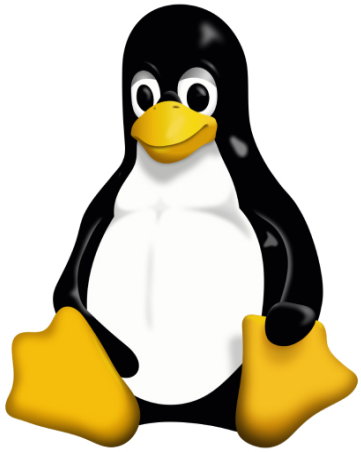
# THE LINUX



# COMMANDS HANDBOOK

Flavio Copes

# Table of Contents


Preface

Introduction to Linux and shells

man

ls

cd

pwd

mkdir

rmdir

mv

cp

open

touch

find

ln

gzip

gunzip

tar

alias

cat

less

tail

wc

grep

sort

uniq

diff

echo

chown

chmod

umask

du

df

basename

dirname

ps

top

kill

killall

jobs

bg

fg

type

which

nohup

xargs

vim

emacs

nano

whoami

who

su

sudo

passwd
ping
traceroute
clear
history
export
crontab
uname
env
printenv
Conclusion

# Preface

The Linux Commands Handbook follows the 80/20 rule: learn in 20% of the time the 80% of a topic.

I find this approach gives a well-rounded overview.

This book does not try to cover everything under the sun related to Linux and its commands. It focuses on the small core commands that you will use the 80% or 90% of the time, trying to simplify the usage of the more complex ones.

All those commands work on Linux, macOS, WSL, and anywhere you have a UNIX environment.

I hope the contents of this book will help you achieve what you want: **get comfortable with Linux**.

This book is written by Flavio. I **publish programming tutorials** every day on my website flaviocopes.com.

You can reach me on Twitter @flaviocopes.

Enjoy!

# Introduction to Linux and shells

Linux is an operating system, like macOS or Windows.

It is also the most popular Open Source and free, as in freedom, operating system.

It powers the vast majority of the servers that compose the Internet. It's the base upon which everything is built upon. But not just that. Android is based on (a modified version of) Linux.

The Linux "core" (called *kernel*) was born in 1991 in Finland, and it went a really long way from its humble beginnings. It went on to be the kernel of the GNU Operating System, creating the duo GNU/Linux.

There's one thing about Linux that corporations like Microsoft and Apple, or Google, will never be able to offer: the freedom to do whatever you want with your computer.

They're actually going in the opposite direction, building walled gardens, especially on the mobile side.

Linux is the ultimate freedom.

It is developed by volunteers, some paid by companies that rely on it, some independently, but there's no single commercial company that can dictate what goes into Linux, or the project priorities.

Linux can also be used as your day to day computer. I use macOS because I really enjoy the applications, the design and I also used to be an iOS and Mac apps developer, but before using it I used Linux as my main computer Operating System.

No one can dictate which apps you can run, or "call home" with apps that track you, your position, and more.

Linux is also special because there's not just "one Linux", like it happens on Windows or macOS. Instead, we have **distributions**.

A "distro" is made by a company or organization and packages the Linux core with additional programs and tooling.

For example you have Debian, Red Hat, and Ubuntu, probably the most popular.

Many, many more exist. You can create your own distribution, too. But most likely you'll use a popular one, one that has lots of users and a community of people around it, so you can do what you need to do without losing too much time reinventing the wheel and figuring out answers to common problems.

Some desktop computers and laptops ship with Linux preinstalled. Or you can install it on your Windows-based computer, or on a Mac.

But you don't need to disrupt your existing computer just to get an idea of how Linux works.

I don't have a Linux computer.

If you use a Mac you need to know that under the hood macOS is a UNIX Operating System, and it shares a lot of the same ideas and software that a GNU/Linux system uses, because GNU/Linux is a free alternative to UNIX.

> UNIX is an umbrella term that groups many operating systems used in big corporations and institutions, starting from the 70's

The macOS terminal gives you access to the same exact commands I'll describe in the rest of this handbook.

Microsoft has an official Windows Subsystem for Linux which you can (and should!) install on Windows. This will give you the ability to run Linux in a very easy way on your PC.

But the vast majority of the time you will run a Linux computer in the cloud via a VPS (Virtual Private Server) like DigitalOcean.

A shell is a command interpreter that exposes to the user an interface to work with the underlying operating system.

It allows you to execute operations using text and commands, and it provides users advanced features like being able to create scripts.

This is important: shells let you perform things in a more optimized way than a GUI (Graphical User Interface) could ever possibly let you do. Command line tools can offer many different configuration options without being too complex to use.

There are many different kind of shells. This post focuses on Unix shells, the ones that you will find commonly on Linux and macOS computers.

Many different kind of shells were created for those systems over time, and a few of them dominate the space: Bash, Csh, Zsh, Fish and many more!

All shells originate from the Bourne Shell, called `sh`. "Bourne" because its creator was Steve Bourne.

Bash means *Bourne-again shell*. `sh` was proprietary and not open source, and Bash was created in 1989 to create a free alternative for the GNU project and the Free Software Foundation. Since projects had to pay to use the Bourne shell, Bash became very popular.

If you use a Mac, try opening your Mac terminal. That by default is running ZSH. (or, pre-Catalina, Bash)

You can set up your system to run any kind of shell, for example I use the Fish shell.

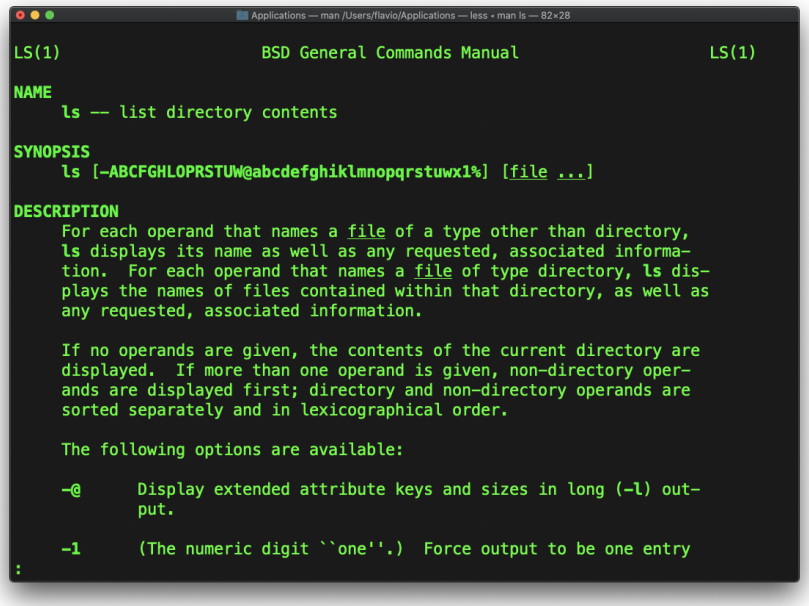
Each single shell has its own unique features and advanced usage, but they all share a common functionality: they can let you execute programs, and they can be programmed.

In the rest of this handbook we'll see in detail the most common commands you will use.

# man

The first command I want to introduce is a command that will help you understand all the other commands.

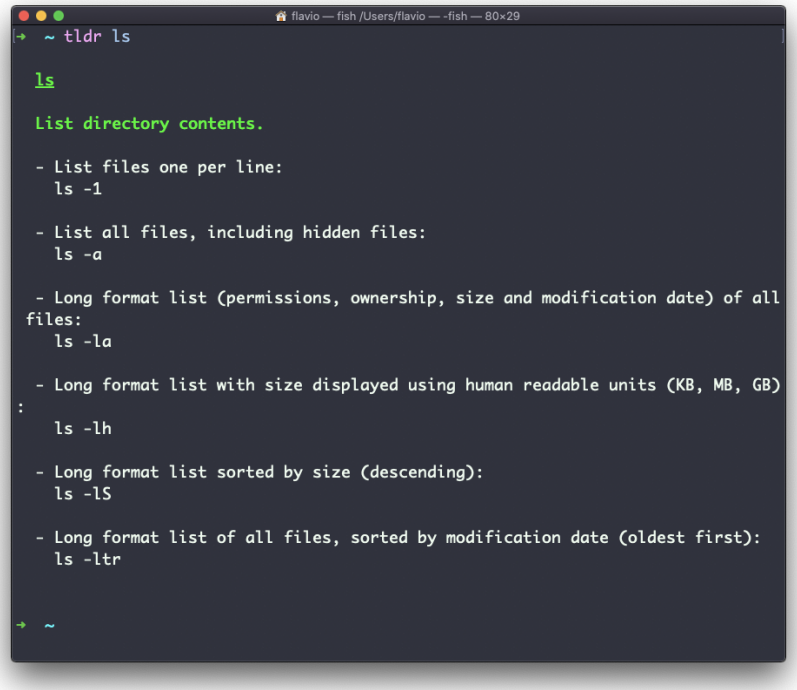
Every time I don't know how to use a command, I type `man <command>` to get the manual:




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.

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.

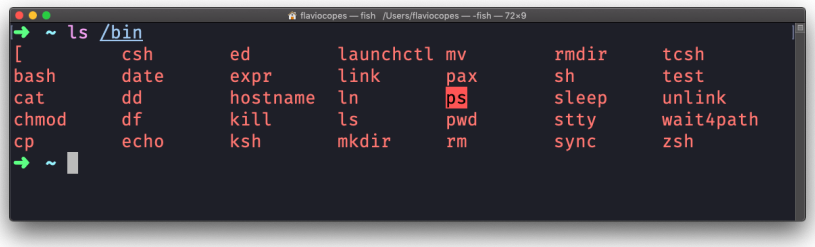
# ls

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

```
ls
```

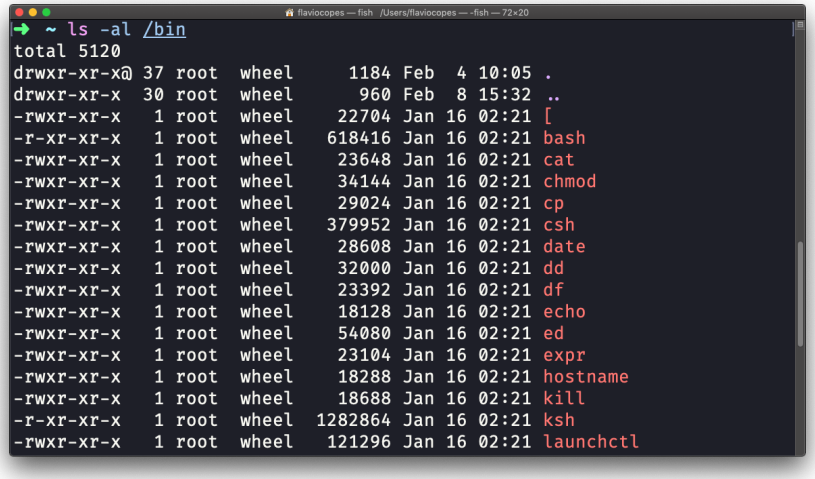
If you add a folder name or path, it will print that folder contents:

```
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 (`.`).

# cd

# 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.

# 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.

# 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.

# 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 fr
```

# 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
```

# open

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

```
open <filename>
```

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

The same command can also be be used to run an application:

```
open <application name>
```

# 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.

# 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.

# ln

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>
```

