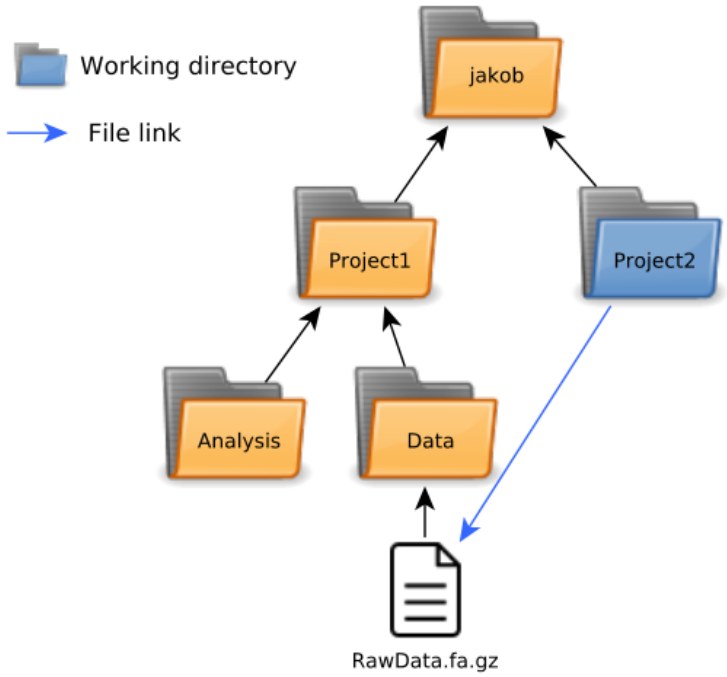

Figure 5.7: Demonstration of file link

To avoid copying it to the second project directory, the file can either be referred to through long paths (often much longer compared to what is shown here), or through a file link which acts as a shortcut.

```
$ pwd
/home/jakob/Project2

# less can read .gz-files
$ less ../Project1/Data/RawData.fa.gz

$ ln -s ../Project1/Data/RawData.fa.gz

$ ls -l
lrwxrwxrwx 1 jakob jakob  12 apr  7 13:58 RawData.fa.gz -> \
    ../Project1/Data/RawData.fa.gz
# Note the beginning 'l' for link filetype

$ less RawData.fa.gz
```

Here, we used the command `ln -s ../Project1/Data/RawData.fa.gz` to create a link to the data file `RawData.fa.gz`. This gives us a shortcut to the file, which we then can use to reach the data.

5.5 UNIX hygiene

There are some good practices when working with biological data which can save significant time and effort and make your analysis less prone to mistakes. When working with biological data, those lessons are often learned the hard way. Here, we will do an attempt to lead you past some of those hurdles.

Key points:

- Remove the write permission from files that you want to protect from being removed or edited.
- Always keep at least one backup of your data. What would happen if your most important hard drive stopped working right now? (Which it at some point will do).
- Keeping projects and analyses in clear file structures makes it much easier to keep track of projects and analyses.
- Document the analyses properly. It is as important as a lab notebook. The analysis must be possible to redo at a later point.
- Let the computer do the repetitive work for you to reduce human errors.
- Give your files descriptive names and correct file endings so that you quickly understand what they represent.

We have mentioned file permissions previously in this chapter, and the point of repetitive work will be revisited in the coming chapters. The remaining points will be discussed here.

5.5.1 Backup your data

```
"The rule of two:
 If you have one copy of something you could as well have zero"
                    - CGP Gray, Cortex (paraphrased)
```

If you are working with any kind of sequencing data, make sure that you have it backed up. Also, make sure that it is easy for you to make the backups so that you do them regularly. At some point, your hard drives will fail. You will drop your computer on the ground, someone will accidentally remove your files, you will pour coffee on it or the hard drive will simply stop working due to age. Make sure that you are ready when this will happen. A good rule of thumb is checking how prepared you are if an accident would happen *right now*. Are you satisfied with how well backed up your system would be right now?

5.5.2 Proper organization of your files

It is easy to start out using none or a few folders for all your data, programs and documentation. In the beginning, it might be possible to keep track of everything. But, as your

projects grow, this will soon not be feasible. It takes more and more time to keep track of the files.

Organizing files properly pays off in the long run. If you organize your files well from the beginning, it will help when the project starts expanding. Some important points:

- Have a clear folder structure for each of your projects, so that you know where each file is supposed to be (raw data, analysis, documentation...)
- Use file links rather than copying your files between projects

5.5.3 Document your analyses

You need to keep track of how you have processed your data. It is as important as making notes when performing work in the lab. Both you and others should be able to trace your analysis.

One way to help the documentation is to gather sequences of UNIX commands into scripts. This will be further discussed in the last chapter of this course.

5.5.4 Name your files properly

When you create files in UNIX, make sure that you assign the proper file endings to them. If you for example are creating a FASTQ file, it should have a file ending quickly showing the user that the file is formatted as a FASTQ file (.fastq or .fq are commonly used).

It is also worth putting some thought into the names of your files. If you name important files `file1`, `file2`, `file3`.. you will have trouble later on keeping track of your files. This is important, both as it saves you time in the long run and reduces the risk that you mix up your files.

5.6 Exercises

In this chapter, we will use a FASTQ file (named MhaptRNASeq.fastq) in the exercises. It is highly recommended to write-protect raw files, such as FASTQ, to safeguard the original sequence data that was delivered by the sequence facility.

5.6.1 Download the files to your home directory

This is the regular procedure. We will use the `wget` command to download the FASTQ file from the Linux server:

```
$ wget http://130.235.244.56/unix/tarballs/MhaptRNASeq.fastq.gz
```

A copy of the FASTQ file should now be located in your current directory. Check by using ls:

```
$ ls
```

Many bioinformatics tools can use gzipped files as input. It is highly recommended to keep large files gzipped whenever possible. In this case however we will unzip the file for the rest of the exercise.

```
gunzip MhaptRNASeq.fastq.gz
```

5.6.2 chmod

1. Make a copy (using `cp`) of the MhaptRNASeq.fastq file and call it MhaptRNASeq2.fastq:

   ```
   $ cp MhaptRNASeq.fastq MhaptRNASeq2.fastq
   ```

2. Change file permissions `-w` of the MhaptRNASeq2.fastq that you just generated:

   ```
   $ chmod -w MhaptRNASeq2.fastq
   ```

3. Use `ls -l` to compare the permissions of the two fastq files. Make sure that you understand the difference of the file permissions before you continue. Also note the difference in the results when using `ls` and `ls -l`.

4. Remove the MhaptRNASeq2.fastq file using the `rm` command:

   ```
   $ rm MhaptRNASeq2.fastq
   ```

   Press y to remove the file.

5. Check the files in the directory by using ls again to ensure that MhaptRNASeq2.fastq was removed. Now write protect the original MhaptRNASeq.fastq file before continuing with the exercises.

5.6.3 Symbolic links

In order to organize your data files in an efficient and clear way it can be convenient to use symbolic links. This will keep the original raw files in their place but still easily accessible in the different projects where they may be used.

1. Make a new directory called Data using the `mkdir` command and use the cd command to enter into the Data directory:

```
$ mkdir Data
$ cd Data
```

2. We want to make a symbolic link in the Data directory that points to the MhaptRNASeq.fastq file.

```
$ ln -s ~/MhaptRNASeq.fastq MhaptRNASeq.fastq
```

   Check with both ls and `ls -l` to see how the newly generated link works.

3. Use head to check the contents of the file that the symbolic link points to:

```
$ head MhaptRNASeq.fastq
```

   Notice that you can work with the symbolic link in exactly the same way as if it was a copy of the orignal file. In addition you save precious harddisk space since many of the files you are using could be many Gb is size.

5.6.4 Further work (*)

Further work if you have some time:

1. * Make a set of directories that you will use to organise your bioinformatics project. Which directories would you need? Think about an organization that you feel comfortable with and that will be useful in your research. One of the directories will contain the raw data for this project.

2. * Make a symbolic link from the FASTQ-file in the raw data directory you just created similar to how you did in the earlier exercise. However, this time name the symbolic link something of your own choice (but keep the FASTQ extension so that it is clear what type of file it is).

Tip! Many raw sequence files will have very cryptic names but the name of the symbolic link can be something that is easier to remember.

5.7 Checkpoint

Before you continue, make sure that you can answer the following:

- File permissions can secure your files from other users. When can it be useful to secure the files from yourself?
- What is the purpose of file links?
- Why is it important to use descriptive file names and a clear folder structure in your project?
- Are your files backed up? Are you comfortable with your answer?

5.7.1 UNIX commands

Consider each command for a second. Make sure that you have an idea about what they do before moving on.

