RNA-Seq data analysis Part I

(jan Buchmann), Björn Usadel

Version alpha 0.10, 2023

Day 1 Basics

During this part we will revisit Linux, and use some basic Linux command line features.

Unix: Command-line basics

This section is for people who are new to using the terminal on a Linux computer. The terminal is a place where you can type in commands to make the computer do things.

Unix is an operating system that is running on Linux machines and Macs. Nowadays the graphical user interface (GUI) of Linux basically looks like any other operating system that you are used to and we recommend that you largely use this GUI and only switch to the terminal if you have to. Alternatively, if you use ubuntu on WSL2 Windows use the windows tools as you know them and usher the commands herin from the "ubunto-linux" command line. Similarly, for the MacOS where you have a unix like operation system.

We will teach you how to access the system through the terminal/shell, how to move through folders and a suite of little commands that are helpful in working with large datasets but also to conduct quality controls and sanity checks (these are extremely important).

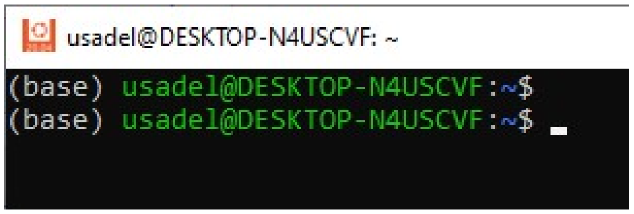
When you first open a terminal or log in to your account, you will see a prompt. This shows some basic information about your current session, like your username, the name of the computer you're using, and the directory (or folder) you're currently in. Everything you type after the prompt will be interpreted as a command for the computer to follow.

The prompt can be broken down into a few parts. The first part shows your username, followed by the name of the computer you're on. The last part shows the current directory you're in. For example, if the prompt shows

```
salk@linux:˜$
```

it means that you're logged in as the user "salk" on a computer called "linux", and you're currently in your home directory (represented by the tilde symbol).

Ubuntu on Windows looks e.g. like this

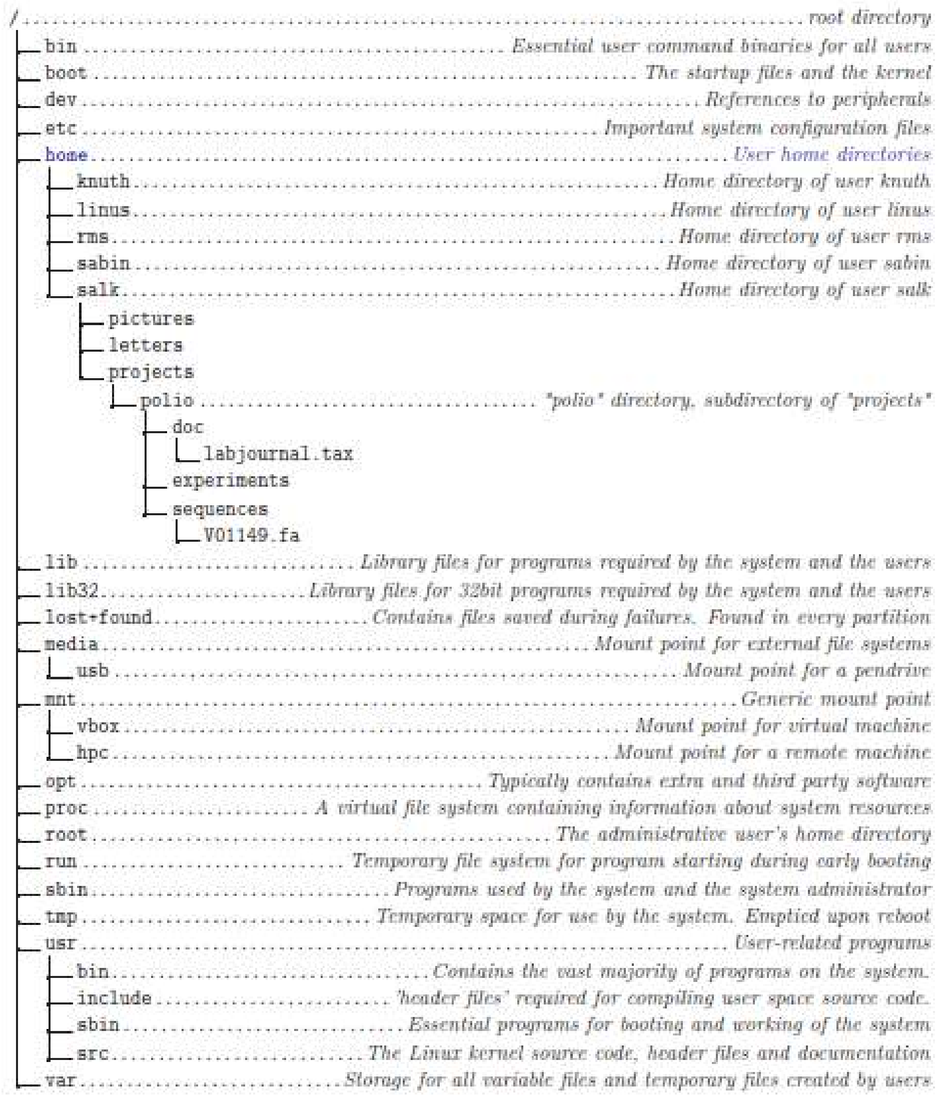


```
(base) usadel@DESKTOP-N4USCVF:~$
(base) usadel@DESKTOP-N4USCVF:~$
```

On a Linux system, everything is considered a file, whether it's a document, program, or device. You don't need to use file suffixes, like .jpg or .pdf, as they are mainly for the user's convenience. For instance, PDF readers can read PDF files without the .pdf suffix, and changing a file from .pdf to .jpg doesn't change its file format.

It is important to understand the Linux filesystem if you are using Linux as your working environment. You need to be aware of “where you are” when using the shell, as it can affect how you interact with the system or set up your analysis.

