Shell Commands Course

Flavio Copes

Updated August 8, 2026

Contents

Learn the shell commands used to navigate files, manage permissions, connect programs with pipes, transform text, and control running processes.

What you'll learn: Use the command line confidently to inspect a system, transform text, automate repeated work, and diagnose common problems.

Navigation basics

The shell, paths, directories, command help, and history.

The Command Line for Complete Beginners

A beginner's guide to the command line: what the terminal and shells like Bash, ZSH and Fish are, plus core commands like ls, cd and man to get started.

Computers are great because using the mouse or touch devices we can do a lot of work, leaving the keyboard to typing our emails, blog posts or reports.

Once upon a time, this didn't exist. Computers were only accessed using the keyboard, typing weird and cryptic messages, called commands, in a terminal.

This was true when computers were big as entire rooms, but also was true when the first personal computers started becoming affordable and they would start with a BASIC command prompt.

Today, well hidden and never used by consumers, we still have this way of using our computer.

We can do so using a **terminal**.

macOS comes with an app called Terminal, appropriately named.

Microsoft provides an app called Windows Terminal.

And Linux users know their terminals very well.

The terminal is not just for programmers. It's essential for every professional computer user, because it unlocks you things that are impossible to do with a GUI (Graphical User Interface).

Oh, I mention GUI. The acronym for the terminal is CLI (Command Line Interface).

There is not just one terminal. You wish. Instead, we have lots and lots of different terminal interfaces, called **shells**.

We have Bash, ZSH, Fish Shell, CSH, and many more. But the most popular ones are Bash and ZSH.

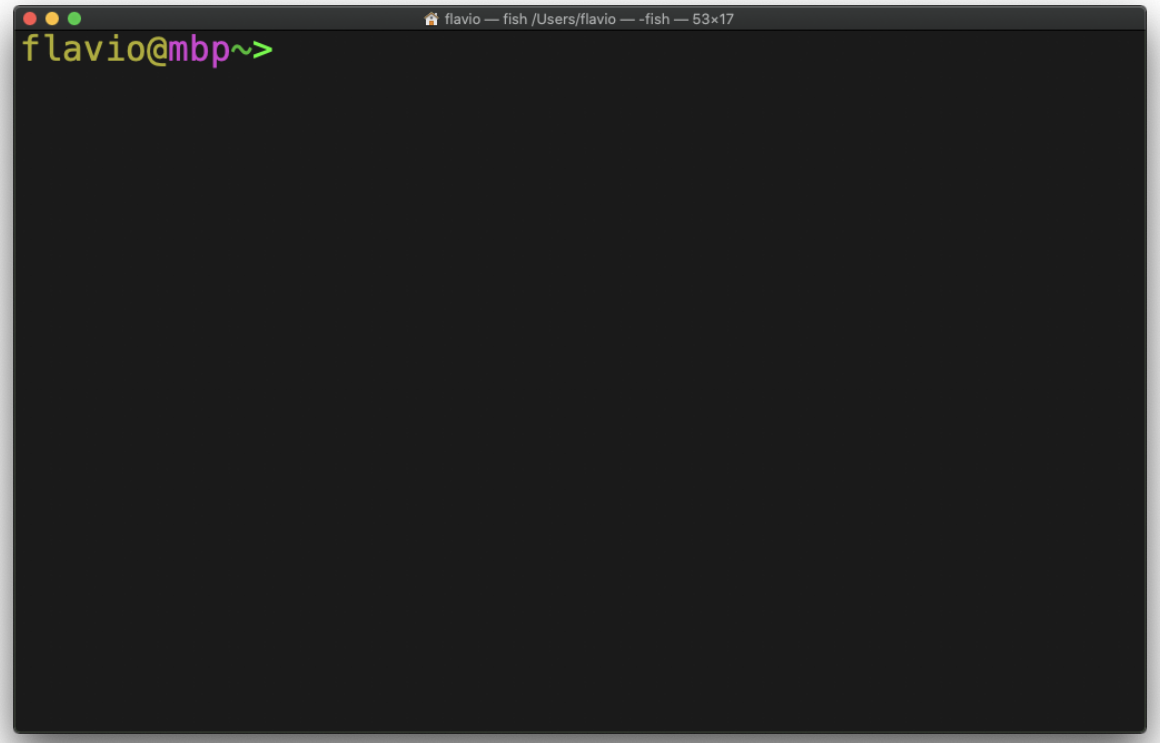
Bash is often the default, and until very recently it was the default macOS shell, but now it's been changed to ZSH.

My favorite shell, to be honest, is Fish Shell. I really like it because it's simple, straightforward, comes with great defaults, and also a web-based configuration. I don't have time to spend configuring my shell prompts and colors manually and when something works out of the box, I take it.

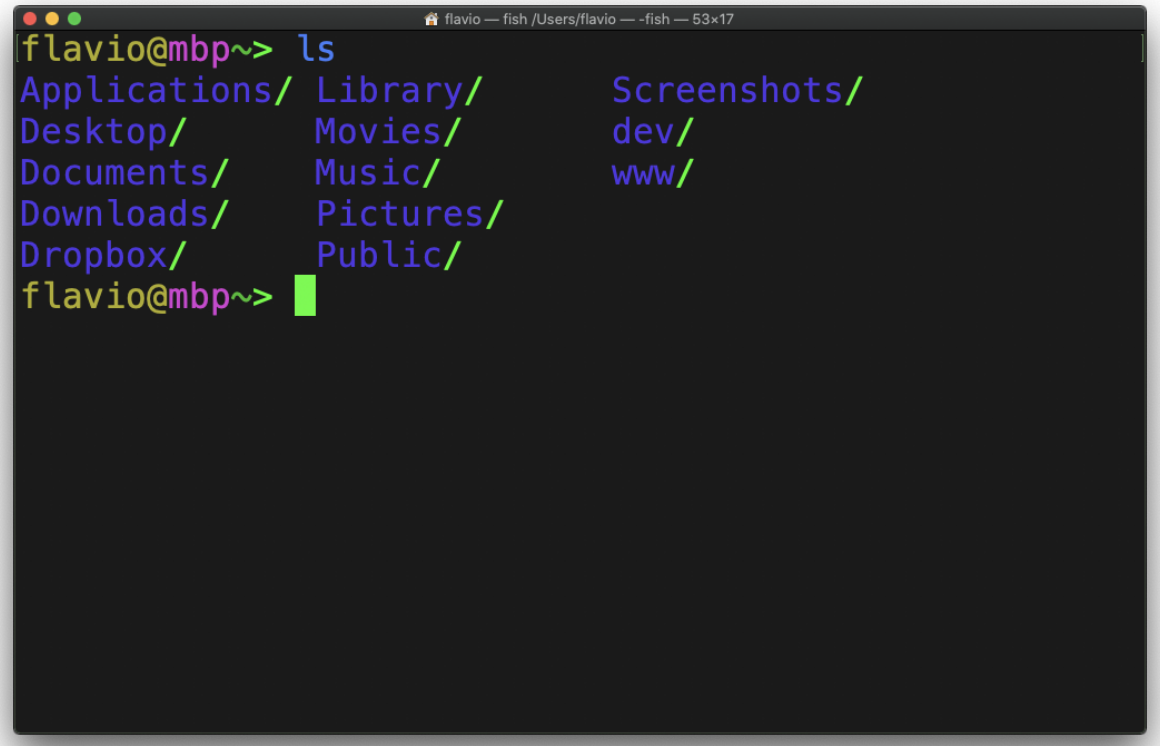
The terminal is also the way you access a **server**. You can create a VPS server on Amazon or DigitalOcean or where you want, and you can access it using SSH, the secure protocol to connect to a remote shell.

You use the shell locally but you could actually be connected to a server at the other side of Earth, which is pretty cool. Except some delay, if the connection is a bit laggy.

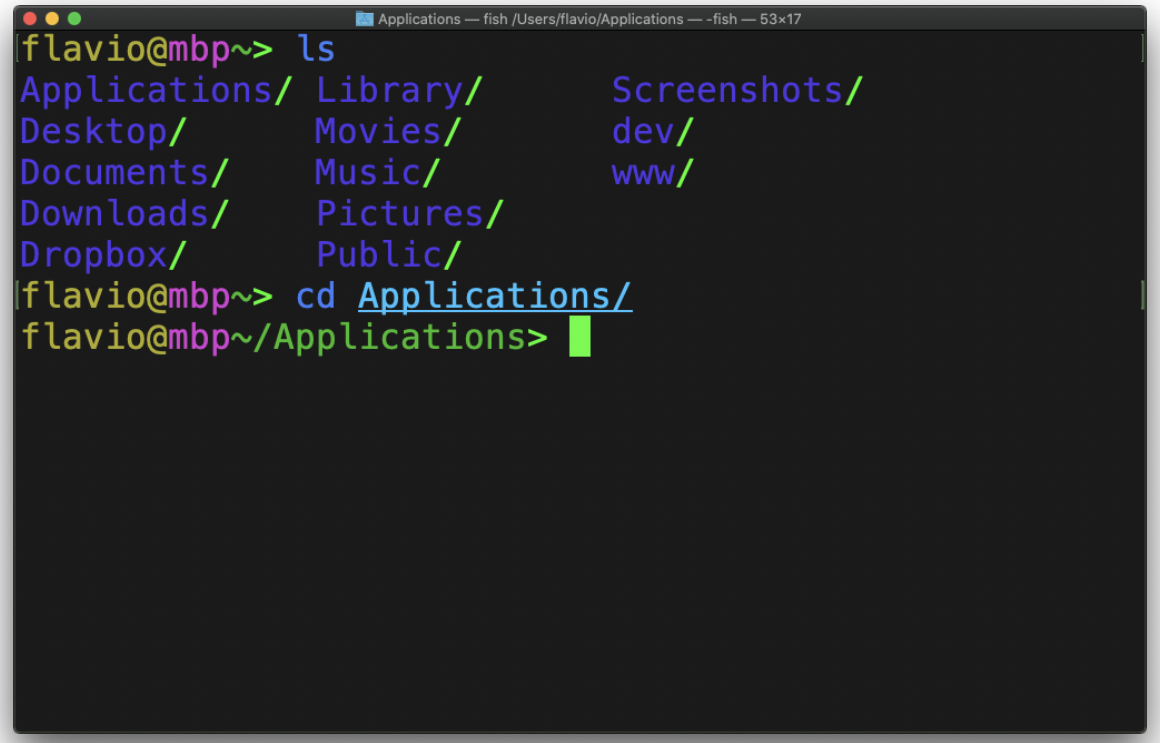
This is my macOS shell:



I can list all the files in my home folder by typing `ls` and pressing enter:



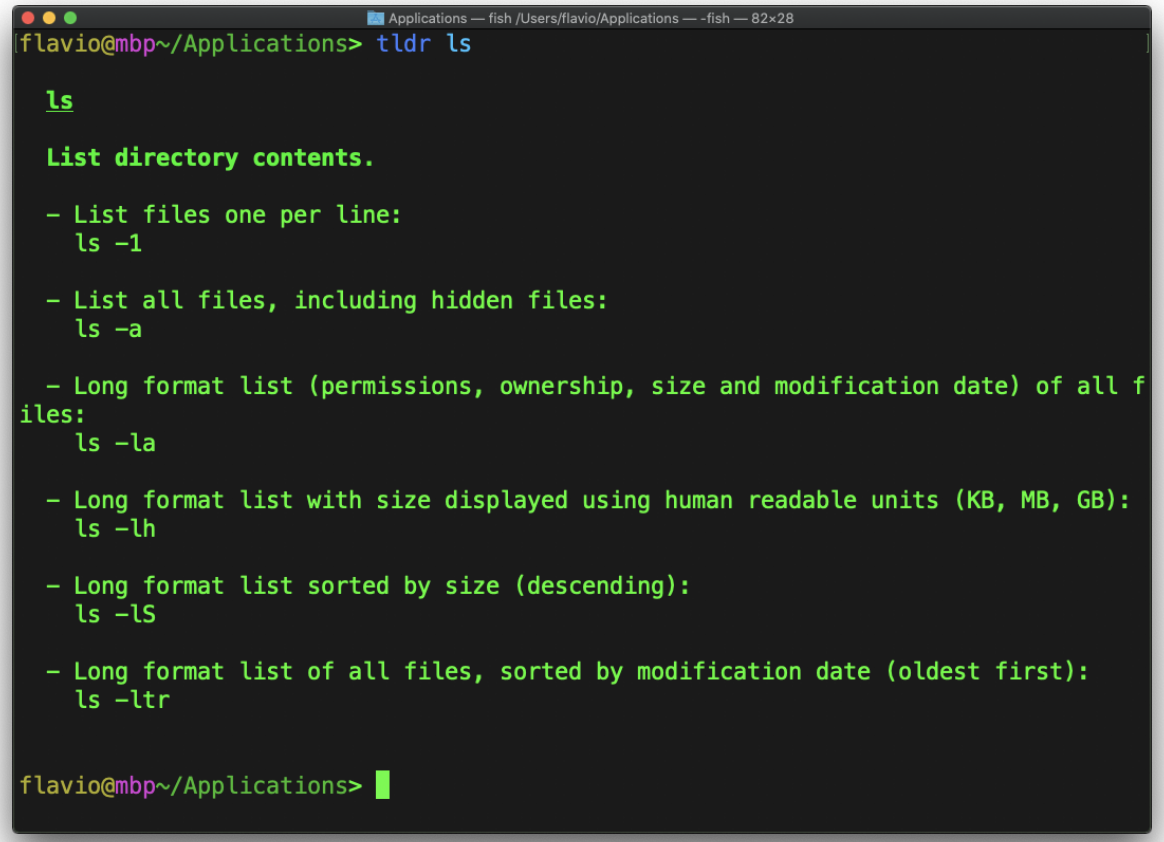
I can change the current working directory to another folder using the `cd` command:



And every time I don't know how to use a command, I type `man <command>` to get the manual:

This is a man page. Although I find man pages to contain too much information, as this is 1 of 14 screens of explanation for the `ls` command.

Most of the times when in need I use this site called *tldr pages*: https://tldr.sh/. It's a command you install, then you run it like this: `tldr <command>`



It just gave me a few useful options with examples.

Anyway, I'm digressing. When typing commands you can move around with your left and right arrow to move the cursor around.

Some systems also let you use the mouse to go to a specific place in the line. For example on macOS I can use option-click to position the cursor anywhere I want.

Pressing the up arrow will show the command you last typed. It's nice when you make a typo and you don't need to re-type all the thing.

If you type a command not found, the shell will tell you:

I made a few tutorials on using shells:

- How to use the macOS terminal
- The Bash shell
- Unix Shells Tutorial
- Introduction to Bash Shell Scripting

There's a lot to read if you want!

Linux commands: pwd

Learn how the Linux pwd command prints your current working directory, so you always know exactly where you are in the filesystem when you feel lost.

Every shell process has a current working directory. Relative paths start there, so the same command can affect different files depending on where you run it.

```
pwd
```

The output is an absolute path such as `/Users/flavio/projects/site`. Use it before a destructive command, after several `cd` operations, or when a script behaves differently in two terminals.

Compare absolute and relative paths:

```
pwd
ls notes.txt
ls "$(pwd)/notes.txt"
```

The last two commands identify the same file. Quoting `$(pwd)` matters because a directory name may contain spaces.

The shell also keeps the current path in `$PWD`:

```
printf '%s\n' "$PWD"
```

`pwd -P` resolves symbolic links and prints the physical path. Plain `pwd` may preserve the logical

route you used to enter a symlinked directory. That distinction matters when debugging scripts that compare paths.

Try it: create a directory, enter it through a symbolic link, then compare `pwd` and `pwd -P`. The command works on Linux, macOS, WSL, and other Unix-like environments.

Linux commands: ls

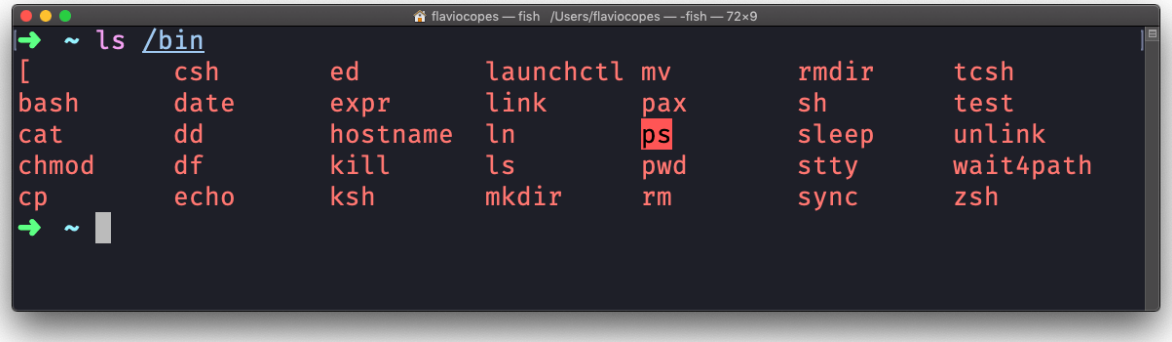
Learn how the Linux ls command lists the files in a folder, and how the -al option adds details like permissions, owner, and size, plus hidden files.

Inside a folder you can list all the files that the folder contains using the `ls` command:

```
ls
```

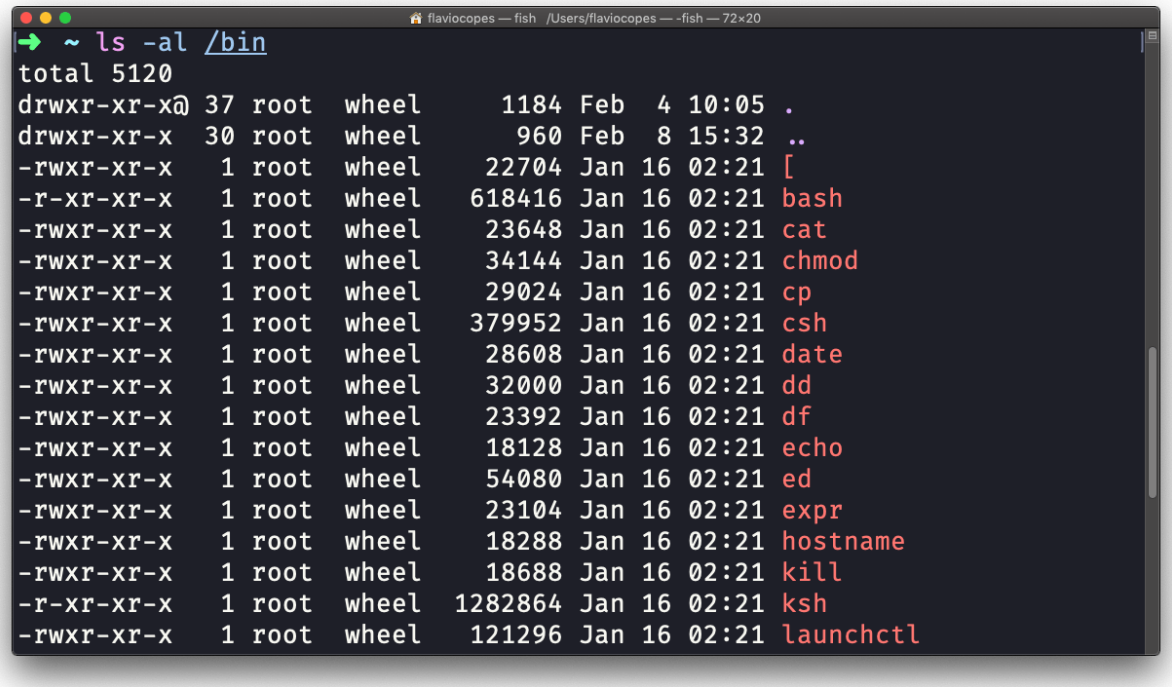
If you add a folder name or path, it will print that folder contents:

```
ls /bin
```



`ls` accepts a lot of options. One of my favorite options combinations is `-al`. Try it:

```
ls -al /bin
```



compared to the plain `ls`, this returns much more information.

You have, from left to right:

- the file permissions (and if your system supports ACLs, you get an ACL flag as well)
- the number of links to that file
- the owner of the file
- the group of the file
- the file size in bytes
- the file modified datetime
- the file name

This set of data is generated by the `l` option. The `a` option instead also shows the hidden files.

Hidden files are files that start with a dot (`.`).

> This command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: cd

Learn how the Linux cd command changes the directory you are in, using a folder name, the .. parent and . current shortcuts, or an absolute path from /.

Once you have a folder, you can move into it using the `cd` command. `cd` means **c**hange **d**irectory. You invoke it specifying a folder to move into. You can specify a folder name, or an entire path.

Example:

```
mkdir fruits
cd fruits
```

Now you are into the `fruits` folder.

You can use the `..` special path to indicate the parent folder:

```
cd .. #back to the home folder
```

The # character indicates the start of the comment, which lasts for the entire line after it's found.

You can use it to form a path:

```
mkdir fruits
mkdir cars
cd fruits
cd ../cars
```

There is another special path indicator which is `.`, and indicates the **current** folder.

You can also use absolute paths, which start from the root folder `/`:

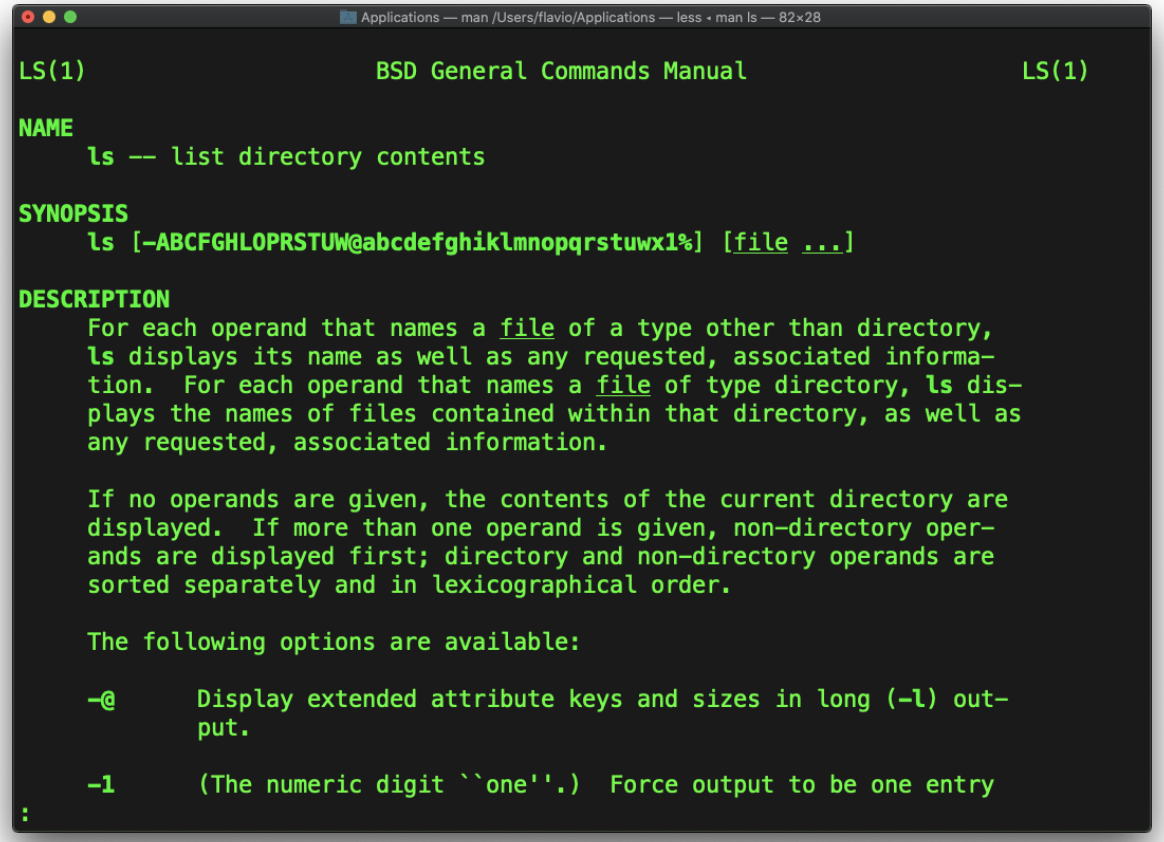
```
cd /etc
```

> This command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: man

Learn how the Linux man command opens the manual page for any command, how its pages are grouped by section number, and how tldr gives you a quick summary.

Every time I don't know how to use a command, I type `man <command>` to get the manual:



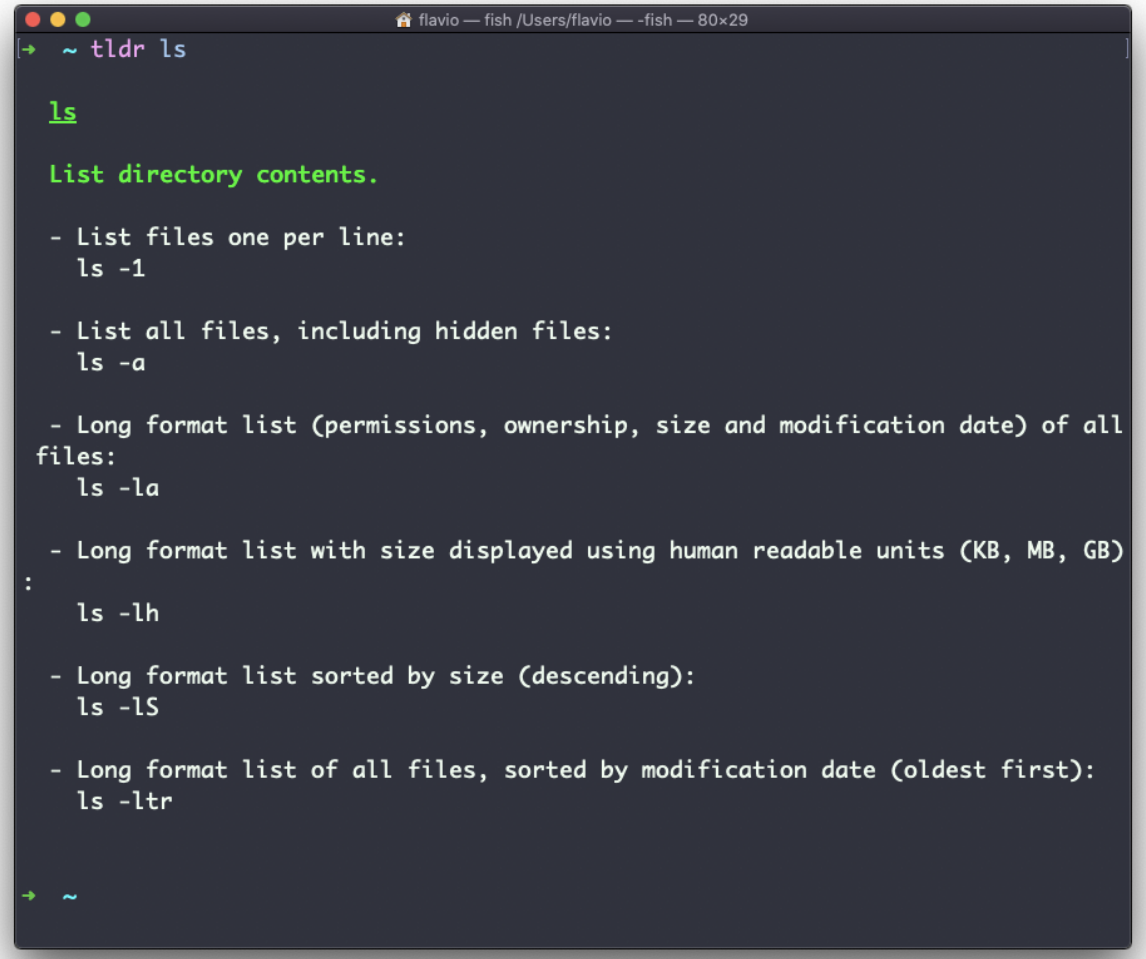
This is a man (from *manual*) page. Man pages are an essential tool to learn, as a developer. They contain so much information that sometimes it's almost too much.

The above screenshot is just 1 of 14 screens of explanation for the `ls` command.

Man pages are diveded into 7 different groups, identified by a number:

- `1` is **user commands**
- `2` is kernel **system calls**
- `3` is **C library functions**
- `4` is **devices**
- `5` is **files formats** and **filesystems**
- `6` is **games**
- `7` is **miscellaneous commands**, conventions and overviews
- `8` is **superuser and system administrator commands**

Most of the times when I'm in need to learn a command quickly I use this site called **tldr pages**: https://tldr.sh/. It's a command you can install, then you run it like this: `tldr <command>`, which gives you a very quick overview of a command, with some handy examples of common usage scenarios:



This is not a substitute for `man`, but a handy tool to avoid losing yourself in the huge amount of information present in a man page. Then you can use the man page to explore all the different options and parameters you can use on a command.

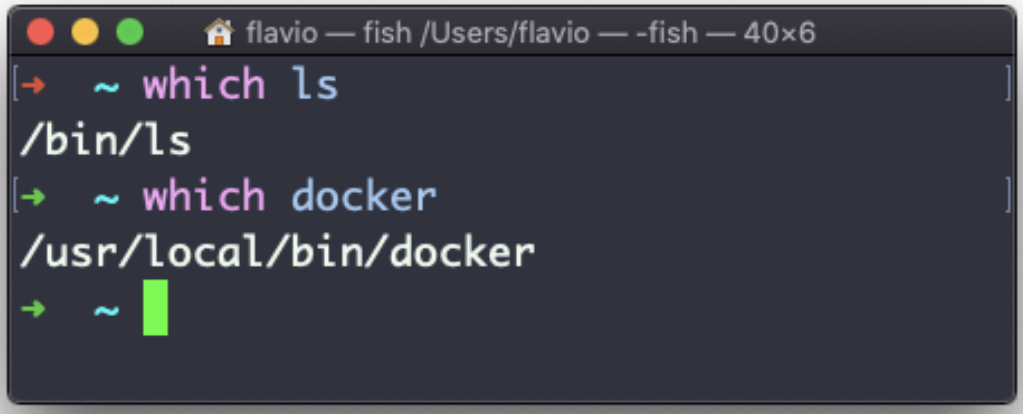
> The `man` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: which

Learn how the Linux which command shows the full path where an executable is stored on disk, and why it cannot locate shell aliases or built-in functions.

When you type a command name, the shell may resolve an alias, function, built-in, or executable found through `$PATH`. `which` commonly searches for an executable path:

You can do so using `which`. The command will return the path to the command specified:



That is useful, but it does not always describe what your current shell will actually run. Prefer the shell-aware command:

```
type ls
type -a node
command -v git
```

`type` can identify aliases, functions, and built-ins as well as files. `type -a` shows every matching definition, which helps when two Node.js installations or package managers compete in `$PATH`. `command -v` is a portable choice in shell scripts.

Inspect the search path itself:

```
printf '%s\n' "$PATH" | tr ':' '\n'
```

Directories are searched from left to right. Adding a directory at the front can shadow a system command; adding it at the end gives existing commands priority. After installing a tool, a shell may cache an older resolution. Starting a new shell or using its cache-refresh command can clear that confusion.

Do not execute an unfamiliar path merely because `which` found it. Inspect ownership and permissions first with `ls -l`, especially on shared systems.

Try it: run `type -a` for `cd`, `ls`, and a language runtime. Explain why their results differ.

Linux commands: history

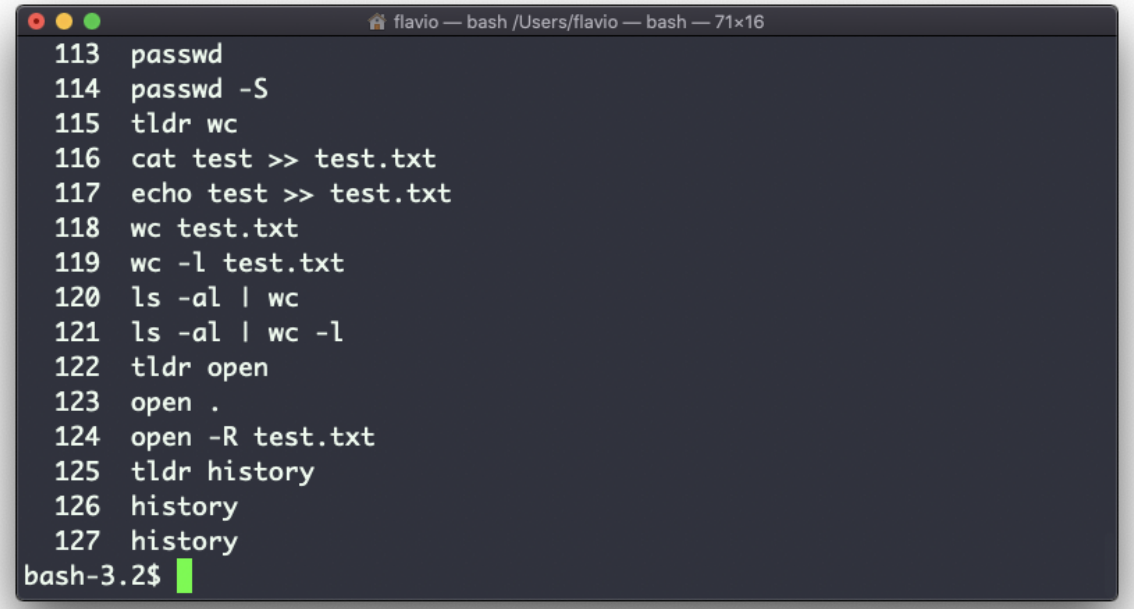
Learn how the Linux history command shows your past commands, how to rerun one with the !number syntax, search them with grep, and clear with history -c.

Interactive shells usually keep command history in memory and periodically save it to a file. The exact file size, save timing, duplicate handling, and sharing between terminals depend on the shell configuration.

You can display all the history using:

```
history
```

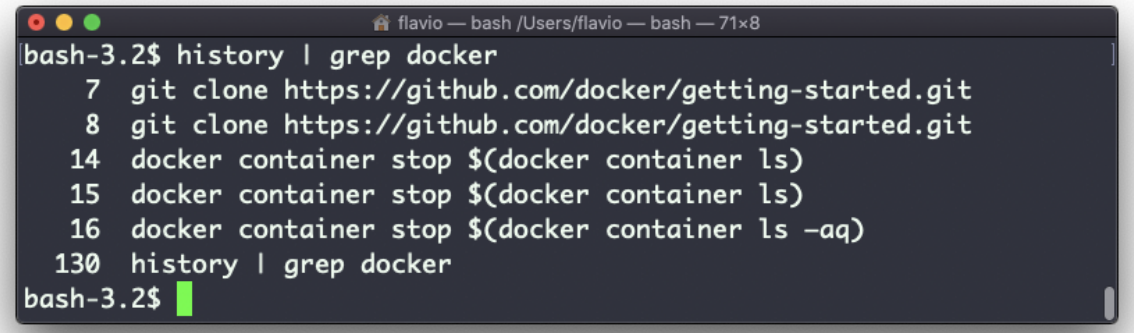
This shows recorded commands with numbers:



In shells with history expansion, `!121` repeats entry 121. Print or edit a command before running it when it can modify data. `!!` repeats the previous command, which is convenient but dangerous after a typo or after changing directories.

Search text with the shell's reverse search, often `Ctrl-R`, or filter the list:

```
history | grep docker
```



History is not a secure audit log. A user can edit or clear it, several shells can overwrite one another's updates, and non-interactive commands may not be recorded. Use system logs or dedicated auditing for accountability.

It is also a secret-leak risk. Tokens and passwords typed as command arguments may be written to the history file. Prefer interactive prompts, standard input where appropriate, or a secret manager. If a secret was exposed, deleting one history line does not revoke it or remove it from process logs, terminal scrollback, backups, or remote systems—rotate the secret first.

`history -c` behavior and persistence vary by shell. Clearing the in-memory list may not remove the saved file, and another open shell may write old entries back. Check your Bash or zsh documentation before relying on it.

Try it: use reverse search to find a harmless earlier command, edit it without executing the original unchanged, then inspect which history file your shell uses.

History features are shell behavior, so Bash and zsh details can differ even on the same operating system.

Check your understanding: terminal navigation

Check shells, prompts, paths, directory navigation, listing options, command history, and built-in help.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Working with files

Create, inspect, copy, move, link, and find files and folders.

The UNIX Filesystem Commands

Learn the core UNIX filesystem commands, from mkdir, cd, ls, and cp to setting permissions with chmod and chown, so you can manage files from the terminal.

I wrote this manual with the goal of making it easy to learn, remember and reference the main UNIX filesystem utilities. macOS and GNU/Linux are both UNIX systems, in practical terms. macOS is a certified UNIX, based on BSD UNIX, while GNU/Linux is UNIX-like, or a UNIX derivative.

After an introduction to the filesystem and how it works, I will cover the details of the following commands:

Working with folders and files

- `mkdir`
- `cd`
- `pwd`
- `rmdir`
- `ls`
- `touch`
- `mv`
- `cp`
- `ln`

Permissions

- `chmod`
- `chown`
- `chgrp`

Files content

- `cat`
- `less`
- `find`
- `cpio`
- `dd`
- `wc`

How the Filesystem works

Every modern computer system relies on a filesystem to store and retrieve programs. Not everything can be kept in memory, which is a precious and limited resource, and we need a safe place where to store data when the computer restarts or is shut off. Memory is completely wiped off any time the computer restarts, while the disk structure is persistent.

