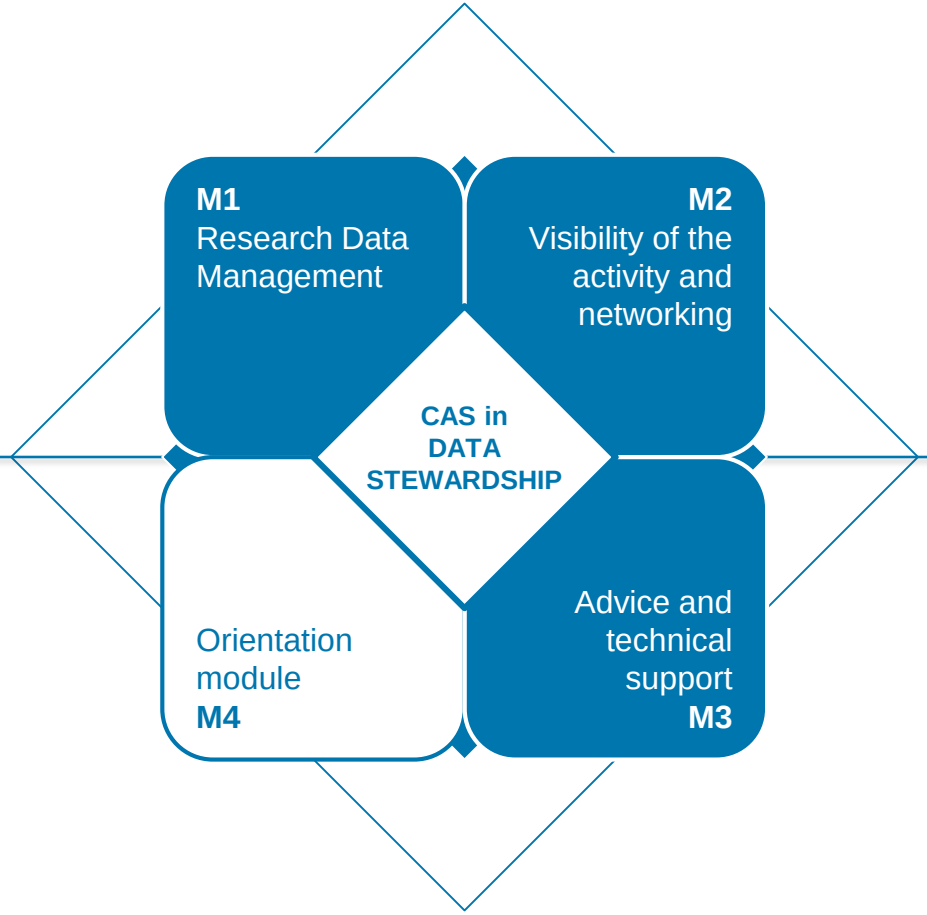


DAY TO DAY MANAGEMENT
INTRODUCTION TO LINUX / UNIX AND THE BASH SHELL

Robin Engler

SIB Swiss Institute of Bioinformatics

Course prepared in collaboration with:

- Gregoire Rossier (SIB training)
- Vassilios Ioannidis (Vital-IT)

ABOUT THIS PRESENTATION

This course is part of the CAS in Data Stewardship. You will find more info at <add link>

Author	Dr Robin Engler, Dr Grégoire Rossier
Provider	SIB Swiss Institute of Bioinformatics
Title	Introduction to Linux and the BASH shell
Education level	Graduates
Language	English
License	CC-BY 4.0
Estimate total time	12h (with interactions)
DOI	<doi + link>
Version	v20240110
How to attribute	ENGLER, Robin (SIB Swiss Institute of Bioinformatics), 2025. Introduction to UNIX. CAS Data Stewardship UNIL. 20 March 2025. DOI: <doi + link>

YOUR HOST(S)

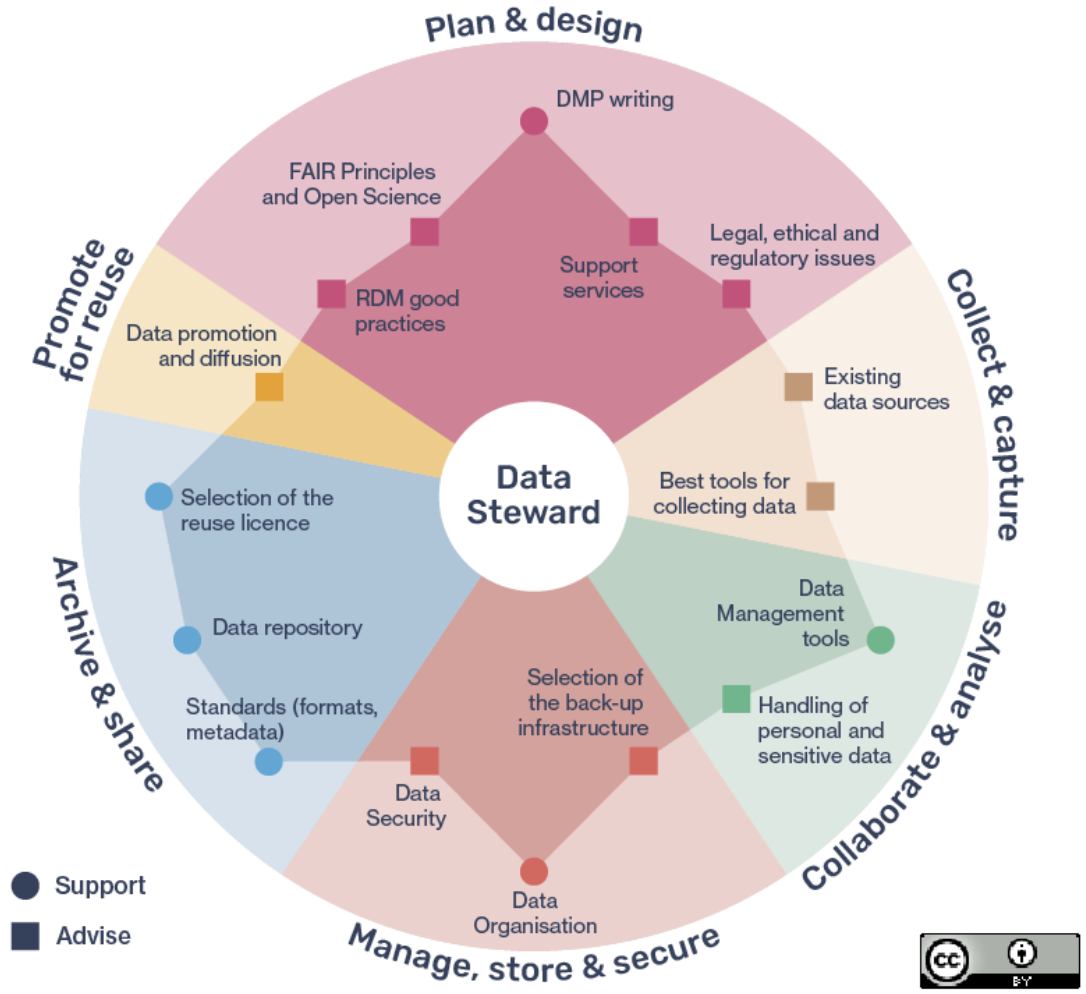
Dr Robin Engler
Senior Computational Biologist, SIB/Vital-IT group

Dr Grégoire Rossier (TBC)
Project Manager, SIB/Training/Vital-IT group
https://orcid.org/0000-0002-0065-5053)

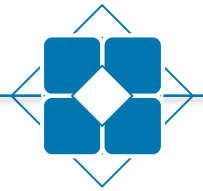
PEDAGOGICAL OBJECTIVES

In this lesson we will work on these competencies:

- PS1 - Understand the functioning and challenges of the research process in an institutional context, including policies, organization and strategy.
- PS1 - Be familiar with research-related professions and their interaction within an institution.
- PS8 - Understand the challenges of Open Science and Open Research Data and what they mean for research data management



SwissDS-ENV project: Data Stewards tasks and contributions at each step of the data lifecycle (lifecycle adapted from FORS). University of Lausanne, 2023.

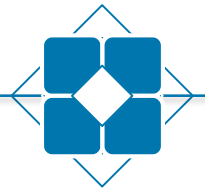


INTRODUCTION TO LINUX / UNIX AND THE BASH SHELL

LEARNING OBJECTIVES

Acquire skills necessary for a **basic usage of Linux / UNIX**.

- Get familiar with the basics of the Linux / UNIX **file system**.
- Learn the basics of the **Bash shell**.
- Learn to use the **most common** Linux / UNIX **commands**.

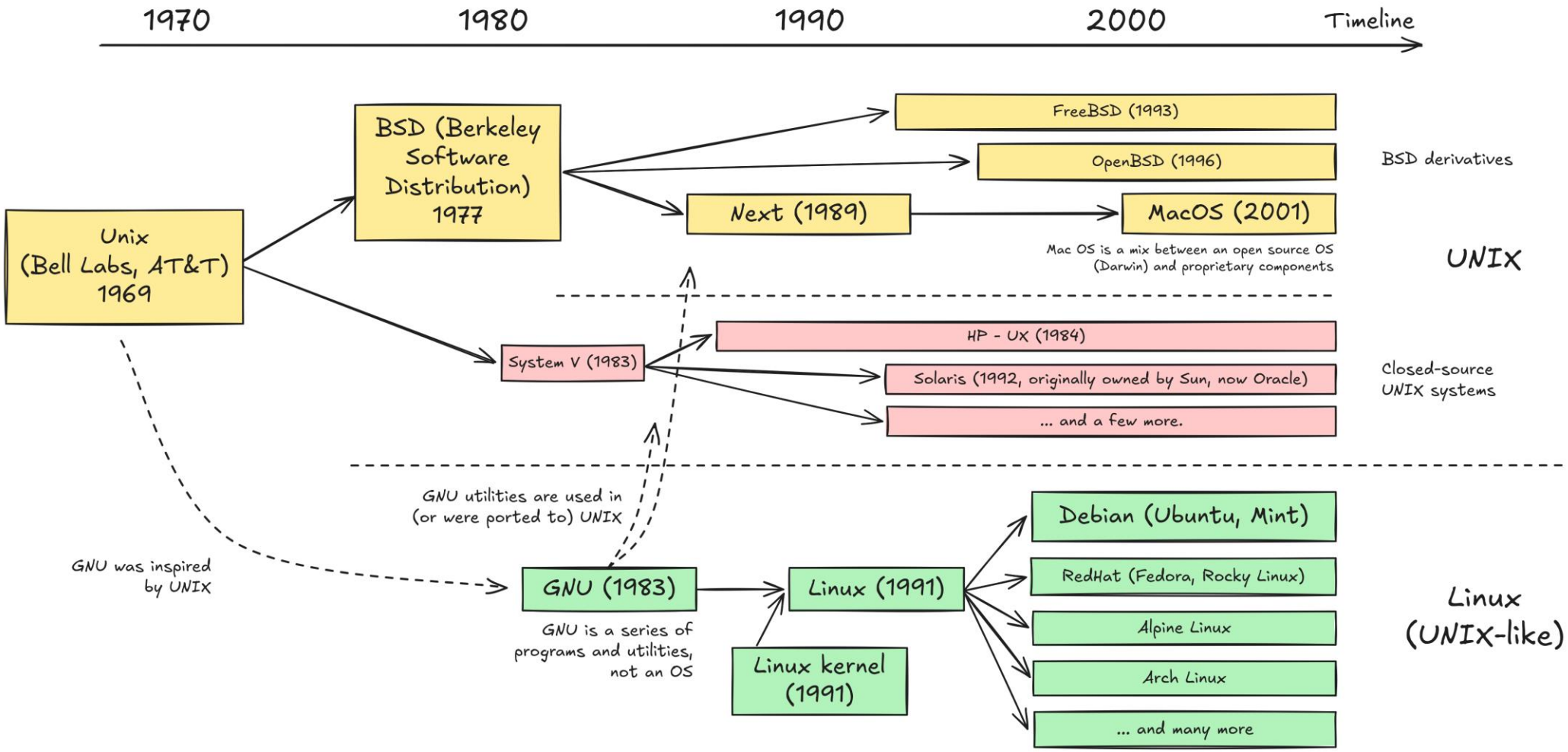


WHAT IS LINUX / UNIX AND WHY YOU MIGHT NEED (OR WANT) TO USE IT ?

Introduction to Linux / UNIX and the Bash Shell

What is Linux and UNIX ?

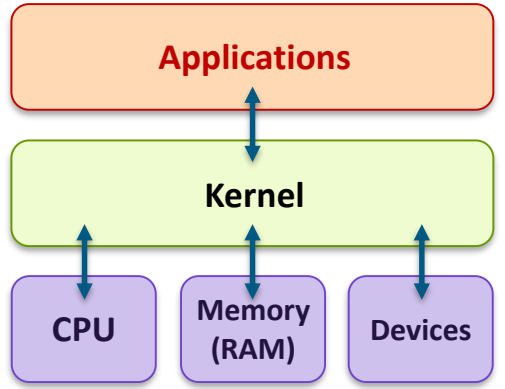
- **UNIX** is a family of (closed and open-source) operating systems that derive from the original AT&T Unics operating system (1969).
- **Linux** is a family of open-source operating systems, whose core software utilities (GNU coreutils) were inspired by UNIX, and which were later (partly) also included in (or ported to) subsequent UNIX operating systems.