Introduction to UNIX

```
man
ssh [-v]
```

Introduction to the file system

```
cd
ls [-l] [-lh]
pwd
```

Working with files in UNIX

```
cat
cp [-i]
file
head [-number]
less
mkdir
mv [-i]
nano
rm [-i] [-r]
rmdir
tail [-number]
```

Working with bioinformatic data

```
cut [-d] -f/-c
diff
grep [-c] [-A] [-v] [-f] [-i]
sort [-n] [-r]
uniq [-c]
wc [-l] [-w] [-m]
```

Organizing files

```
chmod
gunzip
gzip
ln [-s]
scp
tar [-cvzf] [-xvzf]
wget
```


5.8 Further reading

5.8.1 Backup tool: rsync

One of the most commonly used programs on UNIX for creating backups of data is `rsync`. `rsync` is able to mirror the content of a directory to another location. This other location can be another directory on your computer, but can also be specified as a path on another computer. The following command mirrors the local directory `myfiles` to the user `jakob`'s home on another computer. (The colon `:` is used to separate the server address from the path on the server computer).

```
$ rsync -avz myfiles/ jakob@192.168.0.101:/home/jakob
```

Two of the main strengths of `rsync` compared to other copying tools are:

- It is able to detect whether parts of the files already exist in the target directory. This means that when running backups, it will only transfer files which have been edited or created.
- It can keep the data compressed during transfer, which is especially useful when running a backup over an internet connection.

A number of useful examples for the `rsync` command can be found here:

http://www.tecmint.com/rsync-local-remote-file-synchronization-commands/

5.8.2 Organizing projects

Every person working with biological data develops his or her own way of structuring projects. But, there can be a value in learning from the experience of others. The article *A Quick Guide to Organizing Computational Biology Projects* (Lewitter, F. *et al.*) presents one approach with several ideas on how to apply the principle:

"Anyone unfamiliar with your project should be able to look at your computer files and understand in detail what you did and why. This 'someone' will in many cases be yourself returning to a project some months later".

http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1000424

Chapter 6

Working with file streams

6.1 What are file streams?

File streams, or *standard streams*, are standard ways for programs to communicate with the rest of the computer. There are three standard streams used to interact with file content. Those are listed in figure 6.1.

Standard input (stdin) Read file content into a program.

Standard output (stdout) Output content from the program.

Standard error (stderr) Similar to stdout, but used to separate error messages from regular program output.

Figure 6.1: Types of file streams

File streams can be used to input the content of a file to a program, or to capture and control the output from a program. The output of a file is generally directed to one of the three following:

- To the terminal. Per default, the output is printed to the terminal from which you ran the program.
- To a file. The output from a program can be redirected to a file to capture and store the output.
- To another program/command. In UNIX many commands are designed to read input directly from the output of another command.

Up until now, we have simply been printing the output back to the terminal. Now, we will start looking into channelling output from commands to other commands, a process we call `piping`.

6.1.1 echo

Command usage: `echo <text to echo>`

For this chapter, we introduce the command `echo`. `echo` takes a string of text, and then outputs it to the standard output. This is useful for testing purposes.

```
$ echo "This text is given to the echo command"
This text is given to the echo command
```

The string "This text is given to the echo command" is echoed back to the terminal.

6.2 Redirecting input and output

6.2.1 Standard output

If we run the command `echo`, the output will go to the default location of standard output - back to the terminal.

```
$ echo "This is some text"
This is some text
```

If we want to store the output for later usage, we can redirect the standard output to a file using the `>` operator.

```
$ echo "This is some text" > for_later_usage.txt
$ cat for_later_usage.txt
This is some text
```

Note that if we redirect the output to a file, it will no longer be printed back to the screen. Another thing to note is what happens if we redirect the standard output to an existing file.

```
$ cat for_later_usage.txt
This is some text
$ echo "More text" > for_later_usage.txt
$ cat for_later_usage.txt
More text
```

The line "This is some text" is no longer present in the file after writing "More text" to it. When you open a new stream to an existing file using the `>` operator, the existing content is removed and replaced with the new file content. If you instead want to append lines to an existing file, you can use the `>>` operator.

```
$ cat for_later_usage.txt
This is some text
$ echo "More text" >> for_later_usage.txt
$ cat for_later_usage.txt
This is some text
More text
```

Note

A common way of accidentally removing important files is by redirecting the output from commands into them. We will learn how to use *file permissions* to protect your valuable data from this kind of accident in the next chapter.

6.2.2 Standard error

There is a second output stream that can be used to output information called `standard error`.

Many commands print useful error messages when they are used incorrectly which can help understanding what went wrong. In many cases, those are captured in log files for later reviewing. Standard error can be redirected using the `2>` operator, or `2>>` if you want to add content to an existing file.

```
# "non_existing_directory" could be any non existing path
$ cd non_existing_directory
bash: cd: non_existing_directory: No such file or directory
$ cd non_existing_directory 2> stderr.log
$ cat stderr.log
bash: cd: non_existing_directory: No such file or directory
```

Many larger programs produce both output to standard output and to standard error. We can capture both outputs, and redirect them to separate locations.

```
$ large_program
This line is written to stdout
This line is written to stderr
$ large_program >stdout_example.txt 2>stderr_example.txt
$ cat stdout_example.txt
This line is written to stdout
$ cat stderr_example.txt
This line is written to stderr
```

In this case, ”This line is written to stdout” was written to standard output by the program, and ”This line is written to stderr” was written to standard error. The developer

of the program can decide which output that should go to stdout and which output that should go to stderr.

6.2.3 Standard input

The input can be read directly from a file using the `<` notation.

For example, we visited the `wc -l` in a previous chapter. The command reads input from a file, and prints the file name together with the number of lines.

```
$ echo "Adding a line" > a_short_file.txt
$ echo "Adding another line" >> a_short_file.txt
$ wc -l a_short_file.txt
2 a_short_file.txt
$ wc -l < a_short_file.txt
2
```

In the first case, the output contained information about both the number of lines, and from which file they came. This is not always desired. To avoid this, the file can be read by the command as a file stream. When running `wc -l < a_short_file.txt`, the command doesn't know from where the lines are coming as it is reading them from the standard input, so it simply outputs the number of lines.

6.3 The pipe

Up until this point we have been exploring what can be accomplished using single UNIX commands. They are useful by themselves, but they really start to shine when they are linked together into pipelines. The idea of the pipe is to link the standard output from one command to the standard input of the next. This means that after the first command has processed a file, its output can be fed directly into another command without storing any intermediate stages as files. This provides us with a flexible tool for processing files in any way we want. You will soon see some useful examples, combining the commands we previously have gone through.

You tell UNIX to pipe the standard output into the standard input of the next command by typing a pipe character (`|`) between the two commands.

> On Swedish keyboards, the pipe key is located to right of the left shift key (same key as the `< >` signs), and is typed by pressing `<AltGr> + <pipe key>`.

```
$ echo "This is a line" | wc -w
4
```

Here you see very simple example of a pipe. Here, we use the `echo` command to output "This is a line" to the standard output. This output is then fed to the `wc -m` command

which subsequently counts and outputs the number words in the sentence. We will soon come back to a variety of examples on how to use the pipe.

6.4 Filters

The purpose of filter commands is to make changes to file streams. They are used in the same way as pipes, using the pipe operator (`|`) between the commands. Here, we will introduce the filters `sed` and `tr`.

6.4.1 tr

tr*anslate or delete characters*
Command usage: `tr [-d] <character_set1> (<character_set2>) <file_name>`

The `tr` command acts on single letters, and can either remove them or change them into other letters.

```
$ echo "ATGXXCG" | tr "X" "N"
ATGNNCG
```

In this case, the Xes were changed into Ns. If we instead had wanted to remove them entirely, we could use the following command.

```
$ echo "ATGXXCG" | tr -d "X"
ATGCG
```

We can also use `tr` in combination with the command `rev` (which reverses a string) to get the reversely complementary letters of a nucleotide string. Here, we provided two sets of letters. Note that those needs to be of the same size, and that letters in the first set are replaced by the letter with corresponding position in the second set.

```
$ echo "ATGCG" | tr "ATGC" "TACG" | rev
CGCAT
```

6.4.2 sed

s*tream* ***ed****itor*
Command usage: `sed <edit_string> <target_file>`

The purpose of `sed` is to edit streams. It is a versatile tool, and can be utilized for a wide variety of text manipulations. Here, we will show how to use it to edit or replace segments of text.

```
$ echo "A line of text." | sed "s/line//"
A  of text.
```

The example above shows how `sed` can be used to remove chunks of text (compared with `tr` which targets single letters). To perform substitutions, `sed` is provided a substitution string. The format of this string is shown in figure 6.2.

```
"s/<pattern to match>/<pattern to replace with>/<options>"
```

Figure 6.2: Format of sed substitution string

g Global match - Replace more than one hit.

i Case insensitive match

Example of substitution string with global and case insensitive matching (note the added "g" and "i"): `"s/target/replacement/gi"`

Figure 6.3: Useful sed options

In the case above, we matched the pattern "line", replaced it with nothing, and we used it without specifying any options. Two useful options are shown in figure 6.3.