In this guide I examine in particular the UNIX filesystem, which can be found on macOS and GNU/Linux machines. The Windows filesystem is different, although Windows 10 with the Windows Subsystem for Linux installed allows you to use the same utilities that I describe here.

A filesystem starts from `/`. This is the root node which hosts the first level directories.

Usual directories that you will find in a UNIX system are

- `/bin` contains the main system commands
- `/etc` contains the system configuration
- `/dev` contains the system devices
- `/usr` contains the user files
- `/tmp` contains the temporary files

...there are more, but you get the idea.

How many depends on the type of system used. Linux has standardised the folders using its Linux Standard Base effort, and you usually have:

- `/bin` the main system commands
- `/boot` the files used to boot the machine (not existing on macOS)
- `/dev` system devices
- `/etc` system configuration files
- `/etc/opt` user programs configuration files
- `/home` the home directories of users (/Users in macOS)
- `/lib` the system libraries (not existing on macOS)
- `/mnt` mounted filesystems
- `/opt` user programs
- `/proc` user by kernel and processes (not existing on macOS)
- `/root` the home folder of the root user (not existing on macOS)
- `/run` (not existing on macOS)
- `/sbin` system binaries user for booting the system
- `/tmp` temporary files
- `/usr` holds user software, libraries and tools
- `/usr/bin` user binaries
- `/usr/include` user header files
- `/usr/lib` user libraries
- `/usr/local` used by user software to store installations, like Homebrew on macOS
- `/usr/sbin` system binaries

- `/usr/share` contains architecture-independent data. It can hold a lot of miscellaneous stuff including documentation and man pages
- `/usr/src` contains the source code of installed packages (not existing in macOS)
- `/var` contains temporary files, logs and more

macOS has various different folders, including

- `/Applications` stores the users applications
- `/Library` holds the library (settings and resources) used globally by users of the system
- `/System` holds system files
- `/private` holds system files, logs and more

Each folder contains files and directories, which in turn can contain other files and directories, and so on.

All files and folders have a name.

What makes a valid name? It can be long from 1 to 255 characters, and it must be composed from any alphanumeric character (`a-z`/`A-Z`/`0-9`), the underscore character (`_`), dot (`.`) and comma (`,`) characters.

You can also use space, although generally not recommended as space needs to be escaped with a backslash every time you must reference the file/folder.

Depending on the system, the filesystem can be case sensitive. On macOS the filesystem is **not** case sensitive by default, so you can't have a file/folder named `test` and another called `Test`. They are the same thing.

I am now going to introduce the various utility commands. Those are executed in the context of a shell, like Bash, Fish or Zsh. When you start a shell, you are going to be in your home directory. In macOS, that is found in the `/Users/<yourname>/` path.

The commands don't change if you change the shell you use, because those are the basic, fundamentals commands of UNIX. You will likely use those commands for many decades unless some revolution in the computing world happens (and they happen, but those server-side things usually move slow). To give you some perspective, the `mkdir` command you will now see was introduced in UNIX AT&T Version 1, in the early 80s.

Working with folders and files

mkdir You create folders using the `mkdir` command:

```
mkdir fruits
```

You can create multiple folders with one command:

```
mkdir dogs cars
```

You can also create multiple nested folders by adding the `-p` option:

```
mkdir -p fruits/apples
```

Options in UNIX commands commonly take this form. You add them right after the command name, and they change how the command behaves. You can often combine multiple options, too.

You can find which options a command supports by typing `man <commandname>`. Try now with `man mkdir` for example (press the `q` key to esc the man page). Man pages are the amazing built-in help for UNIX.

cd Once you create a folder, you can move into it using the `cd` command. `cd` means **c**hange **d**irectory. You invoke it specifying a folder to move into. You can specify a folder name, or an entire path.

Example:

```
mkdir fruits
cd fruits
```

Now you are into the `fruits` folder.

You can use the `..` special path to indicate the parent folder:

```
cd .. #back to the home folder
```

The # character indicates the start of the comment, which lasts for the entire line after it's found.

You can use it to form a path:

```
mkdir fruits
mkdir cars
cd fruits
cd ../cars
```

There is another special path indicator which is `.`, and indicates the **current** folder.

You can also use absolute paths, which start from the root folder `/`:

```
cd /etc
```

pwd Whenever you feel lost in the filesystem, call the `pwd` command to know where you are:

```
pwd
```

It will print the current folder path.

rmdir Just as you can create a folder using `mkdir`, you can delete a folder using `rmdir`:

```
mkdir fruits
rmdir fruits
```

You can also delete multiple folders at once:

```
mkdir fruits cars
rmdir fruits cars
```

The folder you delete must be empty.

To delete folders with files in them, we'll use the more generic `rm` command which deletes files and folders, using the `-rf` options:

```
rm -rf fruits cars
```

Be careful as this command does not ask for confirmation and it will immediately remove anything you ask it to remove.

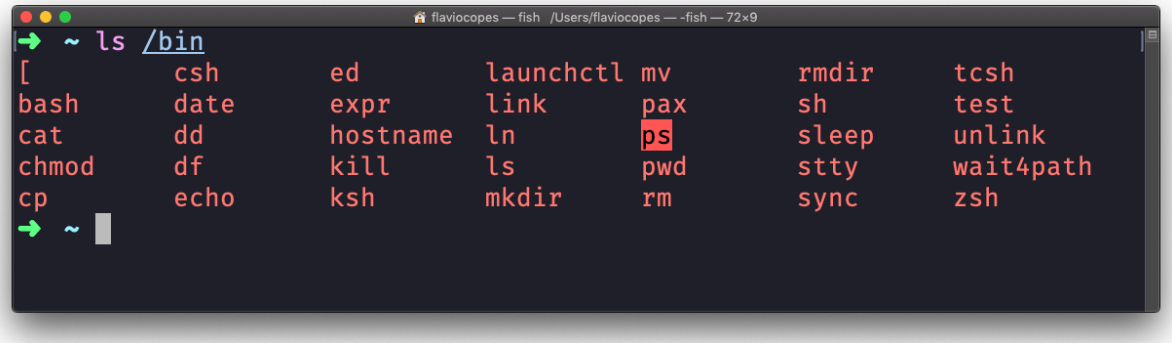
There is no **bin** when removing files from the command line, and recovering lost files can be hard.

ls Inside a folder you can list all the files that the folder contains using the `ls` command:

`ls`

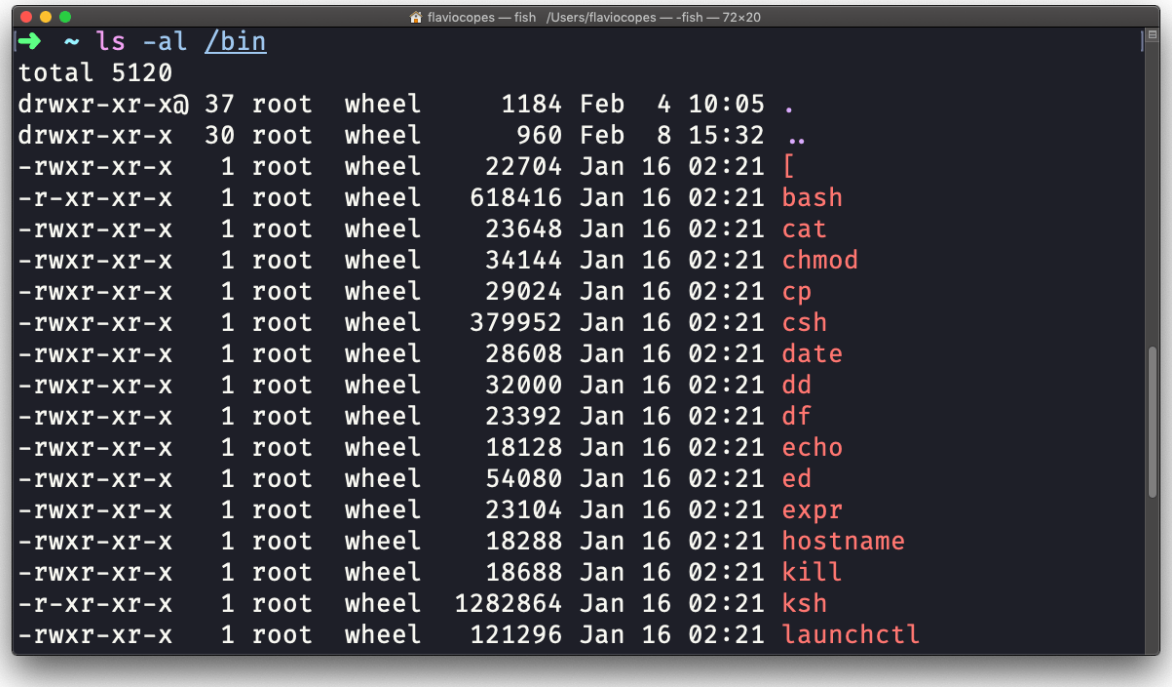
If you add a folder name or path, it will print that folder contents:

`ls /bin`



`ls` accepts a lot of options. One of my favorite options combinations is `-al`. Try it:

`ls -al /bin`



compared to the plain `ls`, this returns much more information.

You have, from left to right:

- the file permissions (and if your system supports ACLs, you get an ACL flag as well)
- the number of links to that file
- the owner of the file
- the group of the file
- the file size in bytes
- the file modified datetime
- the file name

This set of data is generated by the `l` option. The `a` option instead also shows the hidden files.

Hidden files are files that start with a dot (`.`).

touch You can create an empty file using the `touch` command:

```
touch apple
```

If the file already exists, it opens the file in write mode, and the timestamp of the file is updated.

mv Once you have a file, you can move it around using the `mv` command. You specify the file current path, and its new path:

```
touch test
mv pear new_pear
```

The `pear` file is now moved to `new_pear`. This is how you **rename** files and folders.

If the last parameter is a folder, the file located at the first parameter path is going to be moved into that folder. In this case, you can specify a list of files and they will all be moved in the folder path identified by the last parameter:

```
touch pear
touch apple
mkdir fruits
mv pear apple fruits #pear and apple moved to the fruits folder
```

cp You can copy a file using the `cp` command:

```
touch test
cp apple another_apple
```

To copy folders you need to add the `-r` option to recursively copy the whole folder contents:

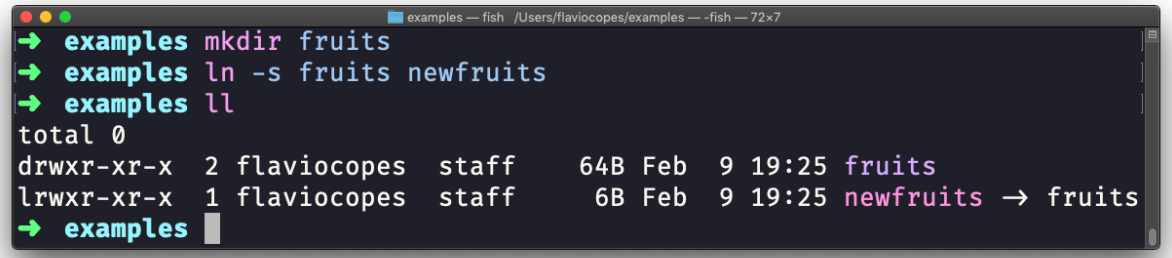
```
mkdir fruits
cp -r fruits cars
```

ln Let's now introduce the concept of links. A link is a file that points to another file. We have those in all the major operating systems.

We can create two types of links: hard links and symbolic (soft) links. We can't create a hard link to a directory, but we can create a symbolic one. For this reason, symbolic links (also called *symlinks*) are much more common.

We create a symbolic link using this syntax: `ln -s original linkname`. Example:

```
mkdir fruits
ln -s fruits newfruits
```



> Notice how I used the `ll` command. This is not a standard command, but an *alias* for `ls -al`. In a shell, you can define aliases that are shortcuts to common command and arguments combinations.

Permissions

I mentioned permissions briefly before, when introduced the `ls -al` command.

The weird string you see on each file line, like `drwxr-xr-x`, defines the permissions of the file or folder.

Let's dissect it.

The first letter indicates the type of file:

- `-` means it's a normal file
- `d` means it's a directory
- `l` means it's a link

Then you have 3 sets of values:

- The first set represents the permissions of the **owner** of the file
- The second set represents the permissions of the members of the **group** the file is associated to
- The third set represents the permissions of the **everyone else**

Those sets are composed by 3 values. `rwx` means that specific *persona* has read, write and execution access. Anything that is removed is swapped with a `-`, which lets you form various combinations of values and relative permissions: `rw-`, `r--`, `r-x`, and so on.

You can change the permissions given to a file using the `chmod` command.

`chmod` can be used in 2 ways. The first is using symbolic arguments, the second is using numeric arguments. Let's start with symbols first, which is more intuitive.

You type `chmod` followed by a space, and a letter:

- `a` stands for *all*
- `u` stands for *user*
- `g` stands for *group*
- `o` stands for *others*

Then you type either `+` or `-` to add a permission, or to remove it. Then you enter one or more permissions symbols (`r`, `w`, `x`).

All followed by the file or folder name.

Here are some examples:

```
chmod a+r filename #everyone can now read
chmod a+rw filename #everyone can now read and write
chmod o-rwx filename #others (not the owner, not in the same group of the file) cannot read,
```

You can apply the same permissions to multiple personas by adding multiple letters before the `+/-`:

```
chmod og-r filename #other and group can't read any more
```

In case you are editing a folder, you can apply the permissions to every file contained in that folder using the `-r` (recursive) flag.

Numeric arguments are faster but I find them hard to remember when you are not using them day to day. You use a digit that represents the permissions of the persona. This number value can be a maximum of 7, and it's calculated in this way:

- `1` if has execution permission
- `2` if has write permission
- `4` if has read permission

This gives us 4 combinations:

- `0` no permissions
- `1` can execute
- `2` can write
- `3` can write, execute
- `4` can read
- `5` can read, execute
- `6` can read, write
- `7` can read, write and execute

We use them in pairs of 3, to set the permissions of all the 3 groups altogether:

```
chmod 777 filename
chmod 755 filename
chmod 644 filename
```

Owner and group You can change the owner of a file using the `chown` command:

```
chown <username> <filename>
```

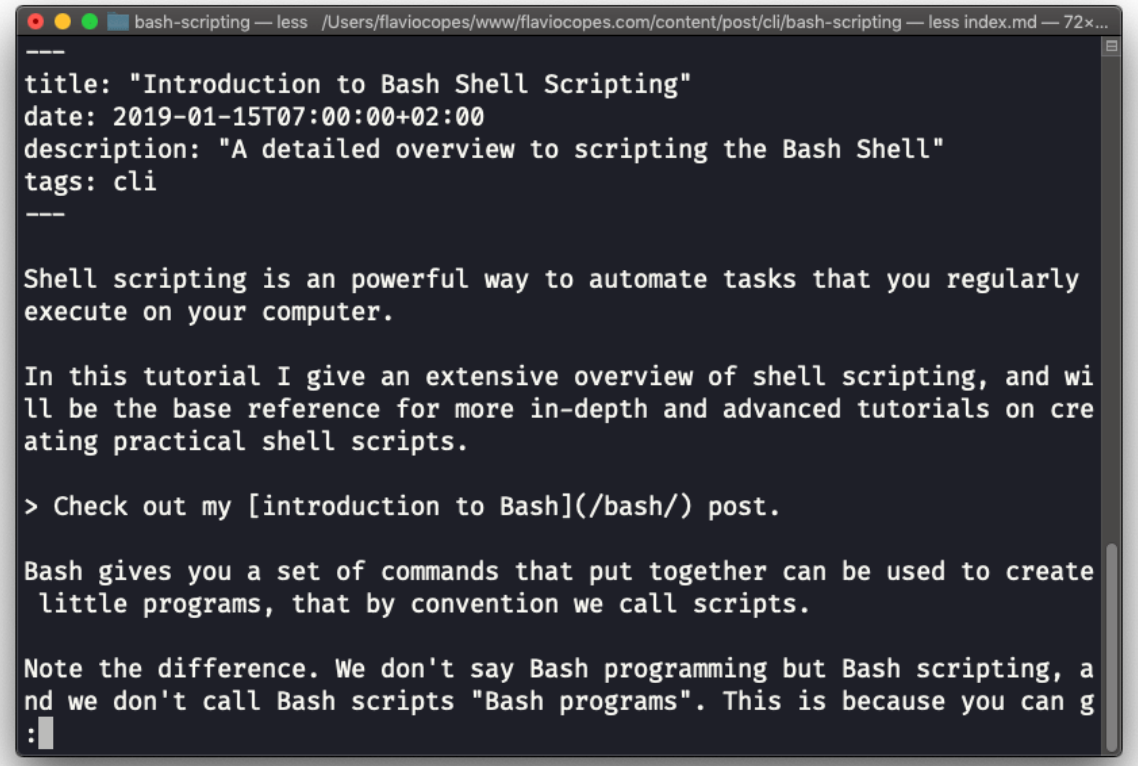
You can change the group of a file using the `chgrp` command:

```
chgrp <group> <filename>
```

Managing files content

less The `less` command is one I use a lot. It shows you the contents stored inside a file, in a nice and interactive UI.

Usage: `less <filename>`.



Once you are inside a `less` session, you can quit by pressing `q`.

You can navigate the file contents using the `up` and `down` keys, or using the `space bar` and `b` to navigate page by page. You can also jump to the end of the file pressing `G` and jump back to the start pressing `g`.

You can search contents inside the file by pressing `/` and typing a word to search. This searches *forward*. You can search backwards using the `?` symbol and typing a word.

This command just visualises the file's content. You can directly open an editor by pressing `v`. It will use the system editor, which in most cases is `vim`.

Pressing the `F` key enters *follow mode*, or *watch mode*. When the file is changed by someone else, like from another program, you get to see the changes *live*. By default this is not happening, and you only see the file version at the time you opened it. You need to press `ctrl-C` to quit this mode. In this case the behaviour is similar to running the `tail -f <filename>` command.

You can open multiple files, and navigate through them using `:n` (to go to the next file) and `:p` (to go to the previous).

tail I just mentioned `tail` in the previous command, so let's use this opportunity to explain what it does.

Its best use case in my opinion is when called with the `-f` option. It opens the file at the end, and watches for file changes. Any time there is new content in the file, it is printed in the window. This is great for watching log files, for example:

```
tail -f /var/log/system.log
```

To exit, press `ctrl-C`.

You can print the last 10 lines in a file:

```
tail -n 10 <filename>
```

You can print the whole file content starting from a specific line using `+` before the line number:

```
tail -n +10 <filename>
```

`tail` can do much more and as always my advice is to check `man tail`.

cat Similar to `tail` in some way, we have `cat`. Except `cat` can also add content to a file, and this makes it super powerful.

In its simplest usage, `cat` prints a file's content to the standard output:

```
cat file
```

You can print the content of multiple files:

```
cat file1 file2
```

and using the output redirection operator `>` you can concatenate the content of multiple files into a new file:

```
cat file1 file2 > file3
```

Using `>>` you can append the content of multiple files into a new file, creating it if it does not exist:

```
cat file1 file2 >> file3
```

When watching source code files it's great to see the line numbers, and you can have `cat` print them using the `-n` option:

```
cat -n file1
```

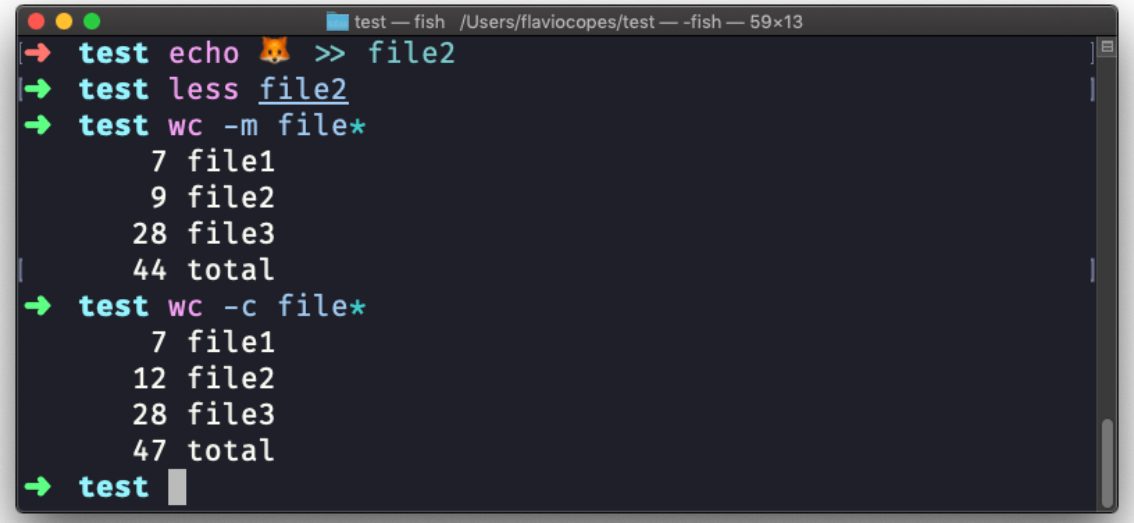
You can only add a number to non-blank lines using `-b`, or you can also remove all the multiple empty lines using `-s`.

`cat` is often used in combination with the pipe operator `|` to feed a file content as input to another command: `cat file1 | anothercommand`.

wc `wc` means *word count*. Here is the most common usage:

- `wc -l file1` count lines
- `wc -w file1` count words
- `wc -c file1` count characters
- `wc -m file1` count characters with multibyte support (i.e. emojis count as 1, not as multiple characters)

You can run `wc` providing multiple files, in which case it will do the calculations on each file separately, and then print a summary:



find The `find` command can be used to find files or folders matching a particular search pattern. It searches recursively.

Let's learn it by example.

Find all the files under the current tree that have the `.js` extension and print the relative path of each file matching:

```
find . -name '*.js'
```

It's important to use quotes around special characters like `*` to avoid the shell interpreting them.

Find directories under the current tree matching the name "src":

```
find . -type d -name src
```

Use `-type f` to search only files, or `-type l` to only search symbolic links.

`-name` is case sensitive. use `-iname` to perform a case-insensitive search.

You can search under multiple root trees:

```
find folder1 folder2 -name filename.txt
```

Find directories under the current tree matching the name "node_modules" or 'public':

```
find . -type d -name node_modules -or -name public
```

You can also exclude a path, using `-not -path`:

```
find . -type d -name '*.md' -not -path 'node_modules/*'
```

You can search files that have more than 100 characters (bytes) in them:

```
find . -type f -size +100c
```

Search files bigger than 100KB but smaller than 1MB:

```
find . -type f -size +100k -size -1M
```

Search files edited more than 3 days ago

```
find . -type f -mtime +3
```

Search files edited in the last 24 hours

```
find . -type f -mtime -1
```

You can delete all the files matching a search by adding the `-delete` option. This deletes all the files edited in the last 24 hours:

```
find . -type f -mtime -1 -delete
```

You can execute a command on each result of the search. In this example we run `cat` to print the file content:

```
find . -type f -exec cat {} \;
```

notice the terminating `\;`. `{}` is filled with the file name at execution time.

Linux commands: touch

Learn how the Linux touch command creates a new empty file, and how running it on a file that already exists just updates that file's modification timestamp.

`touch` updates a file's access and modification timestamps. When the path does not exist, it creates an empty file:

```
touch apple.txt
```

Inspect the result:

```
ls -l apple.txt
touch apple.txt
ls -l apple.txt
```

The second `touch` does not open the file in write mode and does not erase its contents. It changes timestamps. This distinction makes it safe for build tools that use modification time to decide whether work is stale.

Use `-c` when you want to update an existing path without creating it:

```
touch -c optional.txt
```

You can set a specific timestamp with `-t`, although the exact accepted format is easy to mistype. Prefer copying a known timestamp when that expresses the goal:

```
touch -r source.txt destination.txt
```

Creating a file with `touch` says nothing about its contents or permissions beyond the normal defaults. Use redirection when you need content, for example `printf '%s\n' 'hello' > greeting.txt`.

Try it: put text in a file, record `ls -l` and `stat` output, run `touch`, and verify that the text stays while the modification time changes.

Linux commands: mkdir

Learn how the Linux mkdir command creates folders, makes several at once in a single call, and builds nested directories in one go with the -p option.

You create folders using the `mkdir` command:

```
mkdir fruits
```

Verify it worked with `ls`, which now lists the new folder.

If a folder with that name already exists, `mkdir` refuses and tells you:

```
mkdir fruits
# mkdir: fruits: File exists
```

Nothing gets overwritten. The existing folder and everything inside it are safe.

You can create multiple folders with one command:

```
mkdir dogs cars
```

You can also create multiple nested folders by adding the `-p` option:

```
mkdir -p fruits/apples
```

The `-p` matters here. Without it, `mkdir fruits/apples` fails with `No such file or directory` when `fruits` doesn't exist yet, because plain `mkdir` only creates the final piece of the path. `-p` creates every missing parent along the way.

`-p` has a second useful behavior: it stays silent when the directory already exists, instead of failing. That makes the command safe to run twice, which is why `mkdir -p` shows up in so many scripts — the script works on the first run and on every run after.

Add `-v` and the command narrates each directory it creates, a nice confirmation when `-p` builds several levels at once:

```
mkdir -pv fruits/pears
```

Options in UNIX commands commonly take this form. You add them right after the command name, and they change how the command behaves. You can often combine multiple options, too — `-pv` above is `-p` and `-v` together.

You can find which options a command supports by typing `man <commandname>`. Try now with `man mkdir` for example (press the `q` key to esc the man page). Man pages are the amazing built-in help for UNIX.

Linux commands: cp

Learn how the Linux cp command copies files from one place to another, and how the -r recursive option lets you copy entire folders and their contents.

`cp` creates a new directory entry with copied contents:

```
cp report.txt report-backup.txt
```

If the destination is a directory, the copy keeps the source filename:

```
cp report.txt backups/
```

Copy directories recursively with `-R`:

```
cp -R site site-backup
```

The most important question is what already exists at the destination. Plain `cp` can overwrite a file without a prompt. During manual work, `cp -i` asks before overwriting and `cp -n` skips existing

destinations on implementations that support it. Check the exit status rather than assuming every file copied.

Metadata behavior matters too. A basic copy may give the destination new timestamps and ownership. `cp -p` preserves selected metadata; archive options vary between systems. For a portable lesson, inspect the files after copying instead of assuming Linux and macOS flags are identical.

Always quote paths that can contain spaces:

```
cp "Quarterly report.txt" "backups/Quarterly report.txt"
```

Before a large recursive copy, list the source and destination, verify available disk space, and consider whether a synchronization tool such as `rsync` better matches incremental updates.

Try it: copy a file, modify the original, and compare them with `cmp`. The copy is independent; later changes to the source do not update it.

Linux commands: mv

Learn how the Linux mv command moves a file to a new path, how it doubles as the way you rename files, and how to move several files into a folder at once.

`mv` changes a path's name or location:

```
touch pear
mv pear new_pear
```

Within one filesystem this is often a metadata change, so moving a large file can be nearly instant. Across filesystems, the implementation must copy the data and remove the source, which takes longer and introduces more failure points.

When the final argument is a directory, move one or more sources into it:

```
touch pear
touch apple
mkdir fruits
mv pear apple fruits/
```

The destination is the main risk. Plain `mv` can replace an existing file. For interactive work, `mv -i` asks before overwriting. `mv -n` avoids overwriting where supported. In scripts, check beforehand and handle the race or use application-level atomic file operations when correctness depends on it.

Quote paths with spaces and use `--` before names that may begin with a dash:

```
mv -- "draft report.md" "final report.md"
```

After a move, verify both sides:

```
test ! -e old-name && test -e new-name
```

Moving an open file does not necessarily stop a running Unix process from using it; the process may still hold the underlying file descriptor. That is why log rotation and deployments need deliberate coordination.

Try it: move a file within a directory, then across two mounted filesystems if available. Compare timing and explain the different failure modes.

Linux commands: cat

Learn how the Linux cat command prints a file to standard output, joins multiple files into one with redirection, and shows line numbers with the -n option.

Similar to `tail` in some way, we have `cat`. Except `cat` can also add content to a file, and this makes it super powerful.

In its simplest usage, `cat` prints a file's content to the standard output:

```
cat file
```

You can print the content of multiple files:

```
cat file1 file2
```

and using the output redirection operator `>` you can concatenate the content of multiple files into a new file:

```
cat file1 file2 > file3
```

Using `>>` you can append the content of multiple files into a new file, creating it if it does not exist:

```
cat file1 file2 >> file3
```

When watching source code files it's great to see the line numbers, and you can have `cat` print them using the `-n` option:

```
cat -n file1
```

You can only add a number to non-blank lines using `-b`, or you can also remove all the multiple empty lines using `-s`.

`cat` is often used in combination with the pipe operator `|` to feed a file content as input to another command: `cat file1 | anothercommand`.

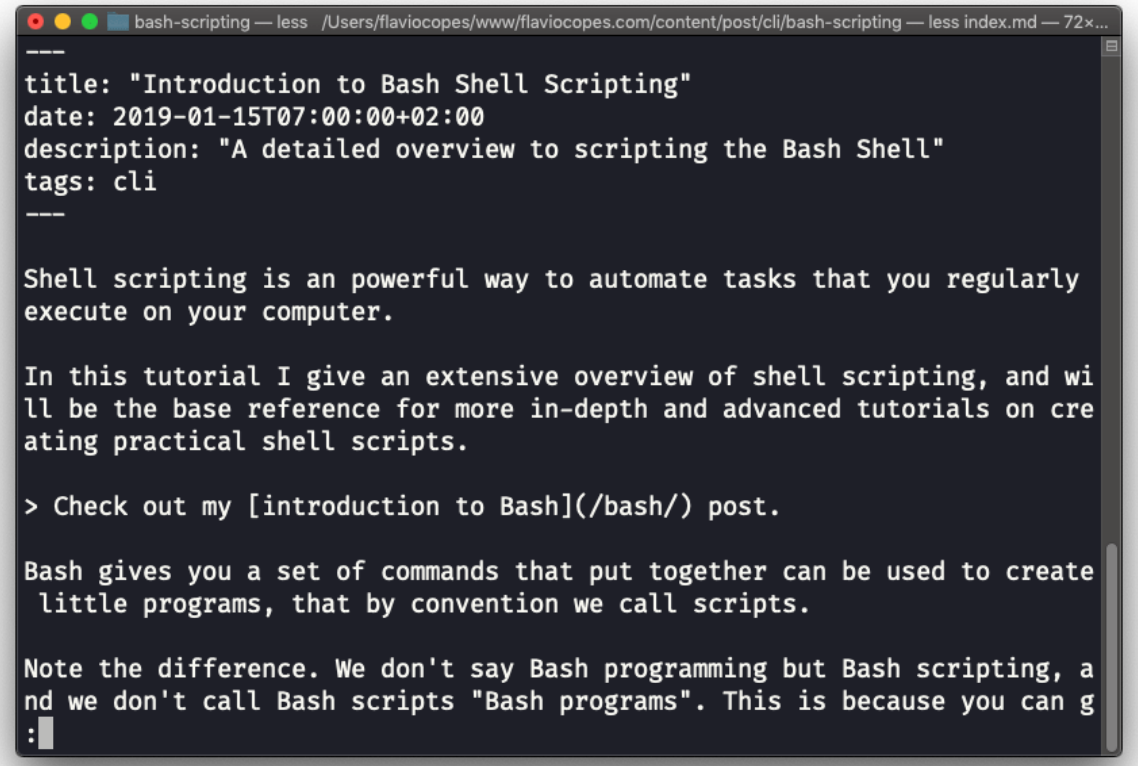
> This command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: less

Learn how the Linux less command shows a file in an interactive viewer, where you scroll, search with /, and press F for live follow mode like tail -f.

The `less` command is one I use a lot. It shows you the content stored inside a file, in a nice and interactive UI.

Usage: `less <filename>`.



Once you are inside a `less` session, you can quit by pressing `q`.

You can navigate the file contents using the `up` and `down` keys, or using the `space bar` and `b` to navigate page by page. You can also jump to the end of the file pressing `G` and jump back to the start pressing `g`.

You can search contents inside the file by pressing `/` and typing a word to search. This searches *forward*. You can search backwards using the `?` symbol and typing a word.

This command just visualises the file's content. You can directly open an editor by pressing `v`. It will use the system editor, which in most cases is `vim`.

Pressing the `F` key enters *follow mode*, or *watch mode*. When the file is changed by someone else, like from another program, you get to see the changes *live*. By default this is not happening, and you only see the file version at the time you opened it. You need to press `ctrl-C` to quit this mode. In this case the behaviour is similar to running the `tail -f <filename>` command.

You can open multiple files, and navigate through them using `:n` (to go to the next file) and `:p` (to go to the previous).

> This command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: find

Learn how the Linux find command searches recursively for files and folders by name, type, size, or age, and runs a command on each match with -exec.

The `find` command can be used to find files or folders matching a particular search pattern. It

searches recursively.

Let's learn it by example.

Find all the files under the current tree that have the `.js` extension and print the relative path of each file matching:

```
find . -name '*.js'
```

It's important to use quotes around special characters like `*` to avoid the shell interpreting them.

Find directories under the current tree matching the name "src":

```
find . -type d -name src
```

Use `-type f` to search only files, or `-type l` to only search symbolic links.

`-name` is case sensitive. use `-iname` to perform a case-insensitive search.

You can search under multiple root trees:

```
find folder1 folder2 -name filename.txt
```

Find directories under the current tree matching the name "node_modules" or 'public':

```
find . -type d -name node_modules -or -name public
```

You can also exclude a path, using `-not -path`:

```
find . -type d -name '*.md' -not -path 'node_modules/*'
```

You can search files that have more than 100 characters (bytes) in them:

```
find . -type f -size +100c
```

Search files bigger than 100KB but smaller than 1MB:

```
find . -type f -size +100k -size -1M
```

Search files edited more than 3 days ago

```
find . -type f -mtime +3
```

Search files edited in the last 24 hours

```
find . -type f -mtime -1
```

You can delete all the files matching a search by adding the `-delete` option. This deletes all the files edited in the last 24 hours:

```
find . -type f -mtime -1 -delete
```

You can execute a command on each result of the search. In this example we run `cat` to print the file content:

```
find . -type f -exec cat {} \;
```

notice the terminating `\;`. `{}` is filled with the file name at execution time.

If building the right `find` command feels like a puzzle, I made a find command builder that puts it together for you.

Linux commands: ln

Learn how the Linux ln command creates links to files: hard links that share content, and soft links made with -s that can point to directories too.

The `ln` command is part of the Linux file system commands.

It's used to create links. What is a link? It's like a pointer to another file. A file that points to another file. You might be familiar with Windows shortcuts. They're similar.

We have 2 types of links: **hard links** and **soft links**.

Hard links

Hard links are rarely used. They have a few limitations: you can't link to directories, and you can't link to external filesystems (disks).

A hard link is created using

```
ln <original> <link>
```

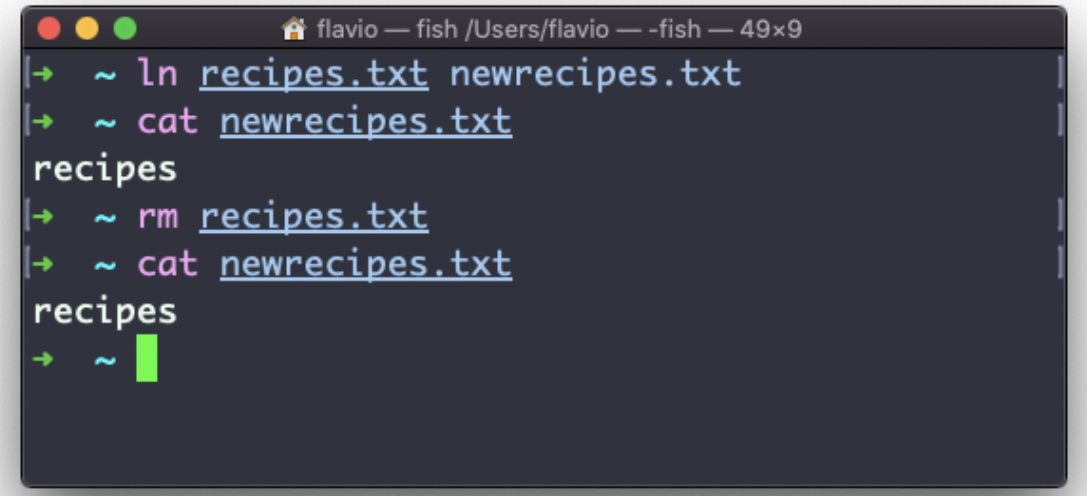
For example, say you have a file called recipes.txt. You can create a hard link to it using:

```
ln recipes.txt newrecipes.txt
```

The new hard link you created is indistinguishable from a regular file:

Now any time you edit any of those files, the content will be updated for both.