The root directory, denoted by '/', is the top-level directory of the filesystem, and it should not be confused with the home directory of the ***root user***, which is typically located at /root/.



```
/ ............................................................ root directory
 ├─bin ............................ Essential user command binaries for all users
 ├─boot ..................................... The startup files and the kernel
 ├─dev .......................................... References to peripherals
 ├─etc ................................... Important system configuration files
 ├─home ............................................. User home directories
 │  ├─knuth ...................................... Home directory of user knuth
 │  ├─linus ...................................... Home directory of user linus
 │  ├─rms .......................................... Home directory of user rms
 │  ├─sabin ...................................... Home directory of user sabin
 │  └─salk ........................................ Home directory of user salk
 │     ├─pictures
 │     ├─letters
 │     └─projects
 │        └─polio ..................... "polio" directory, subdirectory of "projects"
 │           ├─doc
 │           │  └─labjournal.tax
 │           ├─experiments
 │           └─sequences
 │              └─V01149.fa
 ├─lib ..................... Library files for programs required by the system and the users
 ├─lib32 ............... Library files for 32bit programs required by the system and the users
 ├─lost+found ............... Contains files saved during failures. Found in every partition
 ├─media ................................... Mount point for external file systems
 │  └─usb ........................................... Mount point for a pendrive
 ├─mnt ..................................................... Generic mount point
 │  ├─vbox ..................................... Mount point for virtual machine
 │  └─hpc ...................................... Mount point for a remote machine
 ├─opt ............................... Typically contains extra and third party software
 ├─proc ............... A virtual file system containing information about system resources
 ├─root ..................................... The administrative user's home directory
 ├─run ........................ Temporary file system for program starting during early booting
 ├─sbin ........................... Programs used by the system and the system administrator
 ├─tmp ......................... Temporary space for use by the system. Emptied upon reboot
 ├─usr ......................................................... User-related programs
 │  ├─bin ............................... Contains the vast majority of programs on the system.
 │  ├─include ....................... 'header files' required for compiling user space source code.
 │  ├─sbin ............................ Essential programs for booting and working of the system
 │  └─src ......................... The Linux kernel source code, header files and documentation
 └─var ......................... Storage for all variable files and temporary files created by users
```

Knowing how to navigate the Linux filesystem, create and delete files and directories, and understand the concepts of home and working directories is crucial. By understanding the Linux filesystem, you can navigate all Unix-derived filesystems.

Even though Microsoft Windows has a different filesystem, the basic principles are the same, there are folders subfolders etc.

Knowing this information is important, because it helps you keep track of where you are and what you're doing. The **directory**, in particular, is very important, because it tells you which **files** and **directories** you can access and work with. As you move around the system and enter different **directories**, the prompt will update to reflect your current location.

In Linux, everything is treated as a file, including special files like DVDs or pen drives, pipes, sockets, as well as directories which are files containing the names of other files. This abstraction helps with accessing, modifying, handling and exchanging different types of media. The canonical Linux filesystem is described as a tree, with the top-most directory being the root directory ("/").

Paths are used to locate files on the Linux filesystem**. Absolute paths** start with a leading "/" and are anchored to the **root** directory. They describe the full path to a **file** and can be used to access a file from anywhere in the filesystem**. Relative paths** are anchored to the current working directory and do not start with a leading "/". They describe the **path** to a **file** relative to the **current working directory** and can only be used to access a file from within the current directory or its subdirectories.

For example, the **absolute path** to the `errno.h` file is `/usr/include/errno.h`. You can always use an **absolute path** to refer to a file or directory, no matter where your current working directory is.

The current **working directory** is the directory you are currently in, while your home directory is always the same and located at "/home/username". You should always know your current working directory and home directory, as it is the directory that you are currently in and is where any commands you run will be executed. You can see your current working directory by using the **pwd** command (which stands for "print working directory"):

```
$ pwd
/home/salk
```

In this example, the `pwd` command tells us that our current working directory is `/home/salk`. This is the directory that we are currently "in".

If you want to move to a different directory, you can use the `cd` command (which stands for "change directory"). For example, if you want to move to the `/usr/include` directory, you can run:

```
$ cd /usr/include
```

Note the space “ “ between `cd` and `/usr/include`

Now, if you run `pwd`, you'll see that your current working directory has changed:

```
$ pwd
/usr/include
```

You can also use **relative paths** to move between directories. For example, if you want to move from `/usr/include` back to your home directory (`/home/salk`), you can use the following commands:

```
$ cd ../..
$ cd home/salk
```

This moves up two directories (the "`..`" symbol means "the parent directory") from `/usr/include`, so you end up in the root directory. The second command moves you into the /home/salk directory.

Your **home directory** is a special directory that is always the same for your user, and is typically located at `/home/your-username`. You can always get to your home directory by running `cd` without any arguments:

```
$ cd
```

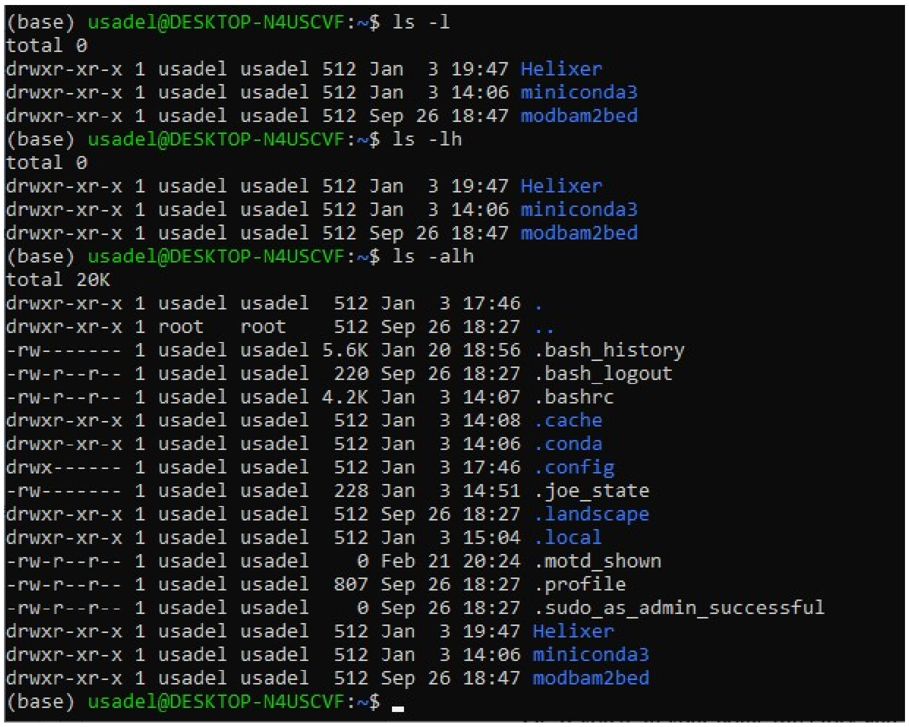
This is equivalent to running cd /home/salk, assuming your username is "salk".