Next, we use it to extract the IDs from a GFF file.

example_annotation.gff

```
ctg123 . gene  1000  9000  .  +  .  ID=gene00001;Name=EDEN
ctg123 . mRNA  1050  9000  .  +  .  ID=mRNA00001;Parent=gene00001
ctg123 . exon  1300  1500  .  +  .  Parent=mRNA00003
ctg123 . exon  5000  5500  .  +  .  Parent=mRNA00001,mRNA00002,mRNA00003
ctg123 . exon  7000  9000  .  +  .  Parent=mRNA00001,mRNA00002,mRNA00003
ctg123 . CDS   1201  1500  .  +  0  ID=cds00001;Parent=mRNA00001
ctg123 . CDS   3000  3902  .  +  0  ID=cds00001;Parent=mRNA00001
ctg123 . CDS   5000  5500  .  +  0  ID=cds00001;Parent=mRNA00001
ctg123 . CDS   7000  7600  .  +  0  ID=cds00001;Parent=mRNA00001
ctg123 . CDS   1201  1500  .  +  0  ID=cds00002;Parent=mRNA00002
```

Figure 6.4: Example GFF file

We revisit the example GFF file from the previous chapter (shown in figure 6.4). We might be interested in retrieving the IDs for the coding sequences. There are some different approaches here. In this case, we could get the coding sequence lines using `grep "CDS"`. This might be risky as it could catch lines with "CDS" in other columns. Grepping for lines containing `"ID=cds"` or even `"ID=cds[0-9]\+;"` is likely safer. The meaning of the one is explained in the next chapter on pattern matching.


```
$ grep "ID=cds" example_annotation.gff
ctg123 . CDS   1201  1500  .  +  0  ID=cds00001;Parent=mRNA00001
ctg123 . CDS   3000  3902  .  +  0  ID=cds00001;Parent=mRNA00001
ctg123 . CDS   5000  5500  .  +  0  ID=cds00001;Parent=mRNA00001
ctg123 . CDS   7000  7600  .  +  0  ID=cds00001;Parent=mRNA00001
ctg123 . CDS   1201  1500  .  +  0  ID=cds00002;Parent=mRNA00002
```

Next, we can pipe this into the `cut` command, to get the column with the IDs (column 9).

```
$ grep "ID=cds" example_annotation.gff | cut -f9
ID=cds00001;Parent=mRNA00001
ID=cds00001;Parent=mRNA00001
ID=cds00001;Parent=mRNA00001
ID=cds00001;Parent=mRNA00001
ID=cds00002;Parent=mRNA00002
```

Now, we can use `sed` to remove the parts before and after the CDS-ID.

```
$ grep "ID=cds" example_annotation.gff | cut -f9 \
    | sed "s/ID=//"
cds00001;Parent=mRNA00001
cds00001;Parent=mRNA00001
cds00001;Parent=mRNA00001
cds00001;Parent=mRNA00001
cds00002;Parent=mRNA00002

$ grep "ID=cds" example_annotation.gff | cut -f9 \
    | sed "s/ID=//" | sed "s/;Parent.*//"
cds00001
cds00001
cds00001
cds00001
cds00002
```

For `sed` matches, `.*` is used to match any number of non-line break characters.

6.5 Exercises

In this exercise, we will continue working with the potato genome files seen in previous chapters. Now, we will start using the `pipe`. You will hopefully soon see how useful it can be.

We will continue working with the same real datasets that we used in previous chapters. If you don't have them, you can download them by running:

```
$ wget http://130.235.244.56/unix/tarballs/genome_files.tar.gz
$ wget http://130.235.244.56/unix/tarballs/MhaptRNASeq.fastq.gz
```

If you do have them you could create links to them in the directory you are using for this exercise.

6.5.1 Extracting information from GFF

Let's start by retrieving some further information from the full GFF file.

1. We will extract the number of different features we have in the GFF file again, but now without creating any intermediate files. Let's start by printing part of the file using the `head` command.

   `head representative_genes.gff`

   When you have the entire command in place, we will replace the `head` command with the `cat` command to process the entire file. Using `head` while building the command makes it much easier to see what is going on as we gradually extend it.

2. Make sure that the comments are removed from the output using `grep -v`. At this point, your command will look something like:

   ```
   $ head representative_genes.gff | grep -v "^#"
   ```

   Run the command and inspect it to make sure that you have the correct output.

3. Extend the command with `cut` to get the column containing the gene features. Double check the output.

4. Sort the gene features, and run the command.

5. Print the number of each gene feature using the `uniq -c` command.

6. Finally, replace the `head` with `cat` and run the entire command. Did you get 39028 genes?

6.5.2 Stream editing

In this chapter we introduced two ways of editing file streams: `tr` and `sed`. `tr` is used to translate single characters into other characters or to delete single characters. `sed` can be used to remove or update chunks of text in lines matching a given pattern.

Building a sed pipeline

We want to retrieve the IDs from the GFF file `representative_genes.gff`. As seen before, the lines look like the following:

```
ST4.03ch01 BGI gene 152322 153489 . . . ID=PGSC0003DMG400015133;name="Defensin"
```

We are after the ID part for each line. In this particular case, we would like to extract `PGSC0003DMG400015133`. There are some different was to approach this. Here, we will use the `sed` command to match and remove text before and after the ID.

When substituting text with `sed` the syntax looks like the following:

```
$ sed "s/<target text>/<replacement>/"
```

When using `sed` we can use the pattern `.*` to match an arbitrary number of (non-linebreak) characters.

1. Start by setting up a basic pipeline to print the content of the GFF-file into the `sed` command. At first, your pipeline might look something like the following:

   ```
   $ head representative_genes.gff | grep -v "^#" | \
       sed "s/.*ID=/replaced /"
   ```

   The first ten lines are printed and comment lines starting with `#` are removed. Remaining lines are fed into `sed`. The `sed` is given the following string as argument:

   ```
   s/.*ID=/replaced /
   ```

   The `s` tells `sed` that we want to do a substitution of the matched pattern. The pattern between the first and second slash ".*ID=" is the part that is matched. In this case, this will match any stretches of text in a line ending with "ID=". The final pattern between the second and the third slash "replaced " is what will be inserted instead of the matched pattern.

2. Run the command and take a look at the output. Do you see what is going on?

3. Now finish off the command. Instead of replacing the lines with the text "replaced ", remove them. Match and remove the part of the line trailing the ID by extending the pipe with another `sed` command matching the text coming after the pattern. In the end you should have a single column containing 354788 IDs.

4. (*) Update the pipeline to calculate the exact number of the four different ID types you have in your data. Did you get 141037 exon IDs?

6.5.3 Useful pipes

Here, we will present some useful combinations of commands for working with GFF-, FASTA- and FASTQ-files.

1. Count the number of nucleotides in a FASTA file. Note that line endings counts as one character. They can be removed using the command `tr -d "\n"`. Below, you see the first part of the command.

   ```
   $ head representative_cds.fasta | grep -v "^>"
   ```

   Build this command step by step by adding one command at a time and try to understand what happens with the output. We use the inverse `grep` here (the `-v` flag) to match non-header sequence content. Remove the line endings and count the nucleotides.

   Did you get 36177057 nucleotides for the full sequence file? Does it matter whether the FASTA is in single- or multi-line format when running this command?

2. The number of genes in the GFF file should be the same as the number of genes in the FASTA file. Let's investigate if that is the case.

   We could easily get the number of coding sequences with a single `grep` command. Get that number. Compare it to the number of genes we have found the GFF file. Are they the same?

   Does the counts match? Actually, they don't. Something is going on here. This is investigated in the final 'case exercise' for this chapter.

3. The FASTA file `representative_cds.fasta` claims to contain coding sequences. This would mean that they should start with `ATG` - the start codon. We can investigate this using a pipe.

   Let's get the `representative_cds.fasta` in single-line format.

   ```
   $ seqtk seq -l0 representative_cds.fasta \
       > representative_cds.single.fasta
   ```

   Now we are ready to start building the pipe. We can get all the sequence lines by using `grep`.

   ```
   $ head representative_cds.single.fasta | grep -v "^>"
   ```

   Look at the output. What do we have at this point?

   Next, `cut` the first three nucleotides for each sequence.

```
$ head representative_cds.single.fasta | grep -v "^>" \
    | cut -c 1-3
```

Check the output. Finally, sort the output and get the counts for each codon.

```
$ head representative_cds.single.fasta | grep -v "^>" \
    | cut -c 1-3 | sort | uniq -c
```

Now we are ready to try it for the whole file. This final output might be easier to study if we add a second `sort` command after the `uniq -c` command. Compare how it is sorted if you sort this output without and with the `-n` flag.