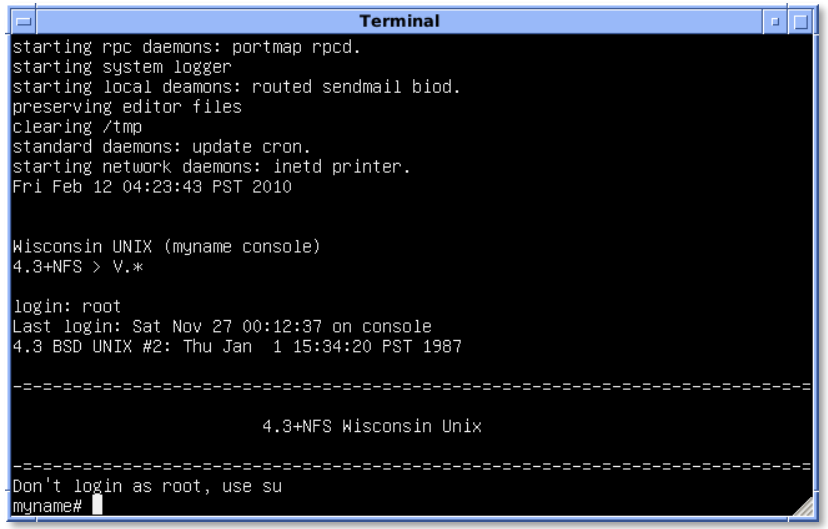
Disclaimer: this diagram is a simplified and non-exhaustive view of UNIX-like operating systems

Basic components of a Linux / UNIX system

- **Kernel** – the brain of the operating system. The kernel is a core program of an operating system: it performs tasks such as running processes, managing CPU, memory and other hardware devices (e.g. hard drives).
- **File system** – for data storage and file management.
- **Shell** – an interface (command line or graphical) for the user to interact with the system.
- **Core utilities** – basic command line tools (programs).
- And a few more: networking, package management, system services, logs.



DEC VT100 terminal, running an early Unix version.
Image credits: Jason Scott, CC BY 2.0
https://commons.wikimedia.org/w/index.php?curid=29457452



"4.3 BSD UNIX" from the University of Wisconsin c. 1987. System startup and login.
Image credits: Huihermit, CC0.
https://commons.wikimedia.org/w/index.php?curid=31125457

What are Shells ?

Shell: a program that exposes an operating system's services to users (humans or other programs).

Shells allow users to perform tasks such as:

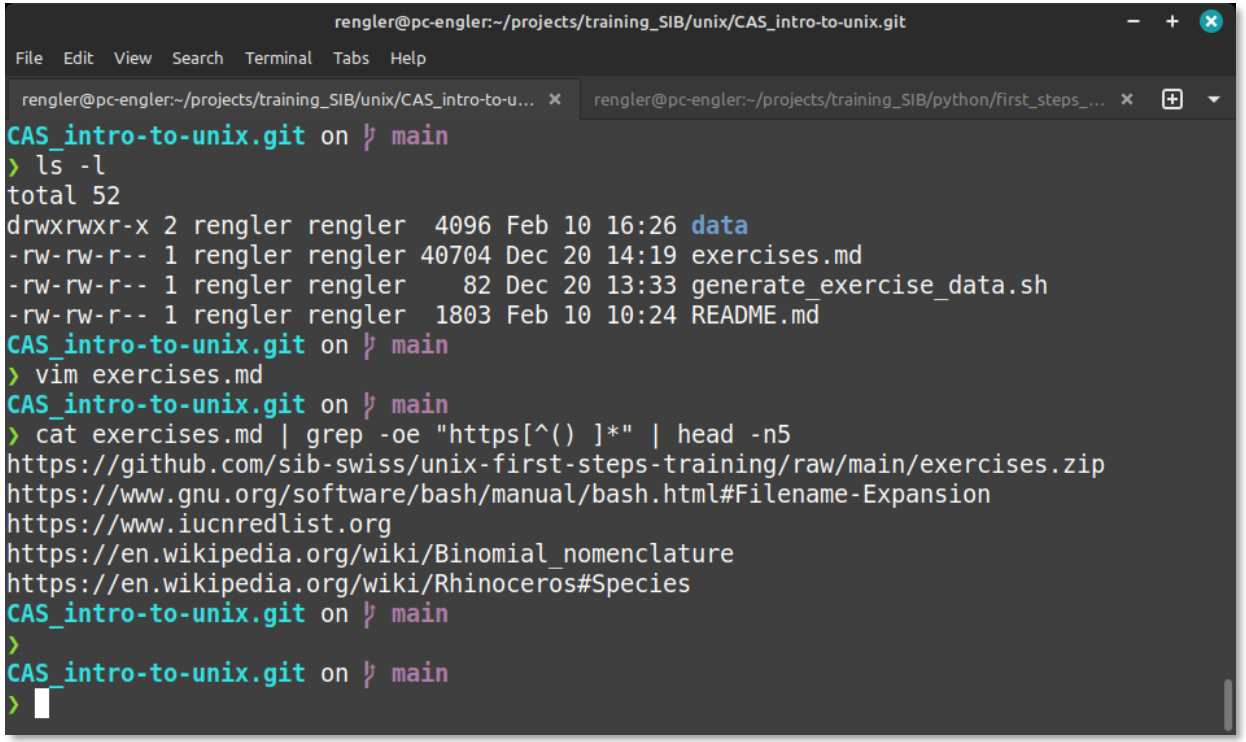
- **Running/starting applications**
- **File management**
- **Connecting applications via pipes** – i.e. passing the output of a program to another.
- **Automating task** (e.g. via scripts) and basic data manipulation (e.g. string manipulation). This will not be covered in this course.

In general, the term **shell** is used to refer to **CLI (Command Line Interface)** shells.

The **Bash** shell ?

Bash (**B**ourne-**a**gain **sh**ell) is an open-source shell created in 1989 for the GNU project, as a replacement for the proprietary UNIX shells, such as the “Bourne shell” (hence its name).

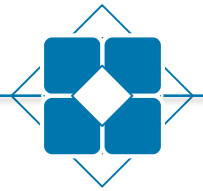
- **Available on almost all Linux/UNIX systems** (also MacOS).
- More modern shell exists (e.g. zsh, fish), but Bash remains the **default shell on most Linux systems**.
- **Good choice for scripting** because you can expect it to run on almost any Linux/UNIX machine.
- Full documentation available at https://www.gnu.org/software/bash/manual.



Screenshot of a terminal running the Bash shell.

Why learn to use Linux and shell commands ?

- **Remote computers (such as compute clusters) are usually running Linux** and generally only offer a command line interface (no graphical interface).
- Some programs are **only available in command line**.
- Some tasks are **simpler/faster to perform with a shell** and basic Linux commands.
- Shell scripts remain one of the simplest ways to **build data processing pipelines**.
- **Stability in time**: basic Linux and Bash commands have been working for the last 30 years, and will likely continue to work for the next 30.

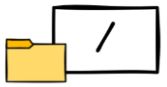


NAVIGATING THE FILE SYSTEM

Introduction to Linux / UNIX and the Bash Shell

Linux / UNIX file system

Directories and files are organized in an inverted tree structure.



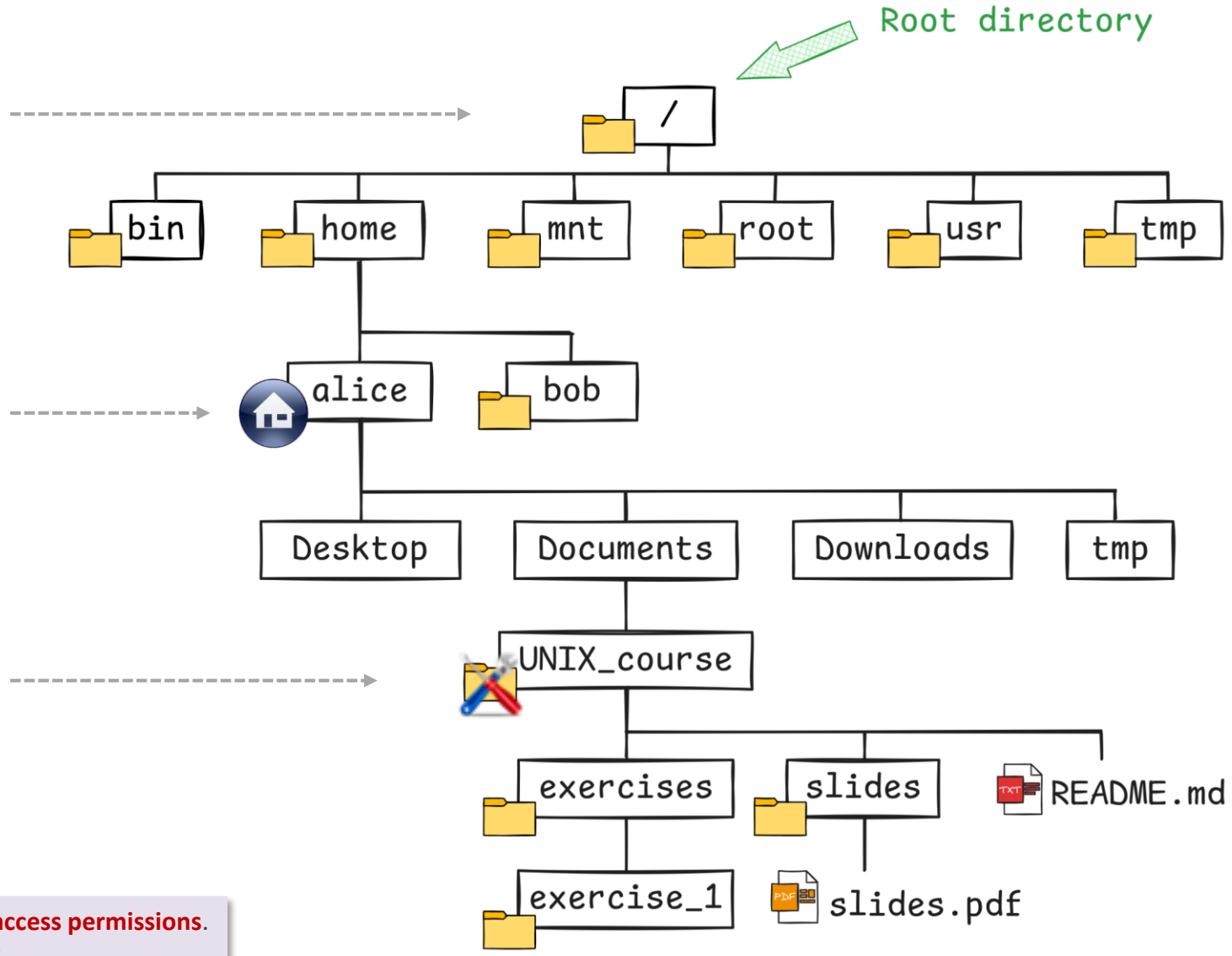
At the top of the tree is **/**, the **root directory** of the file system (cannot go above that).

Each user has a **home directory**. Its location depends on the OS:

Linux: `/home/Alice`
MacOS: `/Users/Alice`
Windows (not UNIX): `C:\Users\Alice`

The **current working directory** is the directory in which we are currently located in our shell.

Access to directories and files is subject to **access permissions**. Not all directories are accessible to all users.



File and directory path

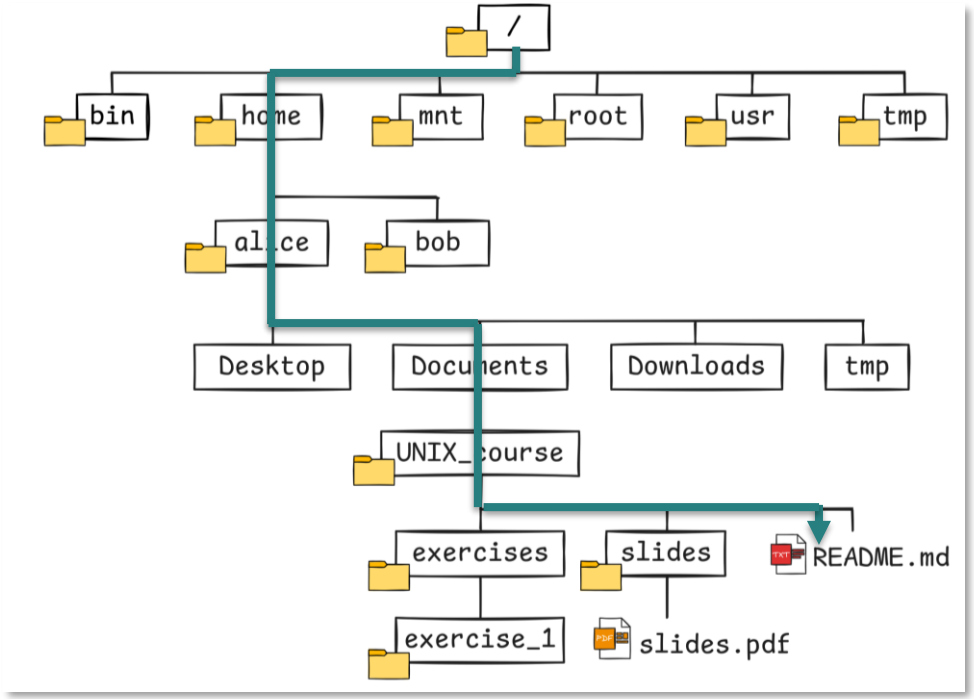
- The location of each file/directory is given by its **path**: a sequence of directory names separated by a **/** (forward slash).
- Paths can be **absolute** or **relative**.