If you delete the original file, the link will still contain the original file content, as that's not removed until there is one hard link pointing to it.



Soft links

Soft links are different. They are more powerful as you can link to other filesystems and to directories, but when the original is removed, the link will be broken.

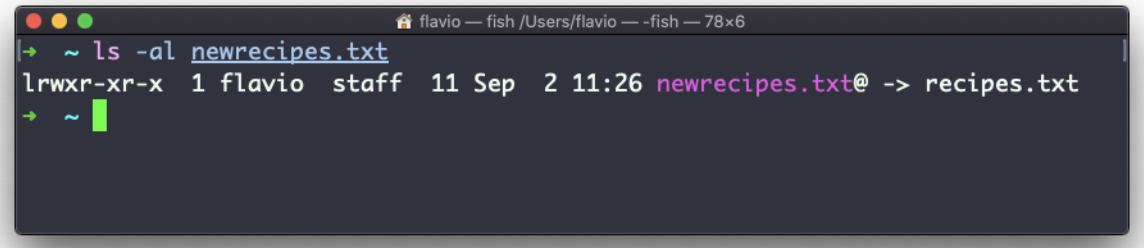
You create soft links using the `-s` option of `ln`:

```
ln -s <original> <link>
```

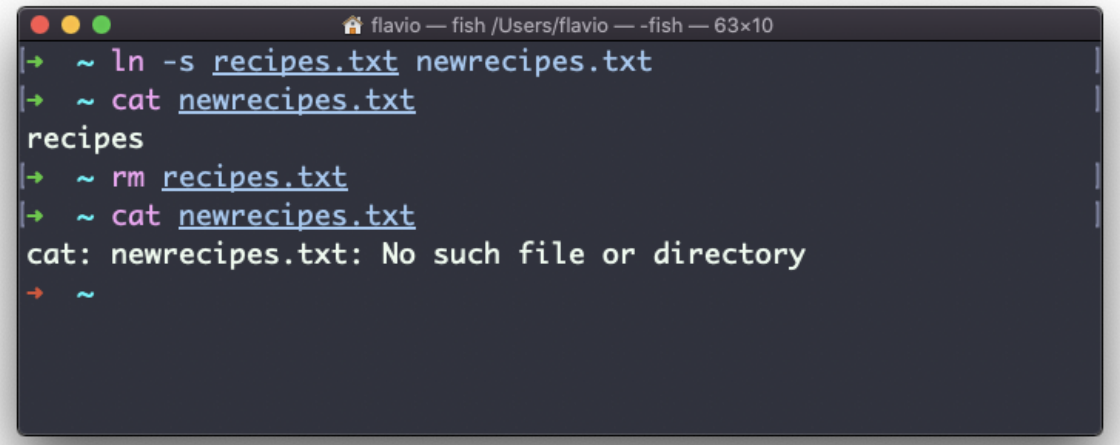
For example, say you have a file called recipes.txt. You can create a soft link to it using:

```
ln -s recipes.txt newrecipes.txt
```

In this case you can see there's a special `l` flag when you list the file using `ls -al`, and the file name has a `@` at the end, and it's colored differently if you have colors enabled:



Now if you delete the original file, the links will be broken, and the shell will tell you "No such file or directory" if you try to access it:



This command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Check your understanding: files and folders

Check file creation, copying, moving, deletion, viewing, links, wildcards, and finding content on the filesystem.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Permissions and users

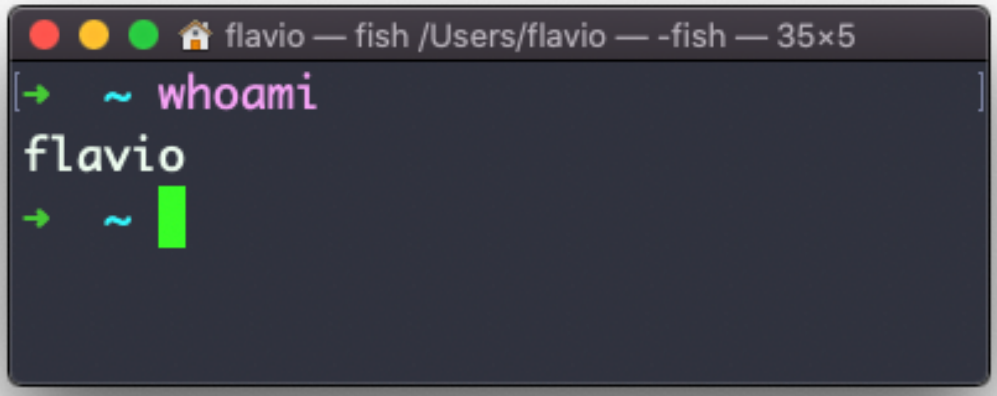
Users, groups, ownership, modes, sudo, and safe access changes.

Linux commands: whoami

Learn how the Linux whoami command prints the user name currently logged in to your terminal session, and how it differs from the more detailed who am i.

`whoami` prints the effective user identity of the current process:

```
whoami
```



The word “effective” matters. After `sudo`, a container switch, or another privilege-changing tool, the process may run as a different user from the account that originally opened the session.

Compare these commands:

```
whoami
id
who am i
```

`id` shows the effective user ID, primary group, and supplementary groups. `who am i` reports the login session when that information is available; it is not a more detailed spelling of `whoami`.

This is a useful first diagnostic for permission errors. Before changing permissions, ask:

- Which user is the failing process actually running as?
- Which groups does it belong to?
- Who owns the file or socket it needs?

For example, a command that works under `sudo` but fails normally usually reveals an ownership, group, or configuration problem. Running everything as root hides that problem and increases the impact of mistakes.

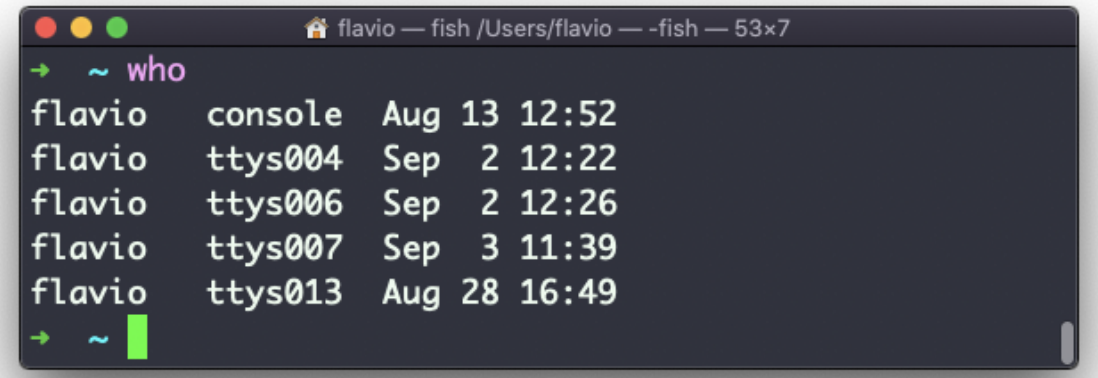
Try it: run `whoami` and `id`, then inspect a file with `ls -l`. Explain whether your effective user can read, write, and execute it from the displayed permission bits.

Linux commands: who

Learn how the Linux who command lists the users currently logged in to the system, why you appear multiple times, and what the -aH flags add to the output.

The `who` command displays the users logged in to the system.

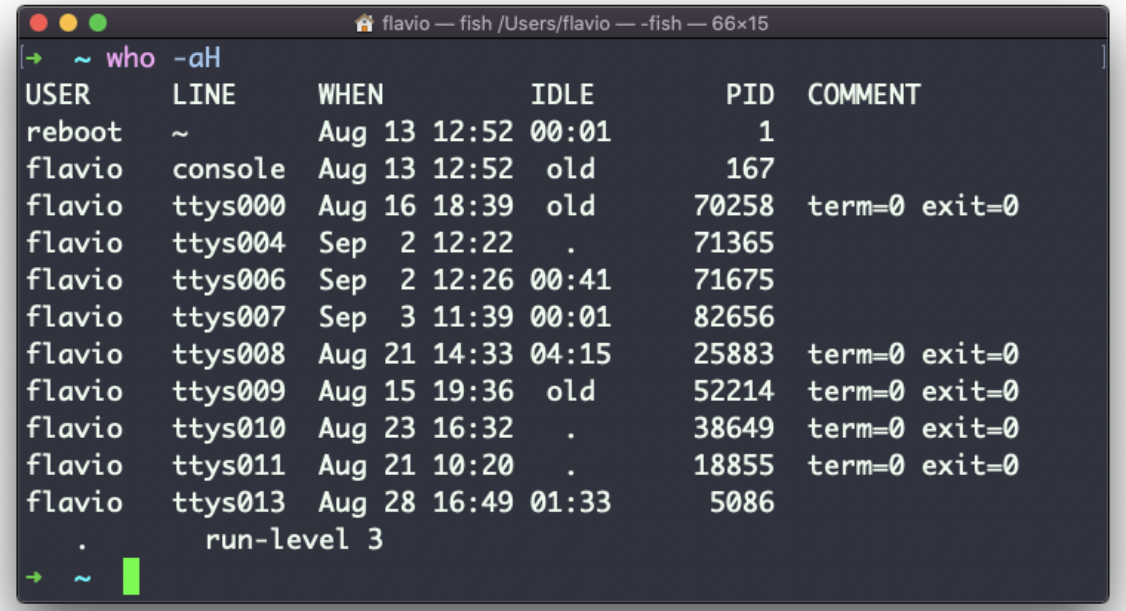
Unless you're using a server multiple people have access to, chances are you will be the only user logged in, multiple times:



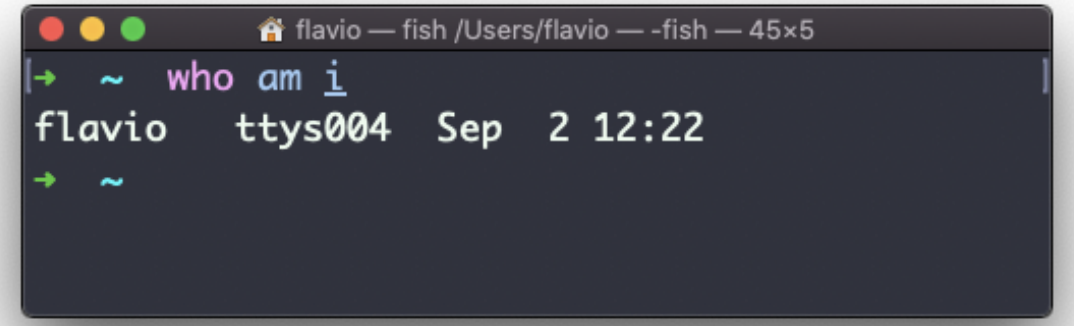
Why multiple times? Because each shell opened will count as an access.

You can see the name of the terminal used, and the time/day the session was started.

The `-aH` flags will tell `who` to display more information, including the idle time and the process ID of the terminal:



The special `who am i` command will list the current terminal session details:



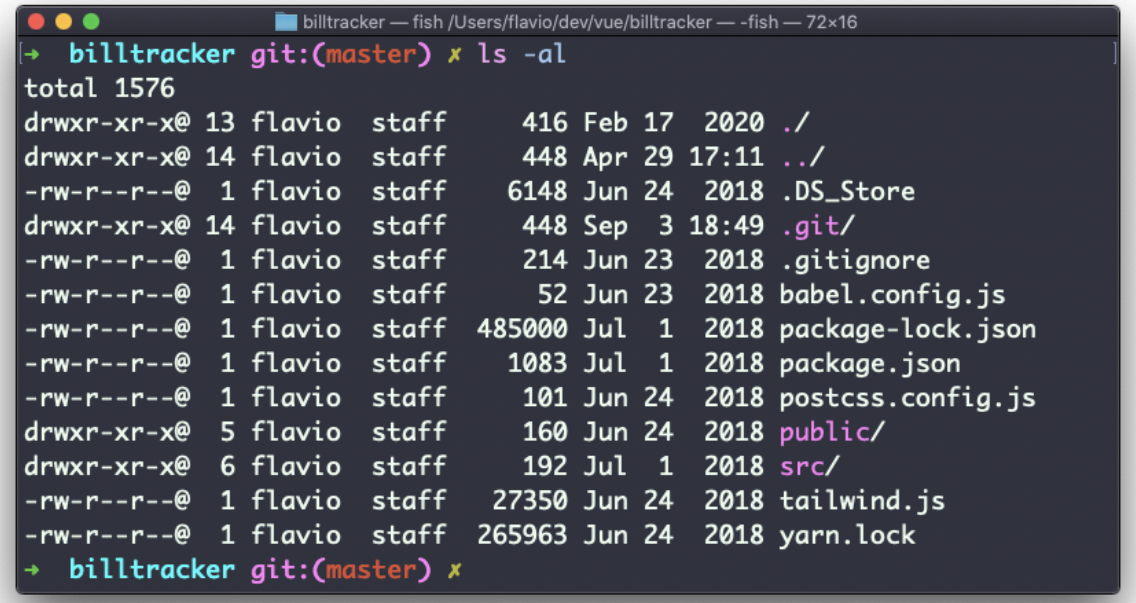
The `who` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: chmod

Learn how the Linux chmod command changes file permissions for the owner, group, and others, using symbolic letters like u+x or numeric modes like 644.

Every file in the Linux / macOS Operating Systems (and UNIX systems in general) has 3 permissions: Read, write, execute.

Go into a folder, and run the `ls -al` command.



The weird strings you see on each file line, like `drwxr-xr-x`, define the permissions of the file or folder.

Let's dissect it.

The first letter indicates the type of file:

- `-` means it's a normal file
- `d` means it's a directory
- `l` means it's a link

Then you have 3 sets of values:

- The first set represents the permissions of the **owner** of the file
- The second set represents the permissions of the members of the **group** the file is associated to
- The third set represents the permissions of the **everyone else**

Those sets are composed by 3 values. `rwx` means that specific *persona* has read, write and execution access. Anything that is removed is swapped with a `-`, which lets you form various combinations of values and relative permissions: `rw-`, `r--`, `r-x`, and so on.

You can change the permissions given to a file using the `chmod` command.

`chmod` can be used in 2 ways. The first is using symbolic arguments, the second is using numeric arguments. Let's start with symbols first, which is more intuitive.

You type `chmod` followed by a space, and a letter:

- `a` stands for *all*
- `u` stands for *user*
- `g` stands for *group*
- `o` stands for *others*

Then you type either `+` or `-` to add a permission, or to remove it. Then you enter one or more

permissions symbols (`r`, `w`, `x`).

All followed by the file or folder name.

Here are some examples:

```
chmod a+r filename #everyone can now read
chmod a+rw filename #everyone can now read and write
chmod o-rwx filename #others (not the owner, not in the same group of the file) cannot read,
```

You can apply the same permissions to multiple personas by adding multiple letters before the `+`/`-`:

```
chmod og-r filename #other and group can't read any more
```

In case you are editing a folder, you can apply the permissions to every file contained in that folder using the `-r` (recursive) flag.

Numeric arguments are faster but I find them hard to remember when you are not using them day to day. You use a digit that represents the permissions of the persona. This number value can be a maximum of 7, and it's calculated in this way:

- `1` if has execution permission
- `2` if has write permission
- `4` if has read permission

This gives us 4 combinations:

- `0` no permissions
- `1` can execute
- `2` can write
- `3` can write, execute
- `4` can read
- `5` can read, execute
- `6` can read, write
- `7` can read, write and execute

We use them in pairs of 3, to set the permissions of all the 3 groups altogether:

```
chmod 777 filename
chmod 755 filename
chmod 644 filename
```

If you don't want to do the math, I built a free chmod calculator that converts between the `rwx` notation, octal numbers, and `ls -l` output.

> The `chmod` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: chown

Learn how the Linux chown command changes the owner of a file or directory, sets owner and group at once, and applies changes recursively with the -R flag.

Every file/directory in an Operating System like Linux or macOS (and every UNIX systems in general) has an **owner**.

The owner of a file can do everything with it. It can decide the fate of that file.

The owner (and the `root` user) can change the owner to another user, too, using the `chown` command:

```
chown <owner> <file>
```

Like this:

```
chown flavio test.txt
```

For example if you have a file that's owned by `root`, you can't write to it as another user:

You can use `chown` to transfer the ownership to you:

It's rather common to have the need to change the ownership of a directory, and recursively all the files contained, plus all the subdirectories and the files contained in them, too.

You can do so using the `-R` flag:

```
chown -R <owner> <file>
```

Files/directories don't just have an owner, they also have a **group**. Through this command you can change that simultaneously while you change the owner:

```
chown <owner>:<group> <file>
```

Example:

```
chown flavio:users test.txt
```

You can also just change the group of a file using the `chgrp` command:

`chgrp <group> <filename>`

> The `chown` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: sudo

Learn how the Linux sudo command runs a command as root using your own password, how sudo -i opens a root shell, and how -u runs it as any other user.

`sudo` is commonly used to run a command as root.

You must be enabled to use `sudo`, and once you do, you can run commands as root by entering your user's password (*not* the root user password).

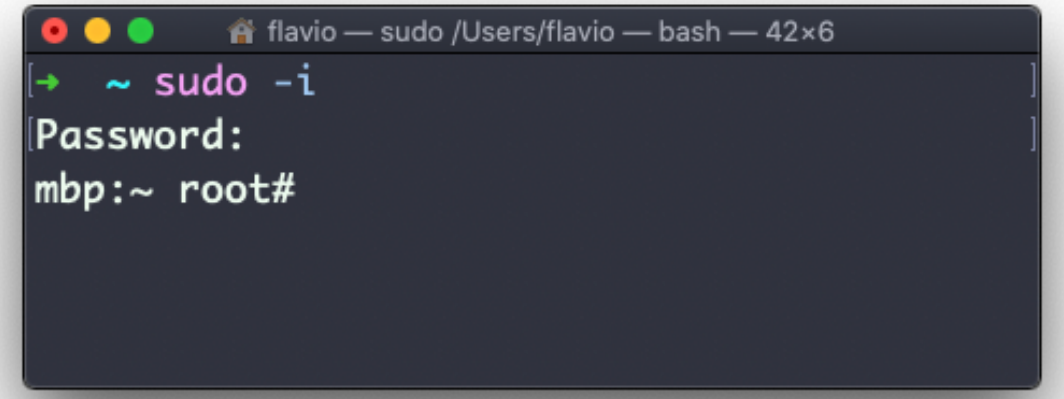
The permissions are highly configurable, which is great especially in a multi-user server environment, and some users can be granted access to running specific commands through `sudo`.

For example you can edit a system configuration file:

```
sudo nano /etc/hosts
```

which would otherwise fail to save since you don't have the permissions for it.

You can run `sudo -i` to start a shell as root:



You can use `sudo` to run commands as any user. `root` is the default, but use the `-u` option to specify another user:

```
sudo -u flavio ls /Users/flavio
```

> The `sudo` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

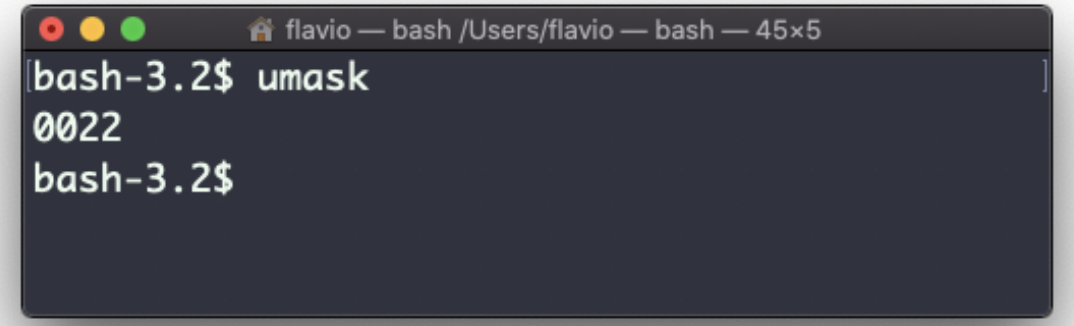
Linux commands: umask

Learn how the Linux umask command sets the default permissions for new files, reading octal values like 0022 and the -S human-readable notation.

When you create a file, you don't have to decide permissions up front. Permissions have defaults.

Those defaults can be controlled and modified using the `umask` command.

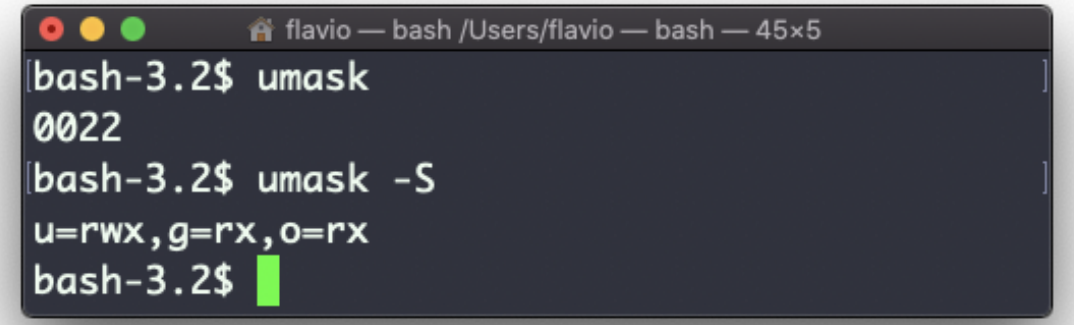
Typing `umask` with no arguments will show you the current umask, in this case `0022`:



What does `0022` mean? That's an octal value that represent the permissions.

Another common value is `0002`.

Use `umask -S` to see a human-readable notation:



In this case, the user (`u`), owner of the file, has read, write and execution permissions on files.

Other users belonging to the same group (`g`) have read and execution permission, same as all the other users (`o`).

In the numeric notation, we typically change the last 3 digits.

Here's a list that gives a meaning to the number:

- `0` read, write, execute
- `1` read and write
- `2` read and execute
- `3` read only

- `4` write and execute
- `5` write only
- `6` execute only
- `7` no permissions

Note that this numeric notation differs from the one we use in `chmod`.

We can set a new value for the mask setting the value in numeric format:

```
umask 002
```

or you can change a specific role's permission:

```
umask g+r
```

> The `umask` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

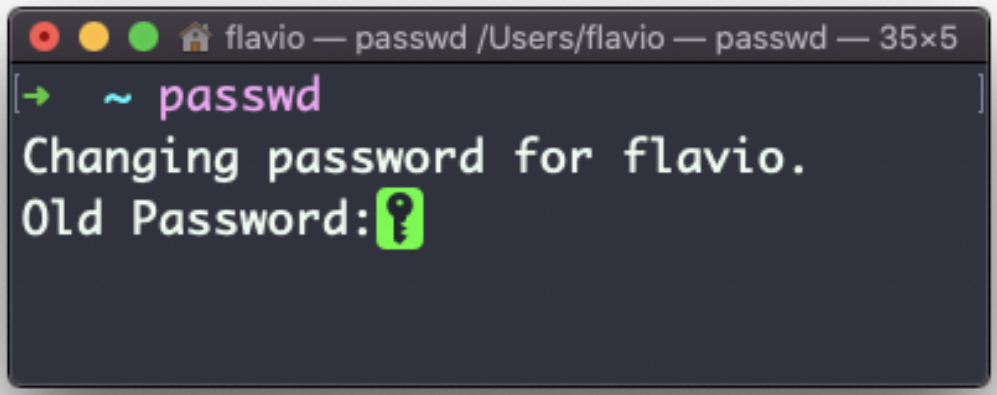
Linux commands: passwd

Learn how the Linux passwd command changes your password through an interactive prompt, and how root can set another user's password without the old one.

Change your own account password interactively:

```
passwd
```

The program reads passwords without displaying them. Depending on the system and authentication setup, it may ask for the current password before accepting a new one:



An administrator can start the interactive flow for another account:

```
sudo passwd username
```

Do not put the new password on the command line. Command arguments may appear in shell history or process listings, and the common `passwd` interface does not take a plaintext password as a second argument.

If a password change is rejected, read the exact message. Systems can enforce minimum length,

history, complexity, account state, or directory-service policy. Repeatedly trying small variations does not diagnose which rule failed.

Changing a local Unix password may not change an SSH key, cloud identity, Apple ID, or organization directory password. First identify which authentication system the login actually uses.

Administrators can inspect account-aging information with platform-specific tools such as `chage` on Linux. Locking, expiring, or resetting an account is security-sensitive: verify the username and keep another administrative session available when working remotely.

Try it on a disposable local account or lab machine. Confirm the command prompts interactively and that no password appears in `history`.

Check your understanding: permissions and users

Check ownership, groups, read-write-execute modes, chmod notation, sudo, umask, and permission failures.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Pipes and text

Standard streams, redirection, filters, environment, and scripts.

Linux commands: echo

Learn how the Linux echo command prints its argument to the terminal, interpolates variables like $PATH, expands globs and ranges, and appends to a file.

The `echo` command does one simple job: it prints to the output the argument passed to it.

This example:

```
echo "hello"
```

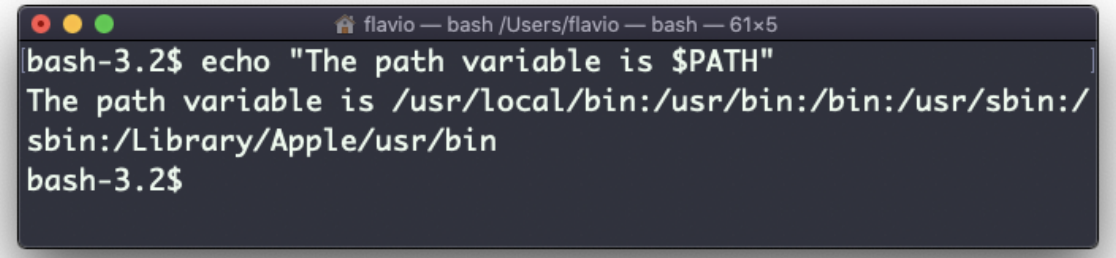
will print `hello` to the terminal.

We can append the output to a file:

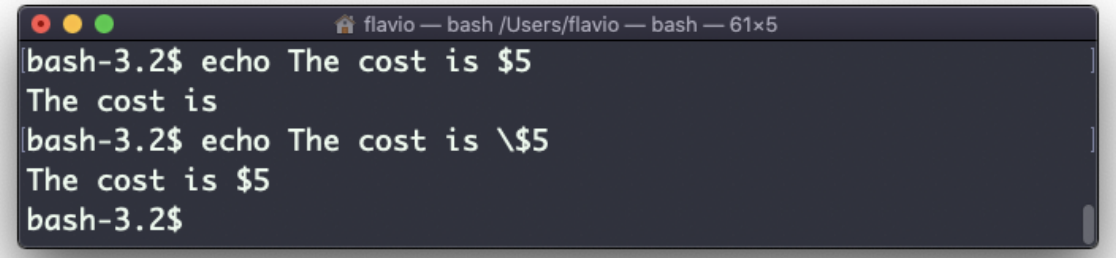
```
echo "hello" >> output.txt
```

We can interpolate environment variables:

```
echo "The path variable is $PATH"
```



Beware that special characters need to be escaped with a backslash `\`. `$` for example:



This is just the start. We can do some nice things when it comes to interacting with the shell features.

We can echo the files in the current folder:

```
echo *
```

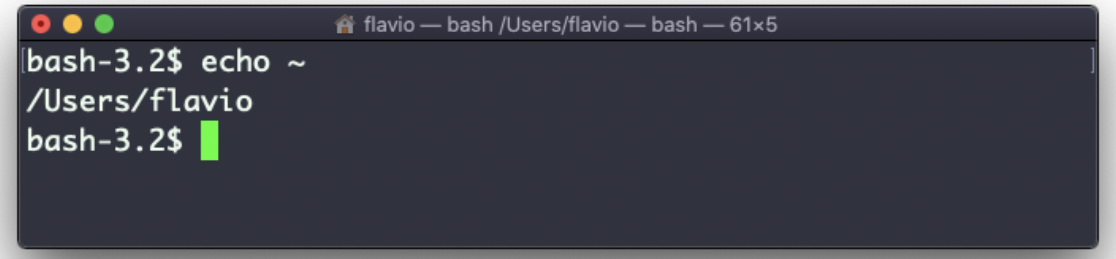
We can echo the files in the current folder that start with the letter `o`:

```
echo o*
```

Any valid Bash (or any shell you are using) command and feature can be used here.

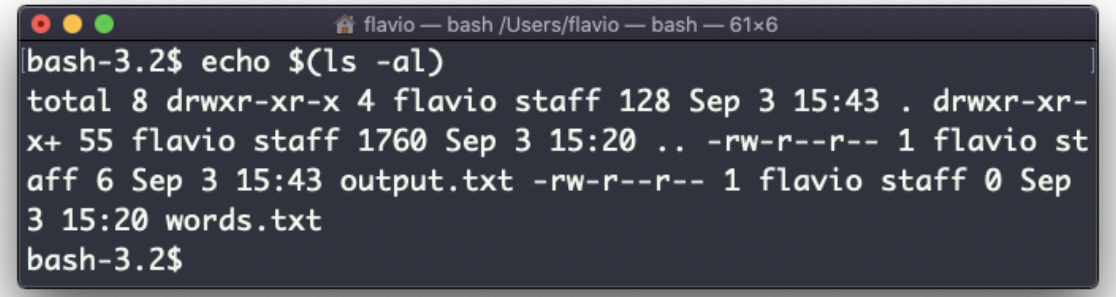
You can print your home folder path:

```
echo ~
```



You can also execute commands, and print the result to the standard output (or to file, as you saw):

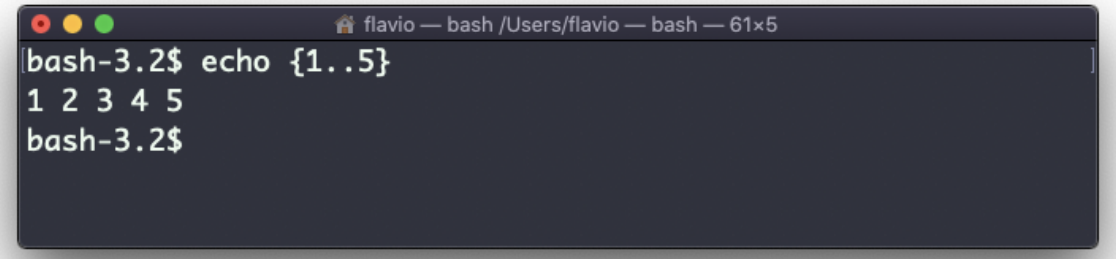
```
echo $(ls -al)
```



Note that whitespace is not preserved by default. You need to wrap the command in double quotes to do so:

You can generate a list of strings, for example ranges:

```
echo {1..5}
```



> The `echo` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: grep

Learn how the Linux grep command matches patterns in files or piped output, with -n for line numbers, -C for context, -i to ignore case, and -v to invert.

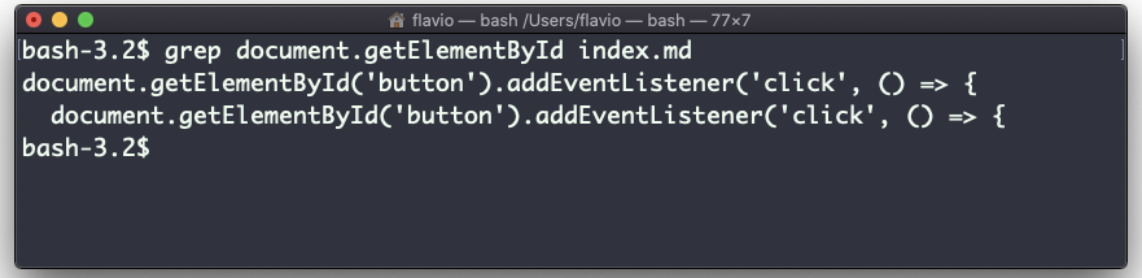
The `grep` command is a very useful tool, that when you master will help you tremendously in your day to day.

> If you're wondering, `grep` stands for *global regular expression print*

You can use `grep` to search in files, or combine it with pipes to filter the output of another command.

For example here's how we can find the occurences of the `document.getElementById` line in the `index.md` file:

```
grep -n document.getElementById index.md
```



Using the `-n` option it will show the line numbers:

```
grep -n document.getElementById index.md
```

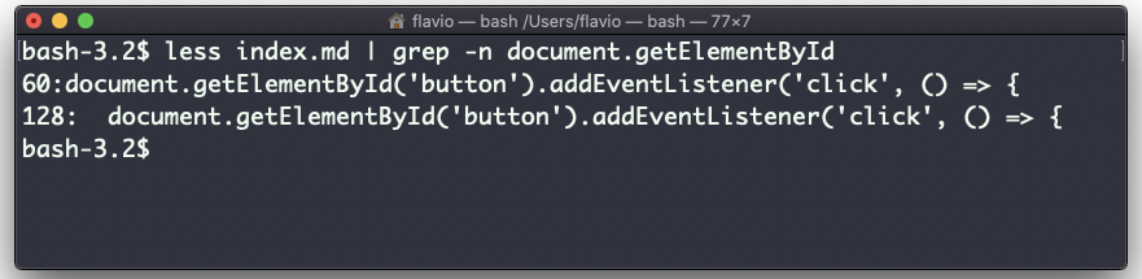
One very useful thing is to tell grep to print 2 lines before, and 2 lines after the matched line, to give us more context. That's done using the `-C` option, which accepts a number of lines:

```
grep -nC 2 document.getElementById index.md
```

Search is case sensitive by default. Use the `-i` flag to make it insensitive.

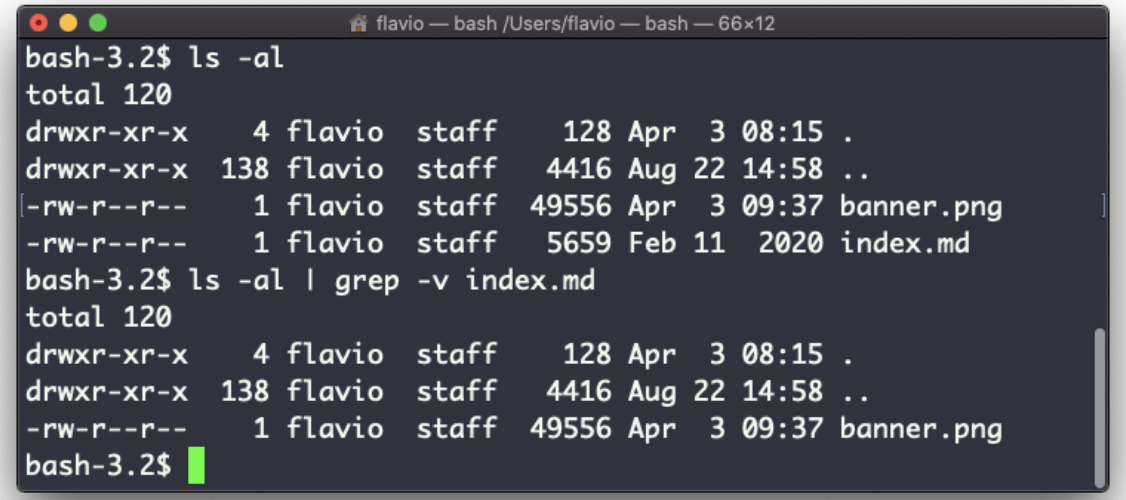
As mentioned, you can use grep to filter the output of another command. We can replicate the same functionality as above using:

```
less index.md | grep -n document.getElementById
```



The search string can be a regular expression, and this makes `grep` very powerful.