You can use the **ls** command to see the contents of a directory. If you want to show the content of the current directory just type "ls".

```
$ ls
```

Or if you want more information you can add "-l" long format (more information); "-a" (all files); "-h" human readable and "-t" sort by time.

You can combine these commands



```
(base) usadel@DESKTOP-N4USCVF:~$ ls -l
total 0
drwxr-xr-x 1 usadel usadel 512 Jan  3 19:47 Helixer
drwxr-xr-x 1 usadel usadel 512 Jan  3 14:06 miniconda3
drwxr-xr-x 1 usadel usadel 512 Sep 26 18:47 modbam2bed
(base) usadel@DESKTOP-N4USCVF:~$ ls -lh
total 0
drwxr-xr-x 1 usadel usadel 512 Jan  3 19:47 Helixer
drwxr-xr-x 1 usadel usadel 512 Jan  3 14:06 miniconda3
drwxr-xr-x 1 usadel usadel 512 Sep 26 18:47 modbam2bed
(base) usadel@DESKTOP-N4USCVF:~$ ls -alh
total 20K
drwxr-xr-x 1 usadel usadel  512 Jan  3 17:46 .
drwxr-xr-x 1 root   root    512 Sep 26 18:27 ..
-rw------- 1 usadel usadel 5.6K Jan 20 18:56 .bash_history
-rw-r--r-- 1 usadel usadel  220 Sep 26 18:27 .bash_logout
-rw-r--r-- 1 usadel usadel 4.2K Jan  3 14:07 .bashrc
drwxr-xr-x 1 usadel usadel  512 Jan  3 14:08 .cache
drwxr-xr-x 1 usadel usadel  512 Jan  3 14:06 .conda
drwx------ 1 usadel usadel  512 Jan  3 17:46 .config
-rw------- 1 usadel usadel  228 Jan  3 14:51 .joe_state
drwxr-xr-x 1 usadel usadel  512 Sep 26 18:27 .landscape
drwxr-xr-x 1 usadel usadel  512 Jan  3 15:04 .local
-rw-r--r-- 1 usadel usadel    0 Feb 21 20:24 .motd_shown
-rw-r--r-- 1 usadel usadel  807 Sep 26 18:27 .profile
-rw-r--r-- 1 usadel usadel    0 Sep 26 18:27 .sudo_as_admin_successful
drwxr-xr-x 1 usadel usadel  512 Jan  3 19:47 Helixer
drwxr-xr-x 1 usadel usadel  512 Jan  3 14:06 miniconda3
drwxr-xr-x 1 usadel usadel  512 Sep 26 18:47 modbam2bed
(base) usadel@DESKTOP-N4USCVF:~$
```

You can also add a directory you want to see. For example, if you want to see the contents of the /usr/include directory, you can run:

```
$ ls /usr/include
```

Or, if you're in your home directory and want to see the contents of the `Helixer` directory relative to your current working directory, you can run:

```
$ ls Helixer
```

Note that in this case, the `ls` command will look for the `Helixer` directory within your current working directory. If it's not there, you'll get an error message. This is why it's important to keep track of your current working directory and use appropriate paths when referring to files and directoriess.

As you will not have it yet you can create the folder with the command **mkdir.**

```
$ mkdir Helixer
```