Absolute path

Location from the root of the file system.
Always start with **/** .

- Absolute path of file `README.md` →

```
/home/alice/Documents/UNIX_course/README.md
```



Relative path

Location relative to the current **working directory**.

- Path from `/home/alice` →

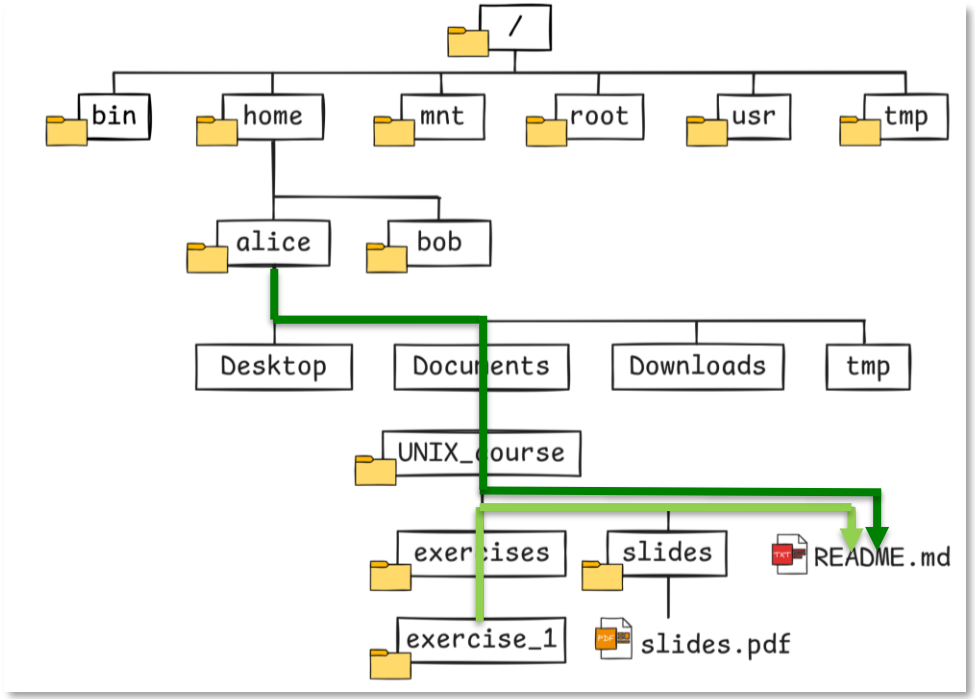
```
UNIX_course/README.md
./UNIX_course/README.md
```

. = **current directory**.

- From `exercise_1` →

```
../../README.md
```

.. = **parent directory**.



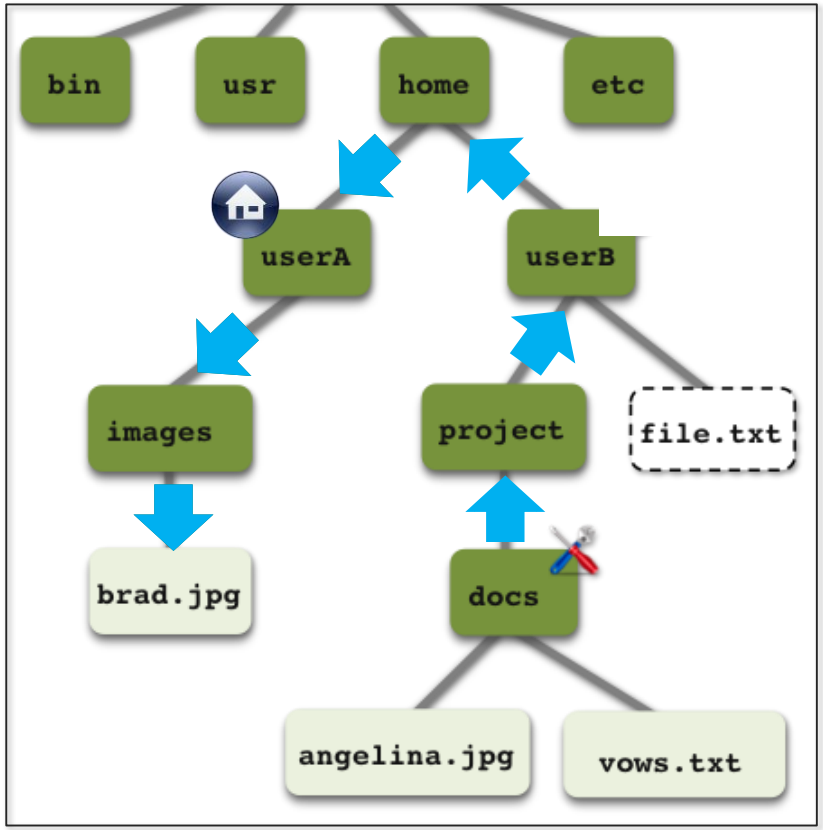
Tilde expansion, current and parent directories

`~` Expanded by the shell to the user's directory (**tilde expansion**).

`~/images` → `/home/userA/images`

`.` Relative path of the **current working directory**.

`..` Relative path of the **parent directory**.



Examples: here the current directory is '`docs`'

- **Absolute path** to brad.jpg

```
~/images/brad.jpg
```

- **Relative path** to brad.jpg

```
../../../userA/images/brad.jpg
```

Parent directory

- Copy brad.jpg to **current directory**

```
cp ../../../home/userA/images/brad.jpg .
```

Navigating the file system: `pwd`, `ls` and `cd`

`pwd`

Prints the **absolute path of the current working directory**.

```
> pwd
/home/alice/documents
```

`ls`

Lists the content of the specified directory.

```
> ls /home/alice/documents/
a_directory  a_file.txt  b_directory  executable_file.sh
```

With no arguments: prints content of current working directory.

```
> ls
directory  file.csv  another_file.md
```

`cd`

Changes the working directory to the specified directory.

```
> cd /home/alice/documents/

# Some shortcuts
cd ..  # Go to parent directory.
cd ~   # Go to home directory.
cd     # Go to home directory.
cd -   # Go back to previous location.
```

`ls` command options

`ls` By default, only file/directory names are listed.

```
❯ ls /home/alice/documents/
a_directory  a_file.txt b_directory  executable_file.sh
```

`ls -l` Adding the `-l` option displays **detailed information**.

```
❯ ls -l
total 16
drwxrwxr-x 2 alice a-team 4096 Jan  9  2023 a_directory
-rw-rw-r-- 1 alice a-team   39 Jan  9  2023 a_regular_file.txt
drwxrwxr-x 2 alice a-team 4096 Aug 23  2023 b_directory
-rwxrwxr-x 1 alice a-team  269 Jan  6  2023 executable_file.sh
```

directory (**d**) vs. file (**-**) | access permissions | owner | group | size [bytes] | last modification date

Some other useful `ls` options

- `-h` prints file and directory sizes in **human readable format** (`--human-readable`). Must be combined with `-l`.
- `-a` shows all files, including hidden files (`--all`).
- `-t` **sort by modification time** (most recent first), default sorting being by name.
- `-r` reverts sorting order (`--reverse`).
- `-R` list content recursively (`--recursive`), i.e. also list content of subdirectories.
- `--group-directories-first` list directories before files.

Case sensitivity

The shell **differentiates between upper and lower case** letters.

- For file/directory names:

```
brad.jpg  ≠  BRAD.jpg  ≠  brad.JPG
```

- And also for command and options:

```
cd  ≠  Cd  ≠  CD
```

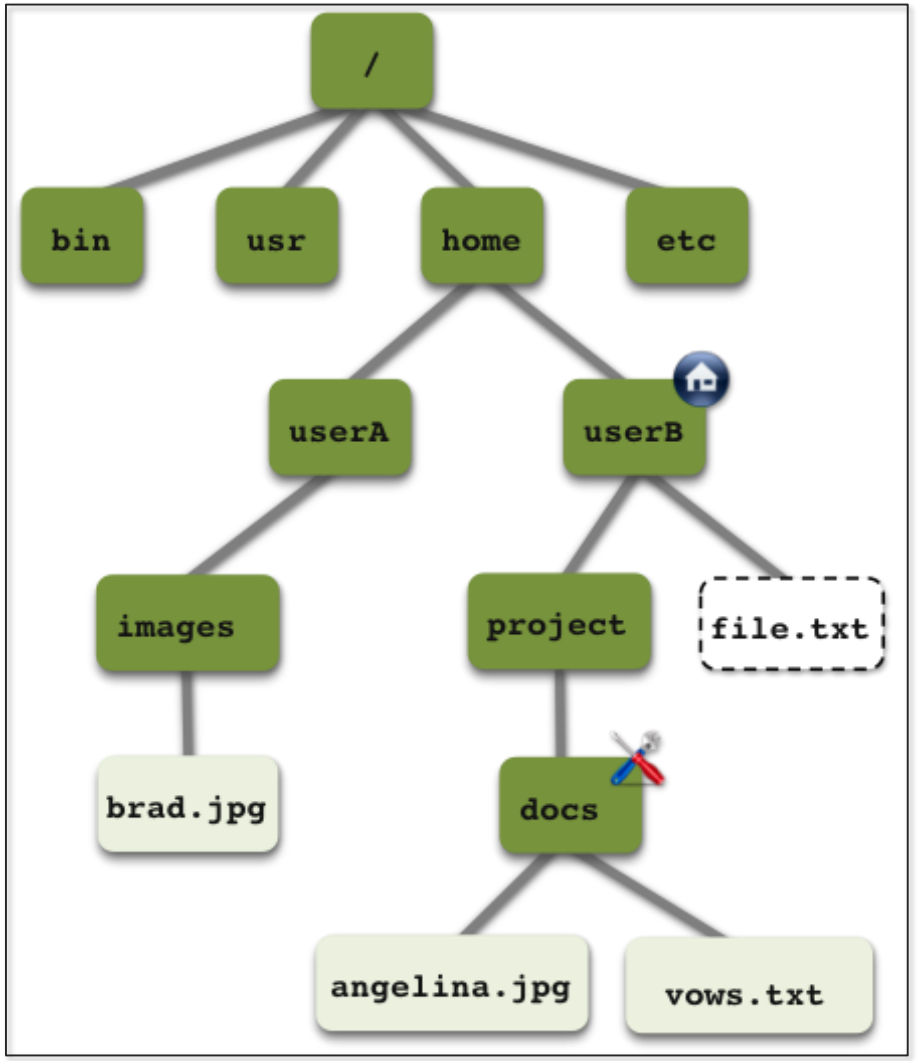
```
ls -r  ≠  ls -R
```

QUIZ

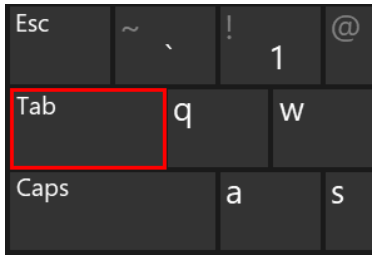
Current working directory: 'docs'.
How would you refer to the file 'brad.jpg' in the 'images' directory?

- ❑ /home/userA/images/brad.jpg
- ❑ ../../../../../../home/userA/images/brad.jpg
- ❑ ../../../userA/images/brad.jpg
- ❑ ~/../userA/images/brad.jpg

All proposed answers are correct



File/directory auto-completion



Manually typing directory and file names is **slow and error prone** !

⇨ use the **Tab** key to **auto-complete file and directory names**.

When you press **Tab** while writing a file/directory name:

- The shell will autocomplete (as much as possible) the file name.
- If there are multiple file name matches for the characters you started to type, the shell will stop the auto-completion at the point where the names diverge (at this point, if you press **Tab 2x**, the shell will list the different possibilities).
- To continue auto-completion, you will need to type additional characters, then press Tab again.