Another thing you might find very useful is to invert the result, excluding the lines that match a particular string, using the `-v` option:

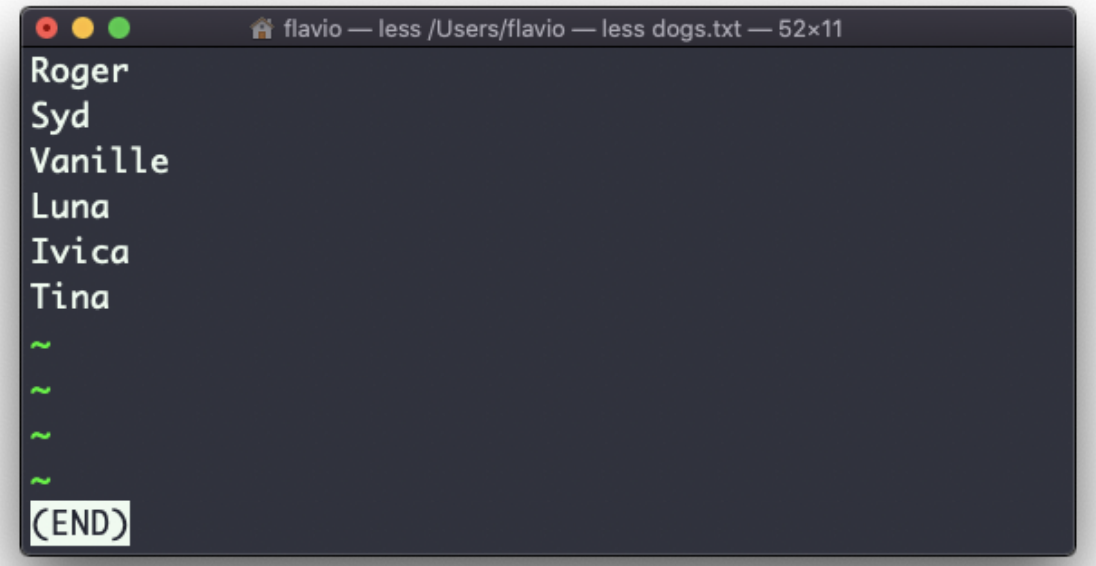


The `grep` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: sort

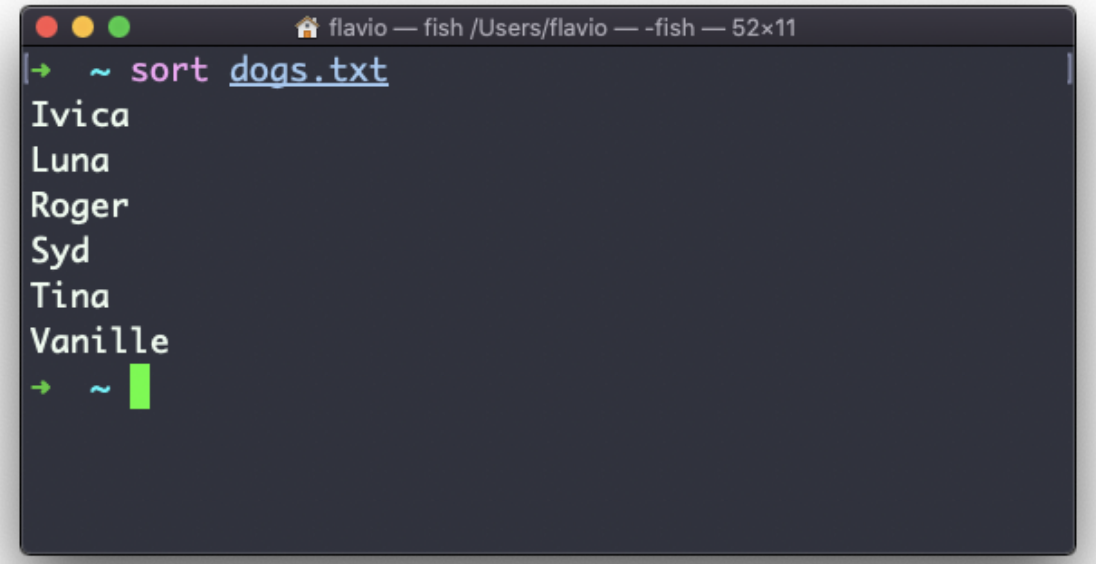
Learn how the Linux sort command orders the lines of a file or piped input, with -r to reverse, -n for numeric order, and -u to drop duplicate lines.

Suppose you have a text file which contains the names of dogs:

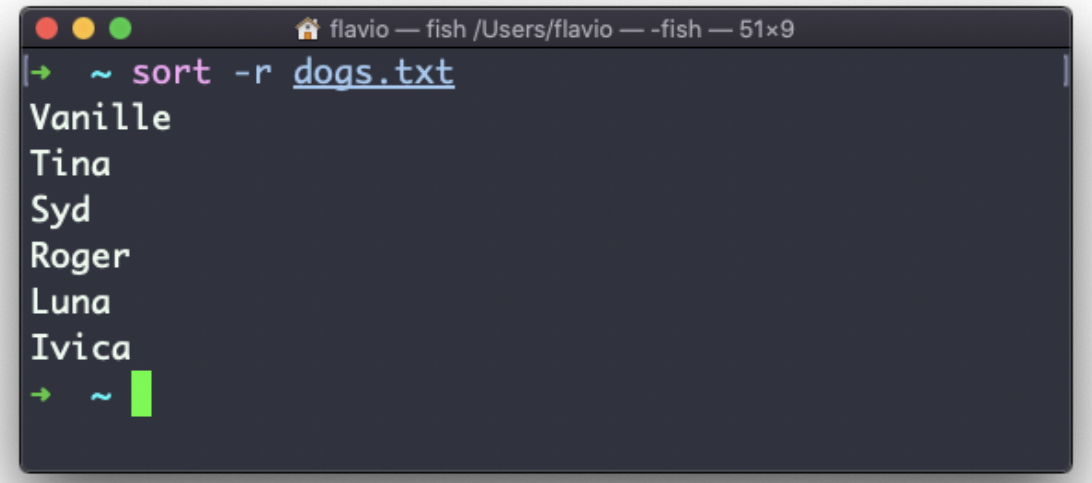


This list is unordered.

The `sort` command helps us sorting them by name:

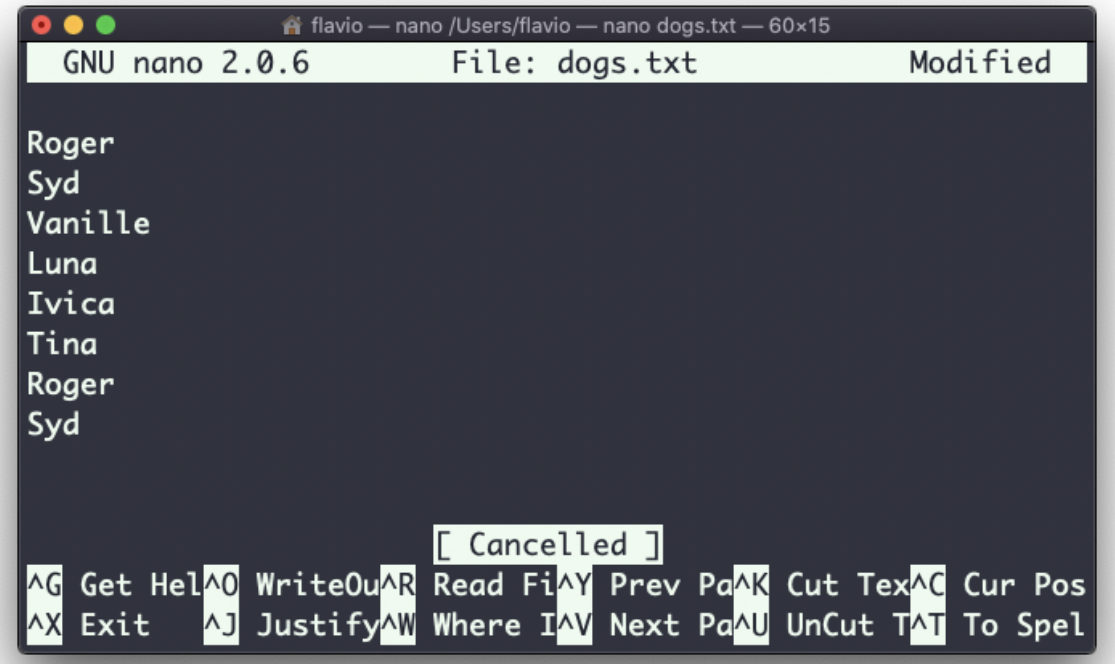


Use the `r` option to reverse the order:

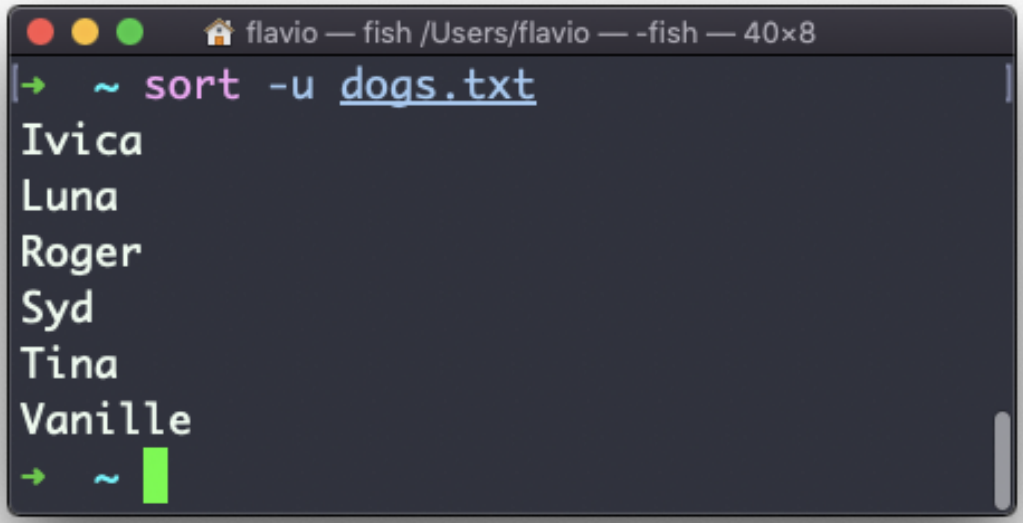


Sorting by default is case sensitive, and alphabetic. Use the `--ignore-case` option to sort case insensitive, and the `-n` option to sort using a numeric order.

If the file contains duplicate lines:



You can use the `-u` option to remove them:



`sort` does not just works on files, as many UNIX commands it also works with pipes, so you can use on the output of another command, for example you can order the files returned by `ls` with:

```
ls | sort
```

`sort` is very powerful and has lots more options, which you can explore calling `man sort`.

The `sort` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

If you have a list in your clipboard and want to sort or dedupe it without the terminal, I built a free line sorter that does both in the browser.

Linux commands: uniq

Learn how the Linux uniq command finds and removes duplicate lines in text, why you pair it with sort, and how -d, -u, and -c change what it reports.

`uniq` is a command useful to sort lines of text.

You can get those lines from a file, or using pipes from the output of another command:

```
uniq dogs.txt
```

```
ls | uniq
```

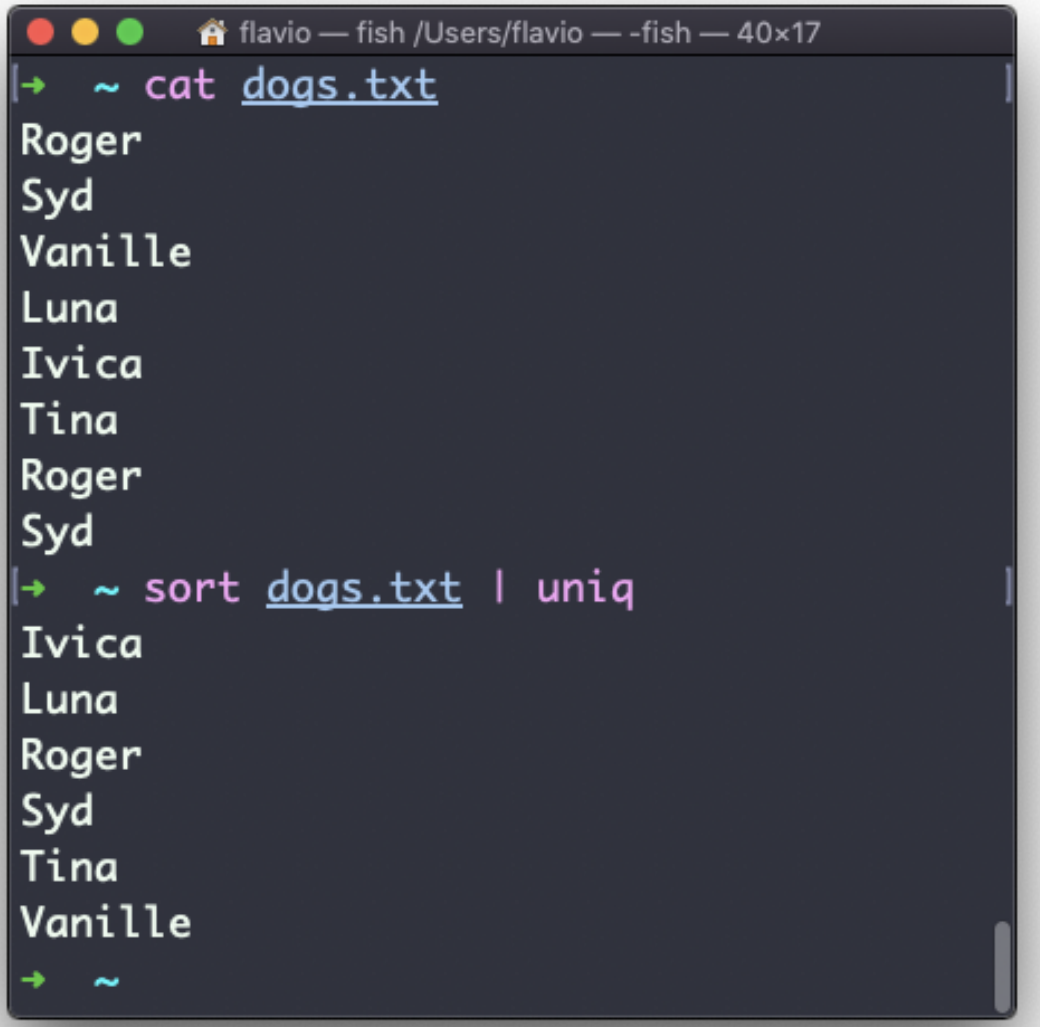
You need to consider this key thing: `uniq` will only detect adjacent duplicate lines.

This implies that you will most likely use it along with `sort`:

```
sort dogs.txt | uniq
```

The `sort` command has its own way to remove duplicates with the `-u` (*unique*) option. But `uniq` has more power.

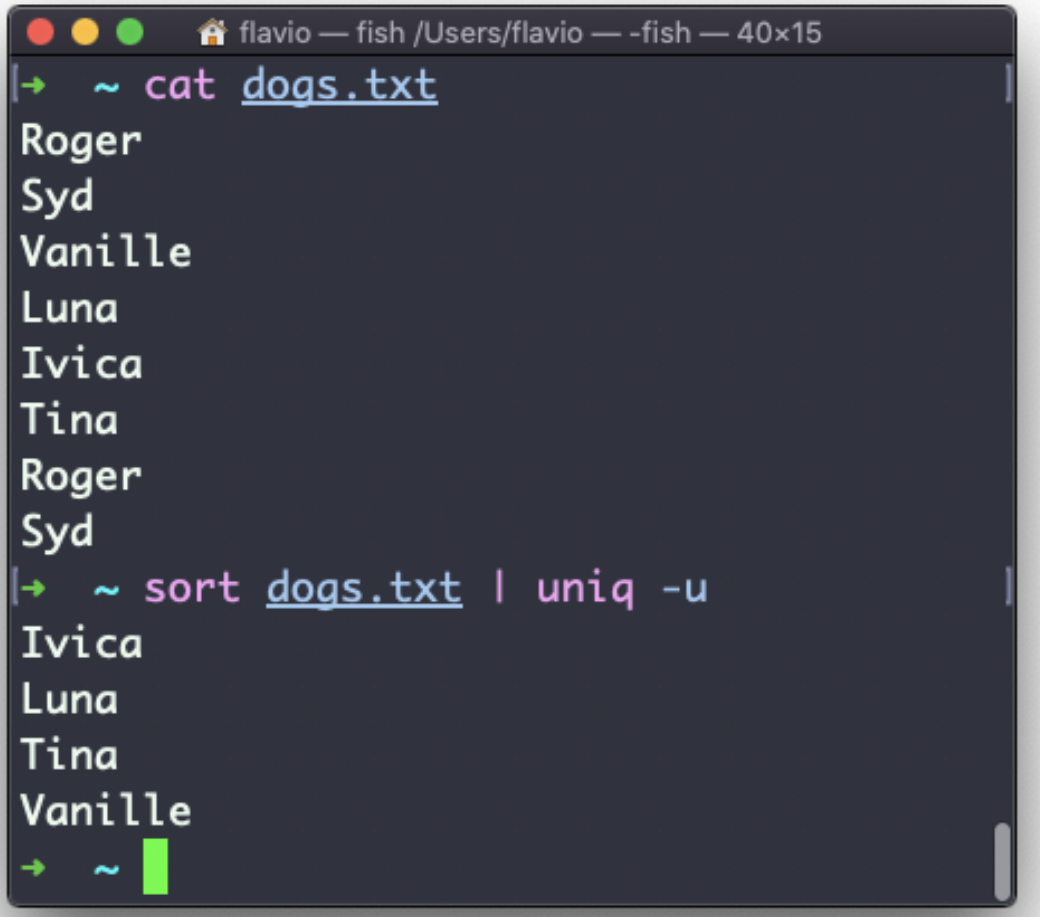
By default it removes duplicate lines:



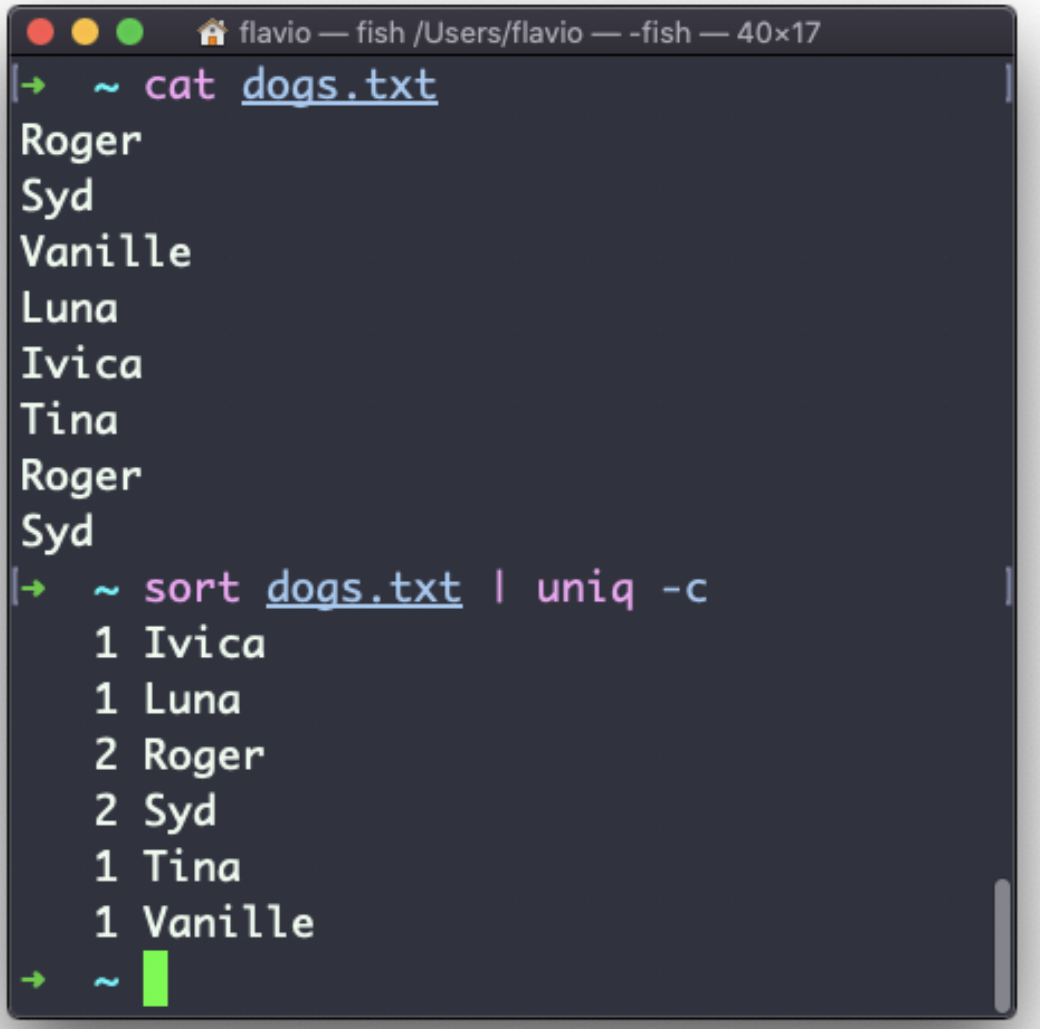
You can tell it to only display duplicate lines, for example, with the `-d` option:

```
sort dogs.txt | uniq -d
```

You can use the `-u` option to only display non-duplicate lines:



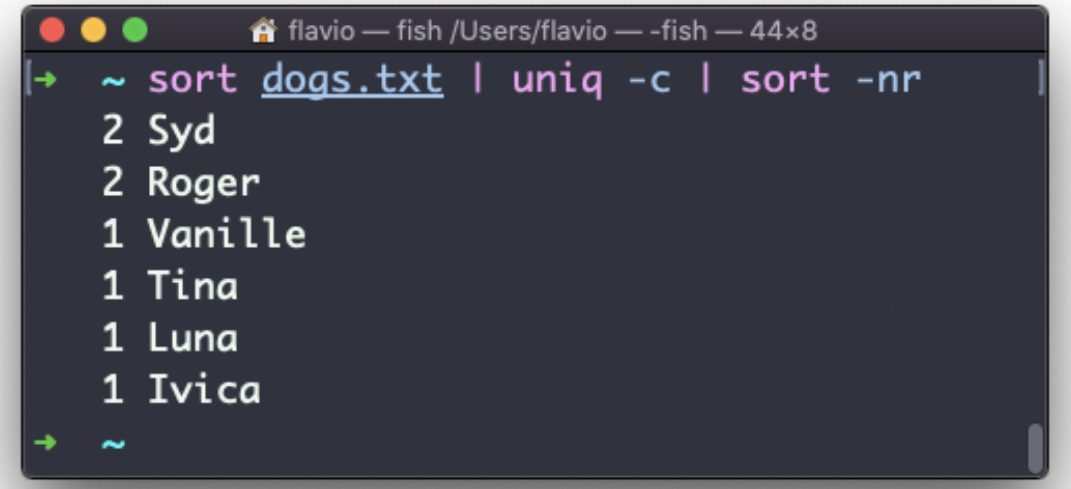
You can count the occurrences of each line with the `-c` option:



Use the special combination:

`sort dogs.txt | uniq -c | sort -nr`

to then sort those lines by most frequent:



The `uniq` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

For a quick paste-and-clean pass over a block of text, I built a free line sorter that sorts lines and removes duplicates.

Linux commands: wc

Learn how the Linux wc command counts the lines, words, and bytes in a file or piped input, plus using -l, -w, -c, and -m for characters in Unicode text.

The `wc` command gives us useful information about a file or input it receives via pipes.

```
echo test >> test.txt
wc test.txt
1       1       5 test.txt
```

Example via pipes, we can count the output of running the `ls -al` command:

```
ls -al | wc
6      47     284
```

The first column returned is the number of lines. The second is the number of words. The third is the number of bytes.

We can tell it to just count the lines:

```
wc -l test.txt
```

or just the words:

```
wc -w test.txt
```

or just the bytes:

```
wc -c test.txt
```

Bytes in ASCII charsets equate to characters, but with non-ASCII charsets, the number of characters

might differ because some characters might take multiple bytes, for example this happens in Unicode.

In this case the `-m` flag will help getting the correct value:

```
wc -m test.txt
```

> The `wc` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: xargs

Learn how the Linux xargs command turns the output of one command into arguments for another, like piping cat into rm, plus the handy -p and -n options.

The `xargs` command is used in a UNIX shell to convert input from standard input into arguments to a command.

In other words, through the use of `xargs` the output of a command is used as the input of another command.

Here's the syntax you will use:

```
command1 | xargs command2
```

We use a pipe (`|`) to pass the output to `xargs`. That will take care of running the `command2` command, using the output of `command1` as its argument(s).

Let's do a simple example. You want to remove some specific files from a directory. Those files are listed inside a text file.

We have 3 files: `file1`, `file2`, `file3`.

In `todelete.txt` we have a list of files we want to delete, in this example `file1` and `file3`:

We will channel the output of `cat todelete.txt` to the `rm` command, through `xargs`.

In this way:

```
cat todelete.txt | xargs rm
```

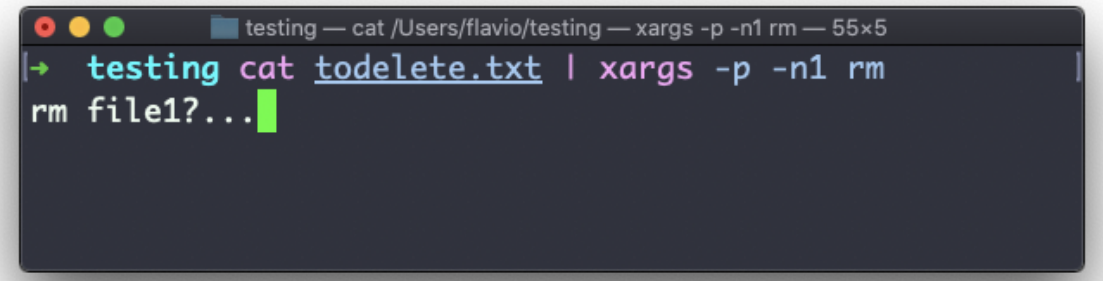
That's the result, the files we listed are now deleted:

The way it works is that `xargs` will run `rm` 2 times, one for each line returned by `cat`.

This is the simplest usage of `xargs`. There are several options we can use.

One of the most useful in my opinion, especially when starting to learn `xargs`, is `-p`. Using this option will make `xargs` print a confirmation prompt with the action it's going to take:

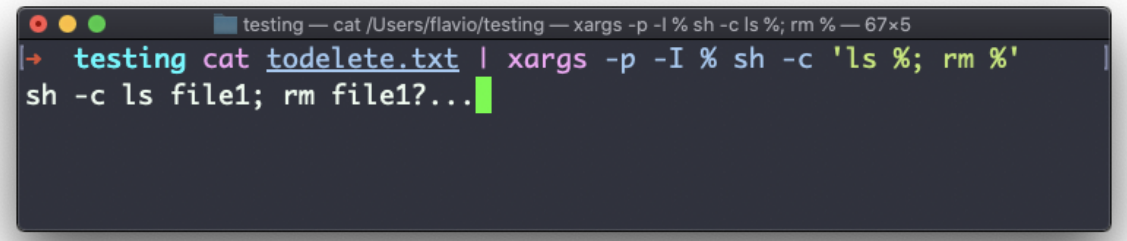
The `-n` option lets you tell `xargs` to perform one iteration at a time, so you can individually confirm them with `-p`. Here we tell `xargs` to perform one iteration at a time with `-n1`:



The `-I` option is another widely used one. It allows you to get the output into a placeholder, and then you can do various things.

One of them is to run multiple commands:

```
command1 | xargs -I % /bin/bash -c 'command2 %; command3 %'
```



You can swap the `%` symbol I used above with anything else, it's a variable

The `xargs` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

How to set environment variables in bash and zsh

Learn how to set environment variables in Bash and zsh with export, how to persist them in .bashrc or .zshrc, and the env prefix trick for the Fish shell.

An environment variable is a name/value pair inherited by child processes. In Bash and zsh, assign a shell variable and export it:

Setting them in the shell is the same:

```
export APP_ENV=development
```

Read it with `printf` or inspect the exported environment:

```
printf '%s\n' "$APP_ENV"
env | grep '^APP_ENV='
```

Typing `$APP_ENV` by itself tries to execute its value as a command. Variable expansion supplies text; it is not a print operation.

Limit a value to one process by prefixing the command:

```
APP_ENV=test node test.js
```

The current shell is unchanged after that command exits. This is ideal for one-off configuration.

To persist settings for future interactive shells, put them in the startup file your shell actually reads, commonly `~/.zshrc` for zsh or `~/.bashrc` for interactive Bash. Reload a known file with `source ~/.zshrc`, or open a new shell. Startup-file behavior differs between login and interactive shells, so diagnose with `echo "$SHELL"` and the shell's documentation instead of editing every dotfile.

Environment variables are visible to the process and often to child processes and diagnostic tools. Do not treat them as encrypted storage. Avoid committing secrets to startup files or printing them in logs.

Remove a variable from the current shell with:

```
unset APP_ENV
```

Try it: export a value, read it from a child `sh -c` process, unset it, and confirm the next child no longer receives it.

Linux commands: export

Learn how the Linux export command makes a variable available to child processes and subshells, how to append to PATH, and how to remove a variable with -n.

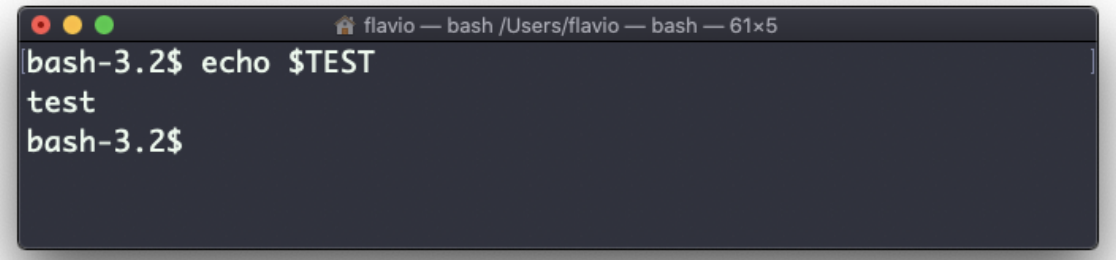
The `export` command is used to export variables to child processes.

What does this mean?

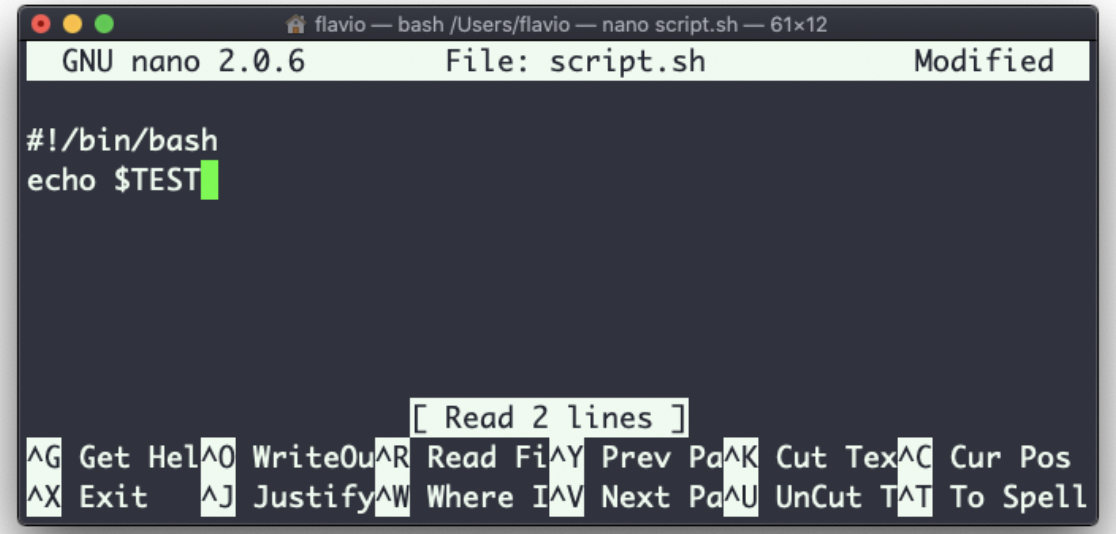
Suppose you have a variable TEST defined in this way:

```
TEST="test"
```

You can print its value using `echo $TEST`:



But if you try defining a Bash script in a file `script.sh` with the above command:



Then you set `chmod u+x script.sh` and you execute this script with `./script.sh`, the `echo $TEST` line will print nothing!

This is because in Bash the `TEST` variable was defined local to the shell. When executing a shell script or another command, a subshell is launched to execute it, which does not contain the current shell local variables.

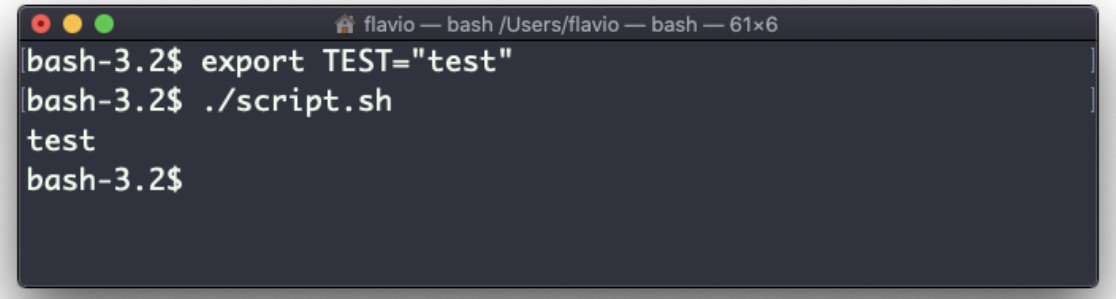
To make the variable available there we need to define `TEST` not in this way:

```
TEST="test"
```

but in this way:

```
export TEST="test"
```

Try that, and running `./script.sh` now should print "test":



Sometimes you need to append something to a variable. It's often done with the `PATH` variable. You use this syntax:

```
export PATH=$PATH:/new/path
```

It's common to use `export` when you create new variables in this way, but also when you create variables in the `.bash_profile` or `.bashrc` configuration files with Bash, or in `.zshenv` with Zsh.

To remove a variable, use the `-n` option:

```
export -n TEST
```

Calling `export` without any option will list all the exported variables.

The `export` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Introduction to Bash Shell Scripting

Learn Bash scripting with executable files, variables, quoting, conditions, loops, functions, arrays, positional arguments, and getopts.

Bash scripts put shell commands into a file so you can repeat a task.

Bash is both a command interpreter and a programming language. It has variables, conditions, loops, arrays, functions, and input/output redirection.

Shell scripts are excellent for connecting command-line tools. When a script grows into a large application, another language may be easier to test and maintain.

> Check out my introduction to Bash first if the shell is new to you.

Create a script

Create a file named `hello.sh`:

```
#!/usr/bin/env bash

echo "Hello from Bash"
```

The first line is the **shebang**. It asks `env` to find `bash` in the current `PATH`.

Use an exact path such as `#!/bin/bash` when the environment guarantees that location and you want that specific interpreter.

Make the file executable:

```
chmod u+x hello.sh
```

Run it:

```
./hello.sh
```

Alternatively, pass the file to Bash:

```
bash hello.sh
```

In that form, Bash is already the interpreter and does not use the shebang to choose one.

Check syntax without running the script:

```
bash -n hello.sh
```

Add comments

A comment starts with `#`:

```
#!/usr/bin/env bash

# Print a greeting
echo "Hello"
```

The shebang is the special first-line exception.

Create variables

Assign a value without spaces around `=`:

```
name="Flavio"
count=3
```

Read a variable with `$`:

```
echo "$name"
echo "$count"
```

Quote variable expansions unless you specifically need word splitting or filename expansion:

```
file_name="My notes.txt"
cat "$file_name"
```

Without quotes, Bash can split one value into several arguments and expand wildcard characters.

Environment variables are inherited by child processes only when exported:

```
export APP_ENV="production"
```

Capture command output

Use command substitution:

```
current_date=$(date +%F)
echo "Today is $current_date"
```

`$(...)` is easier to nest and read than the older backtick syntax.

Work with exit statuses

Every command returns an exit status.

`0` means success. A nonzero value means failure.

Run a second command only when the first succeeds:

```
mkdir -p backup && cp notes.txt backup/
```

Run a fallback when the first command fails:

```
cp notes.txt backup/ || echo "The copy failed" >&2
```

Negate a command's status with `!`:

```
if ! grep -q "ready" status.txt
then
  echo "Not ready"
fi
```

Write conditions

Use `[[ ... ]]` for Bash string and file tests:

```
dog_name="Roger"

if [[ -z $dog_name ]]
then
  echo "A name is required"
else
  echo "The dog is $dog_name"
fi
```

`-z` tests for an empty string. Use `-n` for a non-empty string.

Compare strings:

```
if [[ $dog_name == "Roger" ]]
then
  echo "Found Roger"
fi
```

Test files:

```
if [[ -f notes.txt ]]
then
  echo "notes.txt is a regular file"
fi
```

Common file tests include:

- `-e` exists
- `-f` is a regular file
- `-d` is a directory
- `-r` is readable
- `-w` is writable
- `-x` is executable

Use `(( ... ))` for integer arithmetic:

```
age=23
minimum=18

if (( age >= minimum ))
then
  echo "Old enough"
fi
```

Inside arithmetic expressions, use operators such as `+`, `-`, `*`, `/`, `%`, `<`, `<=`, `==`, `>=`, and `>`.

The `test` command and `[ ... ]` also work, including in POSIX shell scripts. Their parsing rules differ from `[[ ... ]]`, so do not mix the syntaxes blindly.

Use if, elif, and else

The complete shape is:

```
if command_one
then
  echo "First condition matched"
elif command_two
then
  echo "Second condition matched"
else
  echo "No condition matched"
fi
```

The commands after `if` and `elif` are tested by exit status.

Loop over values

Use `for` to iterate over a list:

```
for city in Copenhagen Rome Lisbon
do
  echo "$city"
done
```

Loop over script arguments without losing their boundaries:

```
for argument in "$@"
do
  echo "$argument"
done
```

`"$@"` expands to one quoted word per argument. Prefer it over `$*`.

Use `while` while a command succeeds:

```
count=1

while (( count <= 3 ))
do
  echo "$count"
  ((count += 1))
done
```

Use `break` to leave a loop and `continue` to skip to its next iteration.