When you type

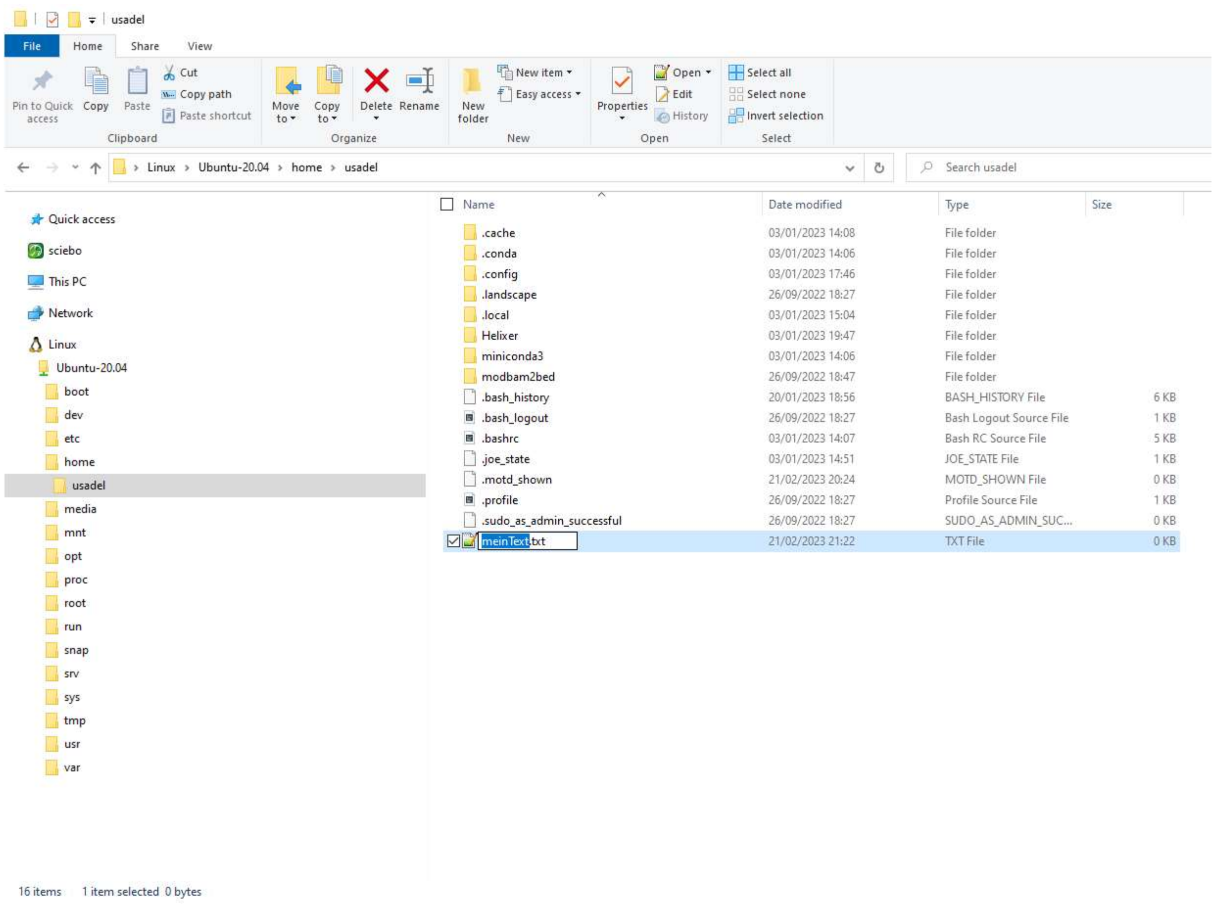
```
$ ls -l
```

now it should have appeared and you can change into this directory.

In terminal/shell, you can access directories just like you can through a GUI. Modern Unix-based operating systems also have convenient GUIs, but you need to know how to work in terminal to run programs.

```
# open terminal by finding the black box icon on the home screen
#list the contents of the directory you are in
ls
#move to your home directory
cd
#list the contents of the home directory
ls
mkdir examples
#make the examples directoy in your home directory
#move to the examples directory
cd examples
#move the directory above the directory you are in note that you need to have a space between cd and “..”
cd ..
```

Many of the programs require that the file you are working on and the program you are working with to be in the same folder. If you do not really know what you are doing, it is a safe idea to use small helper programs by copying them into the folder where the data files are. You can move the programs through the GUI just as you do with other operating systems. The example below shows windows and the penguin (the linux mascot) shows ubuntu linux is running and you can access its directory structure.



Name	Date modified	Type	Size
.cache	03/01/2023 14:08	File folder	
.conda	03/01/2023 14:06	File folder	
.config	03/01/2023 17:46	File folder	
.landscape	26/09/2022 18:27	File folder	
.local	03/01/2023 15:04	File folder	
Helixer	03/01/2023 19:47	File folder	
miniconda3	03/01/2023 14:06	File folder	
modbam2bed	26/09/2022 18:47	File folder	
.bash_history	20/01/2023 18:56	BASH_HISTORY File	6 KB
.bash_logout	26/09/2022 18:27	Bash Logout Source File	1 KB
.bashrc	03/01/2023 14:07	Bash RC Source File	5 KB
.joe_state	03/01/2023 14:51	JOE_STATE File	1 KB
.motd_shown	21/02/2023 20:24	MOTD_SHOWN File	0 KB
.profile	26/09/2022 18:27	Profile Source File	1 KB
.sudo_as_admin_successful	26/09/2022 18:27	SUDO_AS_ADMIN_SUC...	0 KB
meinText.txt	21/02/2023 21:22	TXT File	0 KB

1. Create a file “meinText.txt” in your (linux) `home` directory using the GUI.

2. Look again into your home folder to see what has changed

```
#open terminal
#move to your home folder
cd
#list the contents
ls
#now you can see the new file meinText.txt
```

As the file is really empty we can remove it on the command line now as well using the **rm** command.

```
rm meinText.txt
```

Check in the GUI if it is still there (sometimes you have to reload or just go one directory up and down again for the GUI to refresh).

If a program does not run you might not have the ***permissions*** to run a program. If you are not the ***admin*** on the computer, make sure that you are in future or make your admin install and test all programs that you might want to use. In that case, the admin will give you authority to run the programs. If you need to do it yourself, it works like this

Unix: Setting permissions

There are several cases in which you want to change the permissions of a file or even a folder.

Sometimes you need to use the "**sudo**" command preceding a command to install e.g. into certain directories. This gives you temporary root-permissions (The highest level of permission you can get on a UNIX system). This should only happen seldomly.

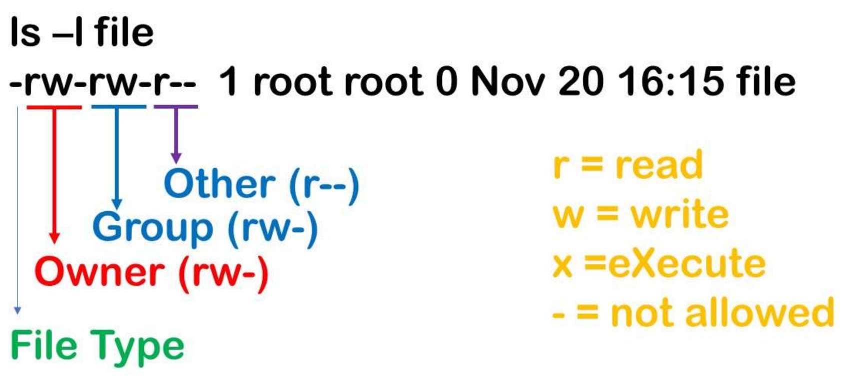
Attention: Using the "sudo" command is like telling a 3-year old child to do something. It WILL do it no matter how lethal that is going to be. Running a malicious script with "sudo" can cause serious issues. Just be sure what you do. And usually you should not need to use this.

If you try to read/write/execute a file you own, you can just add the permission you need. Therefore, you need to understand the permissions given to files in UNIX. A file knows three entities it can be accessed from: u) The file owner (user) g) the usergroup the file belongs to o) any other person (also known as: the world). For each of the entities a file has separate permissions for reading, writing and executing it. First try to understand the permissions of a few files.

```
#print a detailed list of the current directory
ls -al
```

In Linux, each file and directory is associated with a ***user***, and ***group***, and a set of permissions that determine who can access the file, and what actions they can perform on it.