What conclusions can you draw from this output? Does all the coding sequences start with the `ATG` codon?

4. * We have previously been extracting parts of lines using the `cut` command. The `sed` command provides a higher flexibility, and is commonly used to remove or edit part of the lines.

   Your assignment here is to extract a three-column file from the GFF file containing genes together with their chromosome sequence ID from the GFF file and their assigned annotation. The three column file will have rows similar to the following:

   `ST4.03ch03 PGSC0003DMG400013378 Metalloendopeptidase`

   You can do this by using a combination of `grep`, `cut` and `sed`. Make sure that you have the correct number of genes in your final output.

   Hint: If you just wanted to get the first ID for each entry in a GFF file, you could start out by doing something similar to the following:

```
$ head representative_genes.gff | grep -v "^#" \
    | cut -f9 | sed "s/ID=//" | sed "s/;.*//"
```

6.5.4 The gene-count mystery (**)

In the previous exercise we counted the number of genes in the GFF file and the number of coding sequences in the FASTA file. The coding sequences are said to be representative for genes - there should be a single coding sequence per gene. So, what are those extra genes? Are they extra genes, or are some missing from one of the files? This is something that needs to be understood before we do further analysis.

To complicate things further our FASTA have two IDs for coding sequences and transcripts, while the genes in the GFF file have gene IDs. Fortunately, we have the mapping matrix which could be used to figure out to which gene the coding sequences belong. Note that the IDs correspond to genes, coding sequences, peptides and transcripts, and can be distinguished by the letter after "DM".

Our goal is to find the difference in represented genes between the GFF file and the FASTA file. We also want to get the annotation and coding sequences for those genes. We have outlined a possible approach below, but if you can think of other approaches, feel free to try them.

1. * Start with outlining what the problem is. Which files do we have? What is the content of each file? What end result do we want to get? One approach could be to extract two gene lists containing all gene IDs represented in the FASTA and the GFF, which we then can compare and see which IDs that are missing.

2. * Start with extracting the list of gene IDs for the GFF file. The number of gene IDs should correspond to the number of genes we found in previous exercise.

3. * Next, we want to get the gene IDs for the coding sequences. To do this, we need to translate the coding sequence IDs to their gene IDs. A useful starting point could be to create a file containing only the coding sequence IDs. This could then be used to extract corresponding gene IDs from the mapping file. (This step could take up to 10-15 minutes for the computer to calculate, so make sure that it is working for a smaller dataset before running the whole file).

4. * Double check the number of IDs you have. Do they correspond to the number of GFF-file-genes and FASTA coding sequence entries? Check if you have any duplicates in any of the files.

5. * Now, lets prepare the two files so that we can compare and check for diverging IDs. At this point, you should be able to find the differing genes.

6. * Finally, extract the annotations for those entries. Also, retrieve their coding sequences.

7. * Can you draw any conclusions from this? It this a problem for the downstream analysis? What could the reason for this difference be? Feel free to discuss this with the teachers.

6.6 Checkpoint

Before you continue, make sure that you can answer the following:

- What is the purpose of the three different types of file streams: Standard input, standard output and standard error?
- How is piping useful for bioinformatic analyses?

6.6.1 UNIX commands

Consider each command for a second. Make sure that you have an idea about what they do before moving on.

Introduction to UNIX

```
man
ssh [-v]
```

Introduction to the file system

```
cd
ls [-l] [-lh]
pwd
```

Working with files in UNIX

```
cat
cp [-i]
file
head [-number]
less
mkdir
mv [-i]
nano
rm [-i] [-r]
rmdir
tail [-number]
```

Working with bioinformatic data

```
cut [-d] -f/-c
diff
grep [-c] [-A] [-v] [-f] [-i]
sort [-n] [-r]
uniq [-c]
wc [-l] [-w] [-m]
```

Organizing files

```
chmod
gunzip
gzip
ln [-s]
scp
tar [-cvzf] [-xvzf]
wget
```

File streams

```
echo
sed
tr [-d]
```


6.7 Further reading

6.7.1 The tee command

In this chapter we showed how the output from programs can be redirected to different locations. But what if you want the output both printed to the terminal and to a file? One way is by using the `tee` command, which allows you to write the output to a file while also passing the stream on to the standard output. See the following link for some examples.

http://linux.101hacks.com/unix/tee-command-examples/

6.7.2 awk

A common task when working with GFF files is to match lines based on a specific column. This is not trivial to do only using `grep`. A nicer way of doing this is by using the command line tool `awk`. `awk` is a highly powerful tool for working with file streams. If a task is hard to do with `grep`, `awk` is often a good choice. Or, as a wise man said:

```
"The Enlightened Ones say that....

You should never use C if you can do it with a script;
You should never use a script if you can do it with awk;
Never use awk if you can do it with sed;
Never use sed if you can do it with grep."
```

For example, if you want to print lines in a GFF file for which the third column is `gene`, you could do this by simply using the following awk-line:

```
awk '$3 == "gene"' my_file.gff
```

For a number of useful `awk` and `sed` examples, check the following link:

https://github.com/stephenturner/oneliners#awk--sed-for-bioinformatics

Chapter 7

Pattern matching, variables, subshells and loops

In this chapter we will take a closer look at some more conceptually challenging parts of UNIX. We will start using using basic programming concepts which allows us to automate the analysis and to process multiple files at once. What we gain from this is a powerful toolset allowing us to easily process dozens or more files at once.

When will this be particularly useful? When working with sequence data, you will frequently work with multiple samples in parallel. You need to do quality control and trimming for all your twenty samples. Instead of running twenty separate commands, which is both tedious and error prone, you can process all twenty samples in a single command. By knowing how to use these concepts, you can let the computer do most of the work for you.

7.1 Pattern matching

We provide specific files or strings to most of the UNIX commands. Using *pattern matching* (also called *regular expressions*) the command can match a group of files or strings. We have already been doing this for the `ls` command where we have listed files matching patterns using the `*`-sign.

We are able to customize our patterns in a variety of ways. For example, figure 7.1 shows how we can use square brackets to match one of multiple alternatives for a character in a string.

7.1.1 Pattern matching UNIX paths

We mentioned that we already have used the `*`-pattern for matching any number of any non-line break character in UNIX file paths. In figure 7.2, other available patterns for matching file names are shown.

The square brackets (`[]`) are used to match one out of a set of characters, or a range of characters (`[0-9]` matches zero to nine).

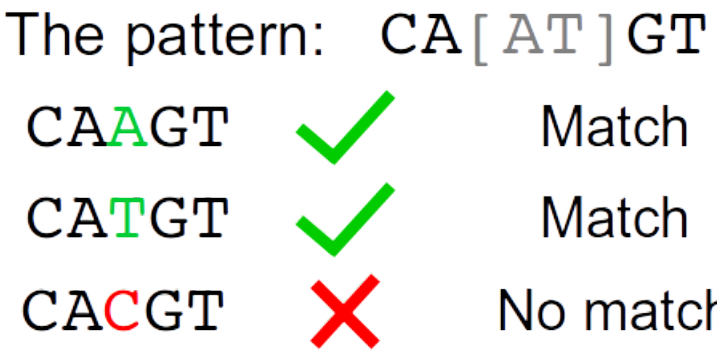

Figure 7.1: Matching a string which could be either 'A' or 'T' in position 3

Note

Unfortunately the syntax for the pattern matching differs slightly between different programs. Here, we are learning how to match file paths in UNIX. When you will try matching in another context another set of symbols will be used.

`?` Matches any single character except newline

`*` Match any number of characters (zero or more)

`{}` Can match one of several comma-delimited words

`[]` Match *one* of the enclosed characters

`[0-9]` Match any number

`[a-z]` Match any lower case character

`[a-zA-Z]` Match any character

\ Turn off the special meaning of the following character

Figure 7.2: Selection of UNIX path patterns

```
# Listing all files in directory
$ ls
sample1.fa sample2.fa sample3.fa sampleA1.fa README.txt

# Listing all sample files in directory
# The asterisk matches any number of characters
$ ls sample*
sample1.fa sample2.fa sample3.fa sampleA1.fa
```

```
# Listing sample files where the name end with 1 or 2
$ ls sample[12].fa
sample1.fa sample2.fa

# Listing sample files where the name end in a number
$ ls sample[0-9].fa
sample1.fa sample2.fa sample3.fa
```

The question mark can be used to match any single character.

```
# Listing all files in directory
$ ls
sample1.fa sample2.fa sample3.fa sampleA1.fa README.txt

# Listing sample files ending with 'A',
# followed by any character or number
$ ls sampleA?.fa
sampleA1.fa
```

The curly brackets ({}) can be used to match one of several words. It can also be useful for creating multiple files or directories in one command.

```
$ ls
sample11.fa sample12.fa sample21.fa sample22.fa
$ ls sample{11,22}.fa
sample11.fa sample22.fa
```

```
$ mkdir {analysis,results}_run1
$ ls
analysis_run1 results_run1
```

7.1.2 Using pattern matching with grep

Pattern matching in `grep` can be used any time you want to extract subsets from files based on patterns rather than exact strings.