Match values with case

`case` is clear when one value can match several patterns:

```
case $1 in
  start)
    echo "Starting"
    ;;
  stop)
    echo "Stopping"
    ;;
  restart|reload)
```

```
    echo "Restarting"
    ;;
  *)
    echo "Usage: $0 {start|stop|restart}" >&2
    exit 2
    ;;
esac
```

The `*)` branch is the fallback.

Read input

Use `read -r` so backslashes are not treated as escapes:

```
read -r -p "Your name: " name
echo "Hello $name"
```

Check whether input succeeded when a script can receive end-of-file:

```
if IFS= read -r line
then
  echo "$line"
fi
```

Setting `IFS=` preserves leading and trailing whitespace.

Use positional arguments

Bash exposes script arguments as:

- `$0`: the script name
- `$1`, `$2`, and so on: positional arguments
- `$#`: the number of positional arguments
- `"$@"`: every positional argument, preserving boundaries

Example:

```
#!/usr/bin/env bash

if (( $# != 1 ))
then
  echo "Usage: $0 FILE" >&2
  exit 2
fi

file=$1
echo "Processing $file"
```

Use `${10}` for argument ten and higher.

Parse options with getopts

The Bash `getopts` builtin parses short options:

```
verbose=false
```

```
name=""

while getopts ":vn:" option
do
  case $option in
    v)
      verbose=true
      ;;
    n)
      name=$OPTARG
      ;;
    :)
      echo "Option -$OPTARG needs a value" >&2
      exit 2
      ;;
    \?)
      echo "Unknown option: -$OPTARG" >&2
      exit 2
      ;;
  esac
done

shift "$((OPTIND - 1))"
```

The option name goes into `option`. An option's value goes into `OPTARG`.

The colon after `n` in `:vn:` means `-n` needs a value. The leading colon lets the script handle errors itself.

After `shift`, the remaining positional arguments start at `$1`.

Use `--` when calling a command to mark the end of options:

```
./report.sh -v -- -july.csv
```

The single hyphen in the old tutorial was incorrect.

Create arrays

Create an indexed array:

```
breeds=(
  "husky"
  "setter"
  "border collie"
)
```

Read one element:

```
echo "${breeds[0]}"
```

Loop over every element:

```
for breed in "${breeds[@]}"
do
```

```
  echo "$breed"
done
```

Get the number of elements:

```
echo "${#breeds[@]}"
```

Always include the parameter expansion syntax. Writing `breeds[0]` by itself does not read the value.

Create functions

Define a function:

```
clean_folder() {
  local folder=$1

  echo "Cleaning $folder"
}
```

Call it like another command:

```
clean_folder "/Users/flavio/Desktop"
```

Function arguments use `$1`, `$2`, and `"$@"`.

Variables are not all global. Use the Bash `local` builtin to keep a function variable inside that function.

Return a status with `return`:

```
is_text_file() {
  [[ $1 == *.txt ]]
}
```

Bash functions return an integer status, not an arbitrary string. Print data to standard output when a caller needs to capture it.

Handle failures

Check failures where you can explain or recover from them:

```
if ! cp "$source" "$destination"
then
  echo "Could not copy $source" >&2
  exit 1
fi
```

Many scripts enable stricter behavior with:

```
set -u
set -o pipefail
```

`set -u` reports unset variables. `pipefail` makes a pipeline fail when any command in it fails.

You will also see `set -e`. Its behavior has important exceptions around conditions, pipelines, subshells, and command lists. Do not add it without understanding how the script handles expected failures.

For the complete language rules and builtin documentation, see the official GNU Bash Reference Manual.

Check your understanding: pipes and text

Check standard input and output, redirection, pipelines, grep, sort, uniq, environment variables, and command substitution.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Processes and jobs

Inspect, pause, resume, schedule, and stop running programs.

Linux commands: ps

Learn how the Linux ps command lists running processes, how ps ax shows them all, and how to read columns like PID and STAT or filter the output with grep.

Your computer is running, at all times, tons of different processes.

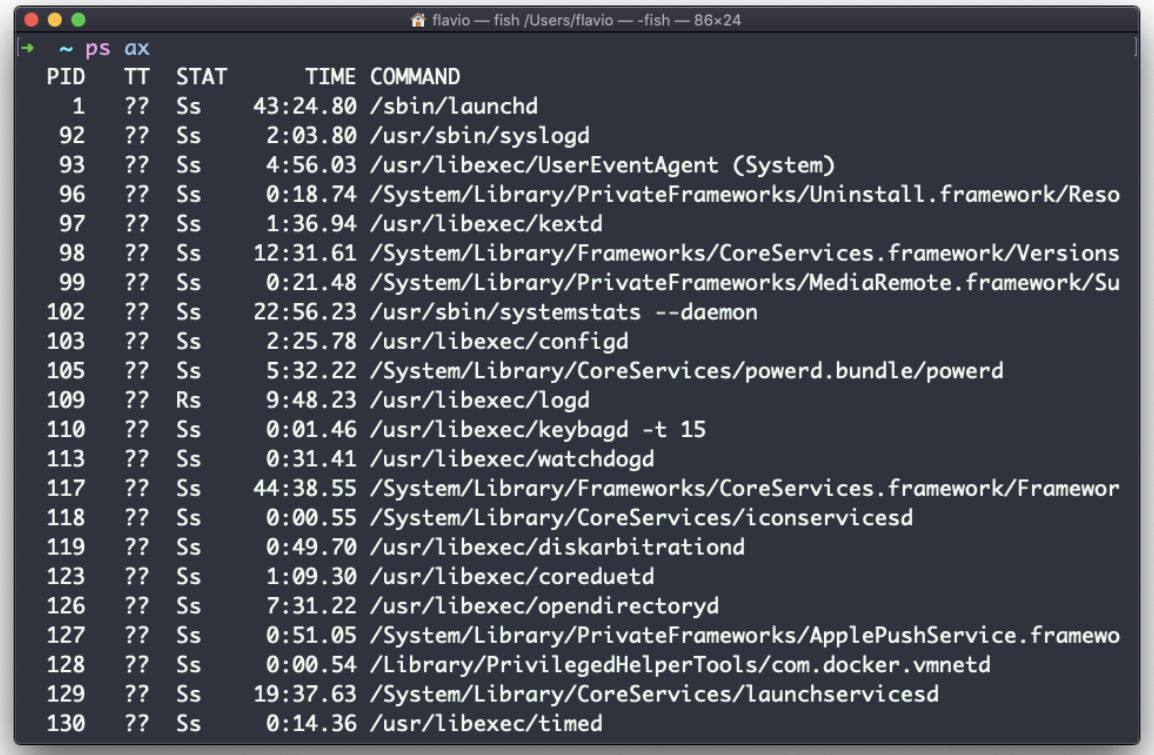
You can inspect them all using the `ps` command:

This is the list of user-initiated processes currently running in the current session.

Here I have a few `fish` shell instances, mostly opened by VS Code inside the editor, and an instances of Hugo running the development preview of a site.

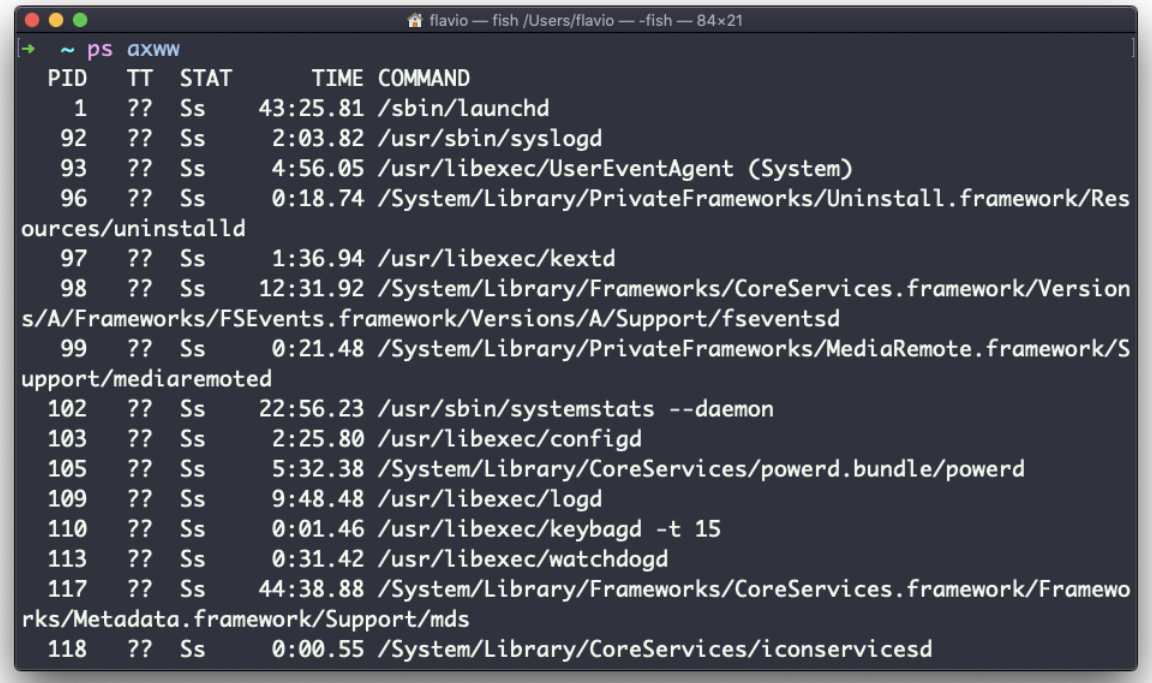
Those are just the commands assigned to the current user. To list **all** processes we need to pass some options to `ps`.

The most common I use is `ps ax`:



The `a` option is used to also list other users processes, not just our own. `x` shows processes not linked to any terminal (not initiated by users through a terminal).

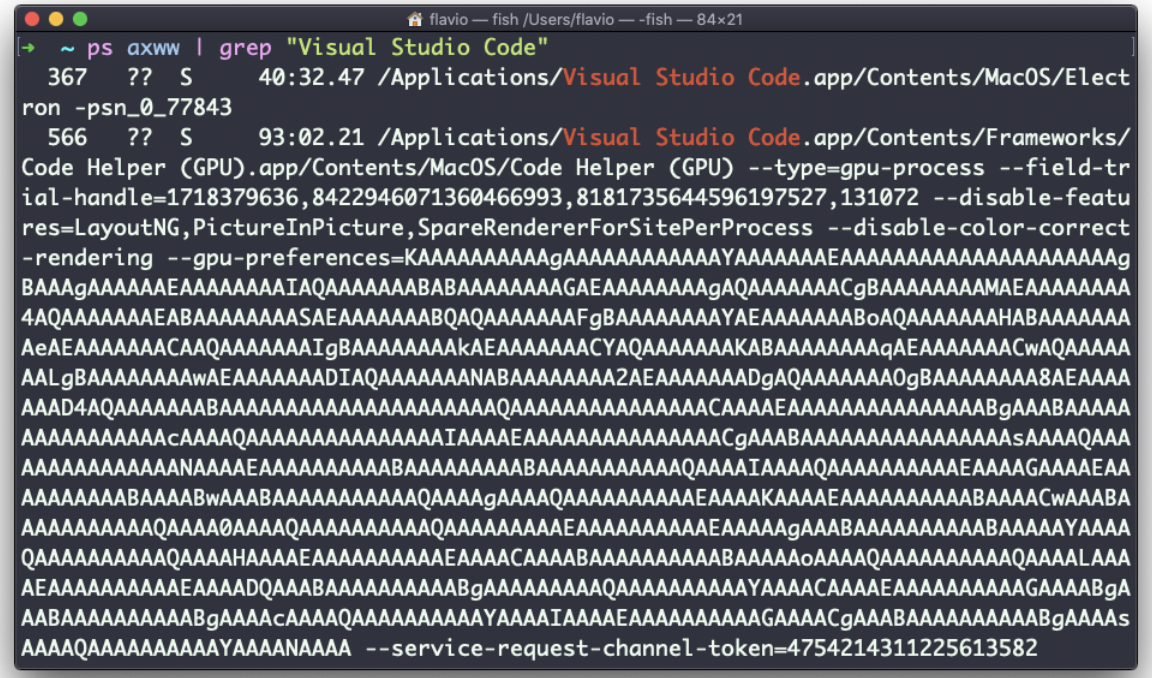
As you can see, the longer commands are cut. Use the command `ps axww` to continue the command listing on a new line instead of cutting it:



We need to specify `w` 2 times to apply this setting, it's not a typo.

You can search for a specific process combining `grep` with a pipe, like this:

```
ps axww | grep "VS Code"
```



The columns returned by `ps` represent some key information.

The first information is `PID`, the process ID. This is key when you want to reference this process in another command, for example to kill it.

Then we have `TT` that tells us the terminal id used.

Then `STAT` tells us the state of the process:

`I` a process that is idle (sleeping for longer than about 20 seconds) `R` a runnable process `S` a process that is sleeping for less than about 20 seconds `T` a stopped process `U` a process in uninterruptible wait `Z` a dead process (a *zombie*)

If you have more than one letter, the second represents further information, which can be very technical.

It's common to have `+` which indicates the process is in the foreground in its terminal. `s` means the process is a session leader.

`TIME` tells us how long the process has been running.

> This command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: top

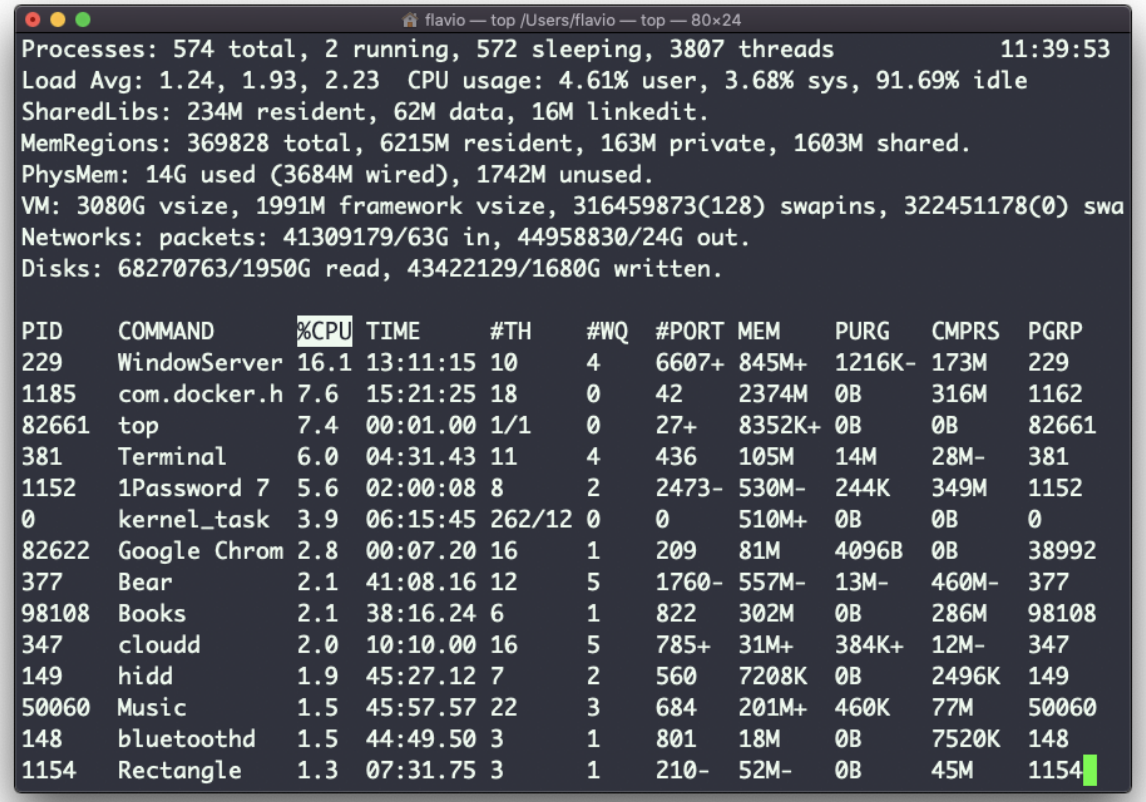
Learn how the Linux top command shows running processes in real time, with live CPU, memory, and system load, and how top -o mem sorts them by memory use.

A quick guide to the `top` command, used to list the processes running in real time

The `top` command is used to display dynamic real-time information about running processes in the system.

It's really handy to understand what is going on.

Its usage is simple, you just type `top`, and the terminal will be fully immersed in this new view:



The process is long-running. To quit, you can type the `q` letter or `ctrl-C`.

There's a lot of information being given to us: the number of processes, how many are running or sleeping, the system load, the CPU usage, and a lot more.

Below, the list of processes taking the most memory and CPU is constantly updated.

By default, as you can see from the `%CPU` column highlighted, they are sorted by the CPU used.

You can add a flag to sort processes by memory utilized:

```
top -o mem
```

> This command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: jobs

Learn how the Linux jobs command lists the background jobs you started with &, giving you the job number for fg and showing process ids with jobs -l.

When we run a command in Linux / macOS, we can set it to run in the background using the `&`

symbol after the command.

For example we can run `top` in the background:

```
top &
```

This is very handy for long-running programs.

We can get back to that program using the `fg` command. This works fine if we just have one job in the background, otherwise we need to use the job number: `fg 1`, `fg 2` and so on.

To get the job number, we use the `jobs` command.

Say we run `top &` and then `top -o mem &`, so we have 2 top instances running. `jobs` will tell us this:

Now we can switch back to one of those using `fg <jobid>`. To stop the program again we can hit `cmd-Z`.

Running `jobs -l` will also print the process id of each job.

> This command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: bg

Learn how the Linux bg command resumes a job you suspended with ctrl-Z, running it in the background so you can keep working in the terminal.

When a command is running you can suspend it using `ctrl-Z`.

The command will immediately stop, and you get back to the shell terminal.

You can resume the execution of the command in the background, so it will keep running but it will not prevent you from doing other work in the terminal.

In this example I have 2 commands stopped:

Terminal output showing jobs command with two stopped jobs labeled [1] and [2], both running top command in different directories

I can run `bg 1` to resume in the background the execution of the job #1.

I could have also said `bg` without any option, as the default is to pick the job #1 in the list.

The `bg` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: fg

Learn how the Linux fg command brings a background job back to the foreground, resuming the last one or a specific job number you find with the jobs command.

When a command is running in the background, because you started it with `&` at the end (example: `top &` or because you put it in the background with the `bg` command, you can put it to the foreground using `fg`.

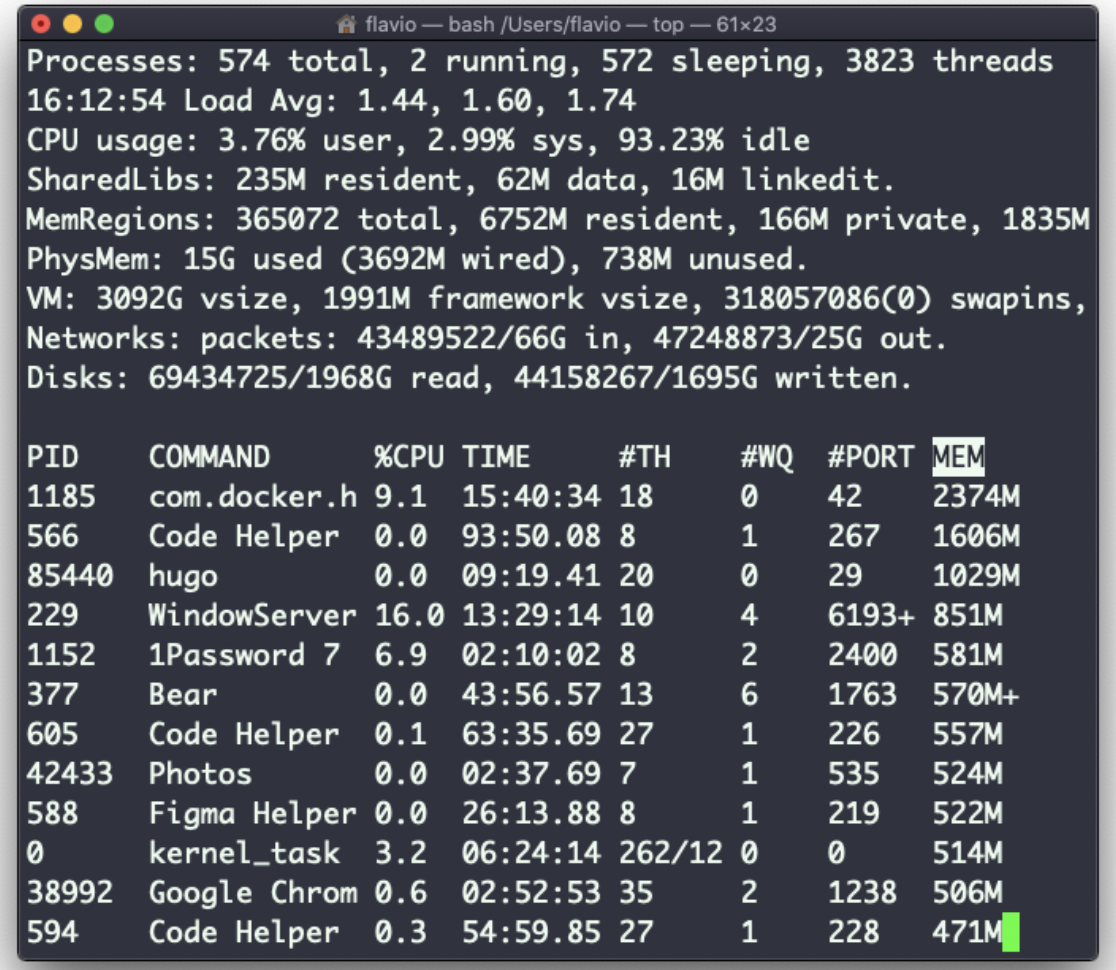
Running

```
fg
```

will resume to the foreground the last job that was suspended.

You can also specify which job you want to resume to the foreground passing the job number, which you can get using the `jobs` command.

Running `fg 2` will resume job #2:



The `fg` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: kill

Learn how the Linux kill command sends signals to a process by PID, defaulting to TERM, and how to send others like KILL, HUP, or STOP, by name or number.

Linux processes can receive **signals** and react to them.

That's one way we can interact with running programs.

The `kill` program can send a variety of signals to a program.

It's not just used to terminate a program, like the name would suggest, but that's its main job.

We use it in this way:

```
kill <PID>
```

By default, this sends the `TERM` signal to the process id specified.

We can use flags to send other signals, including:

```
kill -HUP <PID>
kill -INT <PID>
kill -KILL <PID>
kill -TERM <PID>
kill -CONT <PID>
kill -STOP <PID>
```

`HUP` means **hang up**. It's sent automatically when a terminal window that started a process is closed before terminating the process.

`INT` means **interrupt**, and it sends the same signal used when we press `ctrl-C` in the terminal, which usually terminates the process.

`KILL` is not sent to the process, but to the operating system kernel, which immediately stops and terminates the process.

`TERM` means **terminate**. The process will receive it and terminate itself. It's the default signal sent by `kill`.

`CONT` means **continue**. It can be used to resume a stopped process.

`STOP` is not sent to the process, but to the operating system kernel, which immediately stops (but does not terminate) the process.

You might see numbers used instead, like `kill -1 <PID>`. In this case,

`1` corresponds to `HUP`. `2` corresponds to `INT`. `9` corresponds to `KILL`. `15` corresponds to `TERM`. `18` corresponds to `CONT`. `15` corresponds to `STOP`.

> This command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: nohup

Learn how the Linux nohup command keeps a process running after you log out or close the terminal, which is handy for long-lived jobs on a remote server.

Closing a terminal normally sends a hangup signal to jobs associated with that session. `nohup` starts a command with that signal ignored:

```
nohup ./generate-report >report.log 2>&1 &
```

Each part has a job:

- `nohup` protects the command from the hangup signal.
- `>report.log` records standard output.
- `2>&1` sends errors to the same file.
- `&` returns the shell prompt while the process runs in the background.

Without explicit redirection, many implementations write to `nohup.out`. Record the process ID immediately with `echo $!`, then verify the process and watch its log:

```
job_pid=$!
ps -p "$job_pid"
tail -f report.log
```

Stop the process later with `kill "$job_pid"`. If you lose the process ID, find the process with `ps` before sending a signal. Also watch the log size: `nohup.out` and other unrotated logs can grow until

they fill the filesystem.

Running a command with `&` alone only puts it in the background. It can still receive a hangup signal when the session closes. `nohup` and `&` solve two separate problems.

`nohup` is not a service manager. It does not restart a crashed process, start it after a reboot, rotate logs, manage dependencies, or expose structured status. Use systemd, a container orchestrator, or another supervisor for a production service. Use `tmux` or `screen` when you want to reconnect to an interactive terminal rather than detach a non-interactive job.

Also check whether the program needs input. A background command waiting on the closed terminal can still fail or stop.

Try it with `nohup sh -c 'sleep 10; date' >result.log 2>&1 &`, close or detach the session, then confirm the timestamp appears.

Linux commands: crontab

Learn how the Linux crontab command schedules cron jobs to run at set intervals, listing them with crontab -l and editing them with crontab -e.

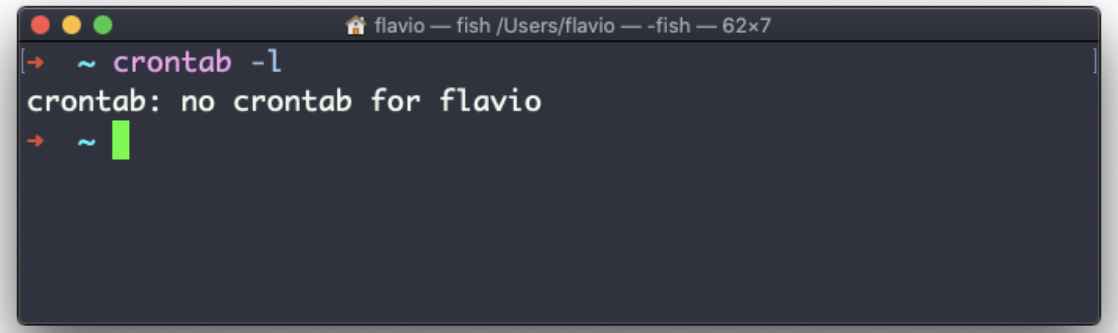
Cron jobs are jobs that are scheduled to run at specific intervals. You might have a command perform something every hour, or every day, or every 2 weeks. Or on weekends. They are very powerful, especially on servers to perform maintenance and automations.

The `crontab` command is the entry point to work with cron jobs.

The first thing you can do is to explore which cron jobs are defined by you:

```
crontab -l
```

You might have none, like me:



Run

```
crontab -e
```

to edit the cron jobs, and add new ones.

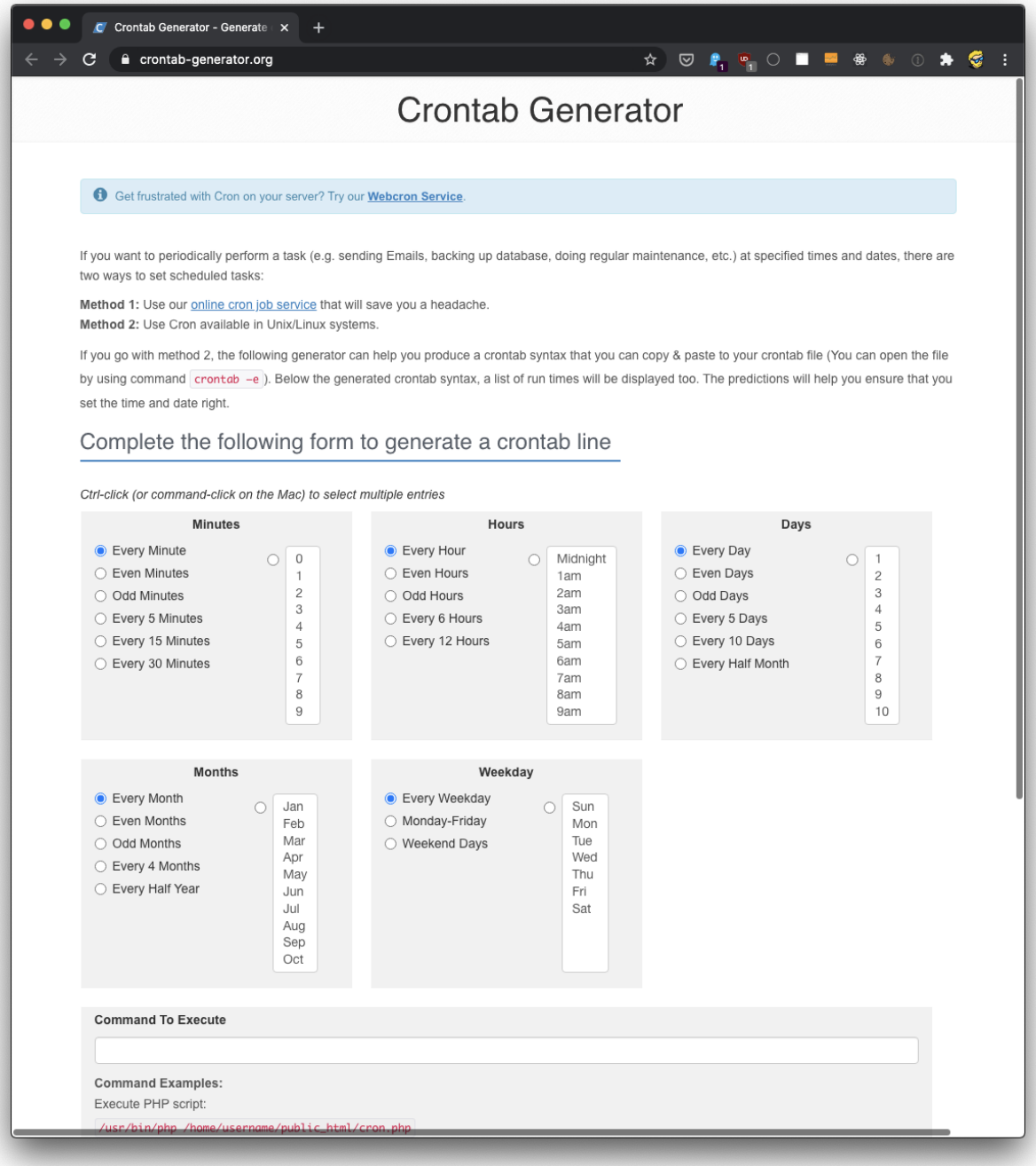
By default this opens with the default editor, which is usually `vim`. I like `nano` more, you can use this line to use a different editor:

```
EDITOR=nano crontab -e
```

Now you can add one line for each cron job.

The syntax to define cron jobs is kind of scary. This is why I usually use a website to help me generate it without errors: https://crontab-generator.org/

I also built my own free cron expression builder that explains the schedule in plain English and shows the next run times.



You pick a time interval for the cron job, and you type the command to execute.

I chose to run a script located in `/Users/flavio/test.sh` every 12 hours. This is the crontab line I need to run:

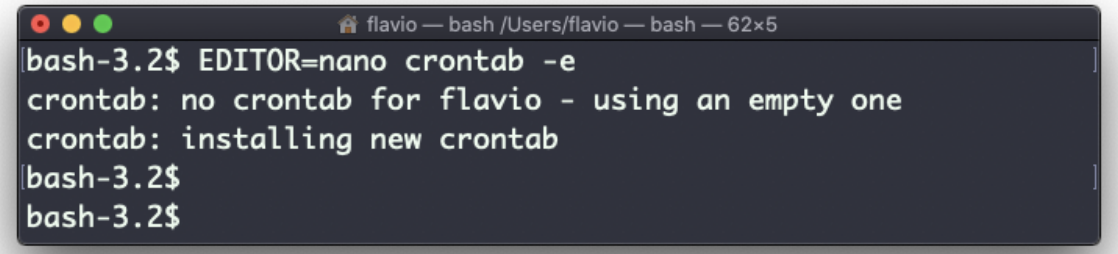
```
* */12 * * * /Users/flavio/test.sh >/dev/null 2>&1
```

I run `crontab -e`:

`EDITOR=nano crontab -e`

and I add that line, then I press `ctrl-X` and press `y` to save.

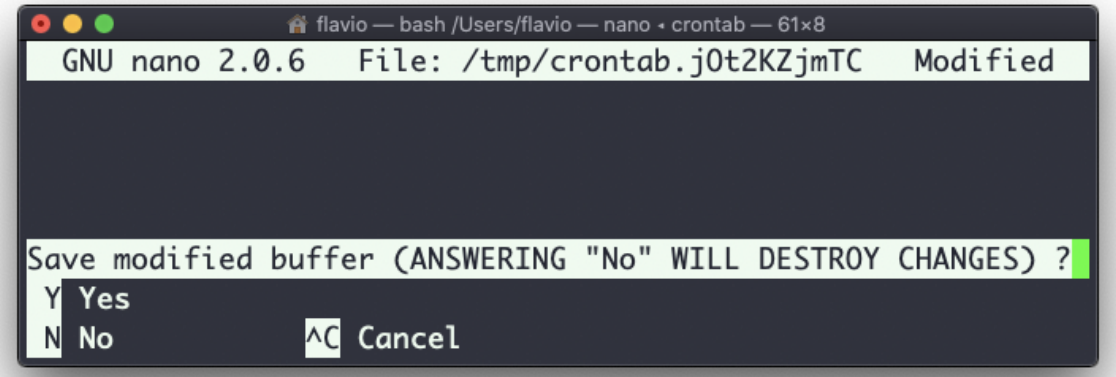
If all goes well, the cron job is set up:

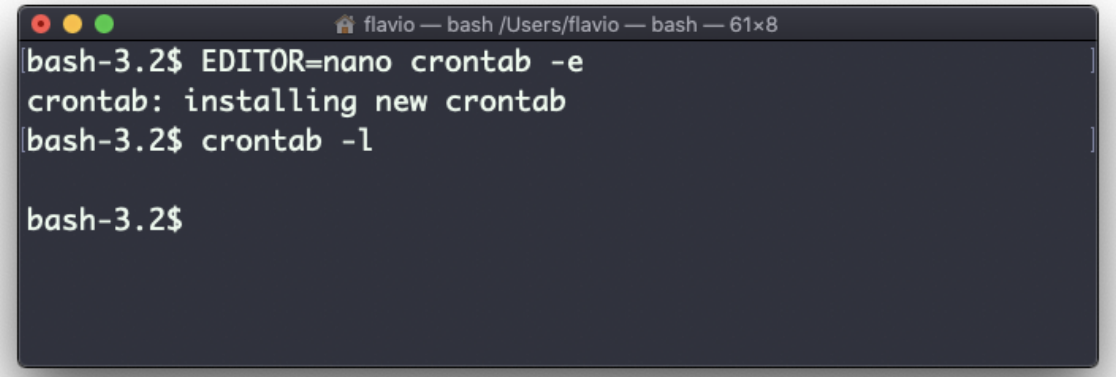


Once this is done, you can see the list of active cron jobs by running:

`crontab -l`

You can remove a cron job running `crontab -e` again, removing the line and exiting the editor:





The `crontab` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Check your understanding: processes and jobs

Check process IDs, foreground and background jobs, signals, monitoring, scheduling, and keeping commands running.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Archives and disk tools

Compare files, inspect disk use, and create or extract common archives.

Linux commands: rmdir

Learn how the Linux rmdir command removes empty folders, one or many at a time, and why you need rm -rf instead to delete a folder that still has files.

`rmdir` removes empty directories:

```
mkdir fruits
rmdir fruits
```

It can remove several empty directories:

```
mkdir fruits cars
rmdir fruits cars
```

If a directory contains a file, another directory, or sometimes an unseen entry, `rmdir` fails instead of deleting the contents. Inspect it:

```
ls -la fruits
```

This refusal is a safety feature. Remove or relocate known contents first. Recursive deletion with `rm -r` is a separate, destructive operation; avoid adding `-f` by habit because it suppresses prompts and some errors.

Before recursive deletion, resolve and print the exact target:

```
target="$PWD/build-output"
printf 'target: %s\n' "$target"
find "$target" -maxdepth 2 -print
```

Only proceed after confirming that the resolved path is narrow and expected. Avoid unquoted variables, globs you have not previewed, and commands run from an uncertain working directory. Command-line deletion often bypasses the desktop trash, so recovery may require a backup.

Some implementations support `rmdir -p a/b/c`, which removes `c` and then empty parent directories. Read the local manual before using options in a portable script.

Try it: create one empty directory and one containing a file. Run `rmdir` on both and explain why one refusal protects data.

Linux commands: open

Learn how the macOS open command opens a file, a folder in Finder, or an application, and why open . is so handy, with xdg-open as the Linux equivalent.

The `open` command lets you open a file using this syntax:

```
open <filename>
```

The file opens in the default app registered for that type, exactly as if you had double-clicked it in the Finder: `open notes.txt` launches your text editor, `open photo.png` your image viewer.

That is the point of this command. It bridges the terminal and the graphical side of macOS: you're already in the shell, already in the right folder, and you don't have to reach for the mouse to see something.