The permissions for a file or directory are represented by a 10-character string that starts with either a "d" or a "-", with the first character indicating whether the file is a directory or not. The remaining nine characters are split into three groups of three characters each, representing the read, write, and execute permissions for the owner, the group, and other users.



Here's an example of what the permissions string might look like: "-rwxr-x---". The first character is "-", indicating that this is not a directory. The next three characters, "rwx", represent the owner's permissions. The owner has read, write, and execute permissions, which means they can read the file, modify it, and run it as a program. The next three characters, "r-x", represent the group's permissions. The group members have read and execute permissions, but they cannot modify the file. The final three characters, "---", represent the permissions for all other users who are not the owner or a member of the group. These users have no permissions at all, which means they cannot read, write, or execute the file.

To change the permissions of a file or directory, you can use the **chmod** command, followed by a three-digit number that represents the new permissions. The first digit represents the owner's permissions, the second digit represents the group's permissions, and the third digit represents the permissions for all other users. Each digit is a sum of the corresponding ***read (4), write (2),*** and ***execute (1)*** permissions. For example, the number ***755*** represents ***read, write, and execute (4+2+1)*** permissions for the owner, and read and execute permissions (***4+1***) for the group and other users.

Here's an example of how to use **chmod** to change the permissions of a file named "example.txt" to read and write for the owner, and read-only for everyone else:

```
chmod 644 example.txt
```

This command sets the permissions to "-rw-r--r--", which means the owner can read and write the file, while all other users can only read it.

Understanding file and directory permissions is an important part of learning to use Linux, and can help you to keep your files and system secure.

In Linux, you can also use symbolic notation to change the permissions of a file or directory. The symbolic notation uses letters to represent the user, group, and other permissions, and symbols to add or remove permissions. The syntax for symbolic notation is:

```
chmod [who][operator][permissions] file/directory
```

The "who" field can be one or more of the following letters:

- u (user/owner)
- g (group)
- o (others)
- a (all/owner, group, and others)

The "operator" field can be one of the following symbols:

+ (adds permissions)

- (removes permissions)

= (sets permissions to the exact values specified)

The "permissions" field can be one or more of the following letters:

- r (read)
- w (write)
- x (execute)

Here are some examples of how to use symbolic notation to change permissions:

To add read and write permissions for the owner of a file:

```
chmod u+rw file.txt
```

Running programs

To run programs, you usually just have to type their name. If you run them from a specific directory you have to add "./" in front of the program name.

Create a New Document in the text editor containing the following two lines:

```
#!/bin/bash
date
```

Save the Document as `mydate.sh` into your Home Directory.

```
#After changing to your home directory, try to execute the script
cd
./mydate.sh
#That won't work, check the permissions and add executable to it and try again.
ls -al
chmod u+x mydate.sh
ls -al
#Note the change. Now start the script again
./mydate.sh
```

The following is just an example you should NOT use this as we now give all permissions to eveyone

```
#Check the current permissions. Make the file grant all permissions to everyone and check again
ls -al
chmod 777 mydate.sh
ls -al
```

Day 2 Unix: Advanced command-line tools

During our work with large datasets there are a few helpful commands that can be used to easily extract information. We will show you a few.

Download the file example.fastq from github

https://github.com/usadellab/course/tree/master/linux

It is also in lsf

less

The less command is a tool used to view the contents of a file in the terminal. It is particularly useful when working with large files, as it allows you to quickly navigate through the file and search for specific content.

As an example, let's say you are working with a Fastq file, which is a common file format used to store DNA sequence data. To view the contents of a Fastq file using the less command, you would open a terminal and navigate to the directory containing the file:

```
cd /path/to/directory
```

Once you are in the correct directory, you can use the `less` command to view the file:

```
less example.fastq
```

This will open the `example.fastq` file in the terminal, allowing you to view its contents. You can use the arrow keys to navigate through the file, **and press `q` to exit `less`**.

If you want to search for a specific sequence within the file, you can use the forward slash (/) followed by the search term. For example, to search for the sequence "ATGC" within the file, you would enter:

```
/ATGC
```

`less` will highlight all occurrences of the search term within the file.

You can also use `less` with other file types, such as text files or code files. It is a versatile tool that can be very helpful for navigating through large files in the terminal.

Look at the examplelong.txt (github)

```
#View a file page by page
less examplelong.txt
#press the Enter key, the file will move forward one line
#press the space key, the file will move forward one page
#press the Q key to leave
```

If you want to look at a file using the GUI on a Windows system, do **NOT** use the text editor. We recommend Notepad++ (http://notepad-plus-plus.org/). This will open even huge files with no to little problems. Notepad++ is also powerful enough to process translations of strings, something that can be really useful to reformat data. We will revisit Notepad++ when we look at extracting biological data.

head

The `head` command is used to print the first few lines of a file. By default, it prints the first 10 lines of the file. You can specify the number of lines to print using the `-n` option, like this:

```
head -n 12 example.txt
```

This will print the first 12 lines of `myfile.txt`.

Example: Let's say you have a file called `example.txt` with the following contents:

```
This is line 1
This is line 2
This is line 3
This is line 4
This is line 5
This is line 6
This is line 7
This is line 8
This is line 9
This is line 10
This is line 11
This is line 12
```

If you run the command:

```
head example.txt
```

The output will be:

```
This is line 1
This is line 2
This is line 3
This is line 4
This is line 5
This is line 6
This is line 7
This is line 8
This is line 9
This is line 10
```

tail

The `tail` command is used to print the last few lines of a file. By default, it prints the last 10 lines of the file. You can specify the number of lines to print using the `-n` option, like this:

```
tail -n 5 example.txt
```

This will print the last 5 lines of `example.txt`.