For example, if you have different isoforms of genes in your FASTA file, where the different isoforms for a particular gene are distinguished with a "_i1", "_i2"... suffix, and you only are interested in the first isoform, those can be extracted in the following way (the `genes_with_isoforms.fa` file is shown in figure 7.4):

`.` Match any *single* character except newline

`*` Match any number of the *preceding* pattern

`^` Match the following expression at beginning of line

`$` Match the preceding regular expression at end of line

`\+` Match the *preceding* pattern one or more times

`[]` Match *one* of the enclosed characters

`[0-9]` Match any single digit

`[a-z]` Match any single lower case character

`[a-zA-Z]` Match any character

`\` Turn of the special meaning of the following character

Figure 7.3: Selection of grep regex

genes_with_isoforms.fa

```
>geneA_i1
ATCGATCGATCG
>geneA_i2
ATCGATCGATGACTCG
>geneB_i1
ATCGATCGATCGAAA
>geneB_i2
ATATGCTAGCCGATCGATCG
>geneB_i3
ATATATTAGCGA
```

Figure 7.4: Example FASTA containing genes with isoforms

```
$ grep -A1 ">.*_i1" genes_with_isoforms.fa
>geneA_i1
ATCGATCGATCG
>geneB_i1
ATCGATCGATCGAAA
```

In a previous chapter we mentioned the `"ID=cds[0-9]\+;"` that could be used to match lines in a GFF file containing coding sequence IDs. This pattern would match the exact

letters `ID=cds`, followed by one or more numbers which finally should be ended by a semicolon (`;`).

This is only a subset of the possible pattern matches. If you need to match a particular pattern - there is probably a way to do it.

7.2 Variables

7.2.1 What is a 'variable'?

A variable is like a labeled box which you can put different things in. The label makes the box easy to refer to. The things you put in the box can for instance be text, numbers or even file paths. Figure 7.5 shows a box labeled `my_variable`. The box contains the string of nucleotides: `AAGTGTACGT...` We can now talk about the box containing the nucleotides, instead of keeping track of the exact nucleotide sequence.

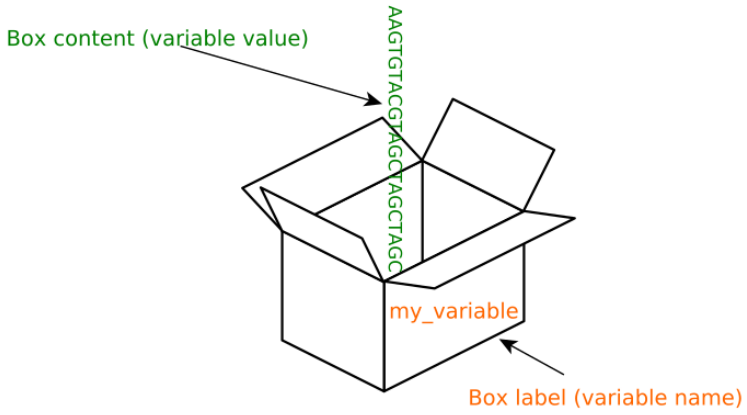

Figure 7.5: Box representing a variable containing a value were the box is labelled "my_variable" and contains a string of nucleotides

A powerful thing with using variables is that we easily can swap its values while retaining the same label. In our example here, we could swap the nucleotide sequence in the box to another sequence, for instance `CCCCCCCC...`. The label of the box would be the same (`my_variable`), but the content has changed (see figure 7.6).

This will prove to be very useful when we want to go through a number of file names. By using this, we can avoid typing out the full filename for each command, and can instead use the label while changing the content of the box.

7.2.2 Variables in UNIX

A variable in UNIX is a way of linking a string of text (the label) to a value (the content of the box). Variables are defined by writing the name of the variable, directly followed by an equals sign and the value (no white spaces are allowed around the equals sign). We retrieve the value of the variable either by simply typing a dollar sign followed by the variable name

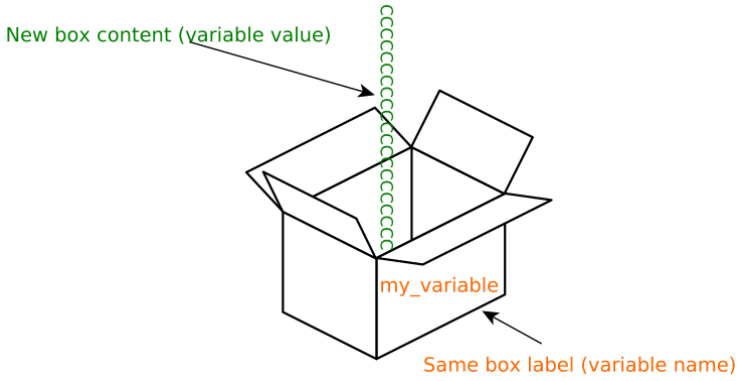

Figure 7.6: Same box as in previous figure, with the same label, but the content has been changed

Note

A common problem when starting out with variables is to mix up the variable *name* (the label of the box) and what the variable *refers* to (what is in the box). This could lead to cases were we are trying to do things with 'variablename1' and 'variablename2' when we actually are interested in the content of the variables.

`$myvar` or using the full syntax and also type out curly braces around the variable name `${myvar}`.

```
# Let's create the variable from our box example
# Here, the name of the variable is "my_variable"
# The content of the variable is "AAGTGT..."
$ my_variable="AAGTGTACGTAGCTAGCTAGC"

# We retrieve the value stored in the variable labeled
# 'my_variable' by prepending the dollar sign and surrounding
# the name with curly braces
$ echo ${my_variable}
AAGTGTACGTAGCTAGCTAGC

# If we not prepend the dollar sign,
# my_variable is used as a regular string
$ echo my_variable
my_variable

# We can update the content of the variable
$ my_variable="CCCCCCCCCCCCCCCCCCCC"

# Now, we get a different output
$ echo ${my_variable}
CCCCCCCCCCCCCCCCCCCC

# We can embed the variables in strings
$ echo "My nucleotide sequence: ${my_variable}"
My nucleotide sequence: CCCCCCCCCCCCCCCCCCCC
```

Note

It is considered good practice to expand the variable using the full `${variable}` syntax rather than the shorter `$variable`. The shorter syntax is mostly fine, but will in some cases fail. For example when expanding "`${variable}moretext`" or "`$variablemoretext`" the first example will get the correct value from the variable, while the second will attempt to try to get the value from the variable "variablemoretext". There are also special cases (mentioned in the "further reading") where the full syntax is required.

A variable can store a value for later reuse. This is both useful if you are using a particular value repeatedly as it gives a short and nice way of referring to it, or if you are generating multiple values which you want to use in a similar context (for example: performing a particular operation on multiple sample files). You will see examples of both cases in the section about loops, and the coming chapter about scripts.

One useful way of using variables is to use them to store long paths.

```
# Typing out full paths can be tedious
$ head ../other_analysis/first_experiment/data/raw_file1.fa
... first ten lines of raw_file1.fa
$ head ../other_analysis/first_experiment/data/raw_file2.fa
... first ten lines of raw_file2.fa

# A variable can save us some sanity here
$ data_path=../other_analysis/first_example/data/
$ head ${data_path}/raw_file1.fa
... first ten lines of raw_file1.fa
$ head ${data_path}/raw_file2.fa
... first ten lines of raw_file2.fa
```

7.3 Subshells

Subshells allows us to run multiple commands within one command. It also allows us to save the output from a command directly into a variable.

There are two ways to write subshells:

- Parenthesis syntax: `$(command)`
- Back-tick syntax: `‘command‘`

It is recommended to use the parenthesis-syntax. This allows for nested subshells (subshells within subshells). In the example below we capture the output from a subshell in the variable `entry_count`. Or in other word, we take the output from the command and store the output in the box labeled `entry_count`.

```
$ grep -c "^>" nucleotides.fa
173
$ entry_count=$(grep -c "^>" nucleotides.fa)
$ echo "My count is: ${entry_count} entries"
My count is: 173 entries
```

This way we can store and reuse output from the command performed in the subshell.

7.4 Loops

By using loops we can execute one or or more commands for a collection of data such as a number of files, a number of lines in a file or a number of words in a string.