You can also open a directory, which on macOS opens the Finder app with the current directory open:

```
open <directory name>
```

I use it all the time to open the current directory:

```
open .
```

> The special `.` symbol points to the current directory, as `..` points to the parent directory

URLs work too. This opens your default browser on the site:

```
open https://flaviocopes.com
```

The same command can also be used to run an application. Pass the `-a` option followed by the application name:

```
open -a Calculator
```

`-a` is also how you open a file with a specific app instead of the default one:

```
open -a "Google Chrome" index.html
```

Note the quotes: an application name containing spaces must be quoted, or the shell splits it into separate arguments.

When something refuses to open, read the error message. `open missing.txt` fails with `The file`

`/Users/flavio/missing.txt does not exist.` — nearly always a typo in the name or the wrong current directory. Run `ls` to check what's actually there, then try again.

> The `open` command works on macOS only. Use `xdg-open` on Linux

`xdg-open` covers the same jobs on Linux: files, folders, and URLs, each handed to the desktop's default application.

Linux commands: tail

Learn how the Linux tail command shows the end of a file, prints the last lines with -n, and follows a log live as it grows with the tail -f option.

`tail` prints the final part of a file. Called with just a filename, it shows the last 10 lines. That alone is useful — the end of a log file is where the newest information lives.

But the best use case of tail in my opinion is when called with the `-f` option. It opens the file at the end, and watches for file changes. Any time there is new content in the file, it is printed in the window. This is great for watching log files, for example:

```
tail -f /var/log/system.log
```

To exit, press `ctrl-C`.

While it runs, you can pipe it into other tools. This watches a web server log and shows only the server errors, as they happen:

```
tail -f /var/log/nginx/access.log | grep " 500 "
```

One thing to know about `-f`: it keeps following the file it originally opened. Log rotation replaces that file with a fresh empty one, and your `tail -f` goes silent, still attached to the old deleted file. When that's a risk, use `-F` instead: it notices the swap and reopens the file by name.

You can print the last 10 lines in a file:

```
tail -n 10 <filename>
```

Change the number to see more or fewer lines. Here is a quick way to verify the behavior with a file you build on the spot:

```
seq 1 100 > numbers.txt
tail -n 3 numbers.txt
# 98
# 99
# 100
```

You can print the whole file content starting from a specific line using `+` before the line number:

```
tail -n +10 <filename>
```

Watch the meaning flip: `-n 10` means ”the last 10 lines”, while `-n +10` means ”everything from line 10 to the end”. Mixing the two up is a classic source of confusion.

The natural companion is `head`, which prints the beginning of a file instead of the end.

`tail` can do much more and as always my advice is to check `man tail`.

> This command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: diff

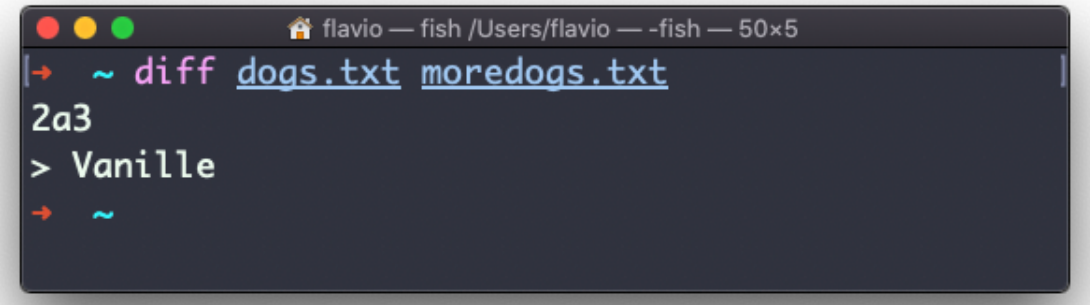
Learn how the Linux diff command compares two files or directories line by line, with -y for a side-by-side view and -u for the unified format Git uses.

`diff` is a handy command. Suppose you have 2 files, which contain almost the same information, but you can't find the difference between the two.

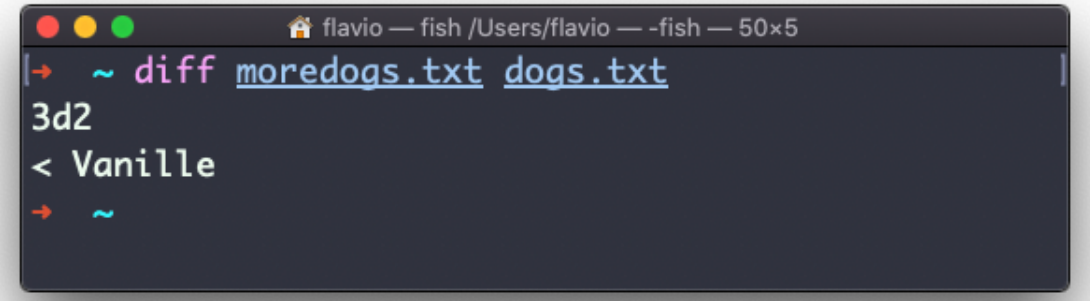
`diff` will process the files and will tell you what's the difference.

Suppose you have 2 files: `dogs.txt` and `moredogs.txt`. The difference is that `moredogs.txt` contains one more dog name:

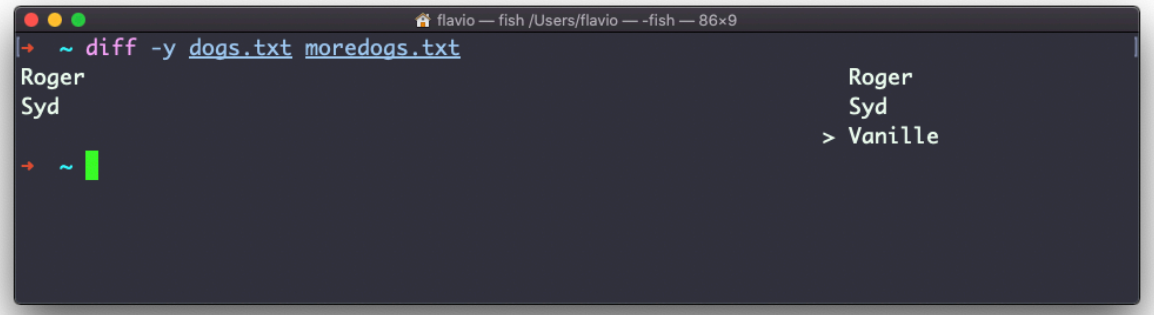
`diff dogs.txt moredogs.txt` will tell you the second file has one more line, line 3 with the line `Vanille`:



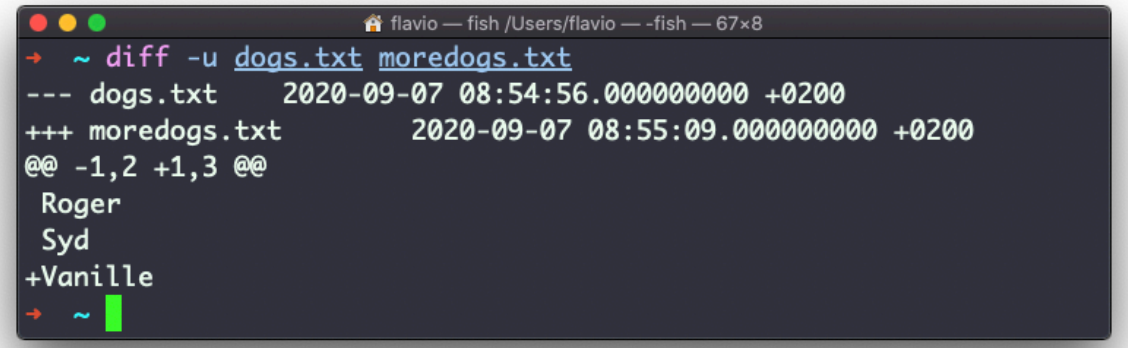
If you invert the order of the files, it will tell you that the second file is missing line 3, whose content is `Vanille`:



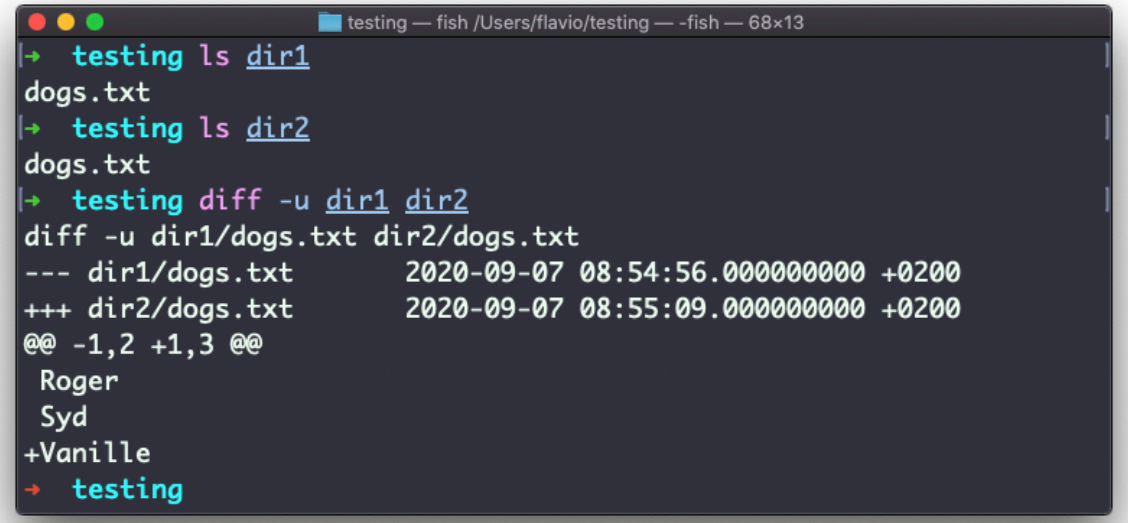
Using the `-y` option will compare the 2 files line by line:



The `-u` option however will be more familiar to you, because that's the same used by the Git version control system to display differences between versions:

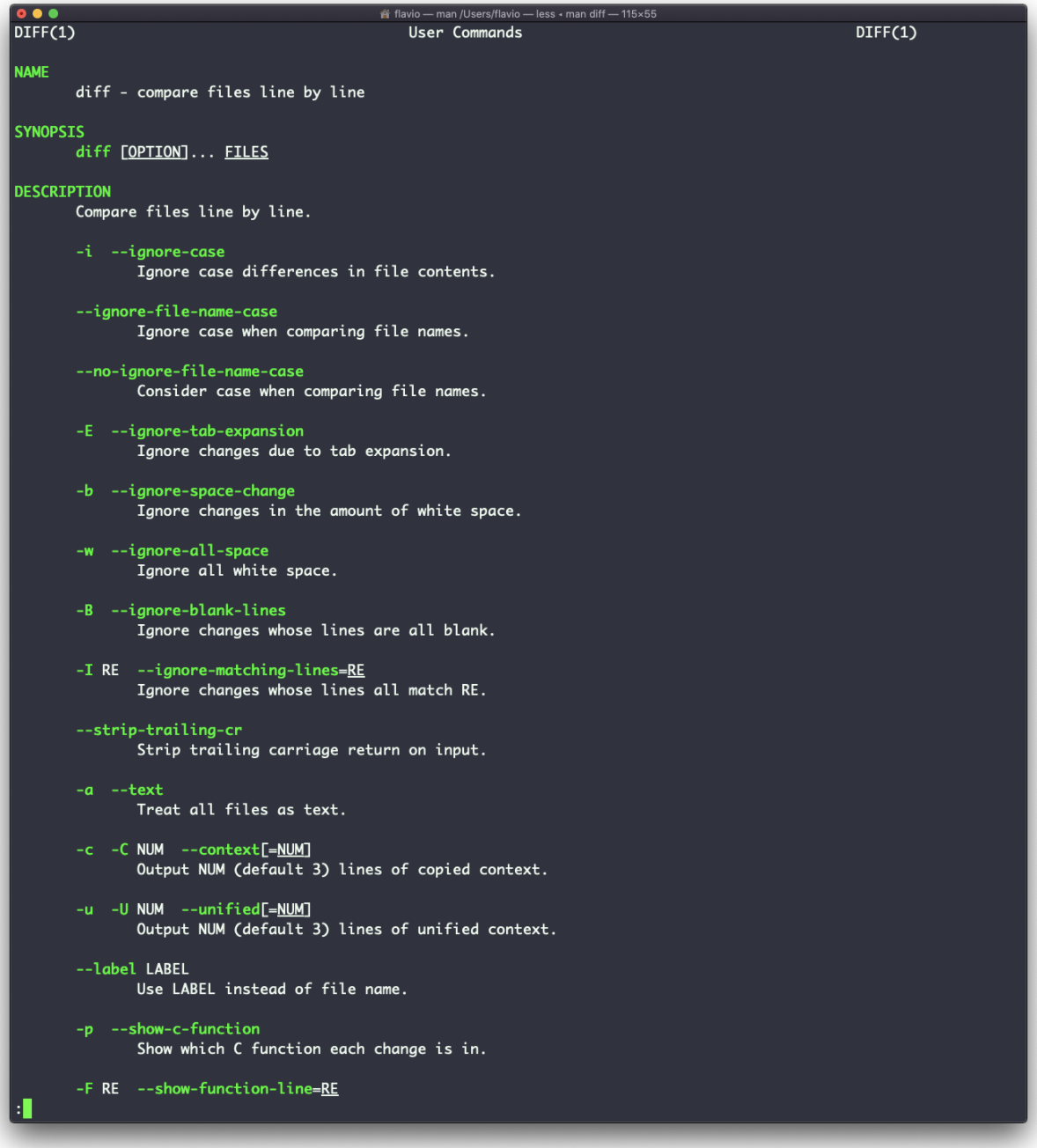


Comparing directories works in the same way. You must use the `-r` option to compare recursively (going into subdirectories):



In case you're interested in which files differ, rather than the content, use the `r` and `q` options:

There are many more options you can explore in the man page running `man diff`:



The `diff` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

If you'd rather compare two texts in the browser, try my free diff checker.

Linux commands: gzip

Learn how the Linux gzip command compresses a file into a .gz archive, how to keep the original with -k, set a level from 1 to 9, and decompress with -d.

You can compress a file using the gzip compression protocol named LZ77 using the `gzip` command.

Here's the simplest usage:

```
gzip filename
```

This will compress the file, and append a `.gz` extension to it. The original file is deleted. To prevent

this, you can use the `-c` option and use output redirection to write the output to the `filename.gz` file:

```
gzip -c filename > filename.gz
```

> The `-c` option specifies that output will go to the standard output stream, leaving the original file intact

Or you can use the `-k` option:

```
gzip -k filename
```

There are various levels of compression. The more the compression, the longer it will take to compress (and decompress). Levels range from 1 (fastest, worst compression) to 9 (slowest, better compression), and the default is 6.

You can choose a specific level with the `-<NUMBER>` option:

```
gzip -1 filename
```

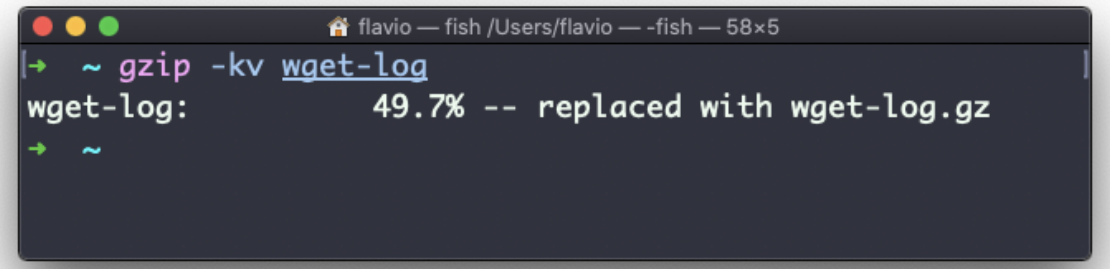
You can compress multiple files by listing them:

```
gzip filename1 filename2
```

You can compress all the files in a directory, recursively, using the `-r` option:

```
gzip -r a_folder
```

The `-v` option prints the compression percentage information. Here's an example of it being used along with the `-k` (keep) option:



`gzip` can also be used to decompress a file, using the `-d` option:

```
gzip -d filename.gz
```

The `gzip` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: gunzip

Learn how the Linux gunzip command decompresses gzipped .gz files, removing the extension, or extracts to a new filename using gunzip -c with redirection.

The `gunzip` command decompresses files that were compressed in the gzip format, the ones ending in `.gz`. It is the counterpart of the `gzip` command: the two are the same program wearing different names, except in `gunzip` the `-d` (decompress) option is always enabled by default.

The command can be invoked in this way:

```
gunzip filename.gz
```

This will gunzip and will remove the `.gz` extension, putting the result in the `filename` file. If that file exists, it will overwrite that.

Notice the compressed file is gone afterwards. Check with `ls`:

```
ls
# filename
```

If you want to keep the `.gz` file around too, add the `-k` option:

```
gunzip -k filename.gz
# now you have both filename and filename.gz
```

You can extract to a different filename using output redirection using the `-c` option:

```
gunzip -c filename.gz > anotherfilename
```

`-c` writes the decompressed data to standard output instead of to a file, and the `>` redirection saves it wherever you want. The original `.gz` file stays untouched in this case too.

Before extracting, you can check what you will get with `-l`, which lists the compressed and uncompressed sizes:

```
gunzip -l filename.gz
```

Two failures you will run into sooner or later. First, running `gunzip filename` on a file without a recognized extension stops with `unknown suffix -- ignored`: gunzip only processes files it knows it created.

Second, gzip compresses a single file, never a folder. When you see `project.tar.gz`, that is a tar archive that was gzipped afterwards. Running `gunzip project.tar.gz` gives you `project.tar`, and you unpack that with the `tar` command — or let `tar -xzf project.tar.gz` do both steps at once.

The `gunzip` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: tar

Learn how the Linux tar command groups files into an archive with tar -cf, extracts them with tar -xf, and makes a gzip-compressed .tar.gz with the z flag.

The `tar` command is used to create an archive, grouping multiple files in a single file.

Its name comes from the past and means *tape archive*. Back when archives were stored on tapes.

This command creates an archive named `archive.tar` with the content of `file1` and `file2`:

```
tar -cf archive.tar file1 file2
```

The `c` option stands for *create*. The `f` option is used to write to file the archive.

To extract files from an archive in the current folder, use:

```
tar -xf archive.tar
```

the `x` option stands for *extract*

and to extract them to a specific directory, use:

```
tar -xf archive.tar -C directory
```

You can also just list the files contained in an archive:

`tar` is often used to create a **compressed archive**, gzipping the archive.

This is done using the `z` option:

```
tar -czf archive.tar.gz file1 file2
```

This is just like creating a tar archive, and then running `gzip` on it.

To unarchive a gzipped archive, you can use `gunzip`, or `gzip -d`, and then unarchive it, but `tar -xf` will recognize it's a gzipped archive, and do it for you:

```
tar -xf archive.tar.gz
```

The `tar` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

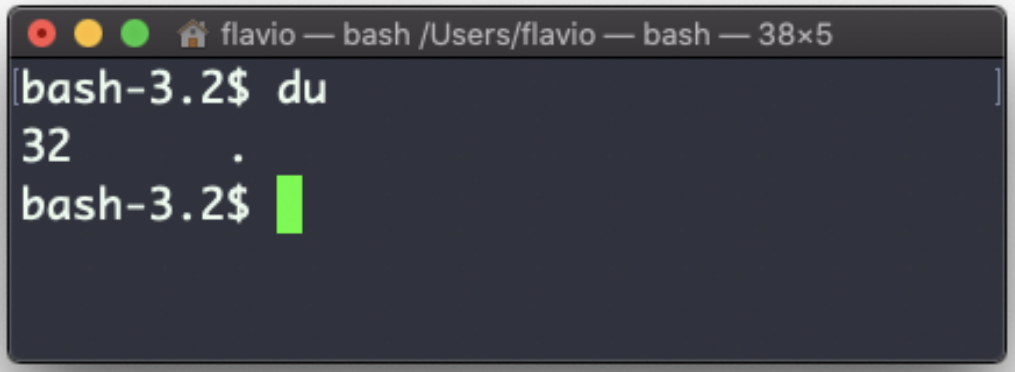
If you can never remember the right flags, I made a tar command builder that composes the command for you.

Linux commands: du

Learn how the Linux du command shows the disk space used by files and directories, with du -h for human-readable sizes and sorting by size using sort -nr.

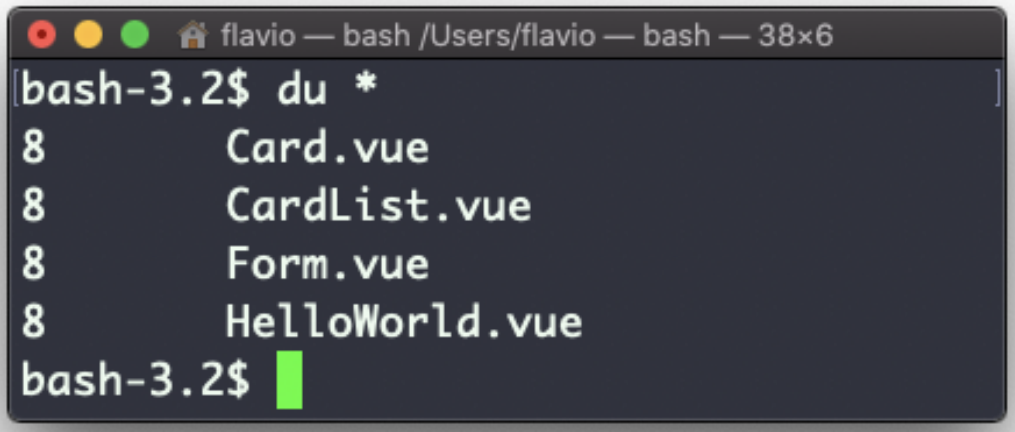
The `du` command will calculate the size of a directory as a whole:

```
du
```



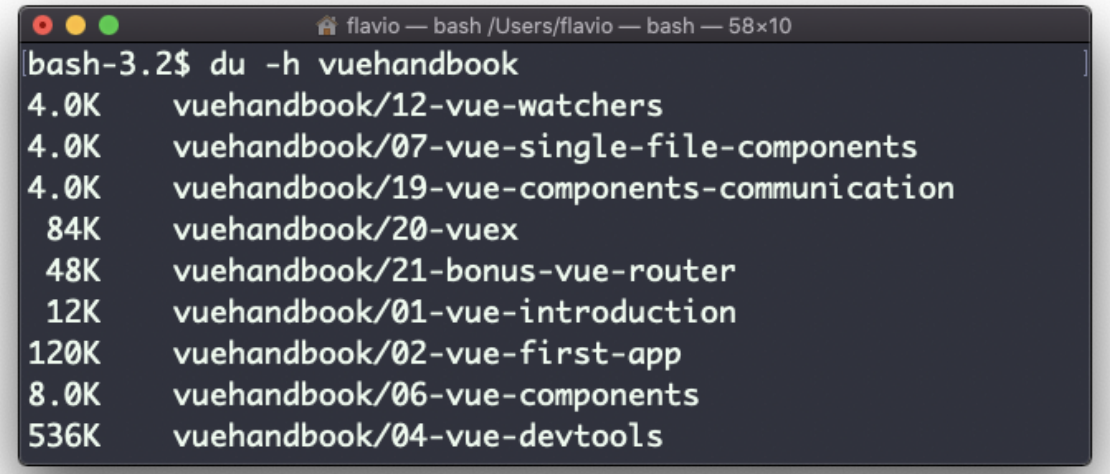
The `32` number here is a value expressed in bytes.

Running `du *` will calculate the size of each file individually:

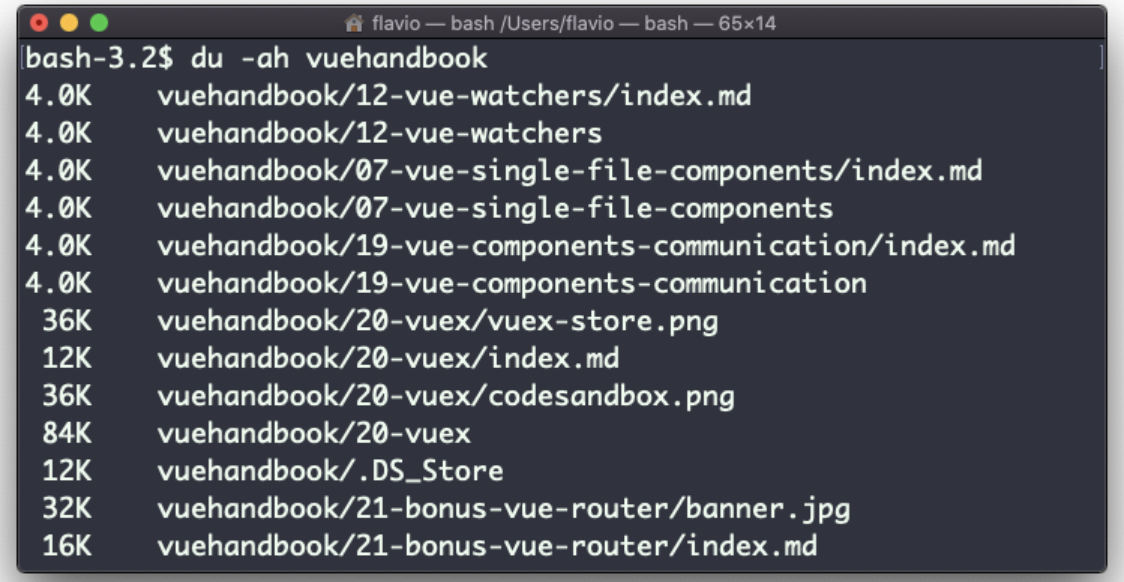


You can set `du` to display values in MegaBytes using `du -m`, and GigaBytes using `du -g`.

The `-h` option will show a human-readable notation for sizes, adapting to the size:



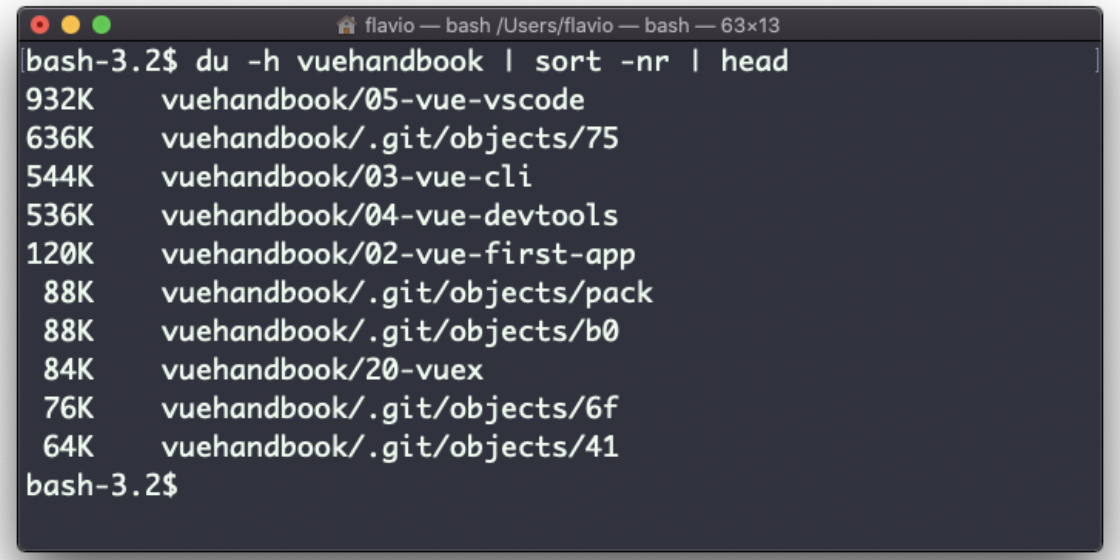
Adding the `-a` option will print the size of each file in the directories, too:



A handy thing is to sort the directories by size:

`du -h <directory> | sort -nr`

and then piping to `head` to only get the first 10 results:



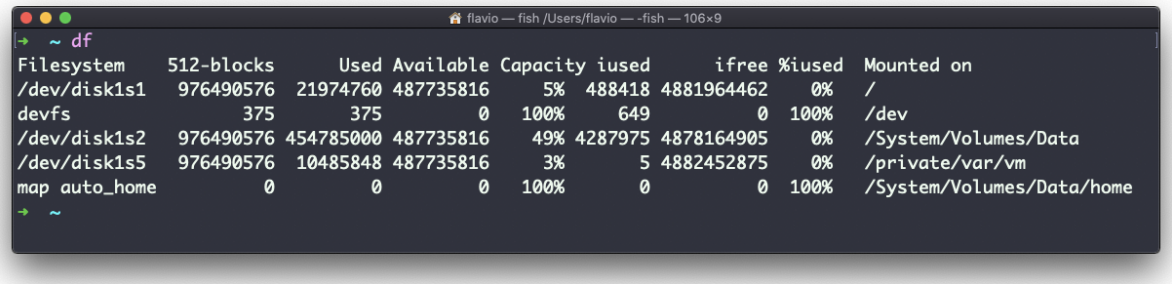
The `du` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: df

Learn how the Linux df command reports disk usage for your mounted volumes, with df -h for human-readable sizes and a path to check the volume it lives on.

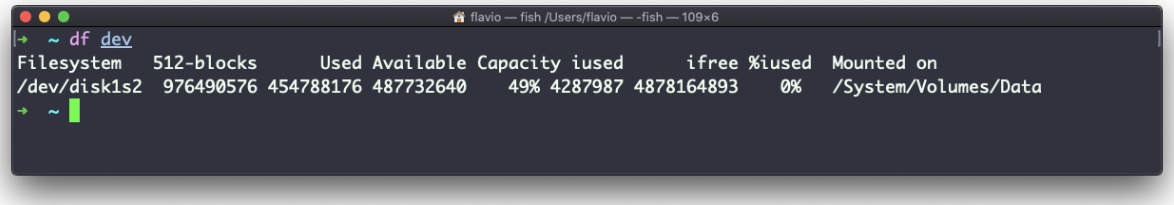
The `df` command is used to get disk usage information.

Its basic form will print information about the volumes mounted:



Using the `-h` option (`df -h`) will show those values in a human-readable format:

You can also specify a file or directory name to get information about the specific volume it lives on:



The `df` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: basename

Learn how the Linux basename command returns the last portion of a path, so basename /Users/flavio/test.txt gives you back just the test.txt filename.

Suppose you have a path to a file, for example `/Users/flavio/test.txt`.

Running

```
basename /Users/flavio/test.txt
```

will return the `test.txt` string:

`basename` strips the directory part from a path and leaves the last portion. It exists for scripts:

when a loop or a variable hands you a full path, and you only need the file name for a message, a log line, or a new destination.

A second argument removes a suffix, too:

```
basename /Users/flavio/test.txt .txt
# test
```

That's handy when renaming or converting files. Here is a realistic loop that copies every `.txt` note to a `.md` file with the same name:

```
for file in /Users/flavio/notes/*.txt; do
  name=$(basename "$file" .txt)
  cp "$file" "$name.md"
done
```

The `$(...)` command substitution captures the output of `basename` into the `name` variable.

If you run `basename` on a path string that points to a directory, you will get the last segment of the path. In this example, `/Users/flavio` is a directory:

A trailing slash makes no difference: `basename /Users/flavio/` also prints `flavio`.

One thing to keep in mind: `basename` works on the string alone. It never touches the filesystem, so it happily processes a path that doesn't exist. Getting output is not a confirmation the file is there — check with `ls` when that matters.

The companion command `dirname` gives you the other half, the directory portion of the path. The two together let you take any path apart.

The `basename` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: dirname

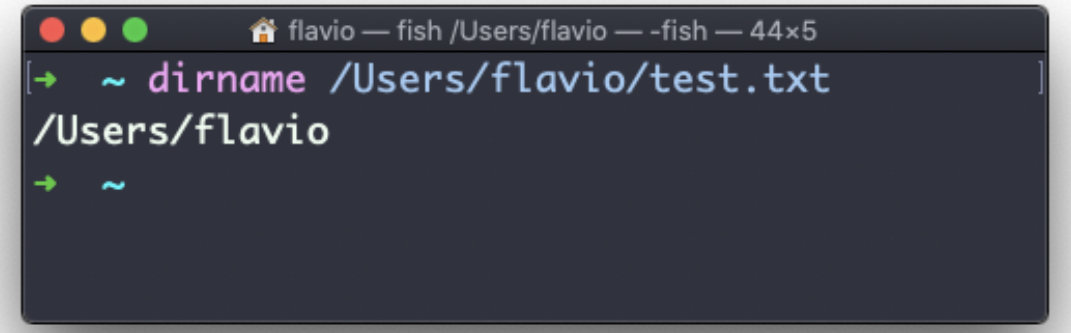
Learn how the Linux dirname command returns the directory portion of a path, so dirname /Users/flavio/test.txt gives you back the /Users/flavio folder.

`dirname` removes the final path component and prints the directory portion:

Running

```
dirname /Users/flavio/test.txt
```

prints `/Users/flavio`:



It manipulates the path string. It does not check whether the file or directory exists:

```
dirname /a/path/that/does/not/exist.txt
# /a/path/that/does/not
```

This makes it useful in scripts. For example, find the directory containing a script:

```
script_dir=$(dirname "$0")
printf 'started from %s\n' "$script_dir"
```

Quote path variables so spaces do not split one path into several arguments. Also remember that `$0` may be relative and may point through a symbolic link; robust script-location code may need to resolve those details.

`dirname file.txt` prints `.`, meaning the parent is the current directory. A trailing slash is normalized, so `dirname /srv/app/` returns `/srv` rather than `/srv/app`.

The companion command `basename /Users/flavio/test.txt` returns `test.txt`. Together they separate a path into its directory and final name.

You can apply `dirname` more than once when you need to walk up a path:

```
dirname "$(dirname /Users/flavio/projects/blog)"
# /Users/flavio
```

Try it with an absolute path, a relative path, a filename with spaces, and a path ending in `/`. Predict each result before running it.

Check your understanding: archives and disk tools

Check empty-directory removal, file comparison, archives, compression, disk usage, and path components.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Shell, system, and network tools

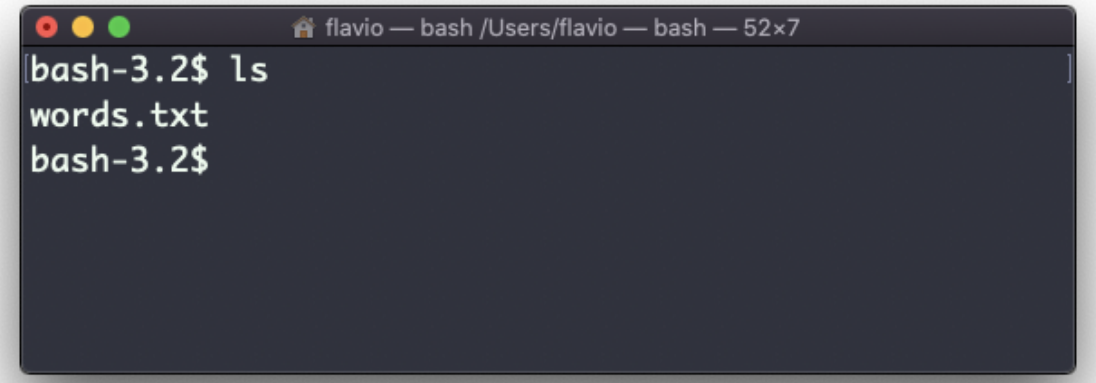
Customize the shell, edit text, identify the system, and test network paths.

Linux commands: alias

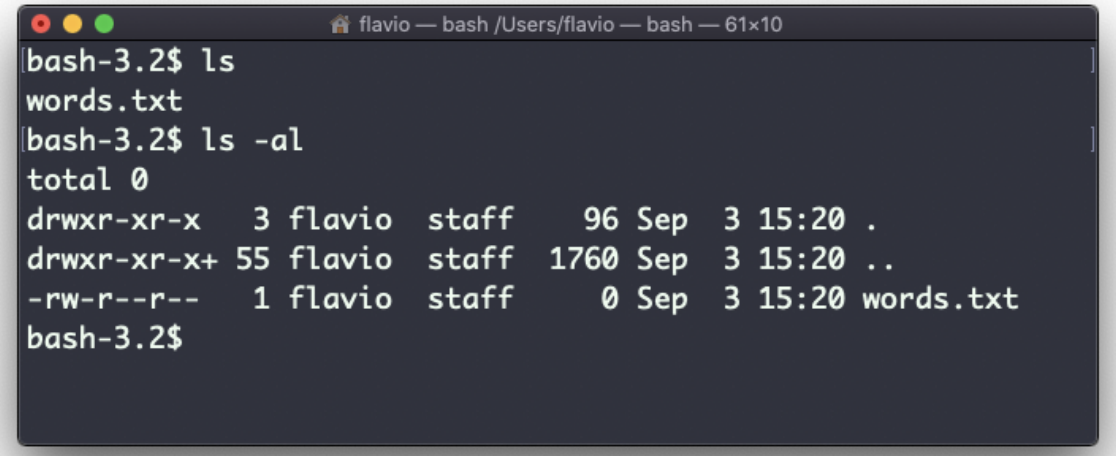
Learn how the Linux alias command creates a shortcut for another command, like alias ll for ls -al, and how to make your aliases permanent in .bashrc.