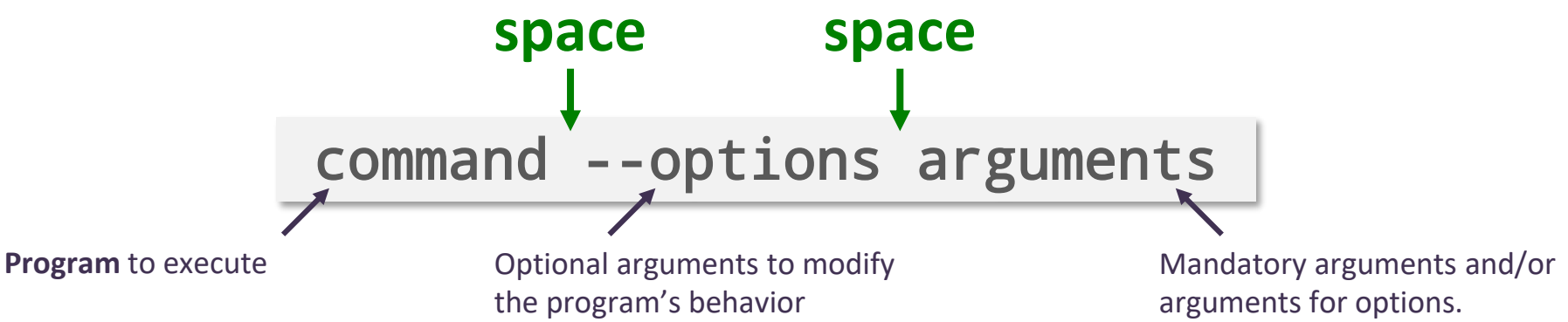
Tab auto-completion is **faster** and **safer** (no typos): use it.

It also works to with command names, as well as command options for some programs (not all).

Shell command syntax

Commands, options and arguments must be **separated by a space** (more than one is OK, but unnecessary).

- By default, commands are executed in the current working directory.



If an argument (e.g. a file name) contains a space, it must be **quoted**.
This is why **spaces in file/directory names are a bad idea!**

```
mv "poor file name.md" better_file_name.md
```

Alternatively, spaces can also be **escaped** by placing a **** in front of them.

```
mv poor\ file\ name.md better_file_name.md
```

Shell command syntax (continued)

Options often have a **short** and a **long form**.

Short form

- Prefixed with a single "**-**"
- Faster to type
- Multiple options can be combined

```
ls -l -t -a -r -h
```

```
ls -ltarh
```

Long form

- prefixed with "**--**"
- More explicit name

```
ls -l -t --all --reverse --human-readable
```

Note: `-l` and `-t` do *not* have a long form.

Command history

The shell keeps a record of the commands you type (up to a limit).

To retrieve commands typed earlier, one can:

- Use the ↑ (up) and ↓ (down) keys to browse through command history.
- Use the command `history`
- To view the history page by page, use `history | less`
- Search a for a command in the history with Ctrl + r, then type the string you search for.

`history`

```
5017  mkdir --help
5018  alias
5019  ls --all --human-readable
5020  ls --all --human-readable --group-directories-first
5021  ls --all --human-readable -d
5022  ls --all --human-readable -G
5023  ls --all --human-readable -g
5024  ls --all --human-readable -l
5025  #ls -lt --all --reverse --human-readable
5026  ls -lt --all --reverse --human-readable
5027  history
5028  history | less
5029  history | less +G
5030  history
5031  ls --all --human-readable
5032  ls --all --human-readable --group-directories-first
5033  ls -l --all --human-readable --group-directories-first
5034  alias
5035  man cd
5036  pwd
5037  history | less
5038  history | less +G
5039  history
```

Displaying help

Commands often have many options: trying to memorize them is a waste of time.

Most commands come with **some form of help**, that can usually be displayed with one (or all) or the following:

```
man command
```

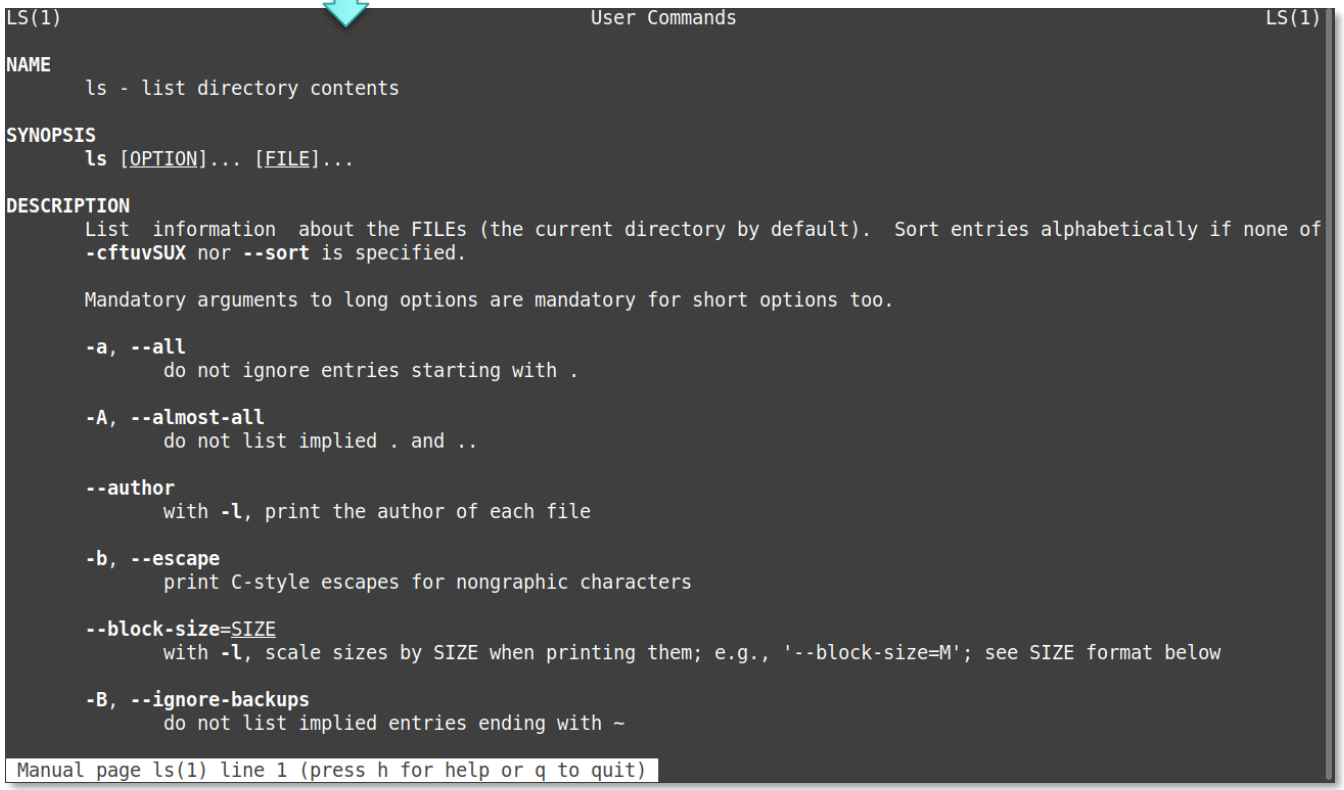
```
command --help
```

Sometimes also:

```
command -h
```

```
man ls
```

To exit **man** , type **q**.

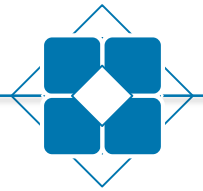


Summary: file system navigation

`pwd`	Outputs the current working directory.
`ls`	List content of a directory. Show details with `ls -l`
`cd`	Change directory (make another directory the current working directory)
`man`	Show help for a command (does not work for all commands).
`history`	Show command history.

`> cd exercise_1/`

Navigating the file system in command line



SHELL EXPANSIONS (BRACE, TILDE, AND FILENAME)

Introduction to Linux / UNIX and the Bash Shell

* There are more expansion types than the 3 shown here, but these are beyond the scope of this course.

Summary: shell expansion

To make a user's life easier and allow for more powerful commands, the shell processes commands *before* running them, and perform a number of **expansions*** in the following order:

1. Brace expansion: the expansion of `{…}` into individual elements.

2. Tilde expansion: the expansion of `~` into the user's home directory absolute path.

`~` → `/home/Alice`

3. Filename expansion: the expansion of wildcards `*`, `?` , `[…]` to match existing file and directory names.

The command you type.

```
ls ~/images/{*.jpeg,*.png}
```

```
ls ~/images/*.jpeg ~/images/*.png
```

```
ls /home/alice/images/*.jpeg /home/alice/images/*.png
```

```
ls /home/alice/images/logo.jpeg /home/alice/images/funny.jpeg
   /home/alice/images/funnier.png
```

The command the shell runs.

Brace expansion

Brace expansion is the expansion of elements between `{…}` into individual elements.

- Elements to expand must be separated by commas: `{item,item2,item3}`.
- Characters before/after the `{…}` are repeated for each expanded element.

```
prefix-{item1,item2,item3}-suffix  ➡  prefix-item1-suffix  prefix-item2-suffix  prefix-item3-suffix
```

Examples

```
mkdir project/{data,scripts,publications}  ➡  mkdir project/data project/scripts project/publications
```

Nesting expansions also works.

```
mkdir {data/{samples,backup},scripts/{python,R}}  ➡  mkdir data/samples data/backup scripts/python scripts/R
```

Brace expansions of **sequence expressions**

```
{1..5}      ➡  1 2 3 4 5
{a..f}      ➡  a b c d e f
{A..Z}      ➡  A B C … X Y Z
{1..10..2}  ➡  1 3 5 7 9
```

Example: generate a sequential list of directories.

```
mkdir sample_{a..f}  ➡
drwxrwxr-x 2 bob a-team 4.0K Feb 15 14:04  sample_a
drwxrwxr-x 2 bob a-team 4.0K Feb 15 14:04  sample_b
drwxrwxr-x 2 bob a-team 4.0K Feb 15 14:04  sample_c
drwxrwxr-x 2 bob a-team 4.0K Feb 15 14:04  sample_d
drwxrwxr-x 2 bob a-team 4.0K Feb 15 14:04  sample_e
drwxrwxr-x 2 bob a-team 4.0K Feb 15 14:04  sample_f
```

Filename expansion (globbing)

The shell has the ability to expand a set of wildcard characters to match *existing* files.
This feature is called **filename expansion** (or **globbing**), and is very useful to apply the same operation to multiple files.

Examples

- List all **`.csv`** files starting with **`sample_A`**

```
› ls -l sample_A1.csv sample_A23.csv sample_A1008.csv
-rw-r----- 2M Feb 12 2025 sample_A1.csv
-rw-r----- 5M Feb 12 2025 sample_A23.csv
-rw-r----- 7M Feb 12 2025 sample_A1008.csv
```

Using filename expansion

```
› ls -l sample_A*.csv
-rw-r----- 2M Feb 12 2025 sample_A1.csv
-rw-r----- 5M Feb 12 2025 sample_A23.csv
-rw-r----- 7M Feb 12 2025 sample_A1008.csv
```

- Move all files ending in **`.txt`** into the directory **`data/`**

```
› mv file.txt another.txt third.txt more.txt data/
```

Using filename expansion

```
› mv *.txt data/
```

Filename expansion (globbing): wildcard characters

As can be seen in some of the examples here, different wildcards can be combined in the same expression. (e.g. "`*.???`" or "`sample_[0-5]*`")

`*` Matches any number of characters (zero or more).

`*.md`	Matches: `README.md` / `notes.md` / `license.md`
`k*.jpg`	Matches: `kangaroo.jpg` / `koala.md` but not `Kookaburra.md`
`sample*`	Matches: `sample.csv` / `sample` / `sample_23.txt`

`?` Matches exactly 1 character. Can be repeated to match a (e.g. "??" matches any group of exactly 2 characters).

`?oose.png`	Matches: `goose.png` / `moose.png`
`*.???`	Matches: `goose.png` / `moose.png` but not `Kookaburra.md`

`[...]` Matches 1 character given in the list or range inside the [ ].

- A list of individual characters: `[ATgc]` / `[.-]` / `[abcdef]`
- A range of characters: `[a-f]` / `[a-fA-F]` / `[0-9]`
- Add `!` to negate the search: `[!abc]` -> any character except lower case "a", "b" and "c".

`[aBx]_sample`	Matches: `a_sample` / `B_sample` / `x_sample` but not `A_sample` / `aB_sample`
`sample_[0-5]*`	Matches: `sample_31` / `sample_5b` but not `sample_7` / `sample_71`

Note: while file expansion wild cards look similar to "regular expressions" (regexp), they are not exactly the same.

Filename expansion (globbing)

Warning: the shell only expands wildcards if there is an actual file/directory that matches !

Trying to creates multiple directories ? This is *not* the way to do it !

```
❯ mkdir sample_[a-f]
❯ ls -l
drwxrwxr-x 2 bob a-team 4.0K Feb 15 14:03 'sample_[a-f]'
```

```
❯ mkdir sample_{a..f}
```


Correct way of creating directories named sample_a, sample_b, … sample_f

Trying to rename multiple files ? This will *not* work !

```
❯ mv Sample_* sample_*
mv: target 'sample_*' is not a directory
```

```
❯ for file in Sample_*; do mv ${file} ${file/S/s}; done
```

Correct way of renaming all files starting with “Sample_” into “sample_”

Note: writing loops is beyond the scope of this course – this is just to show it’s possible.

Preventing filename expansion

Filename expansion is a great feature of the shell, but there are also situations where we do *not* want it to occur.