You often have multiple samples which you want to analyse. Instead of typing out the commands for every single file, you can automate this using loops. This spares you a lot of typing, improves the reproducibility and reduces the risk for typing errors.

Figure 7.7 illustrates this. We walk through four files, and put their paths one and one in our variable `current_file`. We can then process each of them by itself using any desired command. In this case the command `Process $current_file` will be executed four times. One time per file.

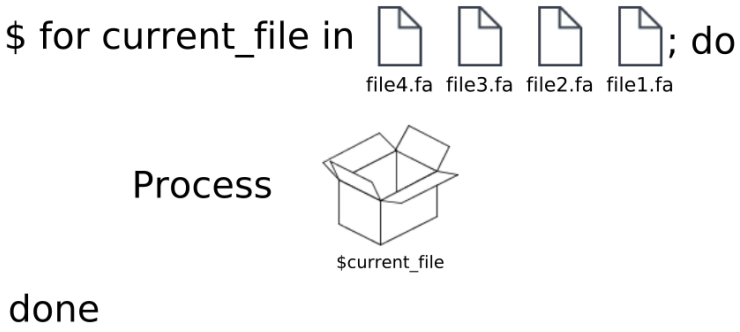

Figure 7.7: Illustration of looping over and processing four files. File names will be placed one by one in the box.

Make sure that you have a good understanding of pattern matching, variables and subshells before moving on. The next part will use them all.

7.4.1 Looping over a set of files

A common way of determining the set of files to loop over is by using pattern matching. For example, if we want to perform a certain set of operations on all files in a directory, we can use *-wild card matching together with the desired commands in a for loop.

```
$ ls
file1.txt file2.txt a_file.txt a_file2.txt

$ for file in *; do echo ${file}; done
file1.txt
file2.txt
a_file.txt
a_file2.txt

$ for file in *; do echo "The current file is: ${file}"; done
The current file is: file1.txt
The current file is: file2.txt
The current file is: a_file.txt
The current file is: a_file2.txt
```

There are several things going on here. A break-down of how the loop is written is found in figure 7.8 with further explanations in figure 7.9.

In the next example, a for loop is used to give an overview of the content for four FASTA files. Here, the number of entries were calculated, but any of the commands you learned can

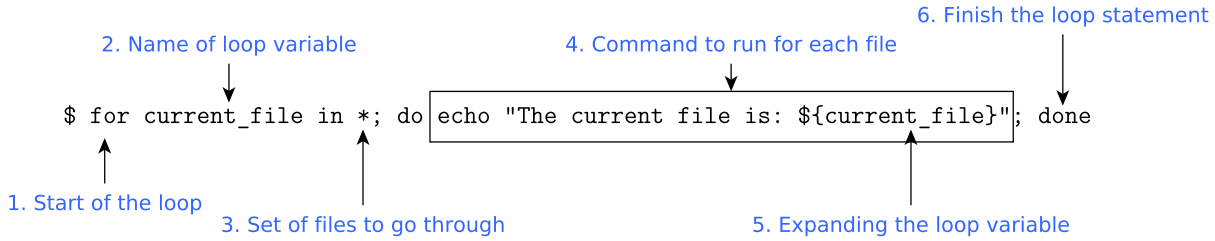

Figure 7.8: The different parts of a for-loop

1. The `for` statement, starting the loop
2. The name of the loop variable which will get the values you are looping over (here, `"current_file"`)
3. Pattern matching the files you want to loop over - **Followed by a semi-colon and the "do" statement**
4. One or more commands to run for each file - **Each ending with a semi-colon**
5. The use of the loop variable, which will contain the name of the file it is currently looping past
6. The `done` statement ends the loop statement

Figure 7.9: A breakdown of the different components in figure 7.8

be used here.

```
$ ls
nucl1.fa nucl2.fa nucl3.fa nucl4.fa

$ for f in *; do count=$(grep -c "^>" ${f}); \
    echo "${f}: ${count} entries"; done
nucl1.fa: 173 entries
nucl2.fa: 215 entries
nucl3.fa: 98 entries
nucl4.fa: 133 entries
```

In this loop two commands are run per iteration. (Are you keeping track of the variables and subshells?)

- `count=$(grep -c "^>" ${f})` uses a subshell to retrieve the number of entries present in the fasta file and store this number in the variable `count`.
- `echo "${f}: ${count} entries"` prints the file name together with the number of

entries found within it.

7.5 Exercises

Here, you will get acquainted with the following concepts:

- Pattern matching
- Variables
- Subshells
- Loops

It can take some time to get used to these concepts. But it is time well spent. These concepts can reduce time and effort for your processing while increasing reproducibility. When you grasp them, you can get a lot done with very few commands.

7.5.1 Variables and subshells

To assign a variable, we type the name of the variable followed by an equals sign and the content of the variable.

```
$ my_variable="Content of the variable"
```

To retrieve the value of a variable we put a dollar sign in front of it, and surround it with curly braces.

```
$ echo ${my_variable}
"Content of the variable"
```

1. Store the text "Hello world" in a variable called "hello_world_var". Use the command `echo` to print the content of the variable.
2. Now, let's use a variable to bookmark your current location in the file system. Retrieve the path by running the `pwd` command in a subshell, and create a variable where you assign the path as value.

   ```
   $ bookmarked_path=$(pwd)
   ```

   The variable should now contain your current path. Check it with the `echo` command.
3. Use the variable to list the files in the bookmarked folder. This could be done by running:

   ```
   $ ls ${bookmarked_path}
   ```

   Did it work?

4. Go to another directory and see whether you can use the `cd` command with your variable to go directly to the bookmarked folder.

5. (*) Use a subshell to calculate the number of FASTA entries in `representative_cds.fasta` and save the number to a variable. Save the number of nucleotides in another variable. Then you should be able to print a string with information about the file and its content like the following:

```
$ echo "File contains ${lines} lines and ${nucl} nucleotides"
```

7.5.2 Loops and pattern matching

Download the tar-ball for this exercise.

```
$ wget http://130.235.244.56/unix/tarballs/unix_course_chapter7.tar.gz
```

Inside it you will find a set of ten FASTA files, each containing one hundred FASTA entries. We will work with all of those ten files at once. We will start out by checking how they are structured.

1. For some commands, we are able to get information about all ten files without using loops. First, get the number of FASTA entries for all the files using `grep` and the `*` pattern instead of the file path. The `*` can be used to match any number of any characters in Bash.

   ```
   $ grep -c "^>" *
   ```

   Try the same thing with the `wc` command. Which of the subsets contains the most data? Why is the number of lines the same, but the number of characters in the files different?

2. Next, we want to calculate the total number of nucleotides in each FASTA. In this case we need to change approach. Try what you get if attempting the following:

   ```
   $ grep -v "^>" * | tr -d "\n" | wc -m
   ```

   Here, we attempted to get the nucleotides for all the files similarly to how we did in the stream chapter. In this case, the pattern matching will not give us output for each single file. The `tr -d` and `wc -m` will see their input as a single stream, and calculates the total number of nucleotides.

3. In order to calculate information for the individual files we will use a for-loop. Take a look at the illustration of a for-loop earlier in this chapter. Do you understand the different parts? We will now build this loop step by step.

4. First specify what you want to loop over. Use a pattern which matches all the FASTA files. Also, specify the name of the variable which will get the values from the different files. End the start statement with `;` and `do`. At this point, our statement looks something like the following:

   ```
   for target_fs in *; do
   ```

5. Next, we will specify what the loop should do, and then terminate it with the `done` statement. As a start, let's just print the file names that we loop over. Do this by entering an echo statement with the `target_fasta` as argument and end the statement with `;` and `done`:

   ```
   for target_fs in *; do echo ${target_fs}; done
   ```

   This loop should be working. Try it!

6. Can you adjust the pattern so that only the first three files are listed? Can you adjust it so that the files with numbers 3, 6, 8 are listed?

7. Create another file in the same directory with another file ending, for example `myfile.txt` Now echo all FASTA files, while excluding the `.txt` file.

8. We can run multiple commands for each step in the loop. If we would like to have two separate `echo` statements, we could do the following:

   ```
   $ for target_fs in *; do echo "New file!"; \
       echo ${target_fs}; done
   ```

   Each command is ended by a semi-colon.

7.5.3 Working with multiple files at once

Your goal is to build a loop steps over a given set of files, calculating the number of nucleotides present in each of them and prints a nice descriptive line together with the number. This principle could be extended to any set of files and to any commands. Needless to say, this is frequently very useful.

There are two parts here. Putting together the loop and putting together the needed commands.

1. Let's print the number of nucleotides in each file. We can calculate this for a single file by running the command:

   ```
   $ grep -v "^>" myfile.fasta | tr -d "\n" | wc -m
   ```

Put this command into a subshell, and save the variable. Try it out on a FASTA file and run `echo` for the content of the variable to verify that it works.