It's common to always run a program with a set of options you like using.

For example, take the `ls` command. By default it prints very little information:



while using the `-al` option it will print something more useful, including the file modification date, the size, the owner, and the permissions, also listing hidden files (files starting with a `.`:

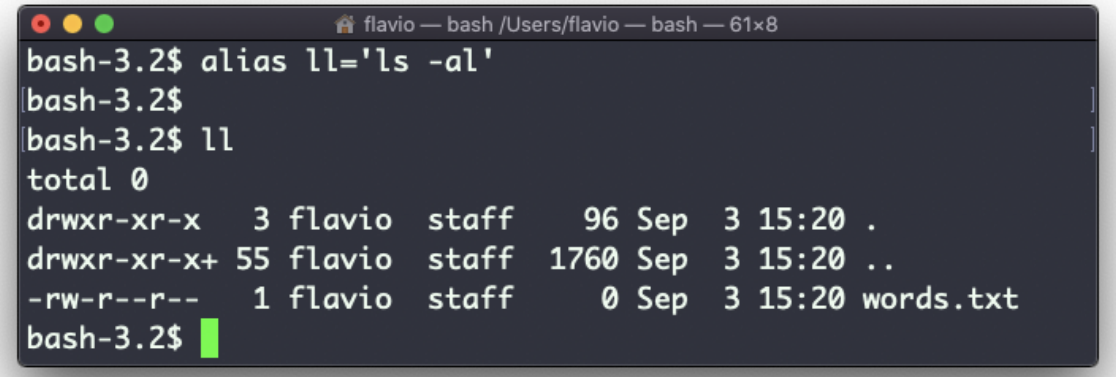


You can create a new command, for example I like to call it `ll`, that is an alias to `ls -al`.

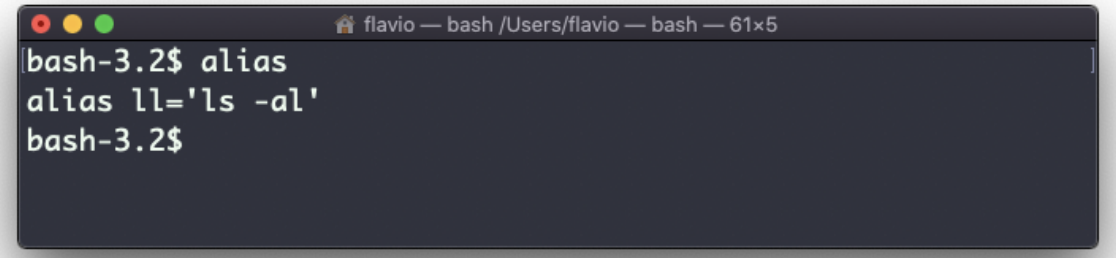
You do it in this way:

```
alias ll='ls -al'
```

Once you do, you can call `ll` just like it was a regular UNIX command:



Now calling `alias` without any option will list the aliases defined:



The alias will work until the terminal session is closed.

To make it permanent, you need to add it to the shell configuration, which could be `~/.bashrc` or `~/.profile` or `~/.bash_profile` if you use the Bash shell, depending on the use case.

Be careful with quotes if you have variables in the command: using double quotes the variable is resolved at definition time, using single quotes it's resolved at invocation time. Those 2 are different:

```
alias lsthis="ls $PWD"
alias lscurrent='ls $PWD'
```

$PWD refers to the current folder the shell is into. If you now navigate away to a new folder, `lscurrent` lists the files in the new folder, `lsthis` still lists the files in the folder you were when you defined the alias.

> The `alias` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: type

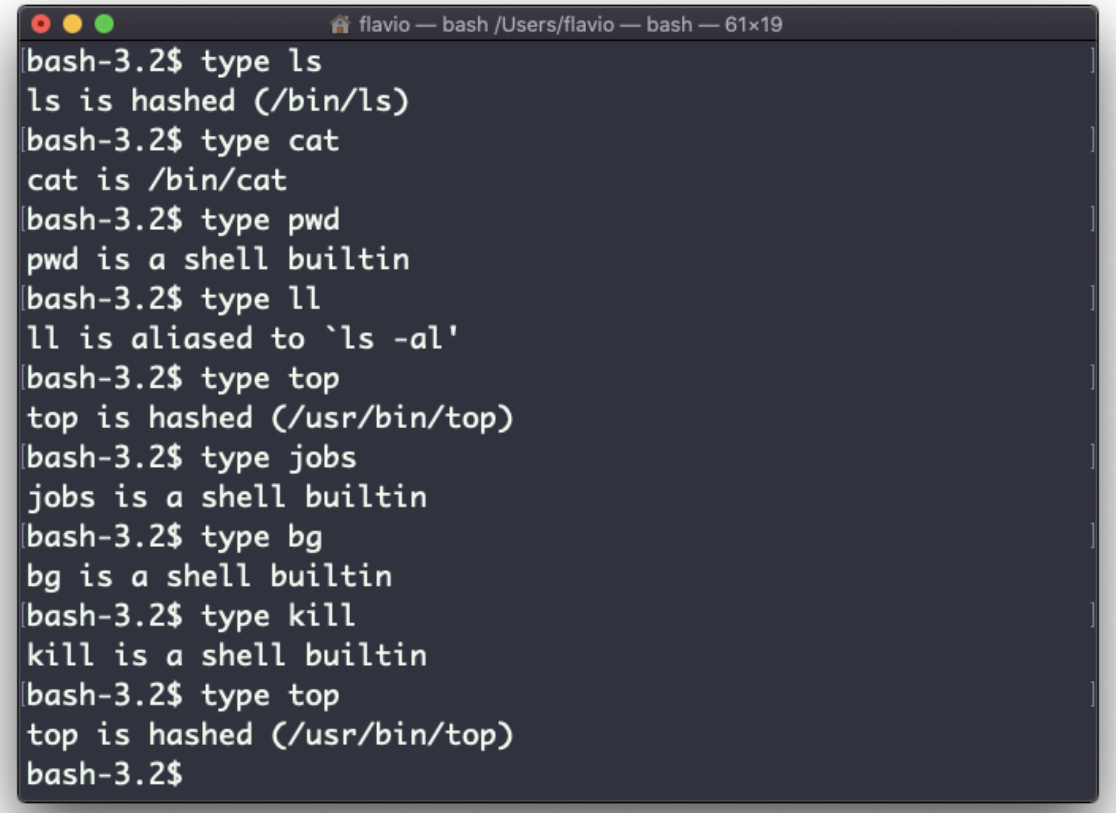
Learn how the Linux type command tells you how a command is interpreted, as an executable, a shell built-in, a function, or an alias, and what it points to.

A command can be one of those 4 types:

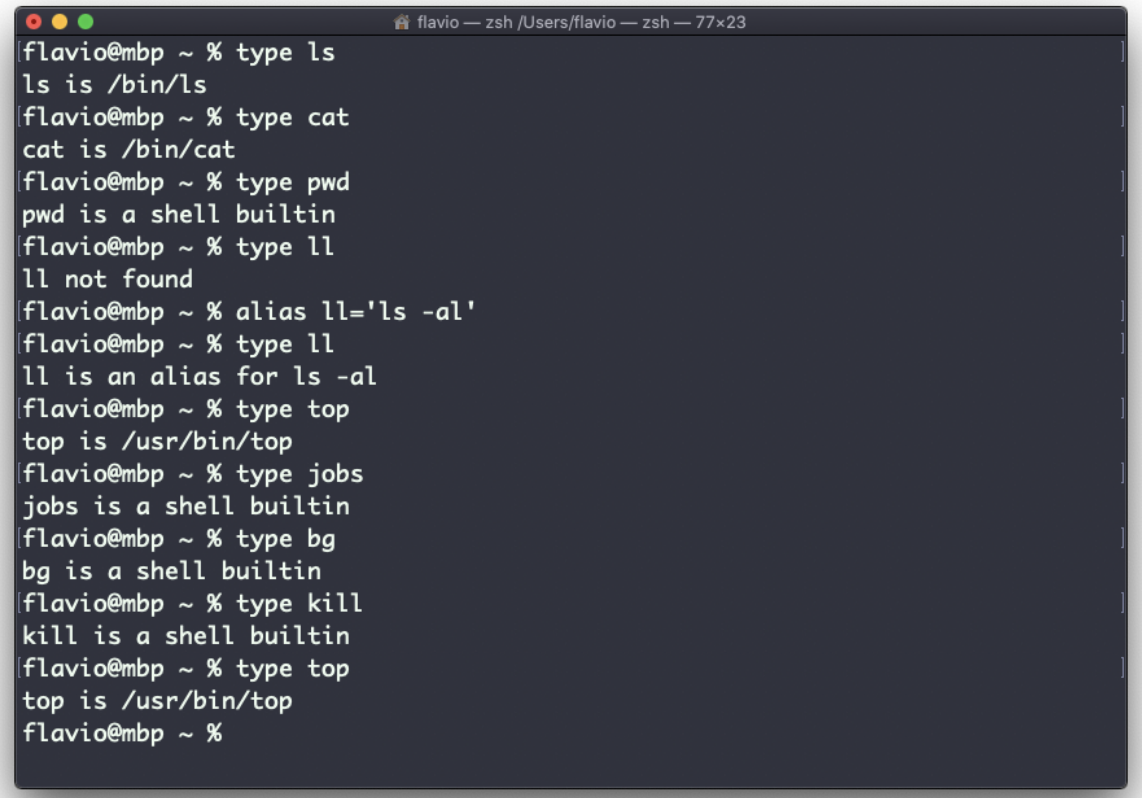
- an executable
- a shell built-in program
- a shell function
- an alias

The `type` command can help figure out this, in case we want to know or we're just curious. It will tell you how the command will be interpreted.

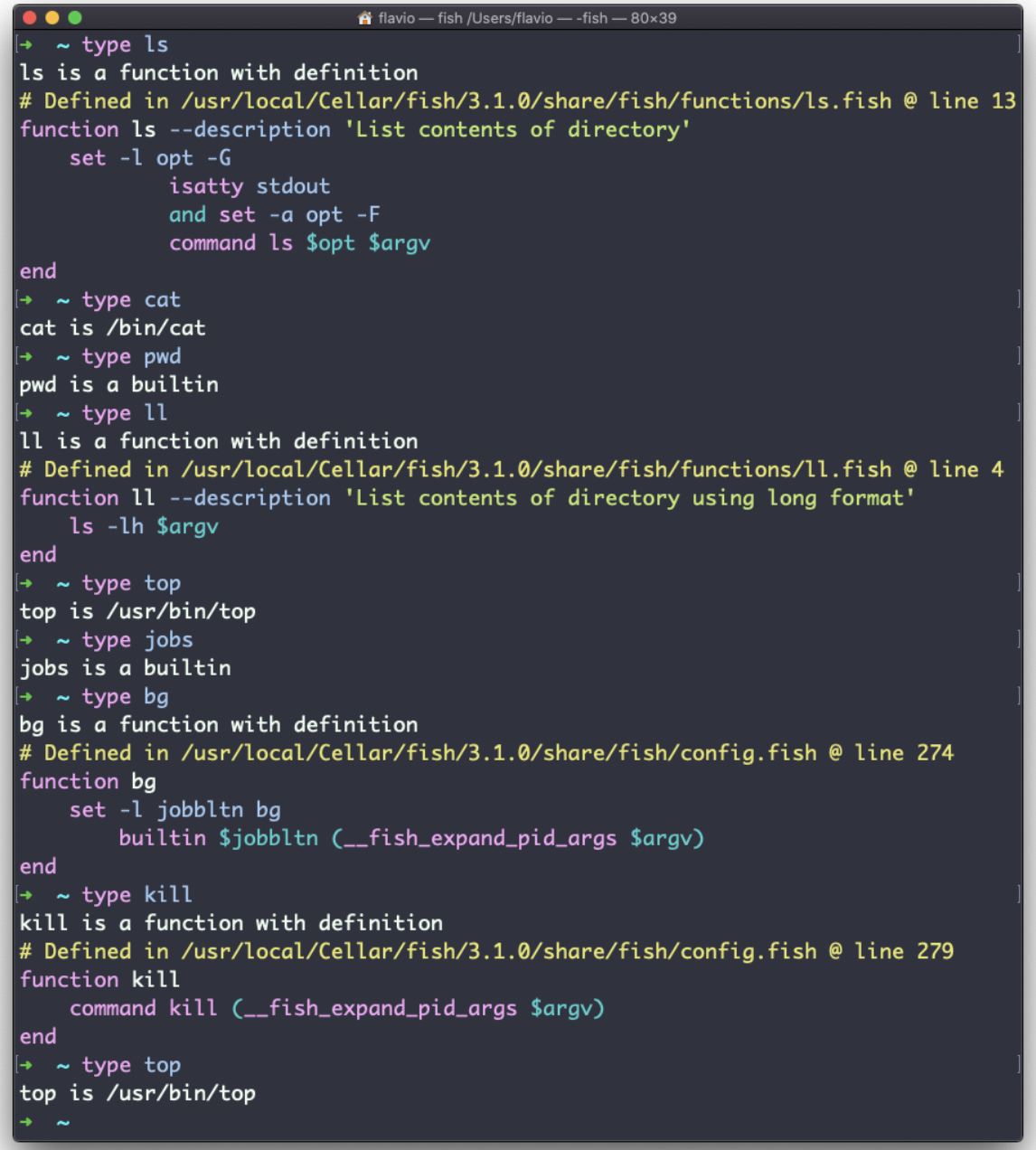
The output will depend on the shell used. This is Bash:



This is Zsh:


flavio — zsh /Users/flavio — zsh — 77×23
flavio@mbp ~ % type ls
ls is /bin/ls
flavio@mbp ~ % type cat
cat is /bin/cat
flavio@mbp ~ % type pwd
pwd is a shell builtin
flavio@mbp ~ % type ll
ll not found
flavio@mbp ~ % alias ll='ls -al'
flavio@mbp ~ % type ll
ll is an alias for ls -al
flavio@mbp ~ % type top
top is /usr/bin/top
flavio@mbp ~ % type jobs
jobs is a shell builtin
flavio@mbp ~ % type bg
bg is a shell builtin
flavio@mbp ~ % type kill
kill is a shell builtin
flavio@mbp ~ % type top
top is /usr/bin/top
flavio@mbp ~ %

This is Fish:



One of the most interesting things here is that for aliases it will tell you what is aliasing to. You can see the `ll` alias, in the case of Bash and Zsh, but Fish provides it by default, so it will tell you it's a built-in shell function.

> The `type` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: clear

Learn how the Linux clear command wipes your terminal screen, the handy ctrl-L shortcut for it, and how clear -x clears but keeps your scrollback.

Type `clear` to clear all the previous commands that were ran in the current terminal.

The screen will clear and you will just see the prompt at the top:

It's a small tool with one honest job: getting visual clutter out of the way. Before starting a new task, or before taking a screenshot, a clean screen makes it much easier to tell fresh output from old output.

> Note: this command has a handy shortcut: `ctrl-L`

The shortcut is provided by the shell, and it's faster than reaching for the command. My advice is to make it a habit.

Once you do that, you will lose access to scrolling to see the output of the previous commands entered.

So you might want to use `clear -x` instead, which still clears the screen, but lets you go back to see the previous work by scrolling up:

```
clear -x
```

To verify the difference, run a few commands, then `clear -x`, then scroll up: the old output is still there. After a plain `clear`, on many terminals, it's gone.

Under the hood there is no magic. `clear` prints escape sequences that your terminal interprets as "wipe the screen". You can make them visible by piping the output through `cat -v`:

```
clear | cat -v
# ^[[3J^[[H^[[2J
```

Those `^[[` codes are instructions for the terminal, not text.

One related failure is worth knowing. If you `cat` a binary file by accident, the terminal can end up garbled, printing strange symbols instead of letters. `clear` will not fix that, because the terminal's own state is broken, not just the screen content. Type `reset` and press enter — even if you can't read what you're typing — and the terminal reinitializes itself.

> The `clear` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: env

Learn how the Linux env command runs a command with extra environment variables, prints all of them when run alone, and clears the environment with -i.

The `env` command can be used to pass environment variables without setting them on the outer environment (the current shell).

Suppose you want to run a Node.js app and set the `USER` variable to it.

You can run

```
env USER=flavio node app.js
```

and the `USER` environment variable will be accessible from the Node.js app via the Node `process.env` interface.

You can also run the command clearing all the environment variables already set, using the `-i` option:

```
env -i node app.js
```

In this case you will get an error saying `env: node: No such file or directory` because the `node` command is not reachable, as the `PATH` variable used by the shell to look up commands in the common paths is unset.

So you need to pass the full path to the `node` program:

```
env -i /usr/local/bin/node app.js
```

Try with a simple `app.js` file with this content:

```
console.log(process.env.NAME)
console.log(process.env.PATH)
```

You will see the output being

```
undefined
undefined
```

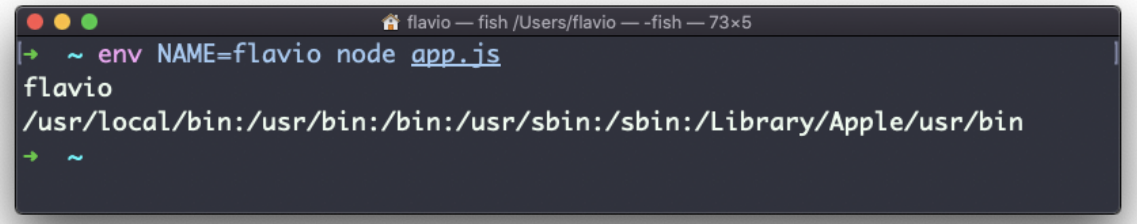
You can pass an env variable:

```
env -i NAME=flavio node app.js
```

and the output will be

```
flavio
undefined
```

Removing the `-i` option will make `PATH` available again inside the program:



The `env` command can also be used to print out all the environment variables, if ran with no options:

```
env
```

it will return a list of the environment variables set, for example:

```
HOME=/Users/flavio
LOGNAME=flavio
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/Apple/usr/bin
PWD=/Users/flavio
SHELL=/usr/local/bin/fish
```

You can also make a variable inaccessible inside the program you run, using the `-u` option, for example this code removes the `HOME` variable from the command environment:

```
env -u HOME node app.js
```

The `env` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: printenv

Learn how the Linux printenv command prints all of your environment variables, or just one like PATH when you pass its name as an argument.

A quick guide to the `printenv` command, used to print the values of environment variables

In any shell there are a good number of environment variables, set either by the system, or by your own shell scripts and configuration. They are how a process receives its settings: where your home folder is, where to look for programs, which editor to launch. Every command you run inherits them.

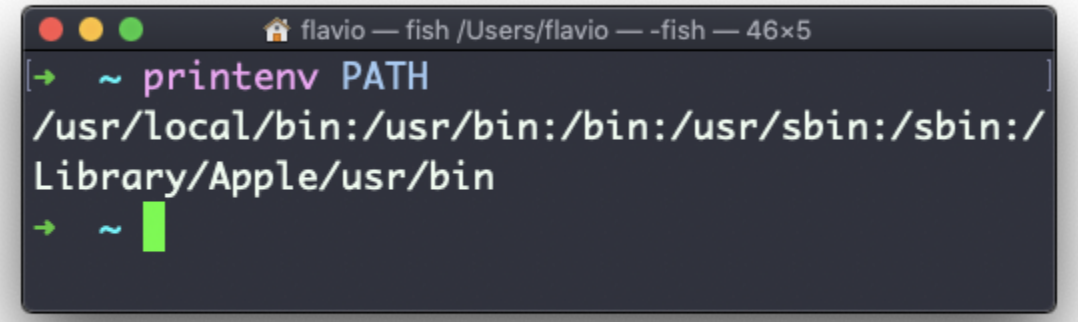
You can print them all to the terminal using the `printenv` command. The output will be something like this:

```
HOME=/Users/flavio
LOGNAME=flavio
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/Apple/usr/bin
PWD=/Users/flavio
SHELL=/usr/local/bin/fish
```

with a few more lines, usually.

You can append a variable name as a parameter, to only show that variable value:

```
printenv PATH
```



If the variable does not exist, `printenv` prints nothing and exits with a non-zero status. You can check that with `echo $?`, which shows the exit status of the last command:

```
printenv NOPE
echo $?
# 1
```

That makes it usable in scripts as a test for whether a variable is set.

Shell variables are not environment variables

Here is the trap that catches everyone once. A variable you assign in the shell is not automatically part of the environment:

```
MYVAR=test
printenv MYVAR
# nothing printed
```

`MYVAR` is a shell variable. Your current shell knows it, and `echo $MYVAR` shows it, but child processes — including `printenv` itself — never receive it. To promote it to an environment variable, use `export`:

```
export MYVAR=test
printenv MYVAR
# test
```

So when a program "can't see" a variable you are sure you set, check with `printenv` first. If it's missing there, the missing piece is the `export`.

The `printenv` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

A short guide to nano

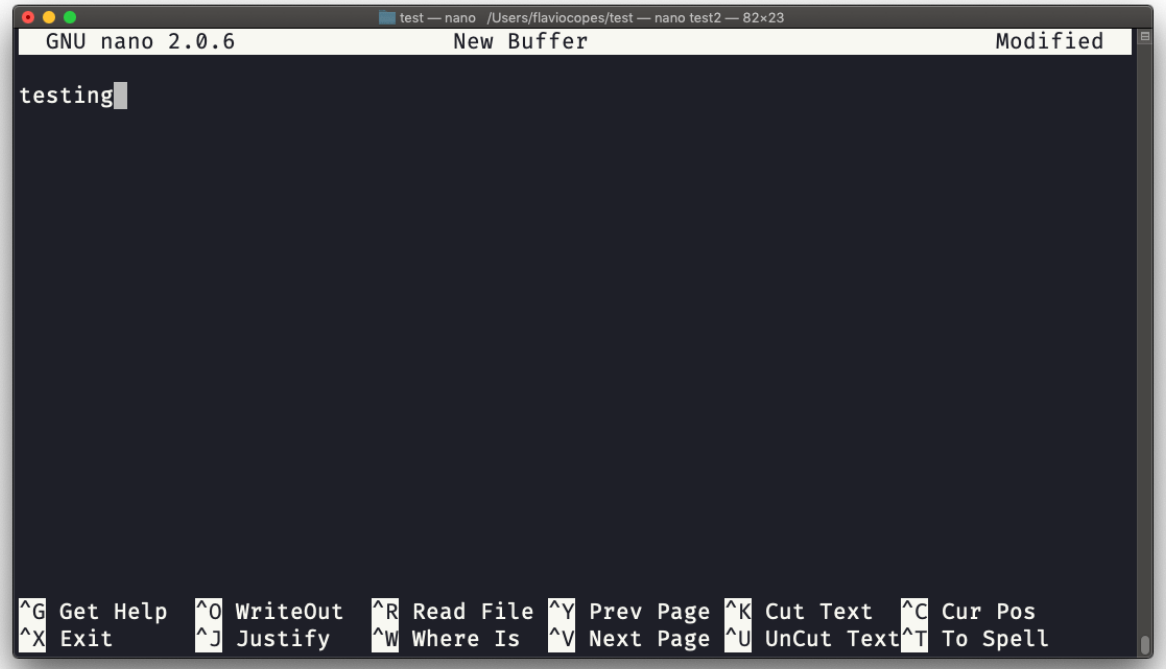
A short guide to nano, the beginner friendly UNIX text editor, where you type directly with no modes and save or quit with the ctrl key shortcuts shown.

`nano` is a beginner friendly editor.

Run it using `nano <filename>`.

You can directly type characters into the file without worrying about modes.

You can quit without editing using `ctrl-X`. If you edited the file buffer, the editor will ask you for confirmation and you can save the edits, or discard them. The help at the bottom shows you the keyboard commands that let you work with the file:



`pico` is more or less the same, although `nano` is the GNU version of `pico` which at some point in history was not open source and the `nano` clone was made to satisfy the GNU operating system license requirements.

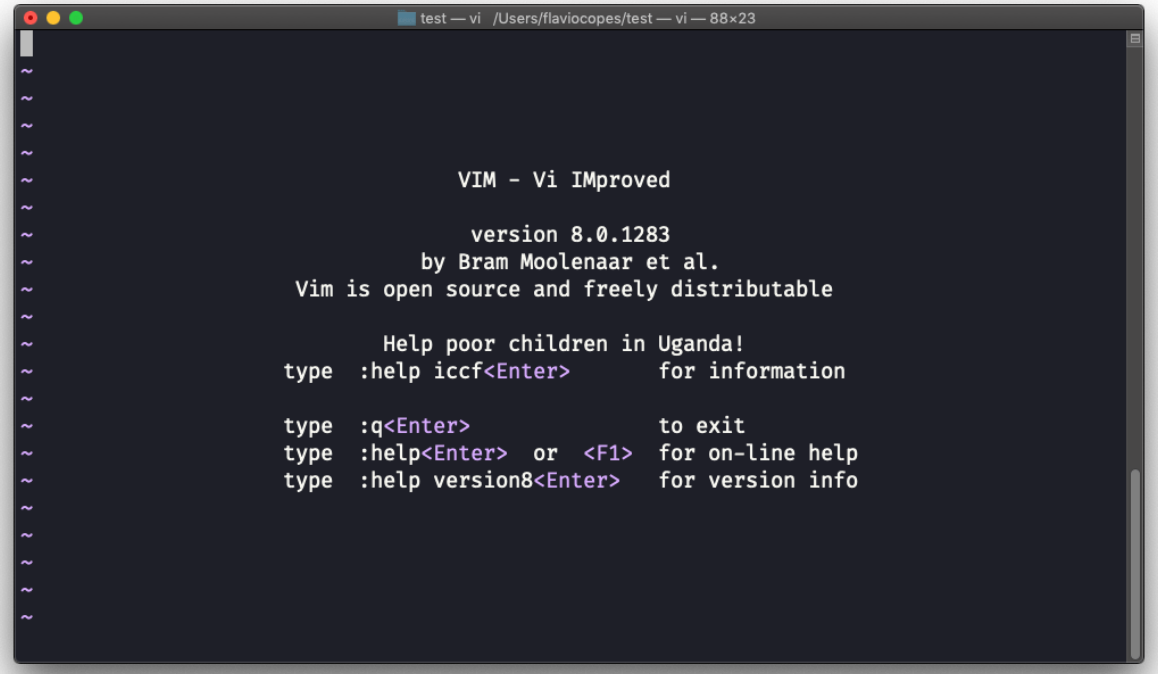
A short guide to vim

A short, beginner-friendly guide to the vim editor: how to open files, switch between command and insert mode, move around, edit text, and save and quit.

`vim` is a **very** popular file editor, especially among programmers. It's actively developed and frequently updated, and there's a very big community around it. There's even a Vim conference!

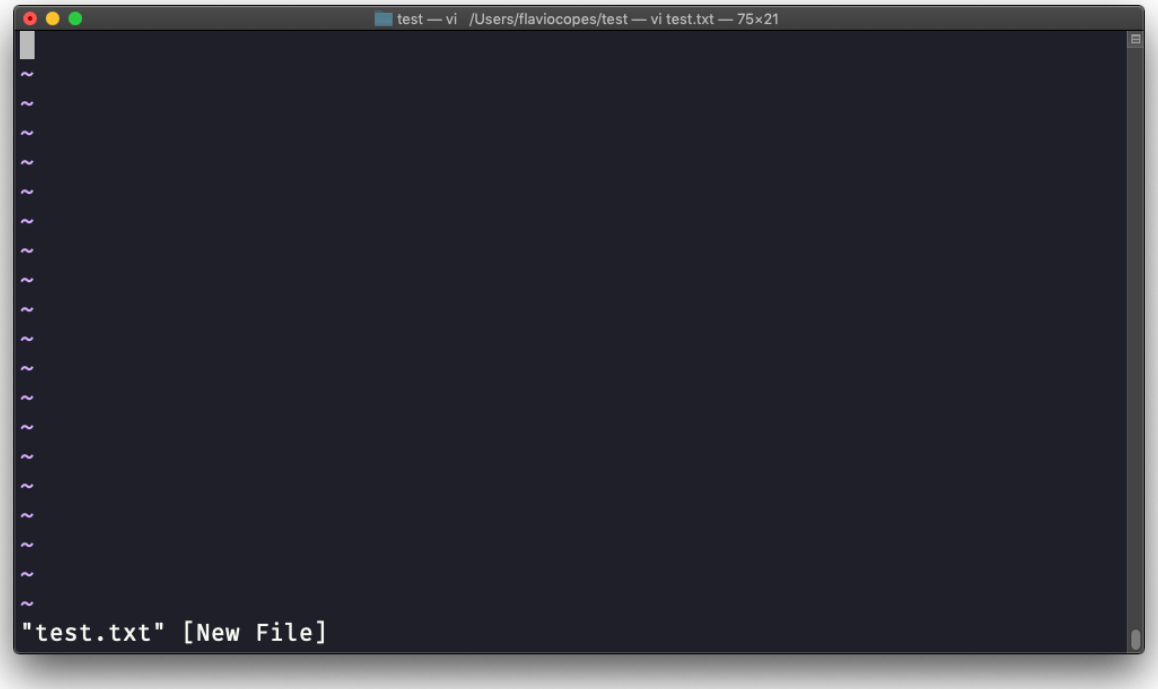
`vi` in modern systems is just an alias to `vim`, which means `vi` improved.

You start it by running `vi` on the command line.



You can specify a filename at invocation time to edit that specific file:

```
vi test.txt
```

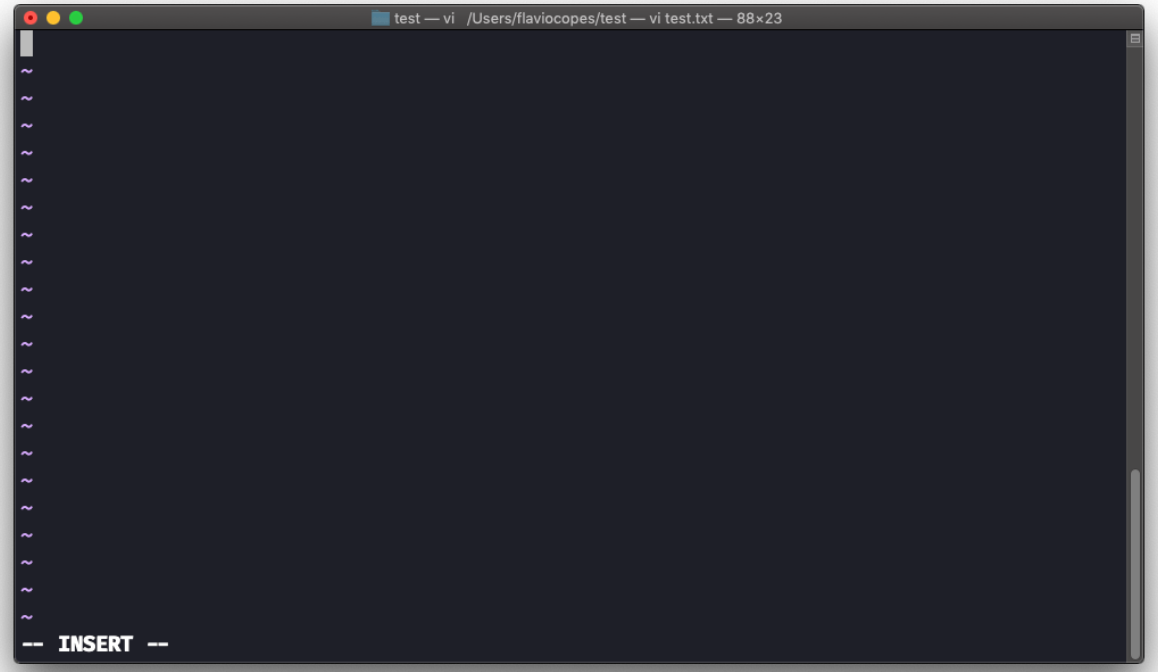


You have to know that Vim has 2 main modes:

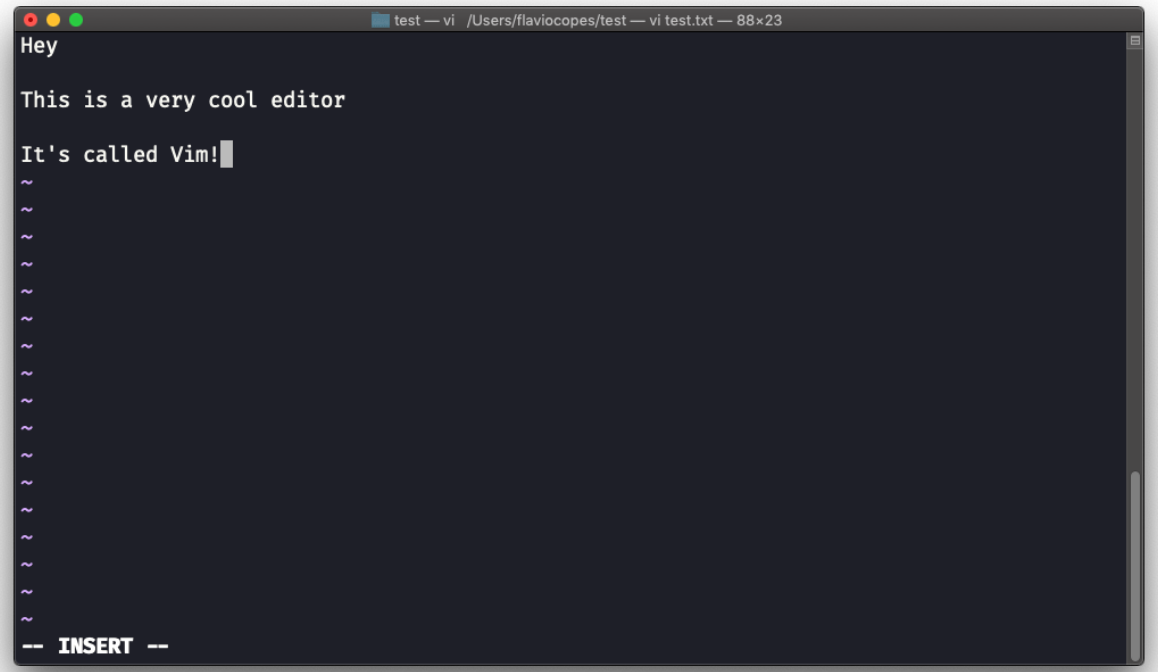
- *command* (or *normal*) mode
- *insert* mode

When you start the editor, you are in command mode. You can't enter text like you expect from a

GUI-based editor. You have to enter **insert mode**. You can do this by pressing the `i` key. Once you do so, the `-- INSERT --` word appear at the bottom of the editor:



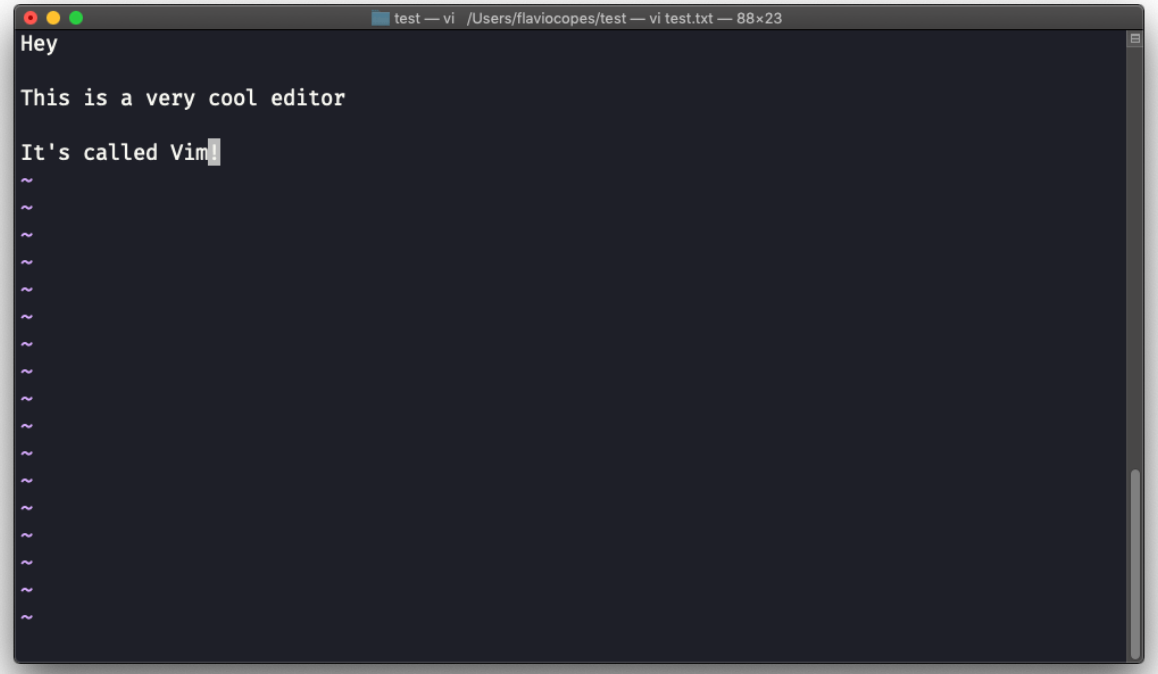
Now you can start typing and filling the screen with the file contents:



You can move around the file with the arrow keys, or using the `h` - `j` - `k` - `l` keys. `h-l` for left-right, `j-k` for down-up.