Problem: since filename expansion happens before the command is run, if there is a file matching the *.csv pattern, the command will either return an error or (even worse!) not the result we expect.

```
❯ find . -name *.csv
find: paths must precede expression: `sample_2.csv'
find: possible unquoted pattern after predicate `-name'?
```

To avoid filename expansion where we do not want it, we can take advantage of the fact that:

wildcards are ***not expanded*** when:

- They are inside quotes – 'single' or "double"
- They are "escaped" - i.e. preceded by a \

Let's apply this to our `find` command:

```
❯ find . -name "*.csv"
❯ find . -name '*.csv'
❯ find . -name \*.csv
./sample_3.csv
./sample_1.csv
./sample_2.csv
```

Problem solved: 3 different - and in this case equivalent - ways to prevent filename expansion.

`echo` - output a line of text

Display a line of text – i.e. print to standard output.

```
echo some text ...
```

Options

`-n` do not output the trailing newline
`-e` enable interpretation of backslash escapes.

Useful for:

- Testing a file expansion or brace expansion pattern without actually running the command.

```
❯ echo rm ./data/sample_[0-5]?.csv
rm ./data/sample_01.csv ./data/sample_02.csv ./data/sample_23.csv ./data/sample_51.csv

❯ echo mkdir -p ./project/{data,script,publication/{figures,text}}
mkdir -p ./project/data ./project/script ./project/publication/figures ./project/publication/text
```

- Quickly creating new text files.

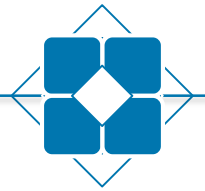
Note: the **>** and **>>** operators perform **standard output redirection**. We will see this later, but it essentially means that the output is saved to a file instead of being printed to screen.

```
❯ echo "This text will go into a new file" > new_file.txt

❯ echo "This will add another line to the file" >> new_file.txt
❯ echo "This will overwrite the file" > new_file.txt
```

`> cd exercise_2/`

Filename expansion with wildcard characters



FILE AND DIRECTORY MANAGEMENT (CREATE, MOVE, COPY, DELETE)

Introduction to Linux / UNIX and the Bash Shell

`mkdir` - create new directories

New directories are created with the "make directory" command: `mkdir <new dir name>`

- Directories can be created using **absolute or relative paths**.

```
# Create a new directory "data" in the current working dir.
❯ mkdir data

# Note: the parent directory "data" must already exist.
❯ mkdir /home/alice/data/samples_day1
```

- **Multiple directories** can be created by passing multiple arguments (directory names).

```
# Create 3 new directories.
❯ mkdir data scripts figures

# Create 3 new directories inside "project" (must already exist).
❯ mkdir project/{data,scripts,figures}
```

The `{}` syntax is a feature of the shell called **brace expansion**.
It is *not* something specific to `mkdir`.

- The `-p/--parents` option allows to create **multiple levels of directories at once**.

```
# Both "data" and "samples_day1" directories are created.
❯ mkdir -p data/samples_day1
❯ mkdir --parents data/samples_day1  # long form.
```

Naming rules (files and directories)

When naming files/directories, it's best to stick to a limited set of characters.

Recommended characters: `a-z A-Z 0-9 - _ .`

- Lower and upper cases letters, numbers, hyphen (dash), underscore and dot.

Forbidden characters: `/`

- `/` is used by the shell as path separator.
- "`.`" and "`..`" are also non-usable names (reserved as relative path of the current and parent directory).

Not-recommended characters:

- characters with a particular meaning for the shell: `space , ; : * ? & % $ | ^ ~ ' '' () [] {} ! \ #`
- international characters, such as for instance: `é à æ ñ ç`

Avoid spaces in file and directory names, as these will have to be escaped when typing the name of the file.

```
❯ ls file name with space.md
ls: cannot access 'file': No such file or directory
ls: cannot access 'name': No such file or directory
ls: cannot access 'with': No such file or directory
ls: cannot access 'space.md': No such file or directory

❯ ls "file name with space.md“
❯ ls file\ name\ with\ space.md

❯ ls "My Documents"
❯ ls My\ Documents
```

```
❯ ls file_name_without_space.md

❯ ls documents
❯ ls my-documents
```

Filename extensions

File extensions are arbitrary and do not have a particular meaning for Linux/UNIX operating systems. However, they are useful for users.

Here are some example of common extension conventions:

`.sh`	Shell scripts
`.py`	Python scripts
`.txt`	Text files with no particular format
`.csv`	Tabulated text files with Comma-Separated Values
`.zip`	Compressed ZIP archive
`.tar`	TAR archive (a bundle of uncompressed files)
`.tar.gz`	Compressed TAR archive file (compressed with gzip)

`mv` - move and/or rename files and directories

The **`mv`** command can be used to:

`mv file-or-dir(s) existing-dir`	**Move** one or more files/directories to another existing directory.
`mv file-or-dir new-name`	**Rename** a file/directory.
`mv file-or-dir existing-dir/new-name`	**Move and rename** a file/directory.

Examples

- Move a file or directory to another location.

```
# Move "sample.csv" into the directory "data"
❯ mv sample.csv data/

# Move multiple files into the directory "data".
❯ mv sample1 sample2 sample3 data/

# Move directory "data" into the directory "new-project".
❯ mv data/ new-project/
```

The trailing `/` of directories is optional.

If the directory "data" does not exist, "sample.csv" gets renamed to "data" (a classical error when doing a typo in the target directory name).

Good Practice: use auto-completion so you are sure the target directory exists.

- Rename a file.

```
❯ mv sample1.csv sample2.csv
```

If the a file with the new name already exists, that file gets overwritten (with no warning by default)

Safe option: to avoid overwriting existing files by mistake, the `-i` / `--interactive` option can be added.

- Move and rename a file.

```
❯ mv data/sample.csv data/samples-A/sample-01.csv
```

`cp` - copy files and directories

To avoid overwriting existing files by mistake, the `-i` / `--interactive` option can be added.

The `cp` command can be used to:

`cp file(s) destination-dir`	**Copy** one or more files.
`cp -r dir destination-dir`	**Copy** a directory (and all its content) by adding the `-r` / `-R` / `--recursive` option.
`cp file-or-dir existing-dir/new-name`	**Copy and rename** files and directories (copy with a different name).

Examples

- Copy files.

```
# Copy a file under a different name (in the same directory).
❯ cp script.sh script_COPY.sh

# Copy a file to the directory "~/templates" and give
# it a different name.
❯ cp process_data.sh ~/templates/process_data_template.sh

# Copy the files "script.sh" and "sample.csv" located
# in directory "~/backup/" to the current working directory.
❯ cp ~/backup/script.sh ~/backup/sample.csv .
```

Reminder "`.`" indicates the current working directory.

- Copy directories.

```
# Copy the directory "data" from the current working
# directory to the existing directory "~/backup".
❯ cp -r data/ ~/backup/

# Same as above, but the directory is renamed.
❯ cp -r data/ ~/backup/project_data_backup
```

The trailing `/` of directories is optional.

`rm` and `rmdir` - delete files and directories

`rmdir directory` Delete one or more **empty directories** (only empty directories can be deleted).

`rm file` Delete one or more **files**.

`rm -r directory` Recursively delete one or more **directories and all of their content**.
The 3 options are equivalent: `-r` / `-R` / `--recursive` .

There is no "undo" for deleting files. **Use with caution**.
Especially in combination with wildcard characters.

Examples

- Delete empty directories.

```
❯ mkdir test_{1..3}
❯ rmdir test_1 test_2 test_3  # or rmdir test_[1-3]
```

- Delete files.

```
❯ rm file-A file-B

# Before deleting, we use "ls" to verify which files
# will get deleted.
❯ ls sample_*.tsv
sample_01 sample_01A sample_32 sample_x86
❯ rm sample_*.tsv
```

Tip: before executing an `rm` command with wildcard characters, simulate it with the `ls` command as a control.

- Delete a directory and all its content (recursive deletion).

```
❯ rm -r data/
```

Summary: file and directory management

`mkdir` Create new directories.

`mv` Move and/or rename files/directories.

`cp` Copy files (and directories with the **`-r`** / **`-R`** / **`--recursive`** option).

`cp -r` Copy directories.

`rm` Delete files (and directories with the **`-r`** / **`-R`** / **`--recursive`** option).

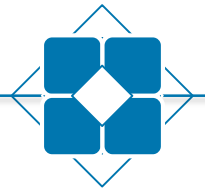
`rm -r` Delete directories and all their content.

There is no "undo" for deleting files. **Use with caution** (especially in combination with wildcard characters).

`rmdir` Delete empty directories

`> cd exercise_3/`

File management basics: create, copy, move, and delete files and directories



FILE ACCESS PERMISSIONS AND OWNERSHIP

Introduction to Linux / UNIX and the Bash Shell

Users and permissions

On Linux and UNIX systems, every file/directory has a set of **3 access permissions** that are set for **3 categories of users**.

3 access permissions

```
read    [r]
write   [w]
execute [x]
```

3 categories of users

```
user    [u] --------> Owner of file.
group   [g] --------> Users who belong to the group associated with the file.
others  [o] --------> Everyone.
```

Each file:
- Has only **1 owner**.
- Belongs to only **1 group**.

```
❯ ls -lh
drwxrwxr-x 2 alice a-team 4.0K Jan  9 2023 a_directory
drwxrwxr-x 2 alice a-team 4.0K Aug 23 2023 b_directory
-rw-rw-r-- 1 alice a-team   39 Jan  9 2023 regular_file.txt
-r--r--r-- 1 alice a-team   41 Aug 24 2023 read_only_file.md
-rwxrwxr-x 1 alice a-team  269 Jan  6 2023 executable_file.sh
```

user (owner) | group | others | owner | group | file size | last modification date | file name

Access permissions | User groups

Users and permissions

User	Group	Others
rwx	rw-	r--

Owner of file

Users who are members of the file's group

All other users on the machine

- `r` : read permission is granted.
- `w` write permission is granted.
- `x` execute permission is granted.
- `-` permission at the given position is not granted.

	Permission on file	Permission on directory
Read [`r`]	Access (view) file content.	List content of directory.
Write [`w`]	Modify file content.	Modify content: add, remove and rename files.
Execute [`x`]	Run file as an executable (script, program).	Enter the directory (cd into it), and access files inside it.

Note: to make a directory properly readable, **read** *and* **execute** permissions must be granted.

The **root user** (system admin) **always has access to all files/directories**, even if you have set them to be only accessible to the user or group.

Technical note: when a new file/directory is created and a user belongs to multiple groups, the *primary group* of the user is used as "group" for the file/directory.

File permissions: examples

Directory and file are only accessible to user “alice”.

```
❯ ls -lh
drwx------ 2 alice a-team 4.0K private_directory
-rw------- 1 alice a-team   39 private_file.txt
```

Directory and file accessible to “alice” and users who belong to group “a-team” (in read-only mode).

```
❯ ls -lh
drwxr-x--- 2 alice a-team 4.0K directory
-rw-r----- 1 alice a-team   39 readable_by_group.txt
```

Directory and file accessible to anyone, but only writable by user and group.

```
❯ ls -lh
drwxrwxr-x 2 alice a-team 4.0K directory
-rw-rw-r-- 1 alice a-team   39 readable_by_anyone.txt
```

Permission modifications

Modifying access permissions is done with the command: `chmod`

`u` = user
`g` = group
`o` = others
`a` = all

`r` = read
`w` = write
`x` = execute

```
chmod [u g o a] [+ - =] [r w x] file_name
```

`+` adds a permission
`-` removes a permission
`=` applies permissions (e.g. "u=rwx,g=rx,o=-")

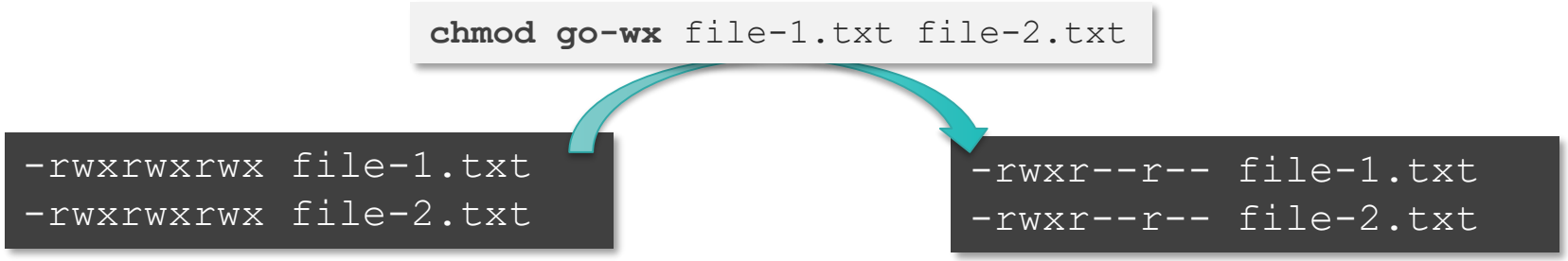
Option:

`-R` to apply changes recursively

`chmod` can only be run by the **file owner or by root** (i.e. system admin)

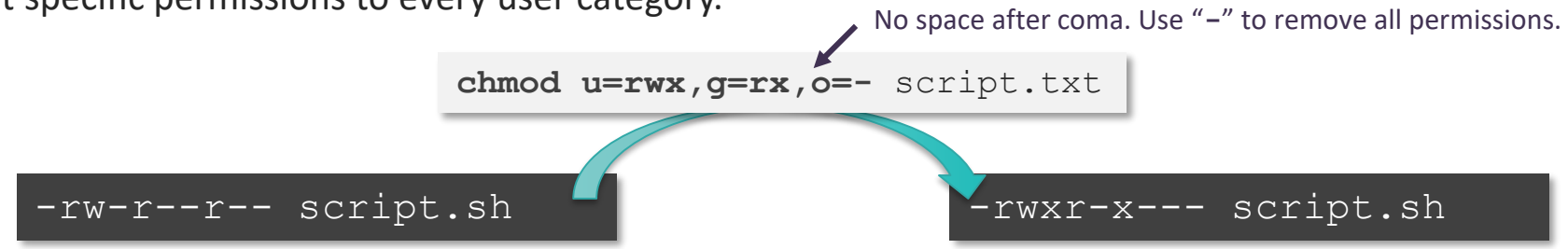
Permission modifications: examples

Example 1: remove write and execution permissions for “group” and “others” .