Example: Using the same file `example.txt`, if you run the command:

```
tail example.txt
```

The output will be:

```
This is line 3
This is line 4
This is line 5
This is line 6
This is line 7
This is line 8
This is line 9
This is line 10
This is line 11
This is line 12
```

cat/zcat

The `cat` command is used to concatenate and display files. It can be used to display the contents of a file or to combine the contents of multiple files into a single file.

Example: If you have two files called `example.txt` and example2`.txt`, you can combine them into a single file using the following command:

```
cat example.txt example2.txt > combined.txt
```

This will create a new file called `combined.txt` that contains the contents of both `file1.txt` and `file2.txt`.

You can also use `cat` to display the contents of a file:

```
cat combined.txt
```

This will display the entire contents of `combined.txt` in the terminal. Note: `cat` can be used to display the contents of binary files like Fastq files, but the output may be unreadable. For viewing Fastq files, the `less` command is recommended.

Order is important so now try

```
cat example2.txt example.txt > combined.txt
```

and again look at the file (yes the file was overwritten)

```
cat combined.txt
```

zcat is a special form of cat that you can use on compressed zipped gz files. This comes in handy for us so we don't decompress our fastq.gz files.

wget

With wget you can download files from a webaddress

```
wget https://raw.githubusercontent.com/usadellab/course/master/linux/example.fastq
```

wc

The **`wc`** command in Linux is short for "word count" and can be used to count the number of lines, words, and bytes in a file. This is particularly useful for analyzing large Next Generation Sequencing (NGS) data files, including FASTQ files.

Counting the number of reads in a FASTQ file:

```
wc -l example.fastq
wc example.fastq
```

This command will output the number of lines in the `example.fastq` file, which should correspond to the number of reads in the file times 4.

cut

The **cut** command in Linux is a utility for cutting out sections from each line of a file. It's particularly useful for parsing and manipulating large text files, like those commonly encountered in next-generation sequencing data analysis.

The basic syntax of the cut command is as follows:

```
cut -d [delimiter] -f [field(s)] [input file]
```

Here, the `-d` option specifies the delimiter character, which is used to separate fields in the input file. The `-f` option specifies the field or fields to be cut out and printed to standard output.

Let's look at some examples of how the `cut` command can be used.

```
cut -d " " -f 4 examplelong.txt
```

This will cut out the first and third fields from each line of the `input.txt` file, separated by commas, and write them to the `output.txt` file.

Sort and uniq

The `sort` command is used to sort the contents of a file in either ascending or descending order. It is useful for sorting large amounts of data, such as next-generation sequencing data. Here is the basic syntax for the `sort` command:

```
sort [options] [filename]
```

Here are some common options:

- `-n`: sort numerically
- `-k`: sort based on a specific column in the file
- -r : reverse order

Now let's see some examples of using the `sort` command with next-generation sequencing data.

Example 1: Sorting

```
Sort -n -r -k4 examplelong.txt
```

This command sorts the input file by the fourth column

The `uniq` command, as its name suggests, is used to remove duplicate lines from a file. It is useful for removing duplicate reads from next-generation sequencing data. Here is the basic syntax for the `uniq` command:

```
uniq [options] [filename]
```

Here are some common options:

- `-c`: count the number of occurrences of each line
- `-d`: only output duplicate lines
- `-u`: only output unique lines (removing duplicates)

Now let's see some examples of using the `uniq` command with next-generation sequencing data.

Example 1: Removing duplicate reads from a BAM file

```
sort -n -k4 examplelong.txt | uniq -u
```

```
sort -n -k4 examplelong.txt | uniq -d
```

Wildcards

In Unix-based operating systems, a wildcard is a character that can be used to represent any character or group of characters in a filename or command. The most common wildcard character is the asterisk (`*`), which represents any sequence of characters.

Here are some examples of how to use the asterisk wildcard in Unix:

1. List all files in a directory:

```
ls *
```

This command will list all files and directories in the current directory.

3. Remove all files with a certain extension:

```
rm *.txt
```

This command will remove all files in the current directory that end with the `.txt` extension.

4. Search for files with a specific pattern:

```
ls *pattern*
```

This command will list all files in the current directory that contain the word `pattern` in their filename.

5. Use the wildcard with commands that take multiple arguments:

```
chmod 755 *.sh
```

This command will set the permission of all files in the current directory that end with the `.sh` extension to `755`.

By using wildcards, you can work on a large numbers of files simulataneously, as you can specify patterns rather than having to type out every file name individually.

tar

tar can combine multiple folders into a virtual tar file. Most often you will see a file called tar.gz which which "tarred" and then compressed you can uncompress such a file like so

```
tar -xvzf FILENAMe.tar.gz
```

touch

The "touch" command in Linux is used to create a new, empty file or update the access and modification times of an existing file.

```
touch completenewfile.txt
```

grep

Another useful command for analyzing files is "grep", which allows you to search for specific patterns in a file. For example, let's say you have a large text file and you want to find all the lines that contain the word "error". You can use the "grep" command to search for this pattern:

```
grep 'not' examplelong.txt
```

In this command, the "grep" command is used to search for the pattern "error" in the "large_file.txt" file. The output of "grep" is all the lines in the file that contain the pattern.

```
grep -A 3 'GGCCCAC' example.fastq > filtered_file.fastq
```

This command searches for the sequence "GGCCCAC " in the file "example.fastq" and outputs the matching lines, as well as the three lines that follow each match (which will contain additional information about the read), to the file "filtered_file.fastq".

The `-A 3` flag tells `grep` to output three additional lines after each match and ">" again directs the output to a new (or existing file).

Similarly `-B 3` would show three additional lines before each match.

Grep can be combined with **regular expressions**

Regular expressions are a powerful tool for searching and manipulating text in Unix-based systems, and are often used with the `grep` command. Regular expressions are a pattern of characters that can be used to match and search for specific sequences of text. Here are some basic regular expression concepts and examples for beginners:

Matching a specific character:

```
grep "a" sample.txt
```

This command will search for the character "a" in the file `sample.txt`.

Matching a range of characters:

```
grep "[a-z]" sample.txt
```

This command will search for any lowercase letters in the file `sample.txt`.

Matching a specific word:

```
grep "hello" sample.txt
```

This command will search for the word "hello" in the file `sample.txt`.