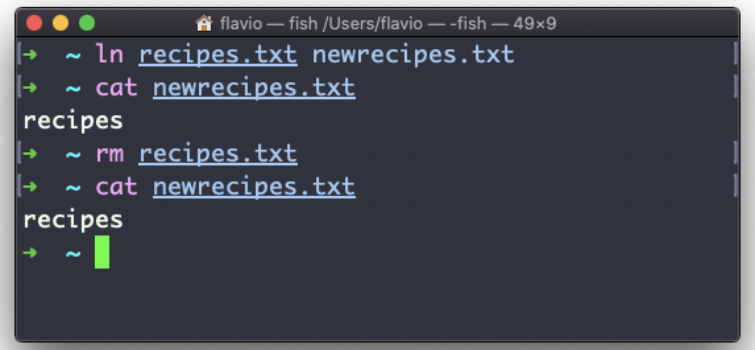
For example, say you have a file called recipes.txt. You can create a hard link to it using:

```
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.

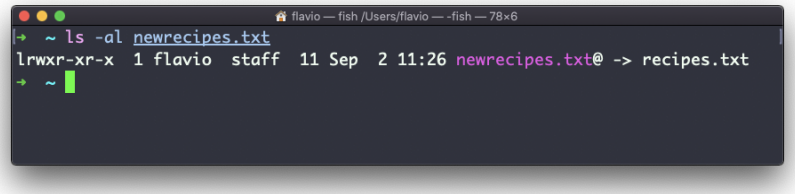
You create soft links using the `-s` option of `ln`:

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

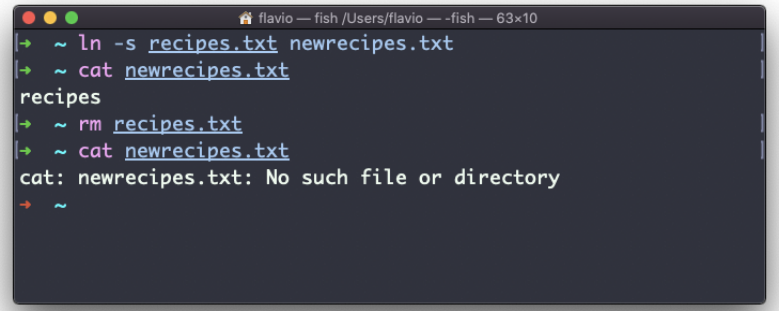
For example, say you have a file called recipes.txt. You can create a soft link to it using:

```
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:



# gzip

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
```

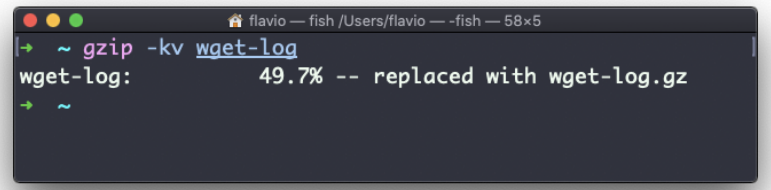
You can compress multiple files by listing them:

```
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
```

# gunzip

The `gunzip` command is basically equivalent to the `gzip` command, except the `-d` 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.

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

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

# tar

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.

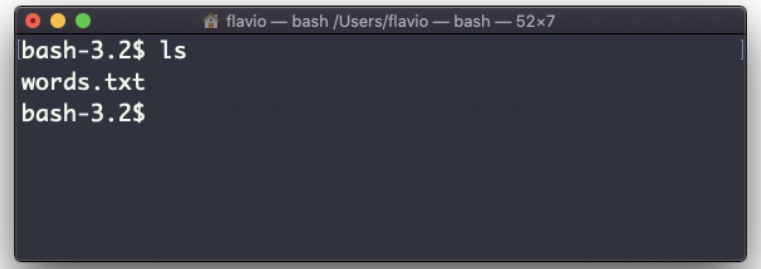
To unarchive a gzipped archive, you can use `gunzip`, or `gzip -d`, and then unarchive it, but `tar -xf` will recognize it's a gzipped archive, and do it for you:

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

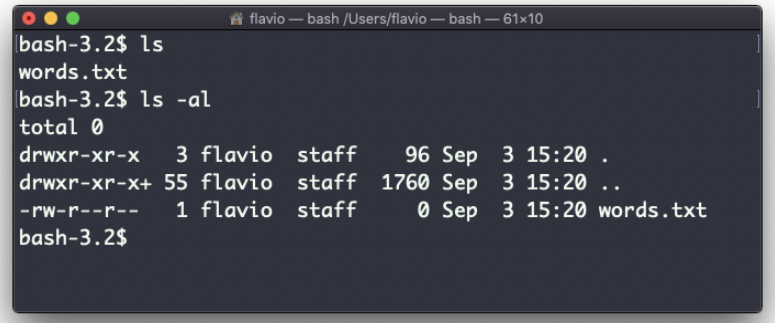
# alias

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 `.`:

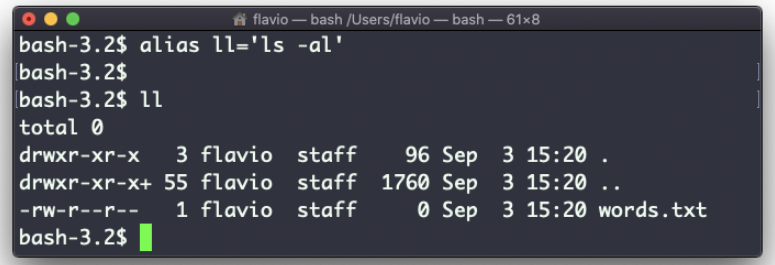



You can create a new command, for example I like to call it `ll`, that is an alias to `ls -al`.

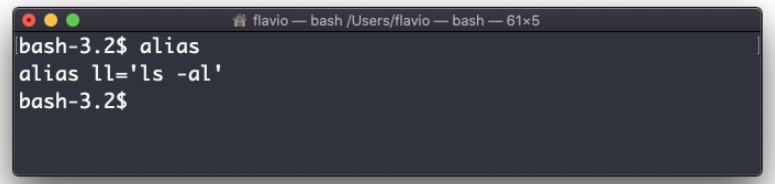
You do it in this way:

```
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.

# 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
```

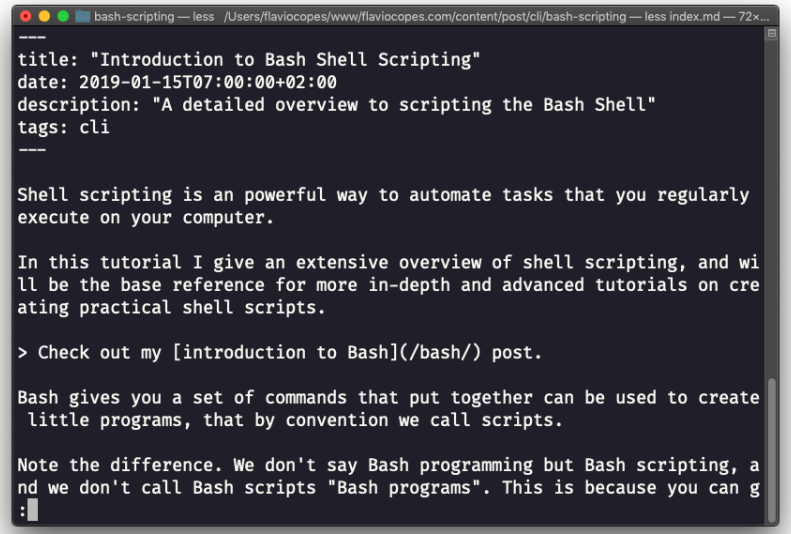
You can only add a number to non-blank lines using `-b`, or you can also remove all the multiple empty lines using `-s`.

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

# less

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).

# tail

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`.

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`.

# wc

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
```

# grep

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
```

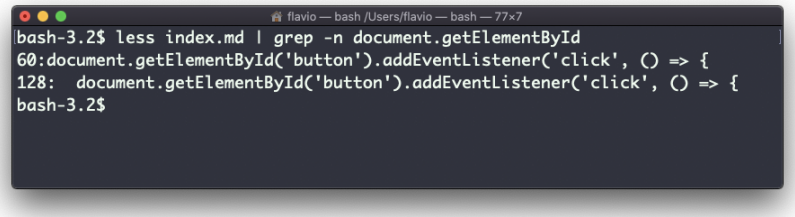
One very useful thing is to tell grep to print 2 lines before, and 2 lines after the matched line, to give us more context. That's done using the `-C` option, which accepts a number of lines:

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

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

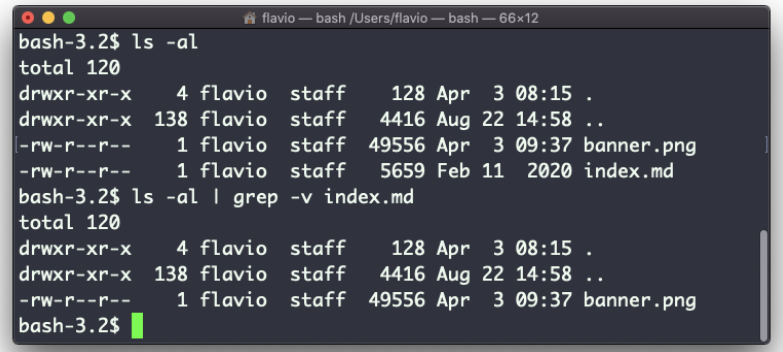
As mentioned, you can use grep to filter the output of another command. We can replicate the same functionality as above using:

```
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:

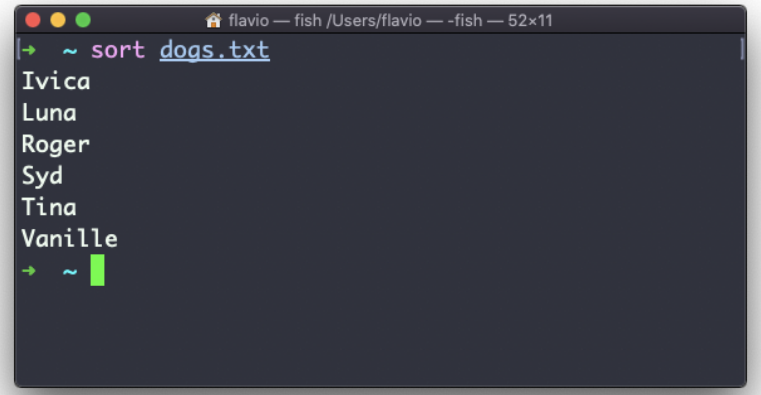


# sort

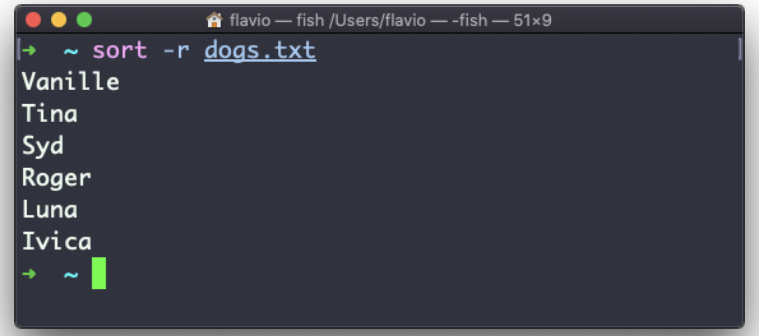
Suppose you have a text file which contains the names of dogs:

This list is unordered.