2. We can now print this value inside a string:

```
$ echo "There is a variable in my string: ${myvar}"
```

Run a command printing `"The file contains <your value> nucleotides"`, and insert the number of nucleotides you received into the string.

3. The loop part is very similar to the one used in the previous exercise. We want to match all FASTA files present in the directory. To match files with the file ending `.fa` we can use the pattern `*.fa` Put together a loop printing the names of each FASTA file in the directory.

4. Now, we have all pieces we need to finish the loop. What you need is:

   - The `for` loop iterating over the FASTA files in the directory.
   - A subshell command retrieving the number of nucleotides present in the current FASTA file.
   - An `echo` command which prints a string with inserted variables for sample name and number of nucleotides calculated by the subshell.

   After running this loop, I got the following output:

```
File: subset10.fa has 91659 nucleotides
File: subset1.fa has 93099 nucleotides
File: subset2.fa has 83619 nucleotides
File: subset3.fa has 95505 nucleotides
File: subset4.fa has 94431 nucleotides
File: subset5.fa has 85503 nucleotides
File: subset6.fa has 98643 nucleotides
File: subset7.fa has 86391 nucleotides
File: subset8.fa has 87846 nucleotides
File: subset9.fa has 95838 nucleotides
```

7.5.4 Processing multiple files (**)

If you have limited time - go through chapter 8 before wrestling with this exercise.

You likely want to use loops any time where you are working with several samples. This is also true when using terminal-based bioinformatic software. Here, we will show an example with the program `seqtk`, but this applies for any software.

Remember how we converted multi-line FASTA files to single-line format? We used the command:

```
seqtk seq -l 0 multi_line.fa > single_line.fa
```

We can use the same command to get multi-line format by adjusting the `-l 0` argument to a particular width (for example `-l 60`). You task is to reformat the ten FASTAs. The biologist asking for the data have some demands:

- The data should be in a single file, containing all the sequences from the ten samples.
- You should be able to distinguish from which sample the different sequences came. This is best done by creating a new set of files where you have added annotation to the header lines.
- The data should be in multi-line format.

Think about the problem for a while. What changes do you need to make to each FASTA? Which step or steps is best done as a part of a loop? Are some steps easier to do before or after combining the different samples? If you get stuck on this one, ask the teachers and you will get some hints.

One hint - If you want to append text to the start of the header lines, you could use the `sed` command, matching the `>` signs and replace them with `"> ${filename}`. One example is shown below (you will need to adjust it to your case).

```
$ loopvar=fasta1.fa
$ cat ${loopvar} | sed "s/>/>${loopvar}/"
```

After delivering the data you got an additional request. Could you provide a subset of the data only containing sequences from sample 2, 5, and 8?

7.6 Checkpoint

Before you continue, make sure that you can answer the following:

- When can we use the two types of pattern matching introduced here? Can you name useful examples in both cases?
- What are variables, and what can they be used for?
- When can subshells be useful?
- How can loops help both reducing error rate and workload in your processing?

7.6.1 UNIX commands

Consider each command for a second. Make sure that you have an idea about what they do before moving on.

Introduction to UNIX

```
man
ssh [-v]
```

Introduction to the file system

```
cd
ls [-l] [-lh]
pwd
```

Working with files in UNIX

```
cat
cp [-i]
file
head [-number]
less
mkdir
mv [-i]
nano
rm [-i] [-r]
rmdir
tail [-number]
```

Working with bioinformatic data

```
cut [-d] -f/-c
diff
grep [-c] [-A] [-v] [-f] [-i]
sort [-n] [-r]
uniq [-c]
wc [-l] [-w] [-m]
```

Organizing files

```
chmod
gunzip
gzip
ln [-s]
scp
tar [-cvzf] [-xvzf]
wget
```

File streams

```
echo
sed
tr [-d]
```

Pattern matching, variables, subshells and loops

7.7 Further reading

7.7.1 Variable expansion

A useful way of manipulating strings in UNIX is the so called *variable expansion*. When retrieving the value for a variable using the `${}` syntax, the retrieved value can be edited using various variable expansions. You can for example easily remove suffixes or file paths by retrieving your value as follows:

```
$ my_file=/home/jakob/Desktop/nucleotides.single.edit.fa

$ echo ${my_file}
/home/jakob/Desktop/nucleotides.single.edit.fa

$ echo ${my_file%.*}
/home/jakob/Desktop/nucleotides.single.edit

$ echo ${my_file%%.*}
/home/jakob/Desktop/nucleotides

$ echo ${my_file##*/}
nucleotides.single.edit.fa
```

The `%` matches a pattern to remove at the end of the file name, while `%%` matches an arbitrary number of time. The `#` does the same for the start of the file name. The syntax is a bit messy, but it is a highly usable tool. Further examples (and explanation of the syntax) can be found at the following link:

https://www.gnu.org/software/bash/manual/html_node/Shell-Parameter-Expansion.html

7.7.2 Regular expressions

You have seen two types of pattern matching/regular expressions used by UNIX in this chapter. There are even more types of regular expression out in the world. When running `grep` together with the `-P` flag you get an even more powerful type of regular expression - The Perl regular expressions. Those are commonly used in more modern languages than the one used in UNIX. An overview of available patterns are found in the following link:

http://regexlib.com/CheatSheet.aspx

Chapter 8

Introduction to scripting

8.1 Bash and scripting

Bash is a programming language, and the most common language used in the UNIX terminals. It is also the type of UNIX terminal that we have been using in this course.

Here, we will start scratching the surface of scripting - putting together small computer programs to run sequences of commands. Scripting is not necessarily as intimidating as it might sound. Building scripts and programs (scripts are in principle small programs) *can* be a highly complex task involving large teams of people, but could as well just be putting together a bunch UNIX commands for later reuse.

Two main advantages of collecting commands into scripts are:

- Reproducibility - If you gather the commands you have been running to process a specific set of data in a script, you (or someone else) will easily be able to re-run the exact same steps at a later point.
- Creating your own tools - Often you find yourself re-running the same commands to process data in different contexts. You can gather those commands into a script, allowing you to perform the same task in a single command by invoking your script.

Scripts are commonly used on computational clusters (like UPPMAX) by their queue systems which use them to run your commands at a later time.

This chapter is a starting point for scripting and programming. It is useful already at this level. But, if you enjoy this, there is a lot more to learn. If you are interested in learning more, take a look in the "Further reading" for this chapter.

8.2 Building a simple Bash script

There is a long going tradition of writing a program that prints "Hello world" as your first program when encountering a new language. In figure 8.1, you see "Hello world" as a Bash script.

The script is shown in figure 8.1. (`sh` is a common file ending for this kind of scripts, and comes from the term ***sh****ell*, which is another name for the terminal). There are two lines in the script. The second one is a simple `echo` command, similar to what we have been using before. The first line tells the script which *interpreter* that the computer should use to read the script. The line starts with `#!`, followed by the *absolute path* to the interpreter which the computer uses to read Bash programs. The initial `#!` is the same for all scripts, while the interpreter is different for each computer language.

hello_world.sh

```
#!/bin/bash
echo "Hello world :)"
```

Figure 8.1: The classical script "hello world" in Bash

In order to run the script we must first set the appropriate permissions - We need to have read (r) and execution (x) rights for the script. Then, we can simply type out the path of the script to run it. Note - If the script is in the present working directory, the path is specified using the current directory sign (`.`).

```
$ ls -l
-rw-rw-r-- 1 jakob jakob 33 apr 10 08:11 hello_world.sh

$ ./hello_world.sh
bash: ./hello_world.sh: Permission denied

$ chmod +x hello_world.sh

$ ./hello_world.sh
Hello world  :)
```

The `.sh`-suffix is commonly used for scripts written in Bash or other terminal languages. So, there are some new things here to keep track of here. They are listed in 8.2.

8.2.1 Comments in Bash scripts

Sometimes, we want to have text in our scripts which isn't interpreted as commands. This can be done by adding a hash-sign (`#`) in front of the text. A commented version of the Hello world-script is seen in figure 8.3. Note that the output from this version of the script is identical to the script we saw in figure 8.1.

```
$ chmod +x hello_world_commented.sh
$ ./hello_world_commented.sh
Hello world :)
```

1. The script consists of an initial line starting with `#!` and the path to the interpreter (`#!/bin/bash`).
2. After the initial line, the script contains one or more Bash-commands (`echo "Hello world :)"`).
3. To run it, the script needs to have read and write permissions (`chmod +x script` and when read permission isn't set: `chmod +r script`).
4. To run it, the path to the script is used. If in the current directory, this is specified by `./script`

Figure 8.2: Building and running a Bash script

hello_world_commented.sh

```
#!/bin/bash

# This script prints "hello world :)"
# These two lines are only here to describe the script
echo "Hello world :)"
```

Figure 8.3: "Hello world" - with comments

8.3 Gathering processing steps in a script

We often need to perform a sequence of commands to process our data. Those commands can be UNIX-commands, or different bioinformatic programs. If those commands are gathered in scripts, this means that you - and other persons - are able to re-run the exact same steps at a later point.