Example 2: give execution permission to everyone (user, group and others).

Example 3: set specific permissions to every user category.



Permission modifications

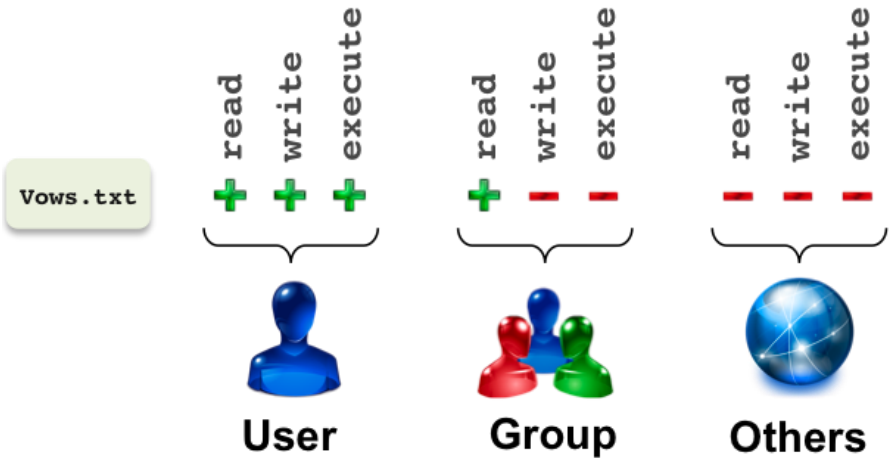
Each configuration of permission can be replaced by a code.
For example ***chmod 700*** = all permissions for owner, but none for all others

ugo

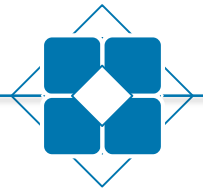
Number	Octal Permission Representation	Ref
0	No permission	---
1	Execute permission	--x
2	Write permission	-w-
3	Execute and write permission: 1 (execute) + 2 (write) = 3	-wx
4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-x
6	Read and write permission: 4 (read) + 2 (write) = 6	rw-
7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwx

QUIZ

Given the permissions of the file vows.txt, check all correct statements



- ❑ The root user, who does not belong to the same unix group as the owner, is not allowed to display the file.
- ❑ Users belonging to the same unix group as the owner can see/display the file.
- ❑ The owner of the file is granted all permissions.
- ❑ Users belonging to the same unix group as the owner can modify the file, but a warning is sent to the owner.
- ❑ Users belonging to the same unix group as the owner can "modify" the file, but they have to do a copy first.



LOCATING FILES: THE FIND COMMAND

Introduction to Linux / UNIX and the Bash Shell

`find` - search for files

Search for files and directories in the file system.

```
find <query location>  <query expression>
```

Path (directory) from where to recursively search, i.e. this directory and all its sub-directories will be searched.

Criteria for a file/directory to be a match.

> **Reminder** of some useful locations
>
> `.` = current working directory.
> `/` = root of file system.
> `~` = user's home directory.

Examples

- Search the current working directory `.` for files named "script.sh" (case insensitive).

```
> find . -type f -iname "script.sh"
./projects/Script.sh
./utils/shell/script.sh
```

- Search the entire file system `/` for any file or directory whose name ends in ".sh".

```
> find / -name "*.sh"
/home/redqueen/projects/off-with-their-heads.sh
/tmp/run_me.sh
```

Some useful query expressions

Multiple query expressions can be combined.

Search for a **file name**

`-name "string"`	case sensitive
`-iname "string_*"`	case insensitive

Wildcard characters, `*` `?` `[]` (same as for filename expansion) can be used in patterns to match.

Search for a **file type**

`-type f`	restrict search to regular files.
`-type d`	restrict search to directories.

Search for a **file size**

```
-size [+-]n[unit]
```

- `+` = larger than
- `-` = smaller than
- no sign = exactly equal to

Unit of value:
- c for bytes
- k for kilobytes
- M for megabytes
- G for gigabytes

```
-size +30M   => files larger than 30 megabytes.
-size -5k    => files smaller than 5 kilobytes.
-size  10k   => files with size equal 10 kilobytes.
```

`find` - examples and tips

- Search patterns containing wildcards should be quoted (single or double), to avoid filename expansion.

```
❯ find /home/redqueen -name '*.sh'
/home/redqueen/run_me.sh
/home/redqueen/projects/off-with-their-heads.sh
```

✓

```
❯ cd /home/redqueen
❯ find . -name *.sh
/home/redqueen/run_me.sh
```

✗

The problem: `*.sh` is expanded into `run_me.sh` before the `find` command runs, and therefore `find` is instructed to search for `run_me.sh` and not `*.sh`.

- Adding `!` in front of a query negates it - finds files that that do *not* match the criteria (only negates the query term that immediately follows it).

```
❯ find demo_data/ ! -name 'sample_[12].csv' -type f
demo_data/sample_3.csv
demo_data/species_names_with_duplicates.txt
demo_data/species_names.txt
```

≠

```
❯ find demo_data/ -name 'sample_[^12].csv' -type f
demo_data/sample_3.csv
```

Only " -name sample_[12].csv" is negated. To also negate "-type f" we would need:

```
! -name sample_[12].csv ! -type f
```

`( )` have a special meaning for the shell (create a subshell), and must therefore be escaped.

- Query terms can be combined with `-and` / `-or` boolean operators, and grouped with `\(  \)` .

```
❯ find . -name "*.mpeg" -size +300M
❯ find . -name "*.mpeg" -and -size +300M
```

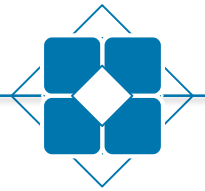
`-and` is the default operator between queries, so these 2 commands are the same

```
# Find files that end in either ".mpeg" or ".avx"
❯ find . -name '*.mpeg ' -or -name '*.avx'

# Find files > 2Gb that end in either ".mpeg" or ".avx"
❯ find . \( -name '*.mpeg' -or -name '*.avx' \) -and -size +2G
```

`> cd exercise_4/`

Finding files



DISPLAYING FILE CONTENT

Introduction to Linux / UNIX and the Bash Shell

`head` and `tail` - output first/last X lines

Output the first (`head`) or last (`tail`) X lines of a file or data stream.

Useful for:

- Taking a quick peek at a file.
- Preview the output of a data processing pipeline (see **Tip** section below).

`head -X` `head -nX`	Output the first X lines
`head -n-X`	Output all except last X lines

```
head file.txt        # Output the first 10 lines (default).
head -2 file.txt     # Output the first 2 lines.
head -n2 file.txt    # Same as above.
head -n-3 file.txt   # Output all lines, except the last 3.
```

`tail -X` `tail -nX`	Output the last X lines
`tail -n+X`	Output all lines from X until the end of the file.

```
tail file.txt        # Output the last 10 lines (default).
tail -2 file.txt     # Output the last 2 lines.
tail -n2 file.txt    # Same as above.
tail -n+3 file.txt   # Output all lines, except the first 3.
```

Tip: **head** can be useful when writing small pipelines to do intermediate checks of the output without each time printing-out the entire data.

```
› cut -f2 input.csv | sort -r | head -3
Zamenis lineatus
Vipera seoanei
Rattus xanthurus
```

`cat` - output entire file content

Prints input files to standard output, con**cat**enating them if multiples files are passed as input.

Useful for:

- Displaying small files in the terminal.
- Merging files together by row (concatenating by row).
- Getting the content of a file to *stdout* (standard output), e.g. to then pipe it into another command.

`cat file`	Output (print to *stdout*) the entire content of a file.
`cat file1 file2 ...`	Output (print to *stdout*) the entire content of all specified files, one after the other. This concatenates the files.

Examples

```
# Display entire content of a file.
❯ cat README.md
# Introduction to Linux and the Bash shell

A half-day course providing an introduction to the open source
[Linux](https://en.wikipedia.org/wiki/Linux) operating system
and the [Bash shell](https://www.gnu.org/software/bash).

...
```

```
# Concatenate (merge by row) 3 data files and save them
# to a new file named "data_all.tsv"
❯ cat data_1.tsv data_2.tsv data_3.tsv > data_all.tsv
```

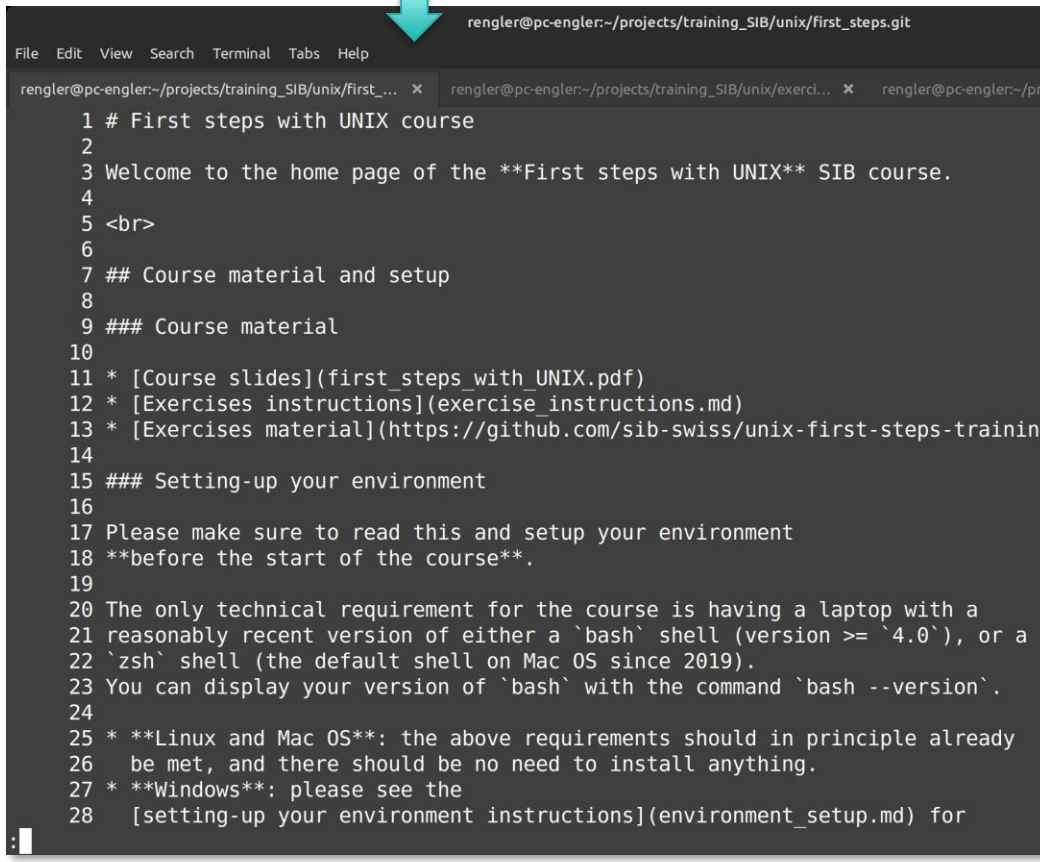
`less` - interactive file viewer

View the content of files (large or small) in an interactive way (scrollable content).

`less file` Opens a file with the `less` program.

Example

```
# Open the README.md file with less.
less README.md
less -N README.md  # with line numbers.
```

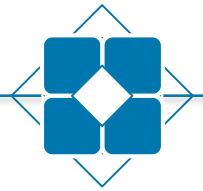


Useful commands (inside the program):

`up` / `down` arrows	Scroll up/down, 1 line at a time.
`space-bar`	Scroll down, 1 screen/page at a time.
`<` or `Home`	Go to start of file.
`>` or `End`	Go to end of file.
`-N`	Show/hide line numbers (can also be passed as option when starting `less`)
`q`	Exit the program

Searching for content (inside the program):

`/string` + `Enter`	Search for the specified string. E.g. `/foo` searches for the string "foo" in the file.
`n`	Go to next match.
`N`	Go to previous match (i.e. search up from current position).