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

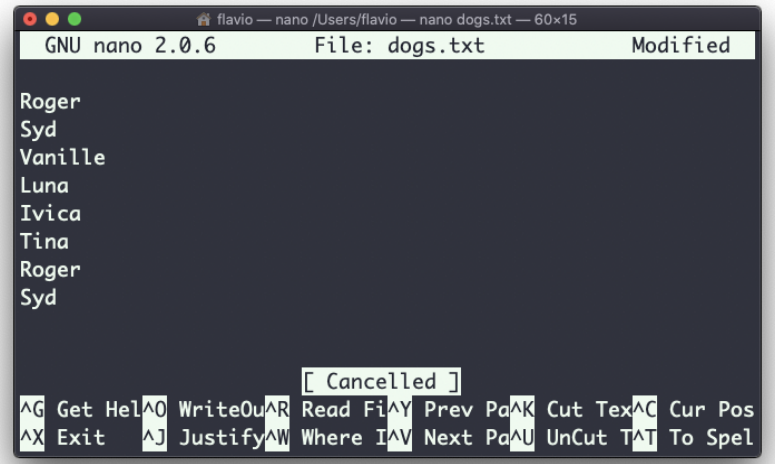



Use the `r` option to reverse the order:

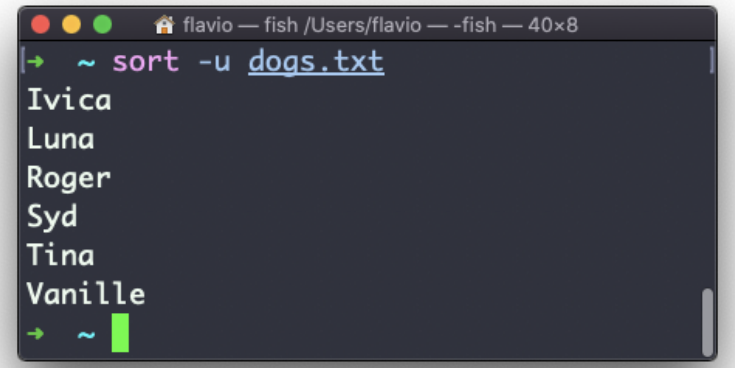



Sorting by default is case sensitive, and alphabetic. Use the `--ignore-case` option to sort case insensitive, and the `-n` option to sort using a numeric order.

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`.

flavio — man /Users/flavio — less • man sort — 115×55
SORT(1) BSD General Commands Manual SORT(1)
NAME
sort -- sort or merge records (lines) of text and binary files
SYNOPSIS
sort [-bcCdfghiRMmnrsuVz] [-k field1[,field2]] [-S memsize] [-T dir] [-t char] [-o output]
[file ...]
sort --help
sort --version
DESCRIPTION
The sort utility sorts text and binary files by lines. A line is a record separated from the subsequent record by a newline (default) or NUL '\0' character (-z option). A record can contain any printable or unprintable characters. Comparisons are based on one or more sort keys extracted from each line of input, and are performed lexicographically, according to the current locale's collating rules and the specified command-line options that can tune the actual sorting behavior. By default, if keys are not given, sort uses entire lines for comparison.
The command line options are as follows:
-c, --check, -C, --check=silent|quiet
Check that the single input file is sorted. If the file is not sorted, sort produces the appropriate error messages and exits with code 1, otherwise returns 0. If -C or --check=silent is specified, sort produces no output. This is a "silent" version of -c.
-m, --merge
Merge only. The input files are assumed to be pre-sorted. If they are not sorted the output order is undefined.
-o output, --output=output
Print the output to the output file instead of the standard output.
-S size, --buffer-size=size
Use size for the maximum size of the memory buffer. Size modifiers %,b,K,M,G,T,P,E,Z,Y can be used. If a memory limit is not explicitly specified, sort takes up to about 90% of available memory. If the file size is too big to fit into the memory buffer, the temporary disk files are used to perform the sorting.
-T dir, --temporary-directory=dir
Store temporary files in the directory dir. The default path is the value of the environment variable TMPDIR or /var/tmp if TMPDIR is not defined.
-u, --unique
Unique keys. Suppress all lines that have a key that is equal to an already processed one. This option, similarly to -s, implies a stable sort. If used with -c or -C, sort also checks that there are no lines with duplicate keys.
-s
Stable sort. This option maintains the original record order of records that have an equal key. This is a non-standard feature, but it is widely accepted and used.
--version
Print the version and silently exits.
:

# uniq

`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
```

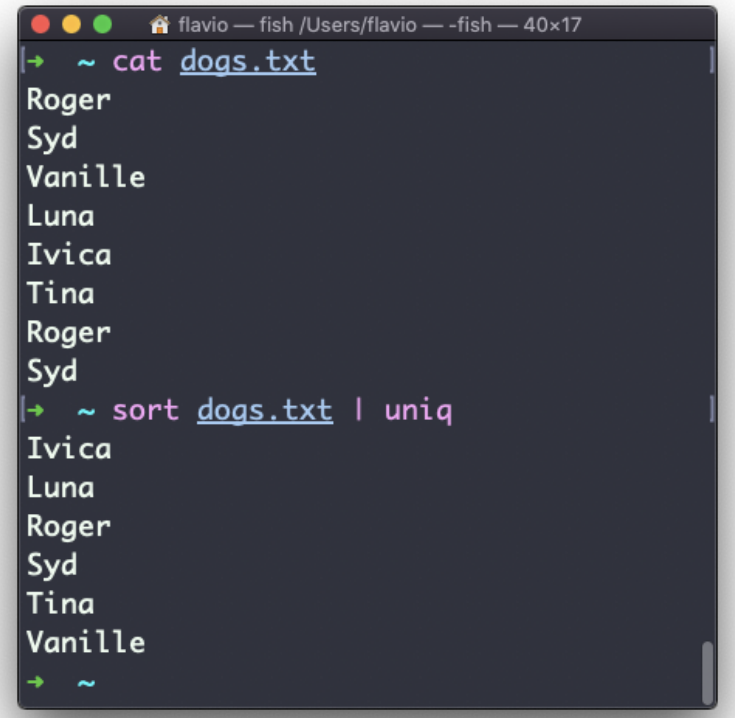
You need to consider this key thing: `uniq` will only detect adjacent duplicate lines.

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

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

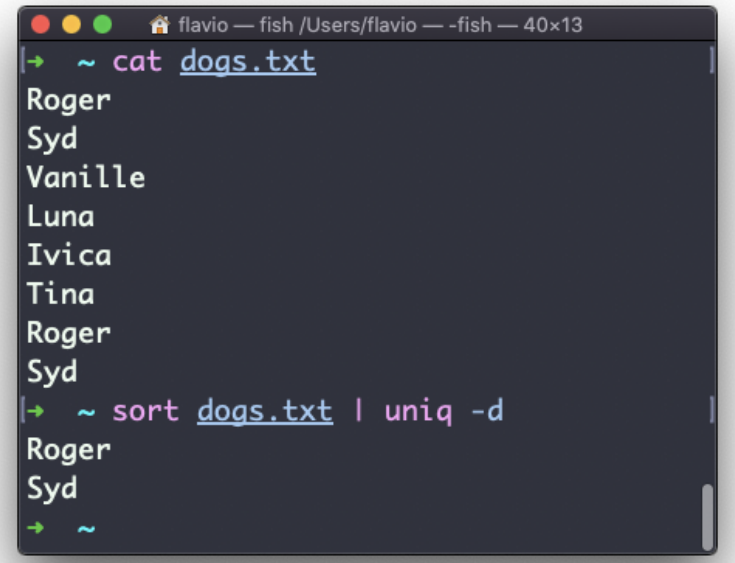
The `sort` command has its own way to remove duplicates with the `-u` (*unique*) option. But `uniq` has more power.

By default it removes duplicate lines:

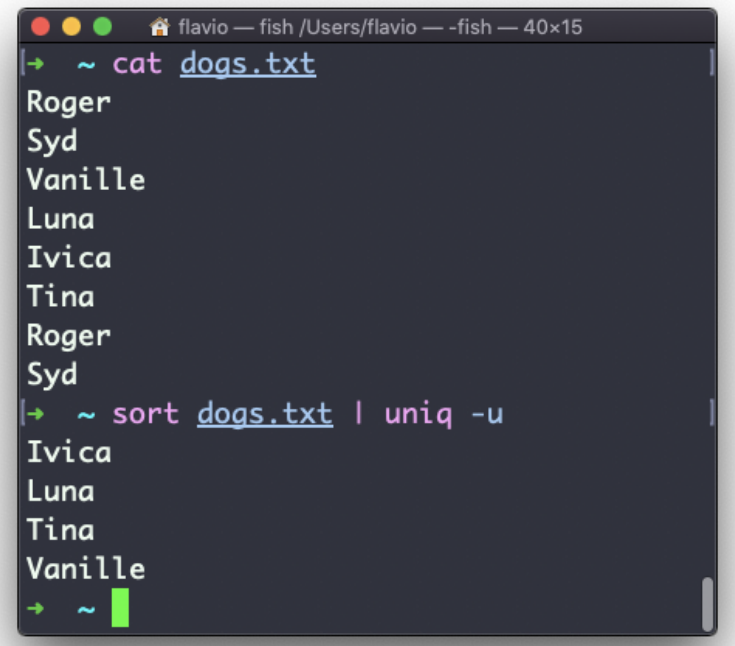



You can tell it to only display duplicate lines, for example, with the `-d` option:

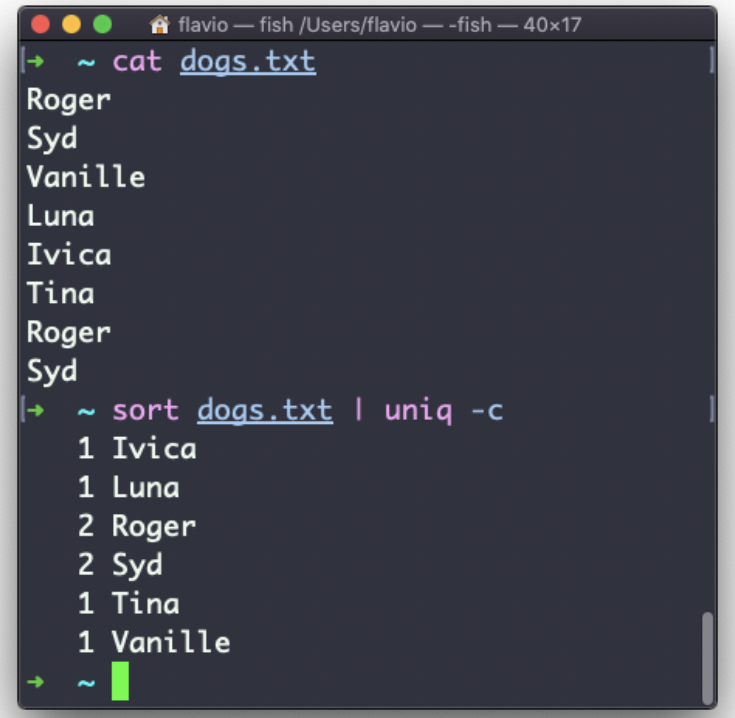
```
sort dogs.txt | uniq -d
```




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



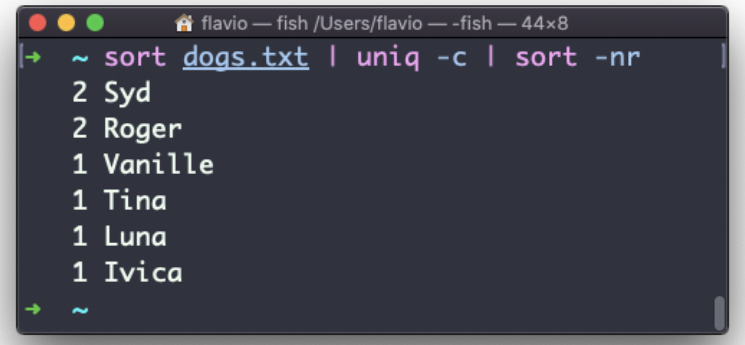

You can count the occurrences of each line with the `-c` option:




Use the special combination:

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

to then sort those lines by most frequent:

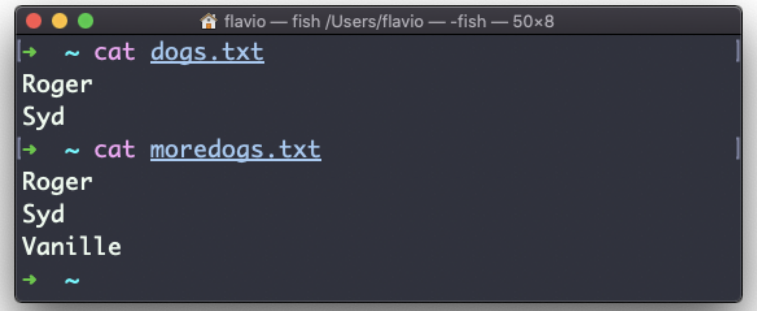


# diff

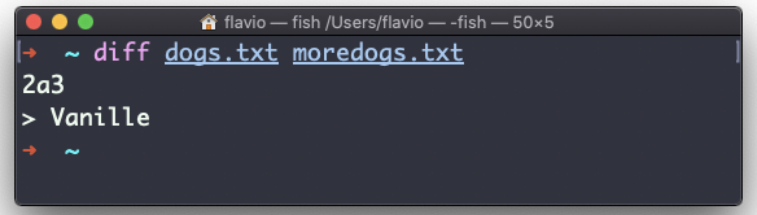
`diff` is a handy command. Suppose you have 2 files, which contain almost the same information, but you can't find the difference between the two.

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

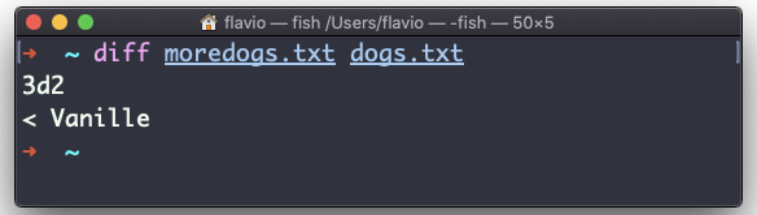
Suppose you have 2 files: `dogs.txt` and `moredogs.txt`. The difference is that `moredogs.txt` contains one more dog name:




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

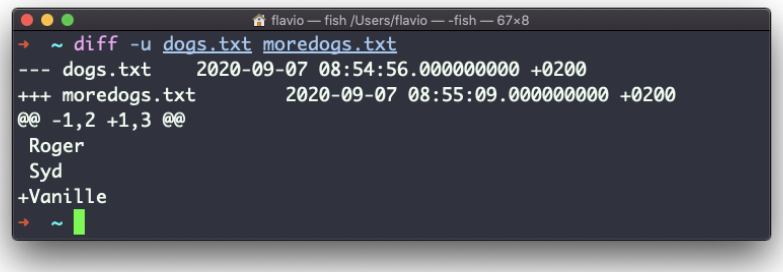


If you invert the order of the files, it will tell you that the second file is missing line 3, whose content is `Vanille`:

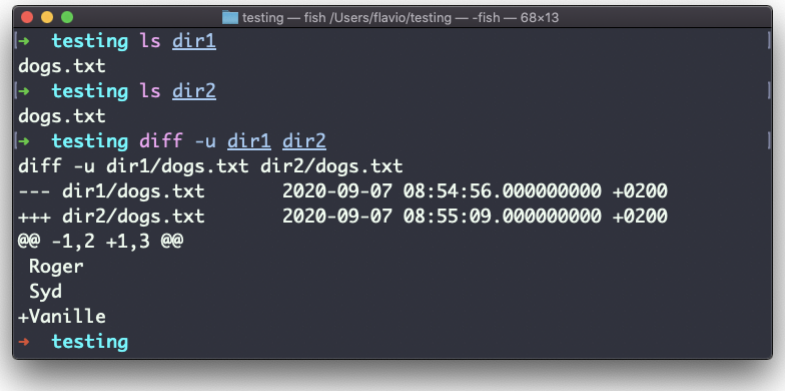



Using the `-y` option will compare the 2 files line by line:

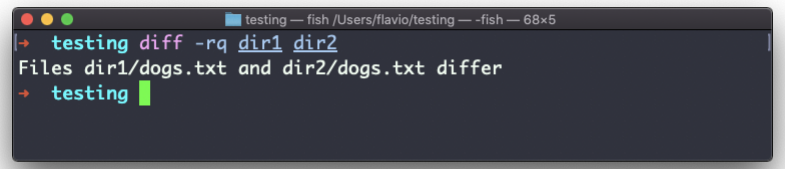
The `-u` option however will be more familiar to you, because that's the same used by the Git version control system to display differences between versions:



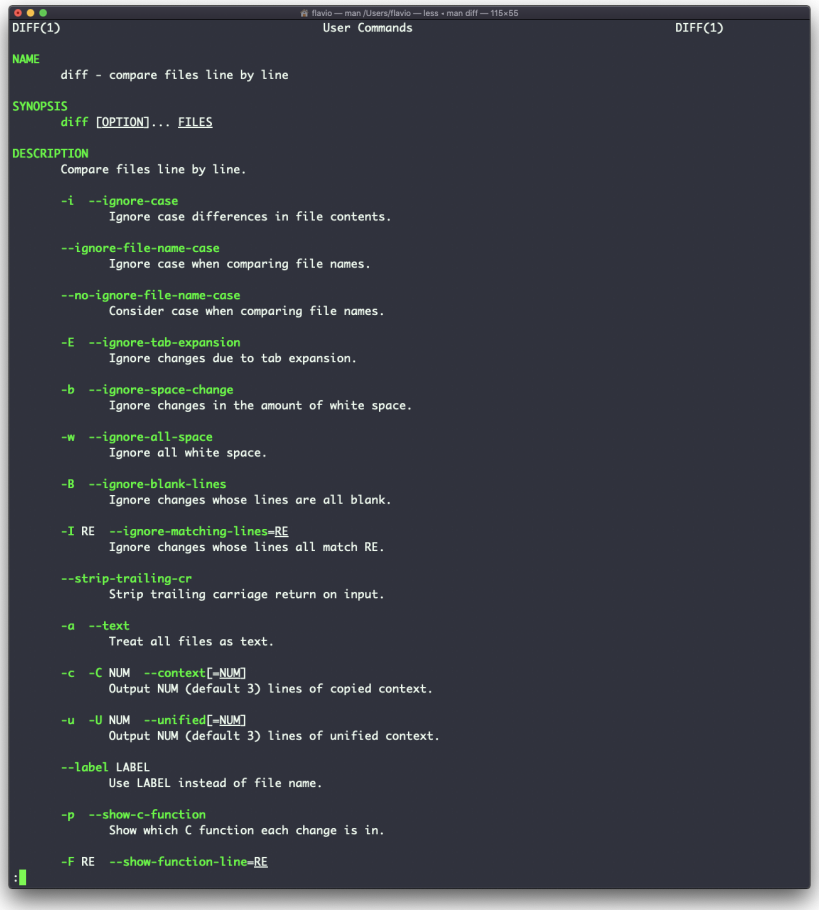

Comparing directories works in the same way. You must use the `-r` option to compare recursively (going into subdirectories):




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`:


flavio — man /Users/flavio — less « man diff — 115×55
DIFF(1) User Commands DIFF(1)
NAME
diff - compare files line by line
SYNOPSIS
diff [OPTION]... FILES
DESCRIPTION
Compare files line by line.
-i --ignore-case
Ignore case differences in file contents.
--ignore-file-name-case
Ignore case when comparing file names.
--no-ignore-file-name-case
Consider case when comparing file names.
-E --ignore-tab-expansion
Ignore changes due to tab expansion.
-b --ignore-space-change
Ignore changes in the amount of white space.
-w --ignore-all-space
Ignore all white space.
-B --ignore-blank-lines
Ignore changes whose lines are all blank.
-I RE --ignore-matching-lines=RE
Ignore changes whose lines all match RE.
--strip-trailing-cr
Strip trailing carriage return on input.
-a --text
Treat all files as text.
-c -C NUM --context[=NUM]
Output NUM (default 3) lines of copied context.
-u -U NUM --unified[=NUM]
Output NUM (default 3) lines of unified context.
--label LABEL
Use LABEL instead of file name.
-p --show-c-function
Show which C function each change is in.
-F RE --show-function-line=RE
:

# echo

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

This example:

```
echo "hello"
```

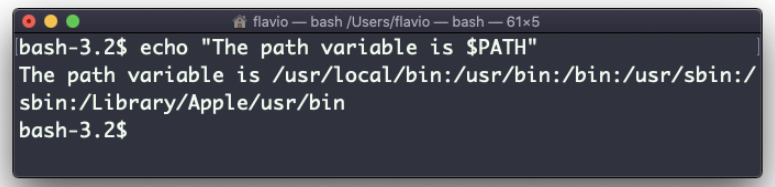
will print `hello` to the terminal.

We can append the output to a file:

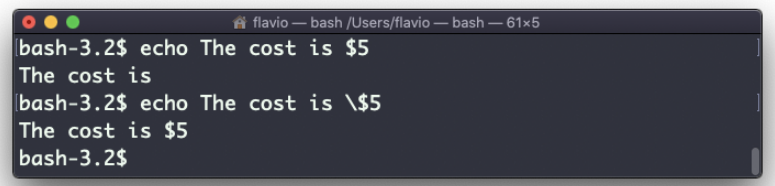
```
echo "hello" >> output.txt
```

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 *
```

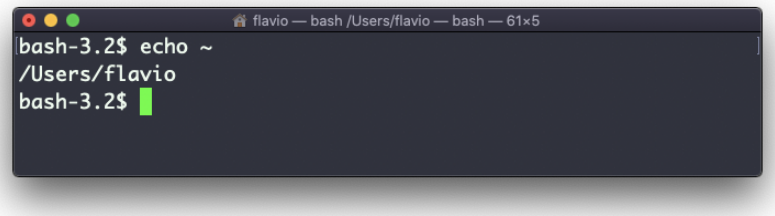
We can echo the files in the current folder that start with the letter `o`:

```
echo o*
```

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

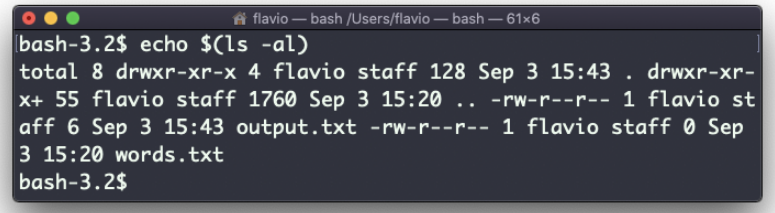
You can print your home folder path:

```
echo ~
```

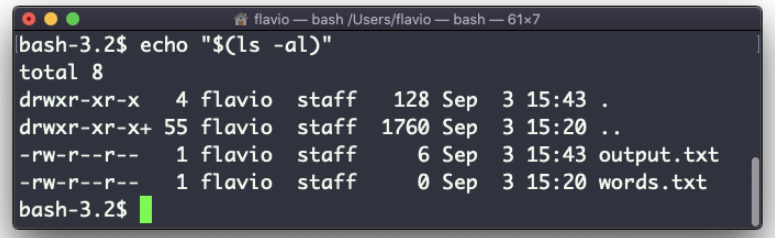



You can also execute commands, and print the result to the standard output (or to file, as you saw):

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

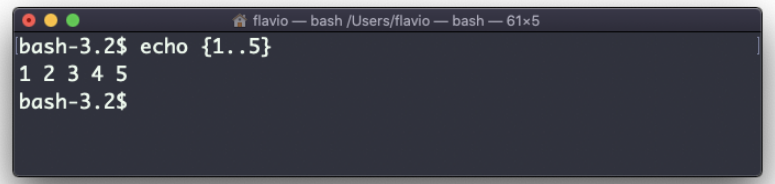



Note that whitespace is not preserved by default. You need to wrap the command in double quotes to do so:




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

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



# chown

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.