Matching any character:

```
grep "." sample.txt
```

This command will search for any character in the file `sample.txt`.

Matching the beginning or end of a line:

```
grep "^hello" sample.txt
```

This command will search for the word "hello" at the beginning of a line in the file `sample.txt`.

```
grep "world$" sample.txt
```

This command will search for the word "world" at the end of a line in the file `sample.txt`.

Matching zero or more occurrences of a character:

```
grep "ab*c" sample.txt
```

This command will search for the sequence "ac", "abc", "abbc", "abbbc", and so on in the file `sample.txt`.

Matching one or more occurrences of a character:

```
grep "ab+c" sample.txt
```

This command will search for the sequence "abc", "abbc", "abbbc", and so on in the file `sample.txt`.

Expressing whitespace in `grep` regular expressions: Whitespace characters like spaces and tabs can be expressed using special escape sequences. For example, if you wanted to search for the string "hello world" where there is a space between "hello" and "world", you could use the escape sequence `\s` to match any whitespace character:

```
grep 'hello\sworld' file.txt
```

In `grep`, the `\d` metacharacter can be used to match any digit character if we boost grep by saying "-P" (perl compatible regular expession). It is equivalent to the character class `[0-9]`. This can be useful when searching for patterns that include numbers or digits.

For example, suppose you have a file called `data.txt` that contains the following lines:

```
John: 123-456-7890

Jane: 555-867-5309

Alice: 999-999-9999
```

If you want to search for any phone number that contains the digits `two three digit blocks and ends in 9999`, you can use the `\d` metacharacter to match any digit:

```
grep -P '\d\d\d-\d\d\d-9999' data.txt
```

The regular expression `\d\d\d-\d\d\d-9999` matches any sequence of three digits followed by a dash, followed by another sequence of three digits and a dash, and then a final sequence of four times the digit 9. The output of this command would be:

```
Alice: 999-999-9999
```

Note that the `\d` metacharacter is a shorthand for the character class `[0-9]`, so you could also use `[0-9]` in place of `\d` in this example:

```
grep '[0-9][0-9][0-9]-[0-9][0-9][0-9]-9999' data.txt
```

Both of these regular expressions would produce the same output as the previous example.

```
echo "John: 123-456-7890
Jane: 555-867-5309
Bob: 999-999-9999" > data.txt
grep -P '\d\d\d-\d\d\d-9999' data.txt
grep '[0-9][0-9][0-9]-[0-9][0-9][0-9]-9999' data.txt
```

These are just a few examples of the basic regular expression concepts that can be used with the `grep` command. Regular expressions can be used to perform complex searches and manipulations of text, but it is important that you familiarize yourself a little bit with at least the basic ones.

One typical mistake is to use points in searches e.g. when looking for numbers. Remember. means anything so if you search for 5.3 it finds 5.3 but also 565 or 5T6 etc.

md5sums

The md5sum command is a commonly used tool in bioinformatics, particularly in the analysis of Next-Generation Sequencing (NGS) data. NGS generates a large amount of data, which can be prone to errors such as read misalignment, sequencing artifacts, and data corruption during transfer or storage. To ensure the integrity of NGS data, researchers often use md5sum to calculate the MD5 checksum of the data files, which is a unique fingerprint of the file's content. This fingerprint can be compared to the expected value to verify that the file has not been altered or corrupted during transfer or storage. By checking the md5sum value before and after file transfer, researchers can detect any changes in the file's content and ensure the accuracy and reliability of their NGS data analysis.

1. To calculate the MD5 checksum of a single NGS data file:

```
md5sum example.fastq
```

This will output the MD5 checksum of the file, which can be compared to the expected value to verify file integrity.

2. To calculate the MD5 checksum of multiple NGS data files in a directory:

```
md5sum *.fastq > checksums.txt
```

This will calculate the MD5 checksum of all files in the directory that match the `*.fastq.gz` pattern, and output the results to a file named `checksums.txt`. The file can be used to verify file integrity at a later time.

3. To compare the MD5 checksum of an NGS data file to the expected value:

```
md5sum -c checksums.txt
```

This will check the MD5 checksum of the file(s) listed in the `expected_checksums.txt` file, which should contain the expected MD5 checksum of each file in the format `checksum filename`. If the calculated checksum matches the expected value, the command will output `filename: OK`. If not, it will output `filename: FAILED`.

Using the md5sum command in these ways can help ensure the integrity of NGS data files, and detect any errors or corruption that may have occurred during transfer or storage.

You never need checksums until you needed them …. Always check them first after receiving files

```
#make 2 files
touch myfile.fastq
touch myfile2.fastq
#calculate checksum of one
md5sum myfile.fastq
#calculate checksum of  all
md5sum *.fastq > checksums.txt
#check files
md5sum -c checksums.txt
#now let’s garble a file
wc -l  myfile.fastq > myfile2.fastq
md5sum -c checksums.txt
```

Unix making your life a little easier

In Linux, the `Tab` key can be used to perform auto-completion, which is a feature that can help you quickly and easily type commands or file names in the terminal. When you type part of a command or file name, and press `Tab`, the system will try to complete the rest of the name for you based on what it finds in the current directory or in the list of available commands.

Here are some examples of how to use auto-completion with the `Tab` key in Linux:

File name auto-completion:

```
ls -a myfile2
```

In this example, typing `ls -a myfile2` and pressing the `Tab` key will auto-complete the rest of the file name to `ls -a myfile2.fastq`

If there are multiple files that match the pattern, the system will list them and you can select the correct file by typing more characters or using the `Tab` key again.

```
ls -a my
```

Directory auto-completion:

```
cd /ho
```

In this example, typing `cd /ho` and pressing the `Tab` key will auto-complete the rest of the directory name to `/home/`

By using auto-completion with the `Tab` key, you can save time and effort when working, as you can quickly and easily complete command and file names without having to type out every character thus also avoiding typos especially for complex and difficult filenames.

You can access previous commands you typed with the up and down arrows on your keyboard.