In figure 8.4 we have a script retrieving information about the content of a specific FASTA file. All this information can then be retrieved and printed by running the script.

investigating_nucleotides.sh

```
#!/bin/bash

# "nucleotides.fa" must be in the same directory as this script

echo "Calculating information about nucleotides.fa"

echo "Number of entries"
grep -c "^>" nucleotides.fa

echo "Number of lines"
wc -l < nucleotides.fa

echo "Number of nucleotides"
grep -v "^>" nucleotides.fa | tr -d "\n" | wc -m
```

Figure 8.4: Script calculating information about a FASTA-file named ”nucleotides.fa”

```
$ ls
investigating_nucleotides.sh nucleotides.fa
$ chmod +x investigating_nucleotides.sh
$ ./investigating_nucleotides.sh
Calculating information about nucleotides.fa
Number of entries
14956
Number of lines
29912
Number of nucleotides
747800000
```

Now we can quickly re-run our analysis at a later point.

8.4 Providing input to a script

In the previous section you saw a way to gather processing steps in a script to easily reproduce the analysis steps.

In some cases our analyses are specific for one single project, and will only be reproduced for that particular case. In other cases, the steps will be repeated many times in different projects - Similarly to how we have been using commands like `grep`, `sort` and `cut` in many different contexts.

It turns out that we with a few changes can take the script in figure 8.4 and generalize it so that it could calculate information about any FASTA file. The only thing we need to

change is making it take an arbitrary file as *input argument*.

Inside the script, there is a special variable called `argv`, which contains all arguments that you have provided to the command line when running it in the terminal. For example, if you have a script called `my_script.sh`, and run it as shown below, we are able to retrieve the provided value `additional_text` from the argv-variable within the script.

```
$ ./my_script.sh additional_text
```

The `argv` values can be accessed within the script by typing `${number}` with `number` replaced by the position of the argument. In the previous case, `${0}` will give `./my_script.sh` and `${1}` will give `additional_text`.

Figure 8.5 shows how our previous script could be generalized to calculate statistics about any provided FASTA file.

fasta_stats.sh

```
#!/bin/bash

# Read an arbitrary FASTA file through argv

echo "Calculating information about ${1}"

echo "Number of entries"
grep -c "^>" ${1}

echo "Number of lines"
wc -l < ${1}

echo "Number of nucleotides"
grep -v "^>" ${1} | tr -d "\n" | wc -m
```

Figure 8.5: Script calculating information about any FASTA provided on command line

Now, we can use this general script to calculate information about any FASTA file.

```
$ ls
fasta_stats.sh nucleotides.fa other_nucleotides.fa

$ chmod +x fasta_stats.sh

$ ./fasta_stats.sh nucleotides.fa
Calculating information about nucleotides.fa
Number of entries
14956
Number of lines
29912
Number of nucleotides
747800000

$ ./fasta_stats.sh other_nucleotides.fa
Calculating information about other_nucleotides.fa
Number of entries
6672
Number of lines
13344
Number of nucleotides
333600000
```

Hopefully this has provided a glimpse of the usefulness of Bash scripts. We will explore some examples in the exercises. If you found this interesting, I encourage you to continue exploring scripting and programming. You are able to do many useful things already with what you have learned here, and at the same time - each additional piece of knowledge opens up new possibilities.

8.5 Exercises

The word "script" could encompass a range of different computer programs. It is often less foreign than it sounds. You will see that the step from your current UNIX knowledge to writing you first scripts is quite small.

When working with UNIX, scripts are especially useful for putting together a sequence of commands performing a specific task. This task could be to perform a particular analysis, or to do a more general task. When doing processing in UNIX, it is generally a good idea to gather processing steps into scripts.

8.5.1 Hello, world!

Let's first write the well-known "Hello, world" script together.

1. Start by opening an empty file in `nano`. Add the `#!` signs and the absolute path to the Bash interpreter at the top.

   ```
   #!/bin/bash
   ```

2. Below this line, you could add any commands. Now, we add an `echo` statement, together with a line similar to "Hello world".

   ```
   #!/bin/bash

   echo "Hello world :)"
   ```

3. Save and exit. You have now written your very first script. Next, change its file permissions to make it executable. In this case the file is named `my_script.sh`.

   ```
   $ chmod +x my_script.sh
   ```

4. Now you can run the script by specifying its path. If it is in your present working directory you need to explicitly show that by typing out `./` before the script (`.` refers to the current directory).

   ```
   $ ./my_script.sh
   ```

   Try it. This is the basic structure of the command. The content of the command could easily be exchanged with sequences of commands that you have learned during this course.

5. If we want to make more general scripts which can act on particular files, we could provide input arguments using the `argv` variable. This variable retrieves information

from the command used to run the script. If we edit our Hello world script to the following:

```
#!/bin/bash

echo "Hello ${1} :)"
```

We can now run it with an argument, saying "Hello" to an arbitrary person.

```
$ ./my_script.sh Jakob
Hello Jakob :)
```

This concept is highly useful if we want a script able to read from and output to arbitrary files.

8.5.2 Retrieving information from a FASTA

1. In this chapter, your were introduced to the script `fasta_stats.sh`. Implement it, and run it on the potato genome FASTA with the coding sequences. How many entries do you find? How many lines? How many nucleotides? (Did you find 36177057 nucleotides?)
2. * Could you think of any other useful information to extract from the FASTA? Add it to the script!
3. ** Extend the script so that it instead of taking a single file takes a directory with FASTAs, and prints information about all the contained FASTA files. Try it on the set of FASTA files we used in the previous chapter.

8.5.3 Building a useful script on your own (*)

As a final exercise, let's build a useful script which we could use later to retrieve information from our files. If you have an idea in mind of something which could be interesting or useful for you, feel free to try doing it! You could discuss with the teachers how to approach it, and what would be reasonable.

One useful example could be to do a FASTA entry retriever. This script could take FASTA file and an ID as input. It should then output the sequence and useful information for that particular entry. Information which might be useful could be:

- The header of the ID (in a nicer format than the regular FASTA header)
- The sequence belonging to the ID.
- Information about the sequence (length, number of different letters)
- If using the potato reference genome, you could make the script read the ID matrix, and output the annotation together with the ID.

8.6 Checkpoint

Before you continue, make sure that you can answer the following:

- How can scripts be used to make your research reproducible?
- Can you think of a Bash script which would be useful for you?

8.6.1 UNIX commands

Consider each command for a second. Make sure that you have an idea about what they do before moving on.

Introduction to UNIX

```
man
ssh [-v]
```

Introduction to the file system

```
cd
ls [-l] [-lh]
pwd
```

Working with files in UNIX

```
cat
cp [-i]
file
head [-number]
less
mkdir
mv [-i]
nano
rm [-i] [-r]
rmdir
tail [-number]
```

Working with bioinformatic data

```
cut [-d] -f/-c
diff
grep [-c] [-A] [-v] [-f] [-i]
sort [-n] [-r]
uniq [-c]
wc [-l] [-w] [-m]
```

Organizing files

```
chmod
gunzip
gzip
ln [-s]
scp
tar [-cvzf] [-xvzf]
wget
```

File streams

```
echo
sed
tr [-d]
```

Pattern matching, variables, subshells and loops

Bash scripting

8.7 Further reading

8.7.1 The PATH variable

The scripts we have been running in this chapter have all been used through their absolute or relative paths. The UNIX commands on the other hands can be accessed from anywhere without specifying any kind of path. The reason for this is that they are found in directories which are stored in the `PATH` variable. This tells the UNIX system where to look for programs. You can see the paths currently present in your `PATH` by printing it:

```
$ echo ${PATH}
```

More information on how to set up and use the `PATH`:

https://kb.iu.edu/d/acar

8.7.2 Further learning materials

If you enjoyed learning about scripting in Bash - Great! But there is a lot more to learn. Both within Bash and in other languages. If you are interested in digging further into programming a good next step could be to take a look at the Python programming language. Python is a powerful language with a clear syntax (Bash is a powerful language with a very messy syntax). Python is widely used to build everything from small scripts to extensive applications.

A nice starting point for getting into Python is Codecademy's interactive introductory course:

https://www.codecademy.com/learn/python

It can sometimes be hard to navigate the available materials. The following page summarizes, grades and reviews some online learning material of you are interested in digging further into Bash:

http://wiki.bash-hackers.org/scripting/tutoriallist

Good luck with your future data processing and scripting!