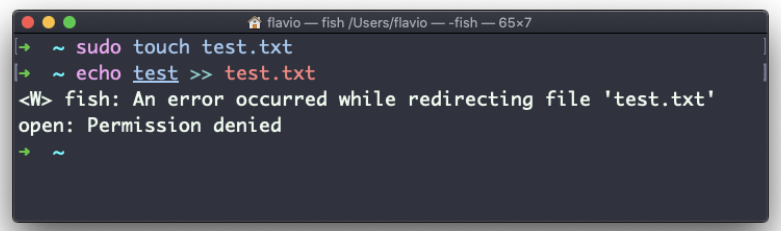
The owner (and the `root` user) can change the owner to another user, too, using the `chown` command:

```
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
```

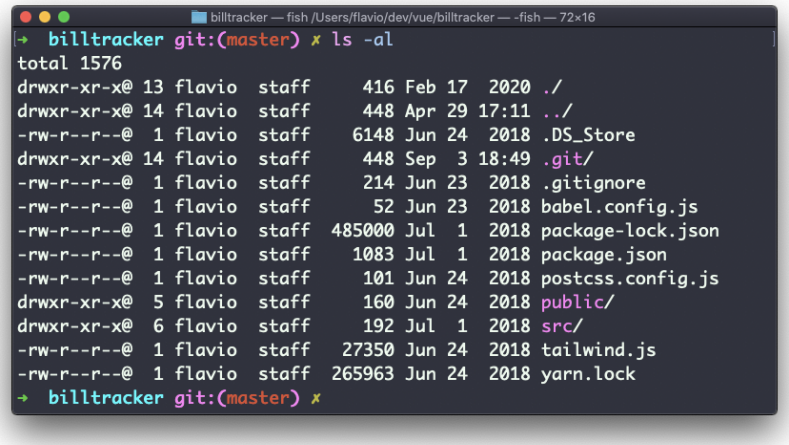
You can also just change the group of a file using the `chgrp` command:

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

# chmod

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
```

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
```

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:

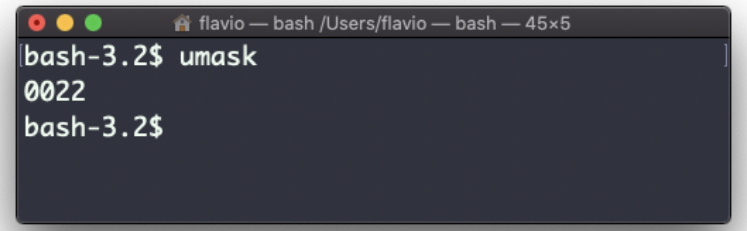
```
chmod 777 filename
chmod 755 filename
chmod 644 filename
```

# umask

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.

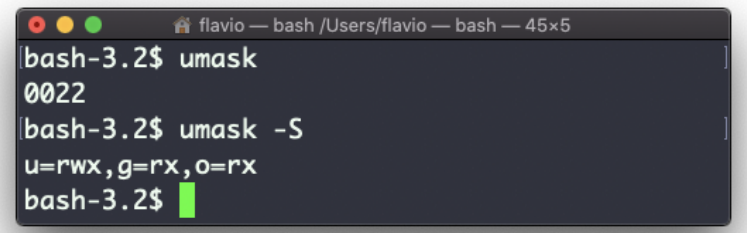
Typing `umask` with no arguments will show you the current umask, in this case `0022`:




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
```

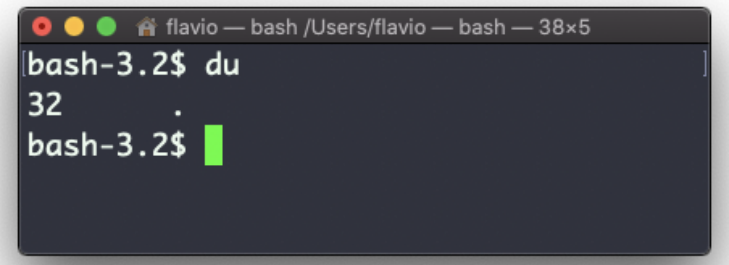
or you can change a specific role's permission:

```
umask g+r
```

# du

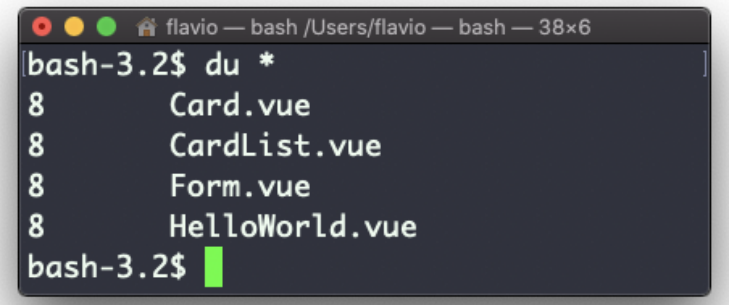
The `du` command will calculate the size of a directory as a whole:

```
du
```



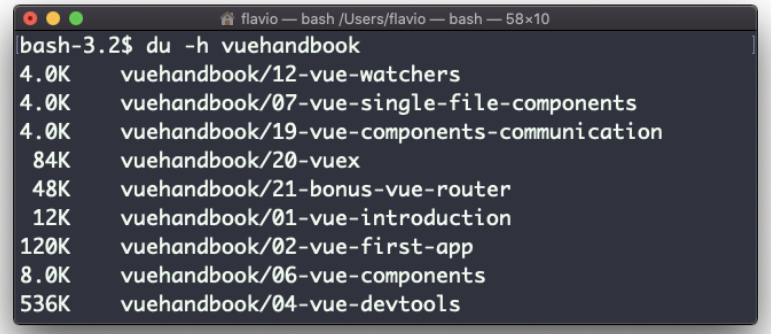

The `32` number here is a value expressed in bytes.

Running `du *` will calculate the size of each file individually:

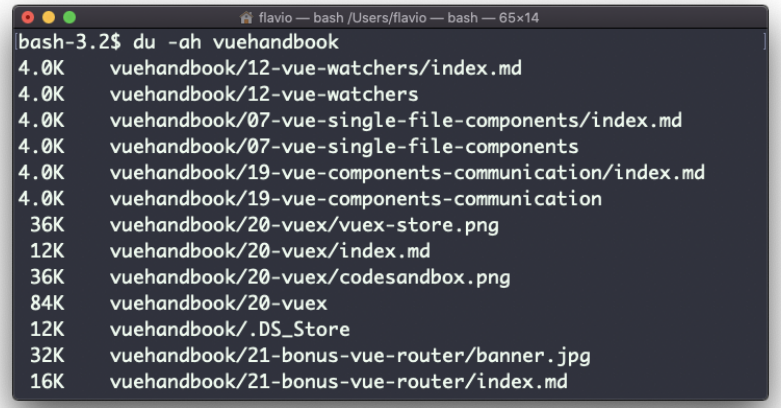



You can set `du` to display values in MegaBytes using `du -m`, and GigaBytes using `du -g`.

The `-h` option will show a human-readable notation for sizes, adapting to the size:



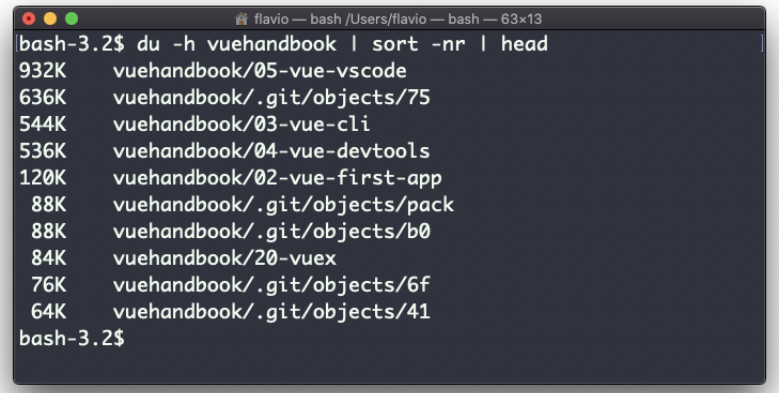

Adding the `-a` option will print the size of each file in the directories, too:




A handy thing is to sort the directories by size:

```
du -h <directory> | sort -nr
```

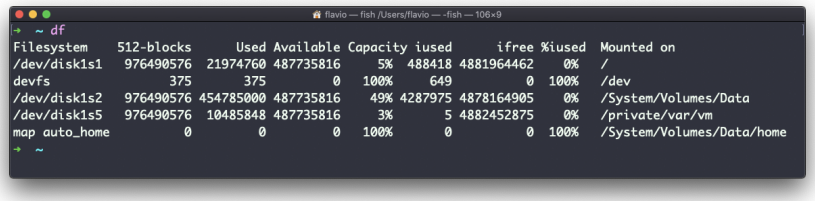
and then piping to `head` to only get the first 10 results:


flavio — bash /Users/flavio — bash — 63×13
bash-3.2$ du -h vuehandbook | sort -nr | head
932K	vuehandbook/05-vue-vscode
636K	vuehandbook/.git/objects/75
544K	vuehandbook/03-vue-cli
536K	vuehandbook/04-vue-devtools
120K	vuehandbook/02-vue-first-app
 88K	vuehandbook/.git/objects/pack
 88K	vuehandbook/.git/objects/b0
 84K	vuehandbook/20-vuex
 76K	vuehandbook/.git/objects/6f
 64K	vuehandbook/.git/objects/41
bash-3.2$

# df

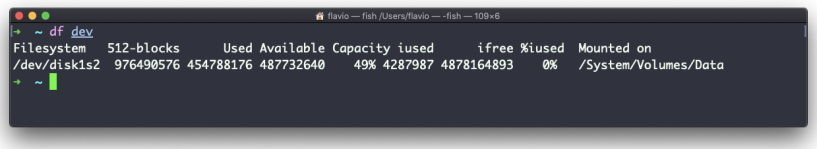
The `df` command is used to get disk usage information.

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:



# basename

Suppose you have a path to a file, for example `/Users/flavio/test.txt`.

Running

```
basename /Users/flavio/test.txt
```

will return the `text.txt` string:

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:

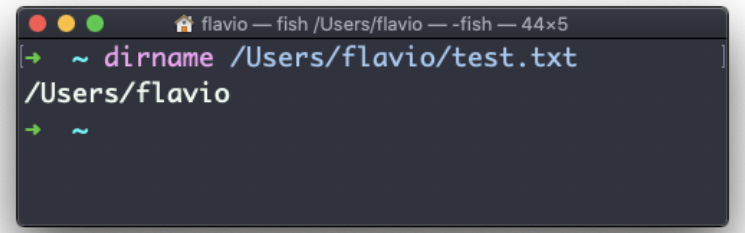
# dirname

Suppose you have a path to a file, for example `/Users/flavio/test.txt`.

Running