Linux Windows and MacOs line endings and other peculiarities

In computing, line endings refer to the characters that are used to mark the end of a line of text in a file. Different operating systems use different conventions for line endings, which can cause compatibility issues when transferring files between systems. Here are the conventions used by Windows, Linux, and macOS, along with some examples:

Windows: Windows uses the combination of a carriage return (CR) character and a line feed (LF) character, represented as "\r\n", to mark the end of a line. For example, the following text file in Windows would have line endings marked with "\r\n":

```
Hello\r\n

World\r\n
```

Linux: Linux uses a single line feed (LF) character, represented as "\n", to mark the end of a line. For example, the following text file in Linux would have line endings marked with "\n":

```
Hello\n

World\n
```

macOS: macOS used to use the same convention as Linux, with a single line feed (LF) character to mark the end of a line. However, some older versions of macOS used a single carriage return (CR) character, represented as "\r", to mark the end of a line. Newer versions of macOS have switched to using the same convention as Linux. For example, the following text file in macOS would have line endings marked with "\n":

```
Hello\n

World\n
```

When transferring files between systems with different line ending conventions, it is important to convert the line endings to the appropriate format for the target system. Many text editors and utilities have the ability to perform line ending conversion automatically, but it can also be done manually using tools like `dos2unix` or `unix2dos` where dos stands for windows.

When transferring windows text files to linux and vice versa mind the line endings. In the worst case your tools will exhibit unexpected behavior

In addition, Linux is case sensitive for file names Myfile and myfile are two totally different files for it.

Also, Linux and Windows “hate” special characters like umlauts and spaces in filenames make your life very, very difficult.

And **don’t** make the mistake of using spaces or special characters e.g. **äöüßçéµł** etc in your file or directory names. Whilst it is possible you will suffer.

Streams and redirections

In this section, we will introduce a number of useful Linux commands that you can use to analyze files. These commands are built into the operating system, which means you can use them without installing any additional software.

In Linux, everything is a file, including input and output streams, which are used to send sequences of characters from one place to another. There are three standard streams or input/output (I/O) streams: STDIN (standard input), STDOUT (standard output), and STDERR (standard error). Every program you run in Linux is connected to these three streams, and each of them has a file descriptor, which is a unique number used to identify the stream.

The STDIN stream (0) is the default input stream that receives input from the user or from another program. STDOUT stream (1) is the default output stream that displays output to the user or sends it to another program, and STDERR stream (2) is used to display error messages or output errors that occurred during the execution of a program.

Since these streams are files, you can treat them as such and perform various operations such as redirection, piping, and manipulation. For example, you can redirect the output of a program from the default STDOUT stream to a file using the ">" operator. Similarly, you can redirect the STDERR stream to a file using the "2>" operator.

Before we start, we need to explain the | (“pipe”) character. If you work on the command line, you can use the pipe character to connect the output of one command to the input of another command. This is useful because it allows you to create complex data processing pipelines by chaining together multiple commands.

Here's an example of how you can use the pipe character to process a text file. Let's say you have a file called "example.txt" that contains a list of numbers, one per line. You want to find the average of these numbers. You can use the "**cat**" command to display the contents of the file, and then pipe the output to the "**wc**" command, which can calculate line number

```
cat example.txt | wc –l
```

In this command, the "cat" command is used to display the contents of the "example.txt" file. The output of "cat" is then piped to the "wc" command, which calculates the number of "-l" lines in the file.

the > symbol in Linux is used to redirect the output of a command to a file, overwriting the contents of that file if it already exists, or creating a new file if it doesn't.

For example, if you have a FASTQ file called `my_reads.fastq` and you want to extract the first 10 lines of the file into a new file called `first_10_lines.txt`, you can use the `head` command to get the first 10 lines, and then redirect the output to a new file using the > symbol like this:

```
head -n 10 my_reads.fastq > first_10_lines.txt
```

This will create a new file called `first_10_lines.txt` in the current directory, which contains the first 10 lines of the `my_reads.fastq` file.

If `first_10_lines.txt` already exists, its contents will be overwritten by the output of the `head` command.

Linux often does not ask for permission to overwrite files if you are allowed to do so! Actually, Linux usually never does (when you need it to)

The STDERR stream is a channel in Linux that is used to output error messages from commands. By default, STDERR is printed to the terminal, but it can also be redirected to a file. In the context of FASTQ files, this can be useful when you want to capture error messages or diagnostic information that may be generated during a command's execution. For this you use "2>".

Here is an example of how to use the STDERR stream with FASTQ files:

```
gzip -cd file.fastq.gz >output.txt 2> error.log
```

In this example, we are unzipping a compressed FASTQ file and printing the uncpmressed data to a new file called "output.txt." The STDERR stream is redirected to a file called "error.log" using the `2>` operator. This means that any error messages generated by the `gzip` command will be written to the "error.log" file instead of being printed to the terminal.

Note that the `2>` operator must be used after the command you want to redirect STDERR for. In this example, we used it after the `gzip -cd file.fastq.gz` command to redirect STDERR to "error.log."

This can be especially useful when dealing with large datasets, where error messages may be numerous and difficult to keep track of in the terminal. By redirecting STDERR to a file, you can keep track of any errors that occur during your analysis without cluttering up your terminal output.

Most bioinformatics tools offer STDERR streams as logs. One should always keep and use them

If you make Linux do something stupid, such as get stuck in a loop, you can always get out by typing C**trl+C**. We will not make you do that in the course but you should remember **Ctrl+C**.

Getting Data

Now let's get some files this will might take a while

As this takes a while we needed to start this on day 2 or before the exercises on day 3

```
mkdir course
cd course
wget ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR176/006/SRR17655906/SRR17655906.fastq.gz
wget ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR176/003/SRR17655903/SRR17655903.fastq.gz
wget ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR176/098/SRR17655898/SRR17655898.fastq.gz
wget ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR176/007/SRR17655907/SRR17655907.fastq.gz
```

Day 3 Working with some real data

Questions and exercises

1. What might these files be?
2. Get a basic idea how much data you downloaded
3. Get a good idea how much bases and reads you have per file
4. Get the md5sum of the files
5. Write a short script that counts the length of each sequence line in a fastq file : use sort und uniq on the output of the script as a basic quality control tool.