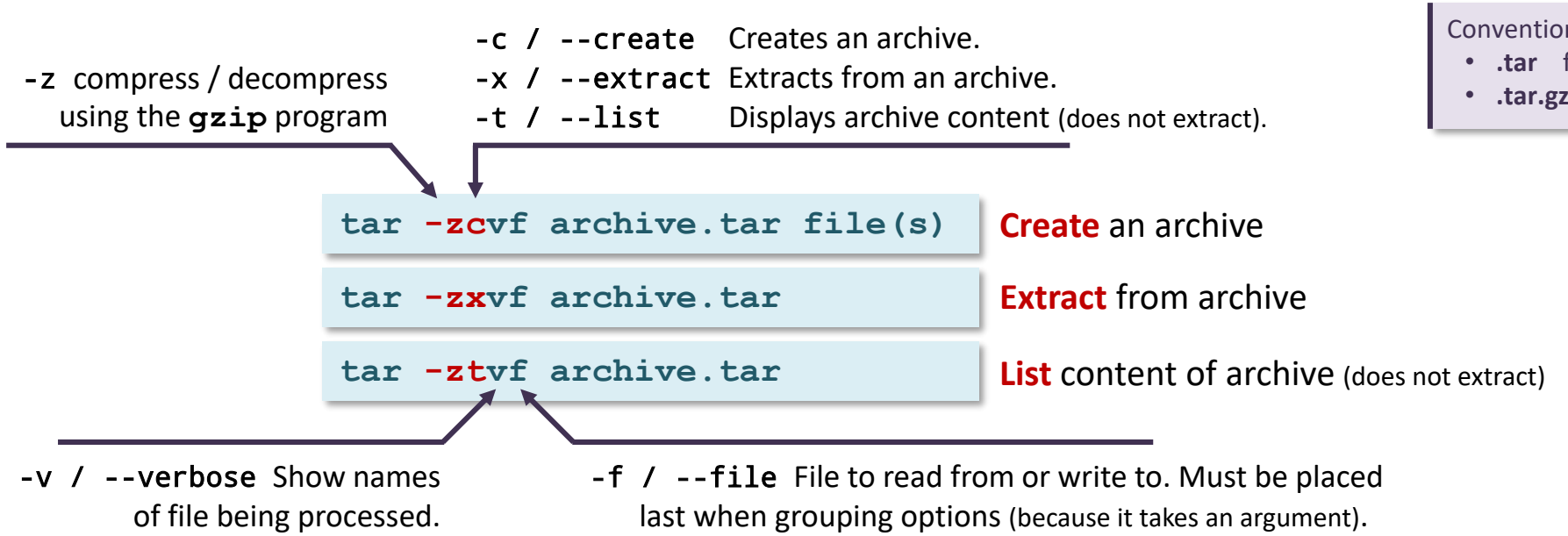


ARCHIVING AND COMPRESSING DATA

Introduction to Linux / UNIX and the Bash Shell

`tar` - archiving files

TAR (**T**ape **AR**chive) is a file format that groups multiple files into a single archive file (often referred to as a "tarball). By default, `tar` does not compress /decompress data, but `gzip` compression can easily be added via the `-z` option.



Convention for filename extensions:
- **.tar** for uncompressed tar archives
- **.tar.gz** for gzip compressed tar archives

Other options

`-m / --touch`	When extracting files, use the current time as last modified date (don't extract file modified time).
`-u / --update`	Append files to an existing tar archive. Only works with uncompressed (.tar) archives.
`-C, --directory`	Decompress in the specified directory (instead of the current working directory).

The TAR format was initially developed to write data to supports such as magnetic tapes, where data is read sequentially. On such medium, it is much more efficient to write/retrieve large sequential blocks of data rather than multiple small files.

`tar` - examples

- Create a `.tar.gz` (compressed TAR) archive that contains the directory **`samples_iris`**, as well as the 3 files **`sample_1.csv`**, **`sample_2.csv`** and **`sample_3.csv`**.

```
❯ tar -zcvf samples.tar.gz samples_iris/ sample_?.csv
samples_iris/
samples_iris/sample_iris_versicolor.tsv
samples_iris/sample_iris_virginica.tsv
samples_iris/sample_iris_setosa.tsv
sample_1.csv
sample_2.csv
sample_3.csv
```

- List the content of the archive we created.

```
❯ tar -ztvf samples.tar.gz
drwxrwxr-x alice/a-team    0 2025-03-01 17:40 samples_iris/
-rw-rw-r-- alice/a-team 1290 2025-03-01 17:39 samples_iris/sample_iris_versicolor.tsv
-rw-rw-r-- alice/a-team 1250 2025-03-01 17:40 samples_iris/sample_iris_virginica.tsv
-rw-rw-r-- alice/a-team 1068 2025-03-01 17:39 samples_iris/sample_iris_setosa.tsv
-rw-rw-r-- alice/a-team   18 2025-02-28 17:06 sample_1.csv
-rw-rw-r-- alice/a-team   23 2025-02-28 17:06 sample_2.csv
-rw-rw-r-- alice/a-team   15 2025-02-28 17:06 sample_3.csv
```

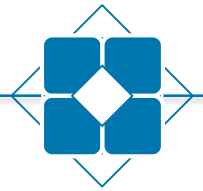
- Decompress/extract the archived files.

```
# Extract to current working directory (default)
❯ tar -zxvf samples.tar.gz
samples_iris/
samples_iris/sample_iris_versicolor.tsv
samples_iris/sample_iris_virginica.tsv
samples_iris/sample_iris_setosa.tsv
sample_1.csv
sample_2.csv
sample_3.csv
```

```
# Extract to a different directory, using -C option.
❯ tar -zxvf samples.tar.gz -C ~/testdir
samples_iris/
samples_iris/sample_iris_versicolor.tsv
samples_iris/sample_iris_virginica.tsv
samples_iris/sample_iris_setosa.tsv
sample_1.csv
sample_2.csv
sample_3.csv
```

```
> cd exercise_5/
```

Display file content



STANDARD INPUT/OUTPUT AND PIPES

Introduction to Linux / UNIX and the Bash Shell

Standard input/output/error (file descriptors)

Linux/UNIX has **3 standard data streams** (also known as "file descriptors").

Standard input (*stdin*): where programs receive data from.

- By default, *stdin* comes from the keyboard.
- Can come from a file using `<` (stdin redirection)
- Also known as "file descriptor 0".

Standard input redirection

```
cmd < file
```

Standard output (*stdout*): where programs write their output data to.

- By default, printed to the terminal window
- Can be piped into another command using `cmd | next-command …`
- Can be written to a file using `>` or `>>` (same as `1>` or `1>>`)
- Also known as "file descriptor 1".

Standard output redirection

```
cmd > file
```

> : Create or **overwrite** file

```
cmd >> file
```

>> : create or **append to** file

Standard error (*stderr*): where programs write their error messages to.

- By default, printed to the terminal window.
- Can be written to a file using `2>` or `2>>`
- Also known as "file descriptor 2".

Standard error redirection

```
cmd 2> file
```

```
cmd 2>> file
```

stdout + stderr redirection

```
cmd &> file
```

```
cmd &>> file
```

Standard input/output/error: examples

- Standard input redirection.

```
# 'tr' is a command that replaces one character with another. It only accepts input
# from stdin (cannot give a file name as argument).
❯ tr "\t" "," < samples.tsv
```

- Standard output redirection.

```
❯ echo "This goes to stdout..."
This goes to stdout...

❯ echo "This goes to stdout..." > new_file

❯ cat new_file
This goes to stdout...

❯ echo "This also goes to stdout..." >> new_file

❯ cat new_file
This goes to stdout...
This also goes to stdout...

❯ echo "Be careful, it's easy to overwrite files (no warning given)" > new_file
❯ cat new_file
Be careful, it's easy to overwrite files (no warning given)
```

A new file named **new_file** is created.

To append to the file - and not overwrite it - we use **>>** .

The existing file **new_file** is overwritten.

- Standard error redirection.

```
❯ ls non-existing-file 2>> error-log
❯ cat error-log
ls: cannot access 'non-existing-file': No such file or directory
```

Pipes: passing *stdout* from one command as *stdin* of the next command

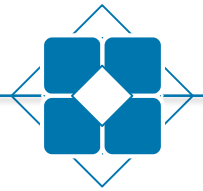
Unix philosophy:

- **Programs do one** thing, and do it well.
- **Programs work together**: the output of one program can be used as input of another.

```
› grep "^GO:" go-data.csv | sort | uniq -c | sort -nr | head -5
```

Pipes allow to pass the output of one program as the input of another

- **Combine functionalities** of individual commands.
- Multiple operations **without creating intermediate files**.
- **Performant** - next command can start even before the previous has completed.



TEXT PROCESSING UTILITIES

Introduction to Linux / UNIX and the Bash Shell

Commonalities (between commands shown in this section)

Input

- Most commands accept data on standard input (*stdin*):
  - **Piped** from another command.
  - From a file, using **standard input redirection**.
- Most also accept **file name(s) as argument**.
  - Depending on the command, multiple files may be passed.

```
… | cmd [options]
cmd [options] < file
cmd [options] file
cmd [options] file1 file2 …
```

Output

- All commands output by default to standard output (*stdout*), where you can then:
  - **Pipe** it to another command.
  - Save it to a file, using **standard output redirection**.
- Some commands accept an **output file name as argument**.

```
cmd | …
cmd > file
cmd >> file
```

Create/overwrite file.

Append to file.

```
cmd -o output_file
```

`-o` is here used as example, but the option can have another name.

Processing

- Most commands process data line-by-line.
  - Not suitable (or difficult) to combine data from multiple lines.

`wc` - file content statistics

Utility to **count lines, words and bytes** (or characters).

Useful to:

- Count number of lines in a file.
- Count number of lines on standard output.

```
wc [options] file(s)
```
← Input from file (passed as argument)

```
… | wc [options]
```
← Input from standard input (via a pipe)

Options

By default, **`wc`** prints lines, words, bytes.

`-l` print the number of lines.
`-w` print the number of words.
`-c` print the number of bytes.
`-m` print the number of characters.

Examples

- Count lines in a file.

```
❯ wc samples.csv
738  1820 13600 samples.txt
❯ wc -l samples.csv
738
```

- Count lines on *standard output*.

```
# Count files in a directory
❯ ls | wc -l
233
```

```
# Count lines across multiple files.
#  * Merge all files together with `cat`.
#  * Count lines with `wc`.
❯ cat sample_*.csv | wc -l
2538  1820 13600 samples.txt
```

- Counting characters: pitfalls.

```
# Some characters are "invisible"
# (by default, echo adds a newline \n)
❯ echo "foo" | wc -c
4
❯ echo -n "foo" | wc -m
3
```

```
# Non-ASCII characters use 2 bytes.
❯ echo "föö" | wc -c
6
❯ echo "föö" | wc -m
4
```

QUIZ

The file 'uniprot-mouse-virus.fasta' contains more than 7000 protein sequences. What will output the following command ?

```
head uniprot-mouse-virus.fasta | wc -l
```

- ❑ 6990
- ❑ 0
- ❑ 7000
- ❑ 10

`tr` - replace (translate) or delete characters

Utility to **replace or delete single characters**.

Replace char **x** by **y**

```
… | tr "x" "y"
```

Delete char **x**

```
… | tr -d "x"
```

Options

`-d / --delete` delete the specified character (instead of replacing).
`-s / --squeeze-repeats` squeeze/squash successive occurrences of a character: `"aaa"` -> `"a"`.

`tr` is one of the only Linux/UNIX utilities that does not accept a file name as argument. It only takes input from standard input.

```
tr "x" "y" file
```

However, we can use **standard input redirection** to stream data from a file to standard input.

```
tr "x" "y" < file
```

`tr` can only replace single characters. To replace multiple characters, use `sed`.

Examples

- Replace.

```
# Replace "g" with "M".
❯ echo "goose" | tr "g" "M"
Moose

# All instances of a character are replaced.
❯ echo "goose" | tr "g" "e"
geese

# Convert a Tab separated into a comma separated file.
❯ tr "\t" "," < input_file.tsv > output.csv
```

- Delete.

```
# Delete "f", then upper-case the letter "o".
❯ echo "fox" | tr -d "f" | tr "o" "O"
Ox
```

- Squeeze.

```
# Squeeze "o" characters.
❯ echo "only loosers spell it with two 'ooo's..." | tr -s "o"
only losers spell it with two 'o's...
```

`sort` - sort lines of text

Sort lines of text, from an input file or from standard input.

Useful for:

- Sorting file content or output of other commands.
- Sorting content before using `uniq` (which requires content to be sorted).

```
sort [options] file(s)
```

```
… | sort [options]
```

Options

The default sort order is alphabetical.

- `-n` for a numeric sort.
- `-r` for reverse sorting order.
- `-u` for unique lines (keeps one instance when identical lines).
- `-k` sort according to specific columns in the file (for tabulated data).