```
dirname /Users/flavio/test.txt
```

will return the `/Users/flavio` string:



# ps

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

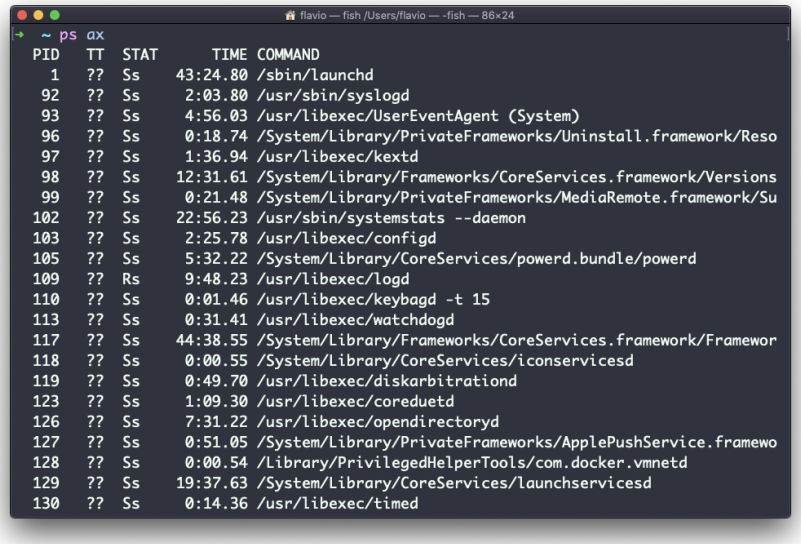
You can inspect them all using the `ps` command:

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.

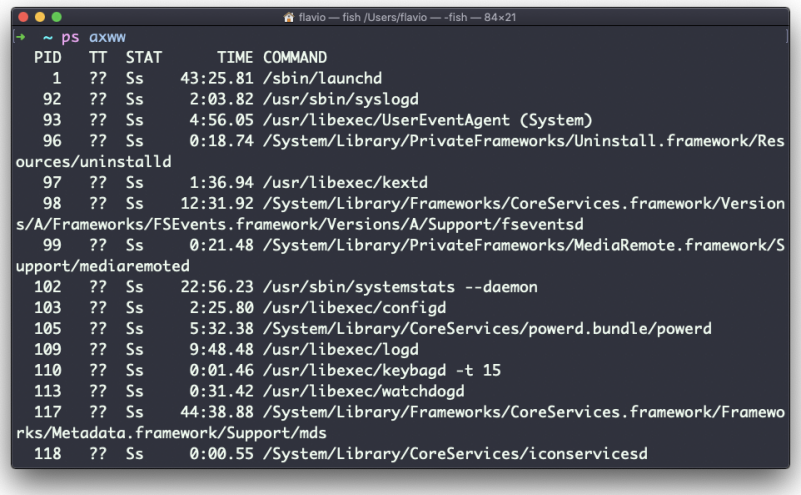
Those are just the commands assigned to the current user. To list **all** processes we need to pass some options to `ps`.

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).

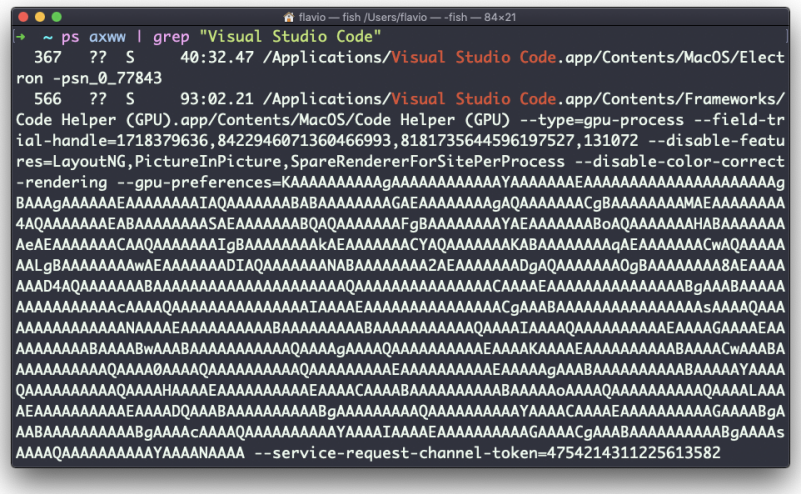
As you can see, the longer commands are cut. Use the command `ps axww` to continue the command listing on a new line instead of cutting it:




> 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 "Visual Studio Code"
```


flavio — fish /Users/flavio — -fish — 84×21
~ ps axww | grep "Visual Studio Code"
367 ?? S 40:32.47 /Applications/Visual Studio Code.app/Contents/MacOS/Electron -psn_0_77843
566 ?? S 93:02.21 /Applications/Visual Studio Code.app/Contents/Frameworks/Code Helper (GPU).app/Contents/MacOS/Code Helper (GPU) --type=gpu-process --field-trial-handle=1718379636,8422946071360466993,8181735644596197527,131072 --disable-features=LayoutNG,PictureInPicture,SpareRendererForSitePerProcess --disable-color-correct-rendering --gpu-preferences=KAAAAAAAAAgAAAAAAAAAAAAYAAAAAAAEAAAAAAAAAAAAAAAAAAAAAgBAAAgAAAAAEAAAAAAAAIAQAAAAAAABABAAAAAAAAGAEAAAAAAAAgAQAAAAAACgBAAAAAAAAAMAEAAAAAAAA4AQAAAAAAAEABAAAAAAAASAEAAAAAAABQAQAAAAAAAFgBAAAAAAAAYAEAAAAAAABoAQAAAAAAAHABAAAAAAAAeAEAAAAAAACAAQAAAAAAAIgBAAAAAAAAkAEAAAAAACYAQAAAAAAAKABAAAAAAAAqAEAAAAAACwAQAAAAAALgBAAAAAAAwAEAAAAAADIAQAAAAAANABAAAAAAA2AEAAAAAADgAQAAAAAAOgBAAAAAAA8AEAAAAAAD4AQAAAAAABAAAAAAAAAAAAAAAAQAAAAAAAAAAAAAACAAAAEAAAAAAAAAAAAAAAAAAA

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.

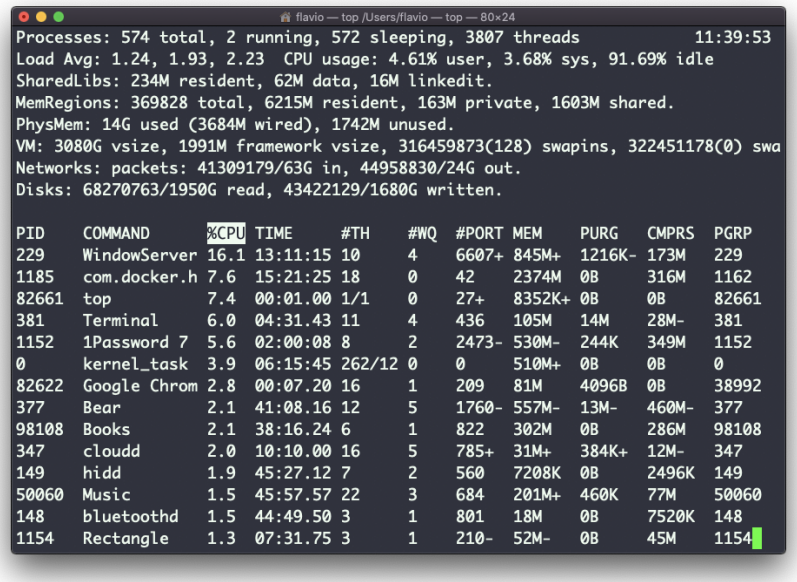
# top

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
```

# kill

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`.

# killall

Similar to the `kill` command, `killall` instead of sending a signal to a specific process id will send the signal to multiple processes at once.

This is the syntax:

```
killall <name>
```

where `name` is the name of a program. For example you can have multiple instances of the `top` program running, and `killall top` will terminate them all.

You can specify the signal, like with `kill` (and check the `kill` tutorial to read more about the specific kinds of signals we can send), for example:

```
killall -HUP top
```

# jobs

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:

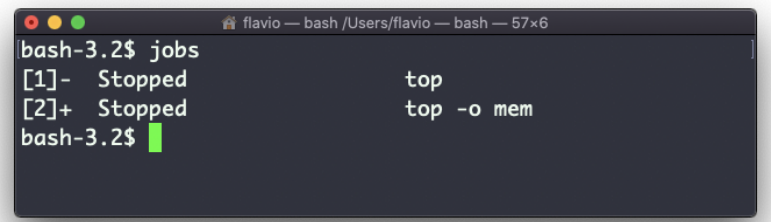
```
top &
```

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.

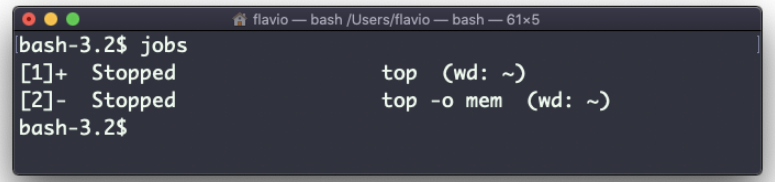
# bg

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:




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.

# fg

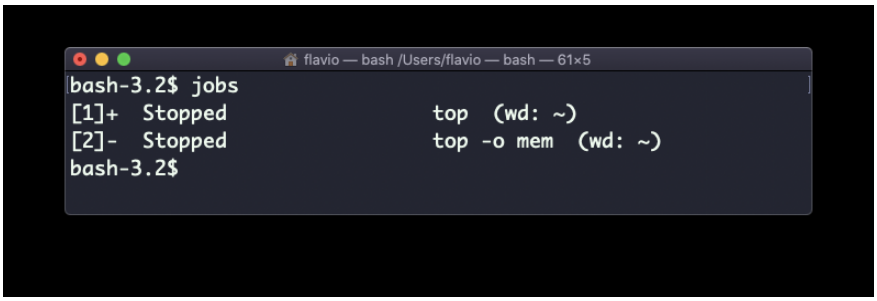
When a command is running in the background, because you started it with `&` at the end (example: `top &` or because you put it in the background with the `bg` command, you can put it to the foreground using `fg`.

Running

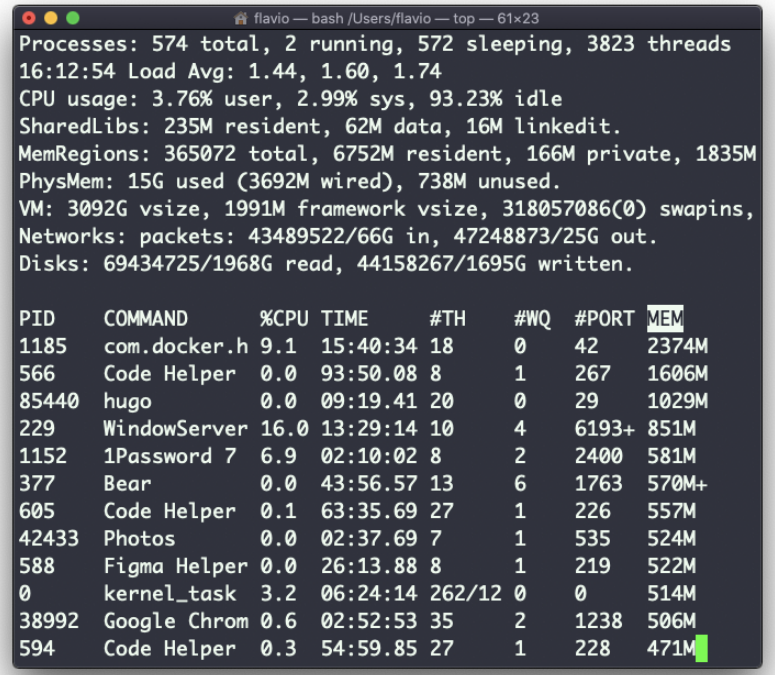
```
fg
```

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:


flavio — bash /Users/flavio — top — 61×23
Processes: 574 total, 2 running, 572 sleeping, 3823 threads
16:12:54 Load Avg: 1.44, 1.60, 1.74
CPU usage: 3.76% user, 2.99% sys, 93.23% idle
SharedLibs: 235M resident, 62M data, 16M linkedit.
MemRegions: 365072 total, 6752M resident, 166M private, 1835M
PhysMem: 15G used (3692M wired), 738M unused.
VM: 3092G vsize, 1991M framework vsize, 318057086(0) swapins,
Networks: packets: 43489522/66G in, 47248873/25G out.
Disks: 69434725/1968G read, 44158267/1695G written.
PID COMMAND %CPU TIME #TH #WQ #PORT MEM
1185 com.docker.h 9.1 15:40:34 18 0 42 2374M
566 Code Helper 0.0 93:50.08 8 1 267 1606M
85440 hugo 0.0 09:19.41 20 0 29 1029M
229 WindowServer 16.0 13:29:14 10 4 6193+ 851M
1152 1Password 7 6.9 02:10:02 8 2 2400 581M
377 Bear 0.0 43:56.57 13 6 1763 570M+
605 Code Helper 0.1 63:35.69 27 1 226 557M
42433 Photos 0.0 02:37.69 7 1 535 524M
588 Figma Helper 0.0 26:13.88 8 1 219 522M
0 kernel_task 3.2 06:24:14 262/12 0 0 514M
38992 Google Chrom 0.6 02:52:53 35 2 1238 506M
594 Code Helper 0.3 54:59.85 27 1 228 471M

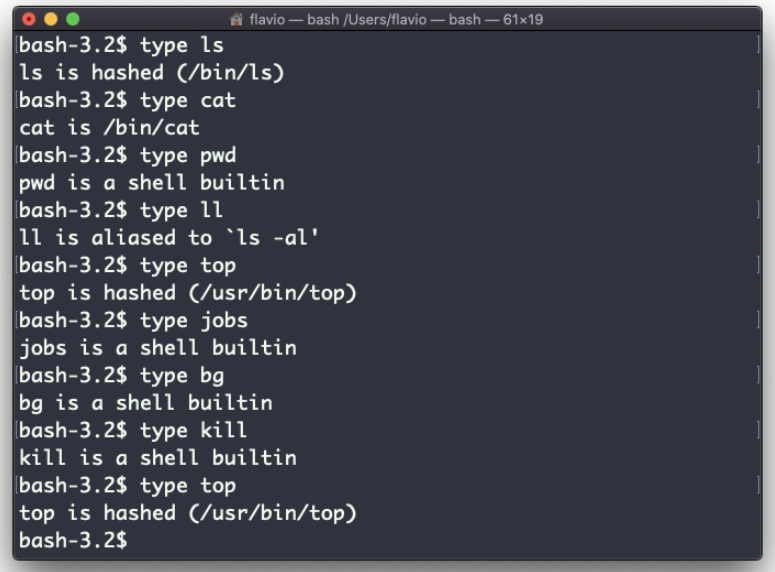
# type

A command can be one of those 4 types:

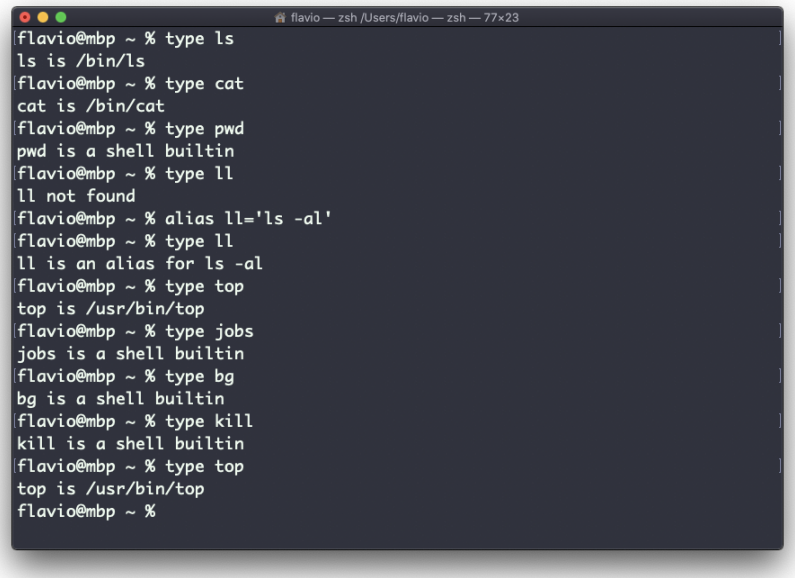
- an executable
- a shell built-in program
- a shell function
- an alias

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.

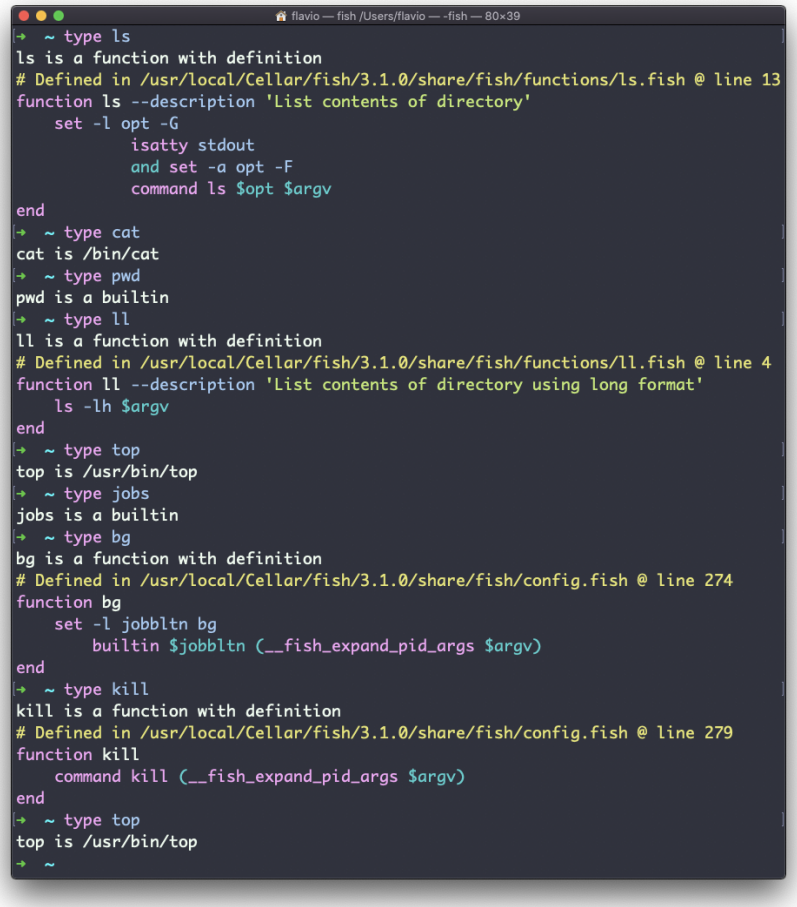
The output will depend on the shell used. This is Bash:




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:

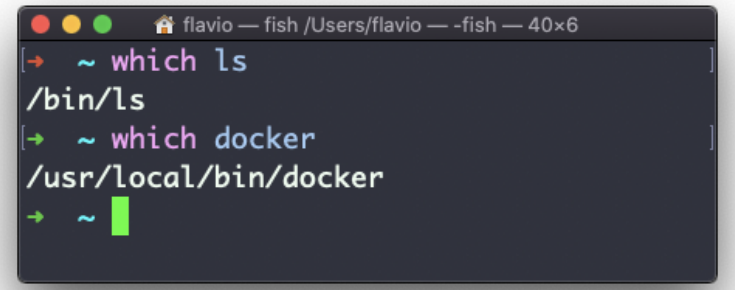

flavio — fish /Users/flavio — -fish — 80×39
➜ ~ type ls
ls is a function with definition
# Defined in /usr/local/Cellar/fish/3.1.0/share/fish/functions/ls.fish @ line 13
function ls --description 'List contents of directory'
set -l opt -G
isatty stdout
and set -a opt -F
command ls $opt $argv
end
➜ ~ type cat
cat is /bin/cat
➜ ~ type pwd
pwd is a builtin
➜ ~ type ll
ll is a function with definition
# Defined in /usr/local/Cellar/fish/3.1.0/share/fish/functions/ll.fish @ line 4
function ll --description 'List contents of directory using long format'
ls -lh $argv
end
➜ ~ type top
top is /usr/bin/top
➜ ~ type jobs
jobs is a builtin
➜ ~ type bg
bg is a function with definition
# Defined in /usr/local/Cellar/fish/3.1.0/share/fish/config.fish @ line 274
function bg
set -l jobbltn bg
builtin $jobbltn (__fish_expand_pid_args $argv)
end
➜ ~ type kill
kill is a function with definition
# Defined in /usr/local/Cellar/fish/3.1.0/share/fish/config.fish @ line 279
function kill
command kill (__fish_expand_pid_args $argv)
end
➜ ~ type top
top is /usr/bin/top
➜ ~

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.

# which

Suppose you have a command you can execute, because it's in the shell path, but you want to know where it is located.

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




`which` will only work for executables stored on disk, not aliases or built-in shell functions.

# nohup

Sometimes you have to run a long-lived process on a remote machine, and then you need to disconnect.

Or you simply want to prevent the command to be halted if there's any network issue between you and the server.

The way to make a command run even after you log out or close the session to a server is to use the `nohup` command.

Use `nohup <command>` to let the process continue working even after you log out.

# xargs

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:

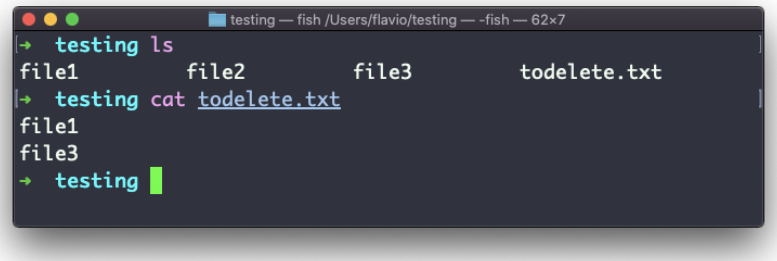
```
command1 | xargs command2
```

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.

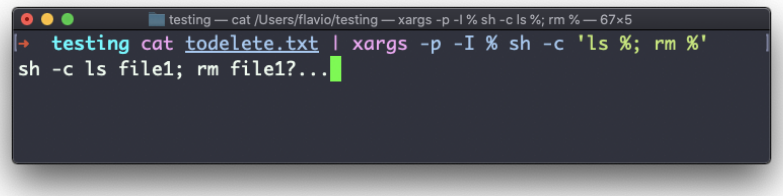
One of the most useful in my opinion, especially when starting to learn `xargs`, is `-p`. Using this option will make `xargs` print a confirmation prompt with the action it's going to take:

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 %; comm
```



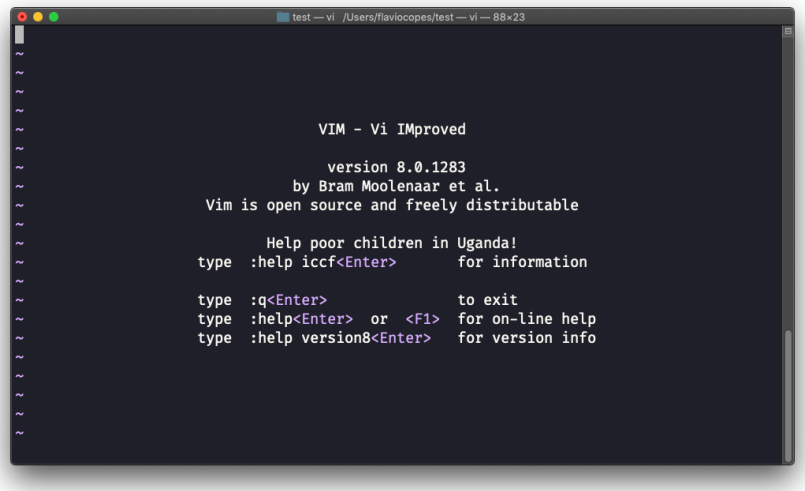

> You can swap the `%` symbol I used above with anything else, it's a variable

# vim

`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!