Once you are done editing you can press the `esc` key to exit insert mode, and go back to **command**

mode.



At this point you can navigate the file, but you can't add content to it (and be careful which keys you press as they might be commands).

One thing you might want to do now is **saving the file**. You can do so by pressing `:` (colon), then `w`.

You can **save and quit** pressing `:` then `w` and `q`: `:wq`

You can **quit without saving**, pressing `:` then `q` and `!`: `:q!`

You can **undo** and edit by going to command mode and pressing `u`. You can **redo** (cancel an undo) by pressing `ctrl-r`.

Those are the basics of working with Vim. From here starts a rabbit hole we can't go into in this little introduction.

I will only mention those commands that will get you started editing with Vim:

- pressing the `x` key deletes the character currently highlighted
- pressing `A` goes at the end of the currently selected line
- press `0` to go to the start of the line
- go to the first character of a word and press `d` followed by `w` to delete that word. If you follow it with `e` instead of `w`, the white space before the next word is preserved
- use a number between `d` and `w` to delete more than 1 word, for example use `d3w` to delete 3 words forward
- press `d` followed by `d` to delete a whole entire line. Press `d` followed by `$` to delete the entire line from where the cursor is, until the end

To find out more about Vim I can recommend the Vim FAQ and especially running the `vimtutor` command, which should already be installed in your system and will greatly help you start your `vim` explorations.

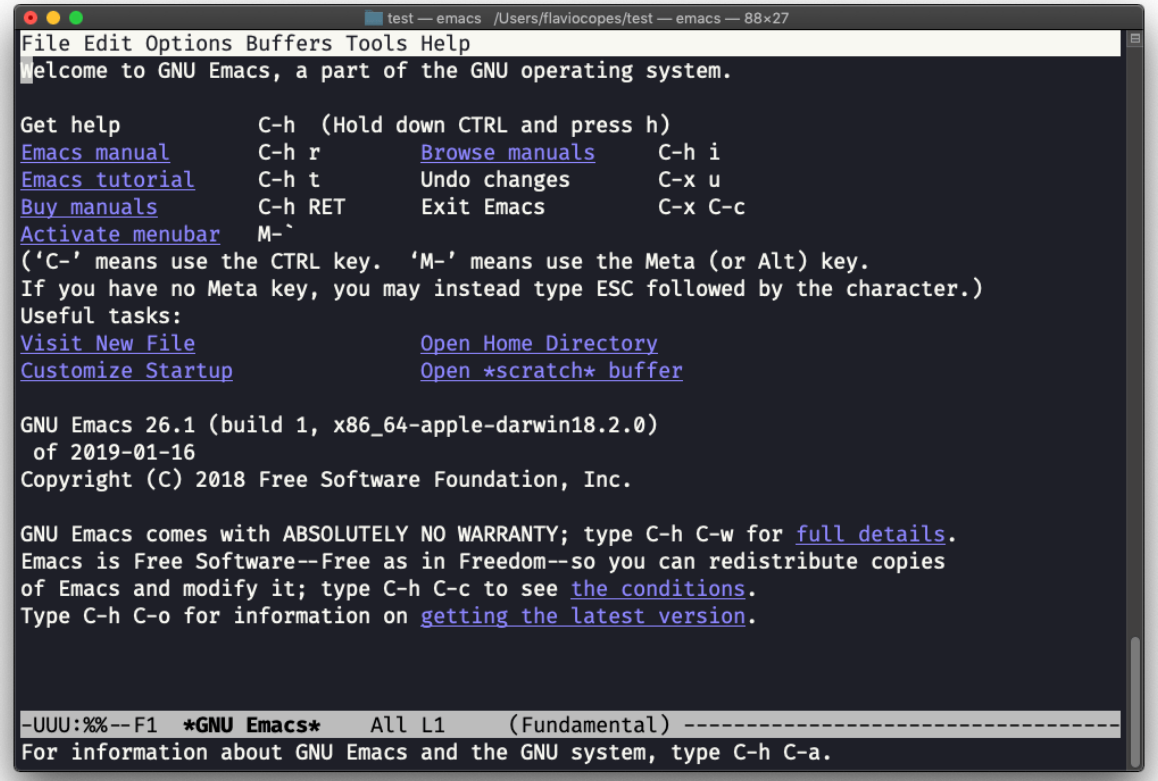
A short guide to emacs

A short guide to the emacs editor on UNIX systems, covering how to open and edit a file, save with ctrl-x ctrl-w, and quit, plus the built-in tutorial.

`emacs` is an awesome editor and it's historically regarded as *the* editor for UNIX systems. Famously `vi` vs `emacs` flame wars and heated discussions caused many unproductive hours for developers around the world.

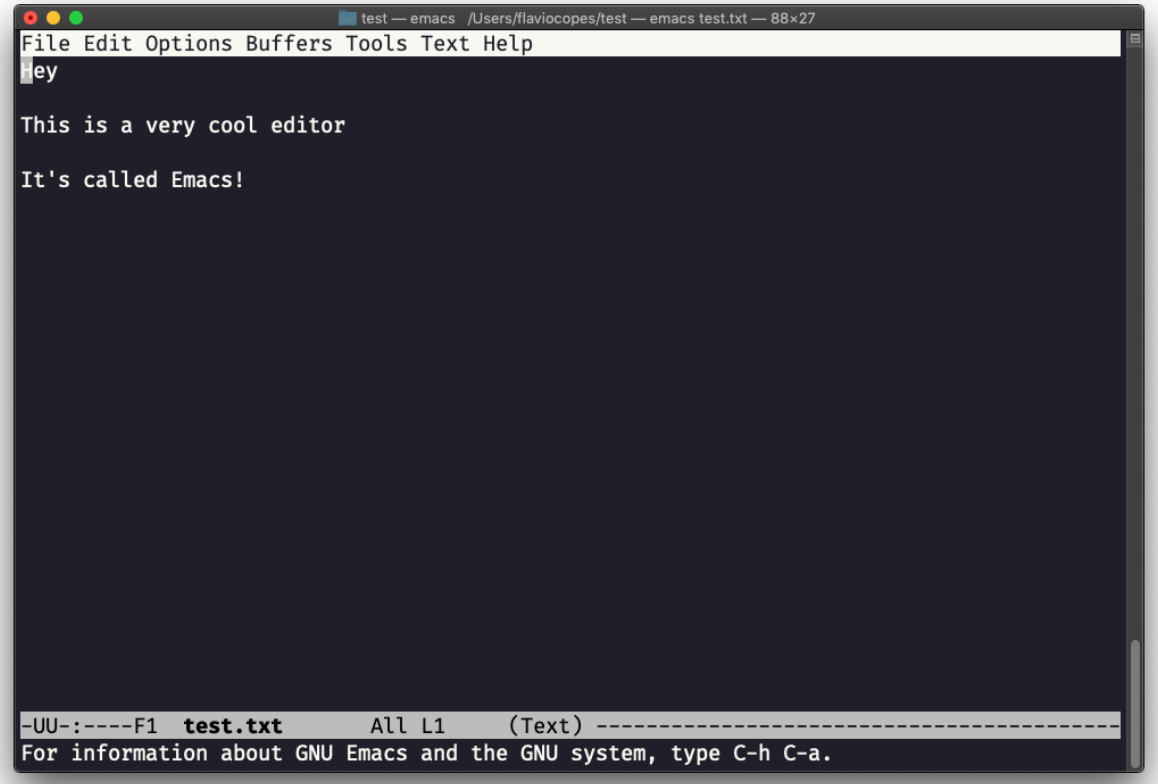
`emacs` is very powerful. Some people use it all day long as a kind of operating system (https://news.ycombinator.com/item?id=19127258). We'll just talk about the basics here.

You can open a new emacs session simply by invoking `emacs`:

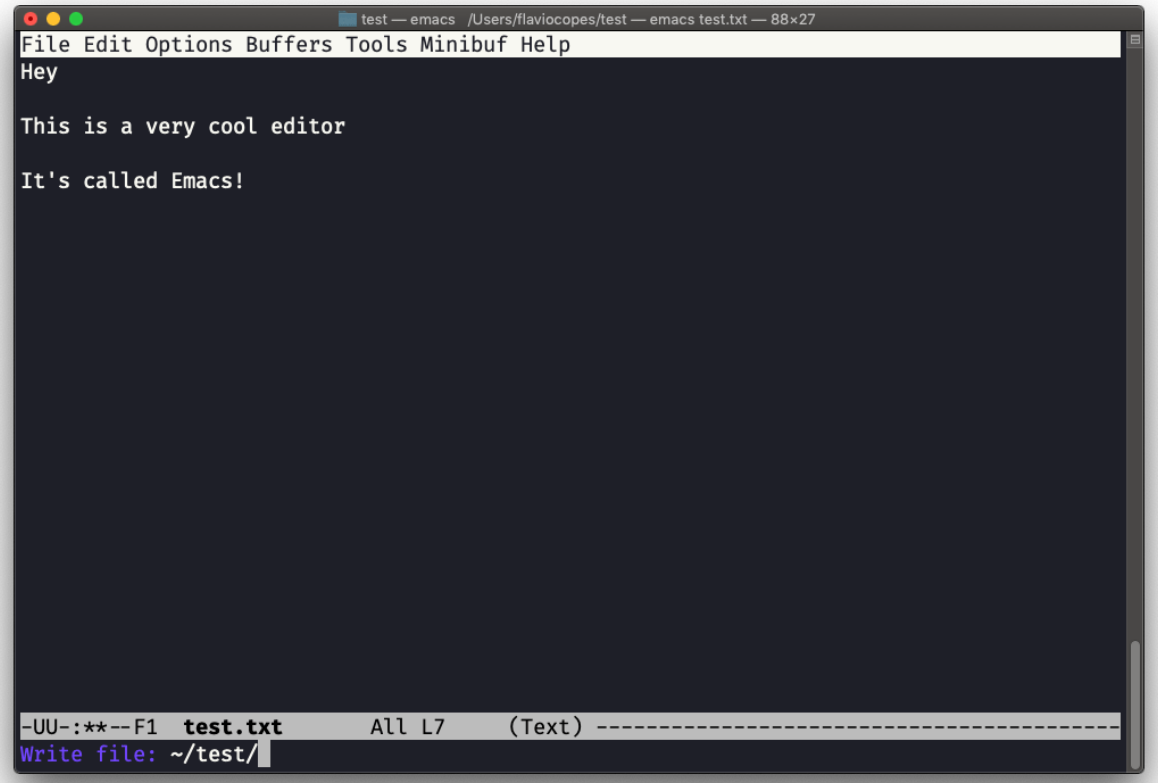


> macOS users, stop a second now. If you are on Linux there are no problems, but macOS does not ship applications using GPLv3, and every built-in UNIX command that has been updated to GPLv3 has not been updated. While there is a little problem with the commands I listed up to now, in this case using an emacs version from 2007 is not exactly the same as using a version with 12 years of improvements and change. This is not a problem with Vim, which is up to date. To fix this, run `brew install emacs` and running `emacs` will use the new version from Homebrew (make sure you have Homebrew installed)

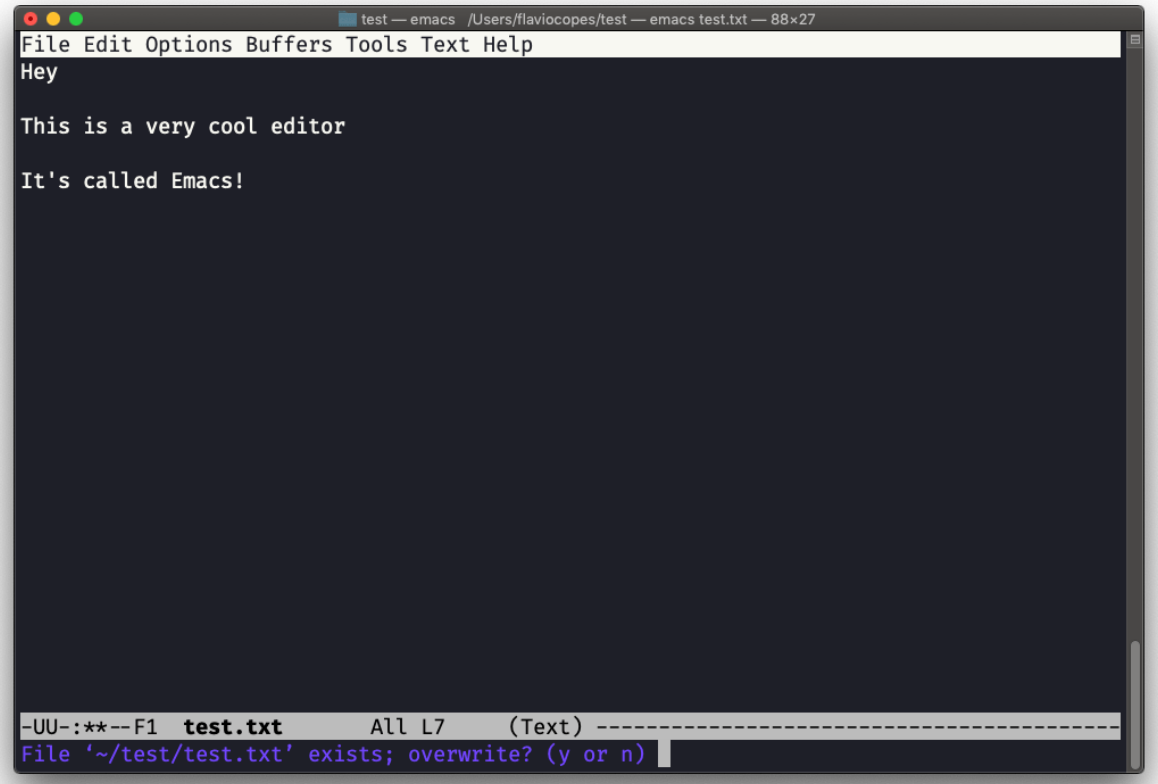
You can also edit an existing file calling `emacs <filename>`:



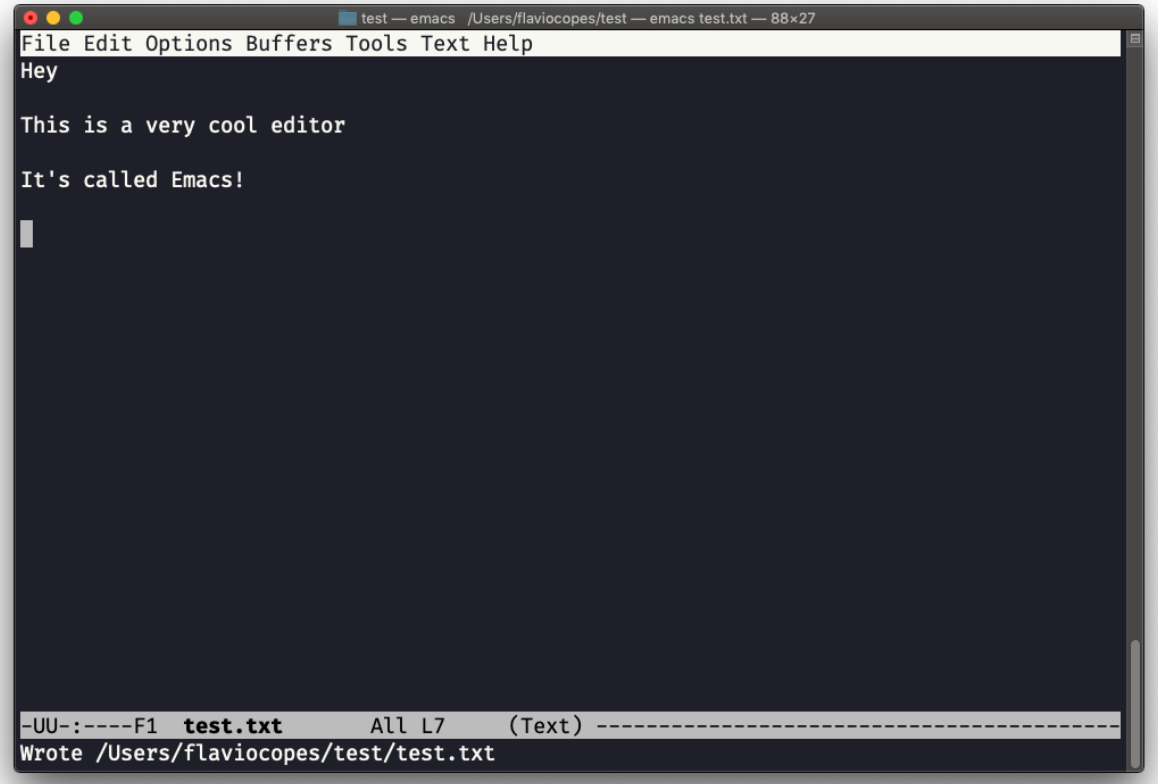
You can start editing and once you are done, press `ctrl-x` followed by `ctrl-w`. You confirm the folder:



and Emacs tell you the file exists, asking you if it should overwrite it:



Answer `y`, and you get a confirmation of success:



You can exit Emacs pressing `ctrl-x` followed by `ctrl-c`. Or `ctrl-x` followed by `c` (keep `ctrl` pressed).

There is a lot to know about Emacs. More than I am able to write in this little introduction. I encourage you to open Emacs and press `ctrl-h r` to open the built-in manual and `ctrl-h t` to open the official tutorial.

Linux commands: su

Learn how the Linux su command switches your shell to another user, starting a new shell, or to root when run alone, with exit to return to your own user.

While you're logged in to the terminal shell with one user, you might have the need to switch to another user.

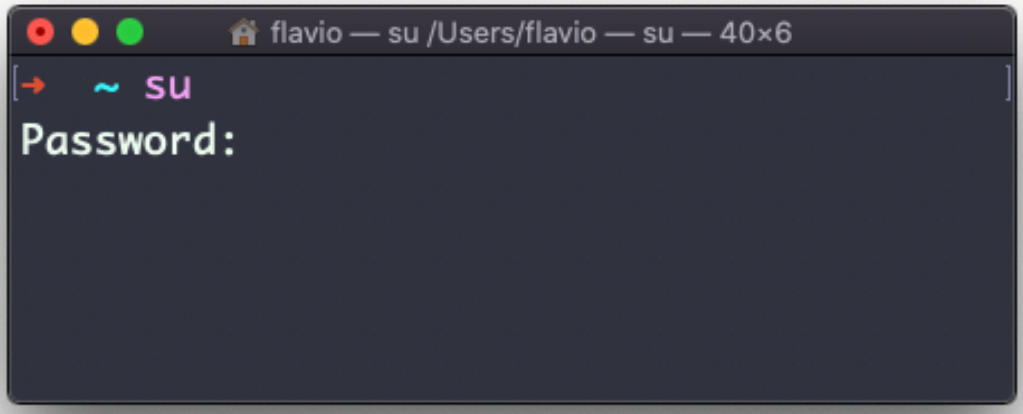
For example you're logged in as root to perform some maintenance, but then you want to switch to a user account.

You can do so with the `su` command:

```
su <username>
```

For example: `su flavio`.

If you're logged in as a user, running `su` without anything else will prompt to enter the `root` user password, as that's the default behavior.



`su` will start a new shell as another user.

When you're done, typing `exit` in the shell will close that shell, and will return back to the current user's shell.

> The `su` command works on Linux. On macOS it will not work unless you enable the root user (tip: you can use `sudo` instead)

Linux commands: killall

Learn how the Linux killall command sends a signal to every process with a name, so killall top ends all running top instances, and how to set the signal.

`kill` targets process IDs. `killall` targets every matching process name, so its scope is wider:

This is the syntax:

```
killall <name>
```

Before sending anything, inspect what will match:

```
pgrep -a top
```

Then ask the processes to terminate normally:

```
killall -TERM top
```

`TERM` gives a program a chance to flush data and clean up. Wait and inspect again. Use `KILL` only when the process cannot respond, because it cannot run cleanup handlers:

```
killall -KILL top
```

Signals can mean program-specific actions. `HUP` often asks a daemon to reload configuration, but that behavior belongs to the program, not to `killall`:

```
killall -HUP top
```

Process-name matching and available flags vary between Linux and macOS. Read the local manual with `man killall`; never assume a command copied from another operating system has identical

semantics.

For one known process, prefer its PID or service manager because the target is explicit. `killall` can terminate your own unrelated sessions with the same name, and elevated privileges widen the damage.

Try it: start two harmless `sleep 60` processes, inspect them with `pgrep`, send `TERM` by name, and confirm both exited.

Linux commands: uname

Learn how uname prints the kernel name, release, version, hostname, and machine hardware on Unix systems, and when macOS users should use sw_vers instead.

Running `uname` without an option prints the kernel name:

```
uname
```

On macOS this prints `Darwin`. On a Linux system it normally prints `Linux`.

The `-m` option prints the machine hardware name:

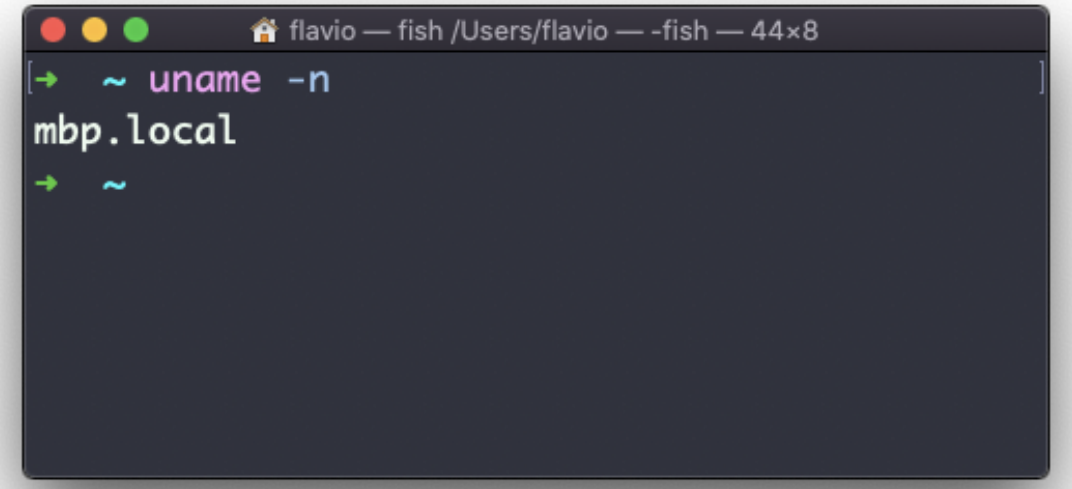
```
uname -m
```

The `-p` option tries to print the processor type, but it is not portable and can print `unknown`. Prefer `-m` in scripts.

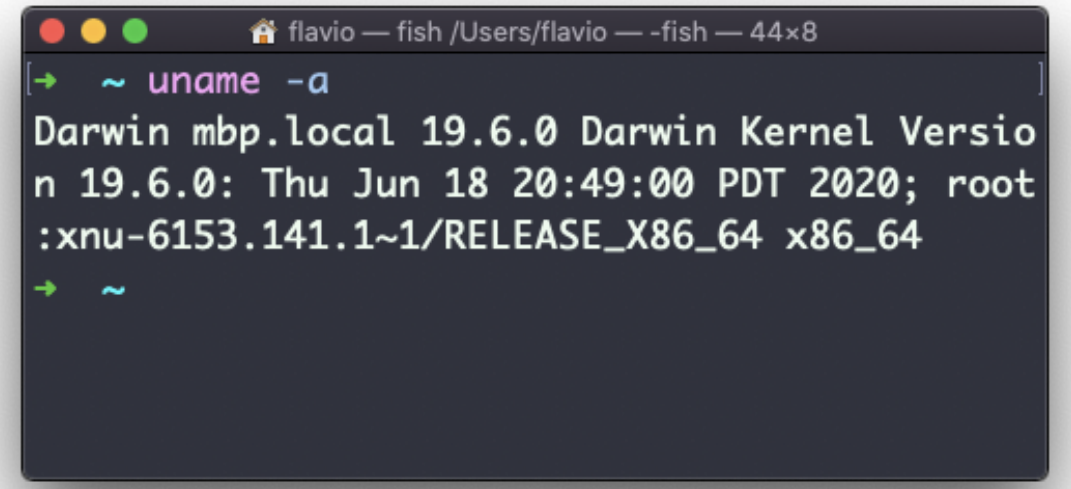
Use `-s` for the kernel name, `-r` for its release, and `-v` for its version:

```
uname -srv
```

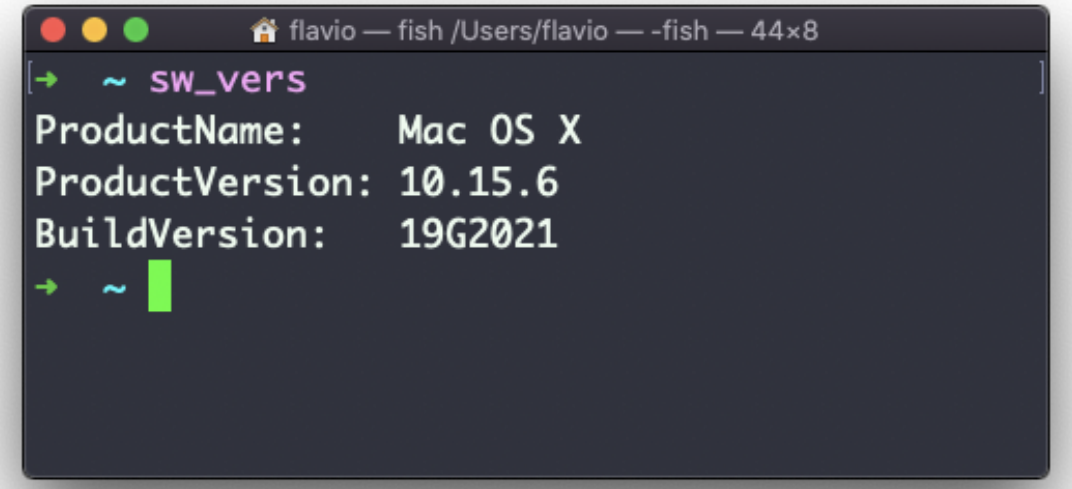
The `-n` option prints the network node name, which is normally the hostname:



The `-a` option prints all available fields:



On macOS, use `sw_vers` when you need the macOS product name, version, and build number. Those values differ from the Darwin kernel version printed by `uname`.



The `uname` command works on Linux, macOS, WSL, and other Unix-like environments. Stick to `-s`, `-n`, `-r`, `-v`, and `-m` when portability matters.

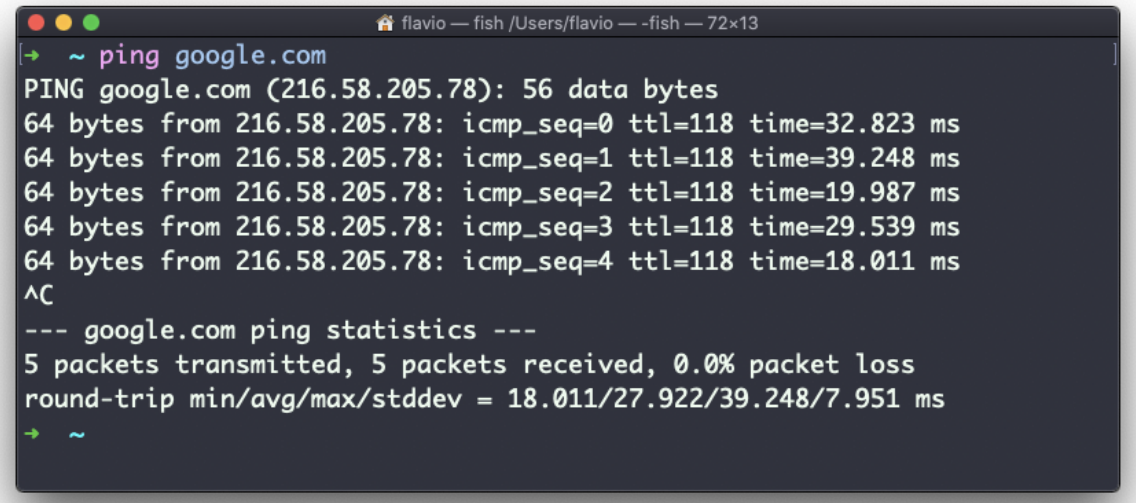
Linux commands: ping

Learn how the Linux ping command checks if a host is reachable over the ICMP protocol, reports round-trip time and packet loss, and limits tries with -c.

The `ping` command pings a specific network host, on the local network or on the Internet.

You use it with the syntax `ping <host>` where `<host>` could be a domain name, or an IP address.

Here's an example pinging `google.com`:



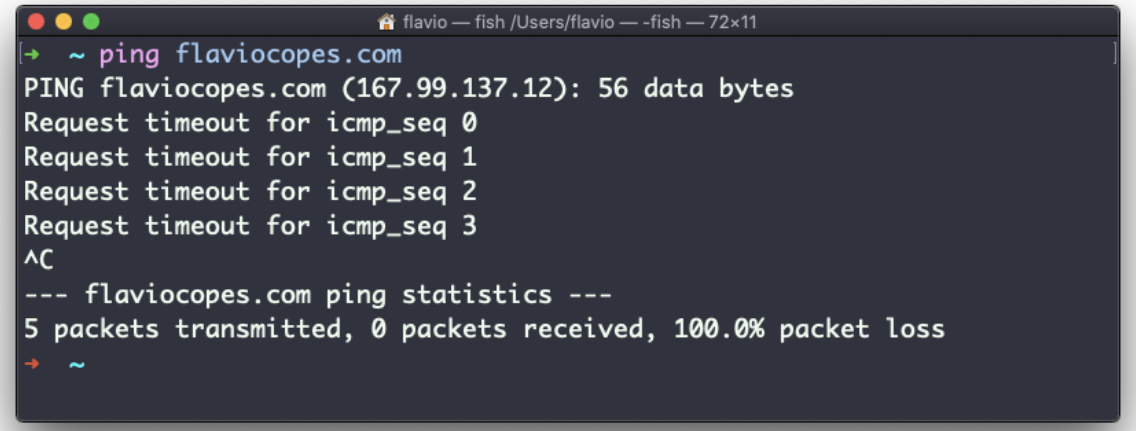
The commands sends a request to the server, and the server returns a response.

`ping` keep sending the request every second, by default, and will keep running until you stop it with `ctrl-C`, unless you pass the number of times you want to try with the `-c` option: `ping -c 2 google.com`.

Once `ping` is stopped, it will print some statistics about the results: the percentage of packages lost, and statistics about the network performance.

As you can see the screen prints the host IP address, and the time that it took to get the response back.

Not all servers support pinging, in case the requests times out:



Sometimes this is done on purpose, to "hide" the server, or just to reduce the load. The ping packets can also be filtered by firewalls.

`ping` works using the **ICMP protocol** (*Internet Control Message Protocol*), a network layer protocol just like TCP or UDP.

The request sends a packet to the server with the `ECHO_REQUEST` message, and the server returns a `ECHO_REPLY` message. I won't go into details, but this is the basic concept.

Pinging a host is useful to know if the host is reachable (supposing it implements ping), and how distant it is in terms of how long it takes to get back to you. Usually the nearest the server is geographically, the less time it will take to return back to you, for simple physical laws that cause a longer distance to introduce more delay in the cables.

The `ping` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Linux commands: traceroute

Learn how the Linux traceroute command maps every router hop your packets take to reach a host, showing the IP and timing of each, and tuning samples with -q.

When you try to reach a host on the Internet, you go through your home router, then you reach your ISP network, which in turn goes through its own upstream network router, and so on, until you finally reach the host.

Have you ever wanted to know what are the steps that your packets go through to do that?

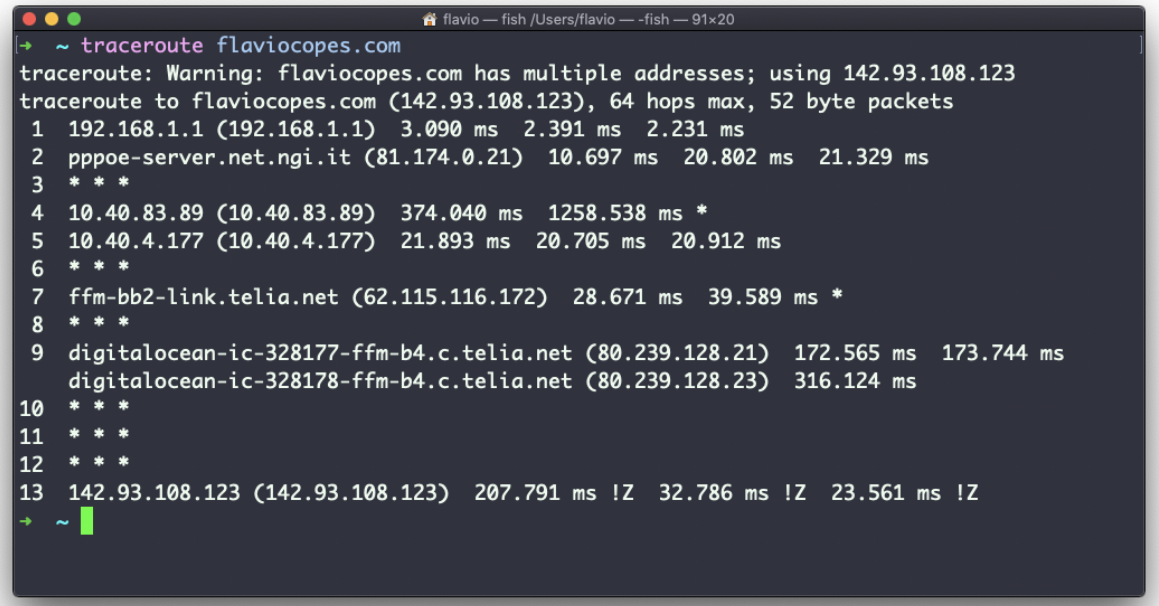
The `traceroute` command is made for this.

You invoke

```
traceroute <host>
```

and it will (slowly) gather all the information while the packet travels.

In this example I tried reaching for my blog with `traceroute flaviocopes.com`:

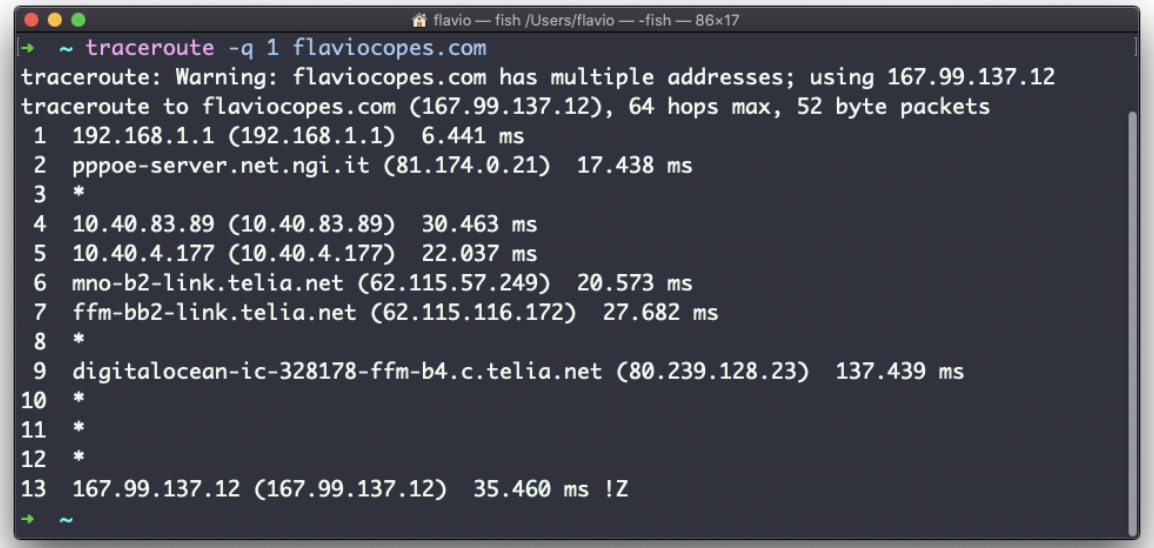


Not every router travelled returns us information. In this case, `traceroute` prints `* * *`. Otherwise, we can see the hostname, the IP address, and some performance indicator.

For every router we can see 3 samples, which means traceroute tries by default 3 times to get you a good indication of the time needed to reach it. This is why it takes this long to execute `traceroute` compared to simply doing a `ping` to that host.

You can customize this number with the `-q` option:

```
traceroute -q 1 flaviocopes.com
```



The `traceroute` command works on Linux, macOS, WSL, and anywhere you have a UNIX environment

Check your understanding: shell, system, and network tools

Check aliases, command lookup, environment inspection, editors, users, processes, system identity, and network diagnostics.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/terminal/.