Examples

- Default sorting (alphabetical).

```
› cat species_names.txt
Rattus xanthurus
Acropora florida
Zamenis lineatus
Actenoides hombroni
Crocidura stenocephala
Vipera seoanei
Aioliops brachypterus
```

```
› sort species_names.txt
Acropora florida
Actenoides hombroni
Aioliops brachypterus
Crossoptilon mantchuricum
Rattus xanthurus
Vipera seoanei
Zamenis lineatus
```

- Default (alphabetical) vs. numeric sorting.

```
› sort numeric.txt
0  foo
10 foo
11 foo
12 foo
1  foo
20 foo
21 foo
22 foo
2  foo
```

```
› sort -n numeric.txt
0  foo
1  foo
2  foo
10 foo
11 foo
12 foo
20 foo
21 foo
22 foo
```

- Sort the output of another command (here `cut`).

```
› cut -f2 species.tsv | sort | uniq -c | head -3
      1  Acropora florida
      7  Crocidura stenocephala
      3  Vipera seoanei
```

`uniq` - delete duplicated lines of text

Filter *adjacent* lines of text (return a single occurrence of duplicated instances).

- Can optionally count the number of instances of each line.
- Warning: `uniq` only filters adjacent lines of text. Its input is therefore typically sorted first: `sort | uniq`.

Useful for:

- Removing duplicate lines.
- Counting the number of occurrences of each line.

```
uniq [options] file
```

```
… | uniq [options]
```

Options

`-c` precedes each reported line with the number of occurrences found.
`-i` case-insensitive when comparing the lines.

Example: count the number of occurrences of each line in a file - here counting frequencies of species.

```
› cut -f2 species.tsv | sort | uniq -c
      1  Acropora florida
      7  Crocidura stenocephala
      5  Rattus xanthurus
      3  Vipera seoanei
      4  Zamenis lineatus
```

Note: to simply sort content without counting occurrences, both `sort | uniq` and `sort -u` can be used.

`cut` - extract text by columns

Extracts selected parts of lines based on field delimiters (columns).

Useful to:

- Manipulate tabulated data.
- Extract specific fields from a text file or from standard input.

```
cut [options] file
```

```
… | cut [options]
```

Options

`-d` specify a custom delimiter character (default is Tab). Must be a single character.
Long form: **`--delimiter=`**
By default, **`cut`** expects files to be Tab delimited.

`-f` fields (columns) to extract on each line. Long form: **`--fields=`**
fields can be a single field (e.g. **`-f1`**), a list of fields (**`-f1,5,7`**) or a range (**`-f2-5`**).

Examples

Extract columns from a tab-delimited file (.tsv): this is the original file.

```
Group     Scientific_name          Common_name                    IUCN_status   Year
MAMMALS   Coendou speratus         Dwarf Porcupine                VU            2024
MAMMALS   Crocidura sokolovi       Sokolov White-toothed Shrew    NT            2024
BIRDS     Acrocephalus kerearako   Cook Islands Reed-warbler      LC            2024
```

Extract 1st, 3rd and 4th column.

```
› cut -f1,3-4 IUCN_RedList_2024.tsv
Group     Common_name                    IUCN_status
MAMMALS   Dwarf Porcupine                VU
MAMMALS   Sokolov White-toothed Shrew    NT
BIRDS     Cook Islands Reed-warbler      LC
```

Extract columns from a comma separated file (.csv)

```
Group,Scientific_name,Common_name,IUCN_status,Year
MAMMALS,Coendou speratus,Dwarf Porcupine,VU,2024
MAMMALS,Crocidura sokolovi,Sokolov White-toothed Shrew,NT,2024
BIRDS,Acrocephalus kerearako,Cook Islands Reed-warbler,LC,2024
```

Extract the 2nd column.

```
› cut --delimiter="," -f2 IUCN_RedList_2024.csv
Common_name
Dwarf Porcupine
Sokolov White-toothed Shrew
Cook Islands Reed-warbler
```

`diff` - compare file content

Compare 2 files and show difference on a line-by-line basis.

Useful for:

- Testing if 2 text files are identical.
- Visualizing differences between 2 text files.

```
diff [options] file1 file2
```

Options

- `-u` report differences in **unified format** (recommended).
- `-s` reports when two files are identical (it does not by default).
- `-q` reports only that files differ, no details of the differences.

Examples

- Output when files are identical.

```
❯ diff -su shopping-list.md shopping-list_copy.md
Files shopping-list.md and shopping-list_copy.md are
identical
```

```
# Without the -s option, nothing is printed
# when both files are identical.
❯ diff -u shopping-list.md shopping-list_copy.md
```

- Output when files differ.

```
❯ diff -u shopping-list.md shopping-list_2.md
--- example_diff_1.md  2025-02-26 17:11:44.805
+++ example_diff_2.md  2025-02-26 17:13:46.076
@@ -1,7 +1,8 @@
 # Shopping list
 apples
-bananas
+strawberries
 eggs
 milk
 pineapple
 passion fruit
+gooseberries
```

Line starting with a `-` : the line is only present in the 1st file.

Line starting with a `+` : the line is only present in the 2nd file.

`cat` and `paste` - merging text content

- **`cat`** merges (concatenates) input by rows.
- **`paste`** merges (concatenates) input by columns.

```
cat file1 file2 ...
```

```
paste file1 file2 ...
```

`paste` options

`-d`/`--delimiters` : use the specified delimiter. By default, **`paste`** uses Tab as field/column delimiter.

Example: merge the following 3 files with either **`cat`** or **`paste`**

```
file1   line1
file1   line2
file1   line3
```

```
file2   line1
file2   line2
file2   line3
```

```
file3
file3
file3
```

`cat`

`paste`

```
› cat file1.tsv file2.tsv file3.tsv
file1   line1
file1   line2
file1   line3
file2   line1
file2   line2
file2   line3
file3
file3
file3
```

```
› paste file1.tsv file2.tsv file3.tsv
file1   line1   file2   line1   file3
file1   line2   file2   line2   file3
file1   line3   file2   line3   file3
```

`cat` and **`paste`** write their output to *stdout*. To save the result to a file, we can use **standard output redirection**.

```
cat file1 file2 > combined_by_row.tsv
```

```
paste file1 file2 > combined_by_col.tsv
```

`> cd exercise_6/`

Convert, sort, compare and merge tabulated data

This exercise has a helper slide (see next slide)

Exercise 6 – helper slide: tabulated data

Free-text (non-tabulated data)

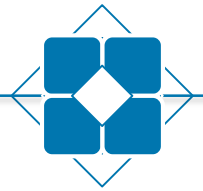
```
Hello, I will be very busy the next couple of months: I started
a course in Bioinformatics!
```

Comma/Semicolon-separated values (CSV)

```
Entry,Entry_name,Protein_name
Q8QMT5;A18_CWPXB;Transcript termination protein A18
Q80DV6;A18_CWPXG;Transcript termination protein A18
```

Tab-separated values (TSV)

```
Entry    Entry_name       Protein_name
Q8QMT5   A18_CPB  Transcript termination protein A18
Q80DV6   A18_CPX  Transcript termination protein A18
```



INTRODUCTION TO GREP AND REGULAR EXPRESSIONS

Introduction to Linux / UNIX and the Bash Shell

`grep` - pattern-based text search

Global **R**egular **E**xpression **P**rint – searches text (from files or *stdin*) for patterns and returns the matching lines (default) or patterns (with **`-o`** option).

```
grep [options] "pattern" file
```

Input can also be from *stdin* (standard input) instead of a file.

Options (some of them)

- **`-i`** performs a case insensitive search.
- **`-c`** displays the number of resulting lines instead of the lines themselves.
- **`-v`** displays lines that DO NOT match the pattern.
- **`-o`** returns only the pattern, not the entire line.
- **`-r`** search all files recursively inside each directory.
- **`-E`** use extended regular expressions.

Examples

By default, the entire matching line is returned. To return only the matching pattern, the **`-o`** option must be added.

From a file containing of the IUCN list of endangered species (**`IUCN_RedList_2024.tsv`**), select all rows with species classified as "vulnerable" (VU):

```
grep "VU" IUCN_RedList_2024.tsv
REPTILES    Vipera aspis            Asp Viper              VU G 2024-1
MAMMALS     Crocidura stenocephala  Kahuzi Swamp Shrew     VU N 2024-1
BIRDS       Burhinus superciliaris  Peruvian Thick-knee    VU N 2024-2
CORALS      Dipsastraea favus       Head Coral             VU N 2024-2
```

From vulnerable species, select only mammals and birds:

```
grep "VU" IUCN_RedList_2024.tsv | grep '^MAMMALS\|^BIRDS'
MAMMALS     Coendou speratus        Dwarf Porcupine        VU N 2024-2
MAMMALS     Crocidura stenocephala  Kahuzi Swamp Shrew     VU N 2024-1
BIRDS       Burhinus superciliaris  Peruvian Thick-knee    VU N 2024-2
```

Extract the binomial name of species from the file:

```
grep -oE " [A-Z][a-z]+ [a-z]+ " IUCN_RedList_2024.tsv
Vipera aspis
Crocidura stenocephala
Calidris falcinellus
```

Count the number of coral species:

```
grep -c "^CORALS" IUCN_RedList_subset_2024.tsv
82
```

Regular expressions (regexp, regex)

https://www.gnu.org/software/grep/manual/html_node/Regular-Expressions.html

Regular expressions are patterns that describe a match pattern in a string. They are somehow similar to the wildcard characters we encountered in filename expansion, but allow for much more powerful/flexible matches.

Here are some of the basic match patterns (this list is not exhaustive):

`.` Match any 1 character (exactly 1)

```
foo.bar  # matches "foo bar", "foolbar", not "foobar"
sn..ze   # matches "snooze", "sneeze", not "snoooze"
```

`*` Match zero or more of the preceding character/pattern

```
foo.*bar  # matches "foo bar", "foobar", "foo XyZbar", not "foo XyZar"
.*\.png   # matches any string ending in ".png"
```

This **"." must be escaped** to be interpreted as a literal "." and not as a wildcard.

`+` Match one or more of the preceding character/pattern

```
foo +bar  # matches "foo bar", "foo   bar", not "foobar", "footbar"
```

`{n}` Match exactly n of the preceding character/pattern.

```
o{2}  # matches "kangaroo", "foobar", not "oxygen", "coool"
```

Only 1 "o" — Too many "o"

`^` Start of line

```
^This  # matches "This is a newline", not " This is a newline." nor "this is a newline"
```

Line starts with a space — "t" is lowercase

`$` End of line

```
words$  # matches "famous last words", not "last words." or "these words don't match"
```

`[…]` Match 1 of the characters in the set

```
[0-59]x   # matches "0x" "3x" "9x" not "7x", "x", "12x"
[a-zT3]   # matches any 1 lowercase letter, "T", or "3".
```

`[^…]` Match any 1 character that is different from the set

```
[^0-9]      # matches any 1 character that is not a number.
[^0-9A-Zc]  # any 1 character that is not a number nor a capital letter, nor "c".
```

`> cd exercise_7/`

Introduction to **`grep`**

This exercise has a helper slide (see next slide)

Exercise 7 – helper slide: FASTA file format

Biological sequences are often represented in a so-called **FASTA format**.

- Text-based format for representing either **nucleotide (DNA, RNA)** or **peptide (proteins)** sequences.
- Each sequence starts with a **header** (always on a single line):
  - Header lines **always start with '>'**, the remainder of the line is free text.
  - Always a **single line**.
- Subsequent lines (one or more) contain the actual sequence (nucleotides or amino acids):
  - Each nucleotide or amino acid is represented by a single letter.
  - No officially enforced line length (but often 80 chars is used).
  - Variable number of lines.

1st sequence	**header**	**>sp|P92981|APR2_ARATH 5'-adenylylsulfate reductase 2 OS=Arabidopsis thaliana GN=APR2 PE=1 SV=2**
	sequence	MALAVTSSSTAISGSSFSRSGASSESKALQICSIRLSDRTHLSQRRYSMKPLNAESHSRS ESWVTRASTLIAPEVEEKGGEVEDFEQLAKKLEDASPLEIMDKALERFGDQIAIAFSGAE RKEAWLVVLYAPWCPFCQAMEASYIELAEKLAGKGVKVAKFRADGEQKEFAKQELQLGSF PTILLFPKRAPRAIKYPSEHRDVDSLMSFVNLLR
2nd sequence	**header**	**>sp|P45947|ARSC_BACSU Protein ArsC OS=Bacillus subtilis GN=arsC PE=1 SV=1**
	sequence	MENKIIYFLCTGNSCRSQMAEGWAKQYLGDEWKVYSAGIEAHGLNPNAVKAMKEVGIDIS NQTSDIIDSDILNNADLVVTLCGDAADKCPMTPPHVKREHWGFDDPARAQGTEEEKWAFF QRVRDEIGNRLKEFAETGK
3rd sequence	**header**	**>sp|Q5RAN9|CASQ2_PONAB Calsequestrin-2 OS=Pongo abelii GN=CASQ2 PE=2 SV=1**
	sequence	MKRTHLFIVGVYVLSSCRAEEGLNFPTYDGKDRVVSLSEKNFKQVLKKYDLLCLYYHEPV SSDKVAQKQFQLKEIVLELVAQVLEHKAIGFVMVDAKKEAKLAKKLGFDEEGSLYILKGD RTIEFDGEFAADVLVEFLLDLIEDPVEIISSKLEVQAFERIEDYIKLIGFFKSGDSEYYK

Thank you for your attention

Acknowledgements

- Gregoire Rossier (SIB training)
- Vassilios Ioannidis (Vital-IT)