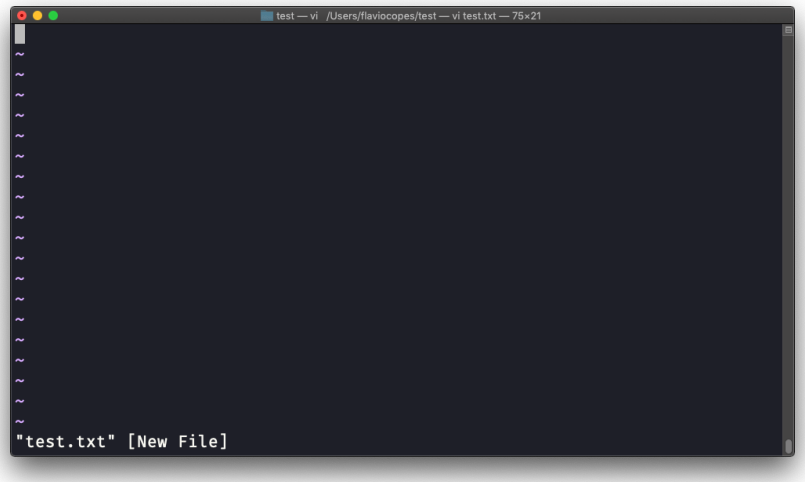
`vi` in modern systems is just an alias to `vim`, which means `vi` i `m` proved.

You start it by running `vi` on the command line.




You can specify a filename at invocation time to edit that specific file:

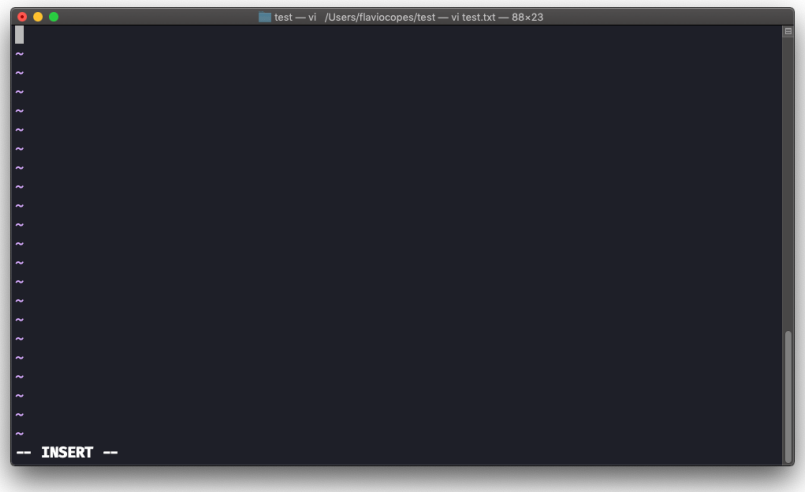
```
vi test.txt
```



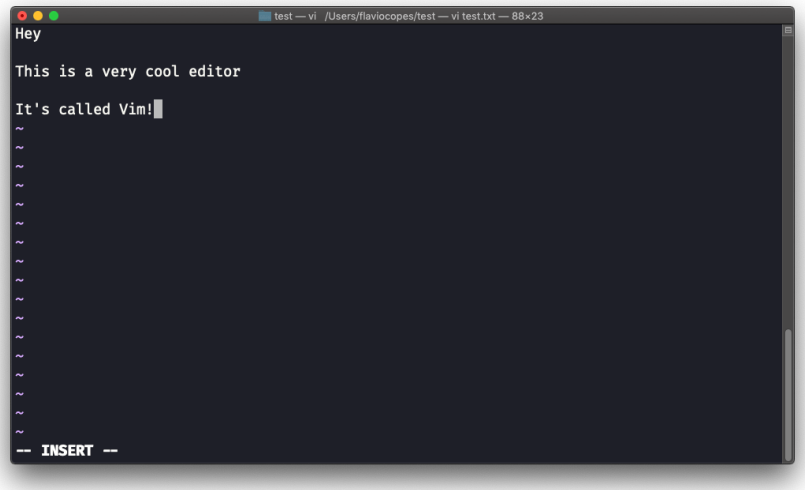

You have to know that Vim has 2 main modes:

- *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:

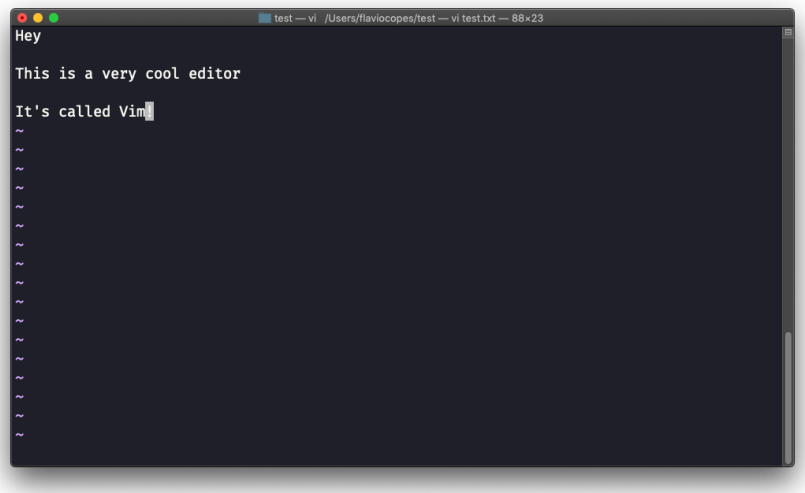



Now you can start typing and filling the screen with the file contents:




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

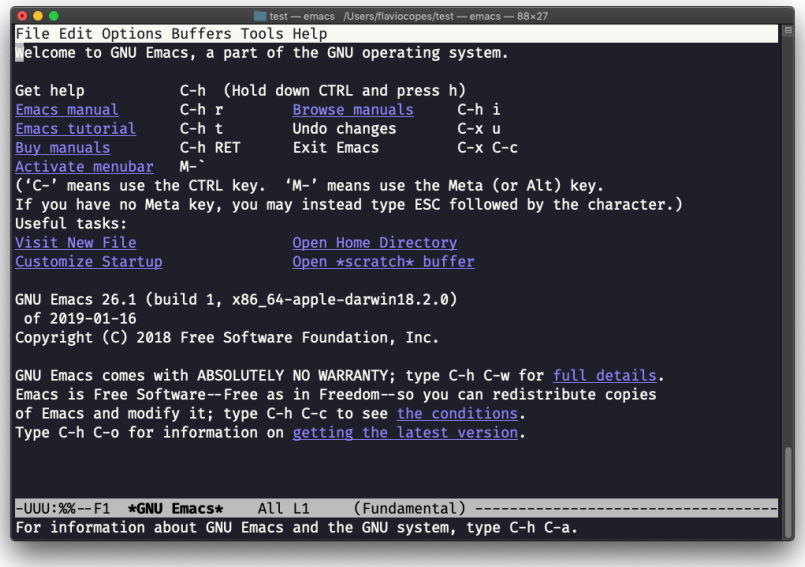
will greatly help you start your `vim` explorations.

# emacs

`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.

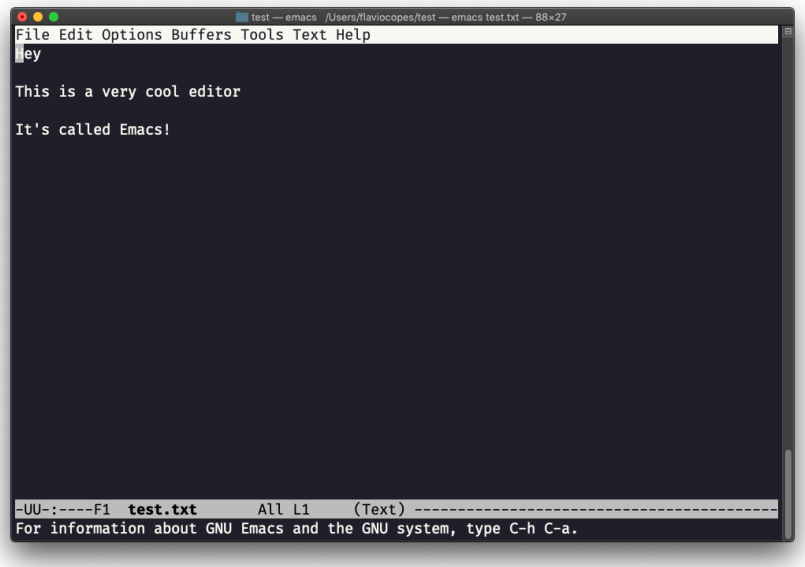
`emacs` is very powerful. Some people use it all day long as a kind of operating system (https://news.ycombinator.com/item?id=19127258). We'll just talk about the basics here.

You can open a new emacs session simply by invoking `emacs`:

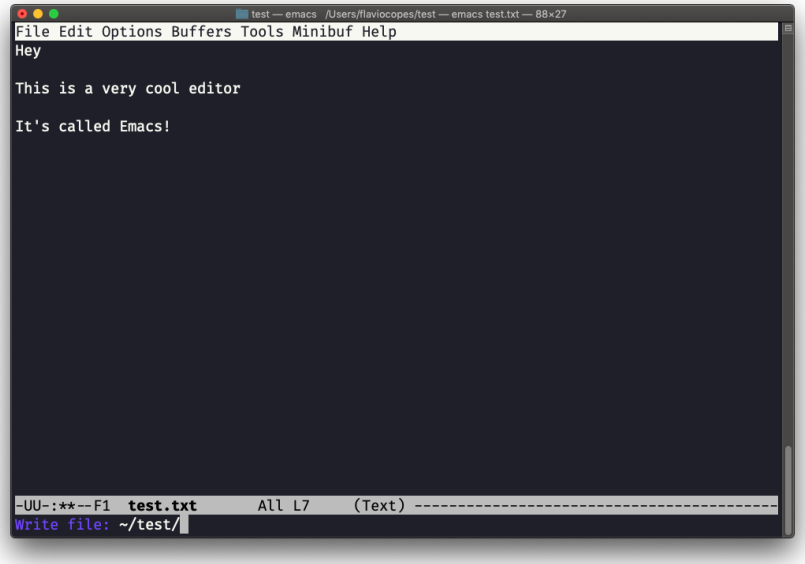


> macOS users, stop a second now. If you are on Linux there are no problems, but macOS does not ship applications using GPLv3, and every built-in UNIX command that has been updated to GPLv3 has not been updated. While there is a little problem with the commands I listed up to now, in this case using 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)

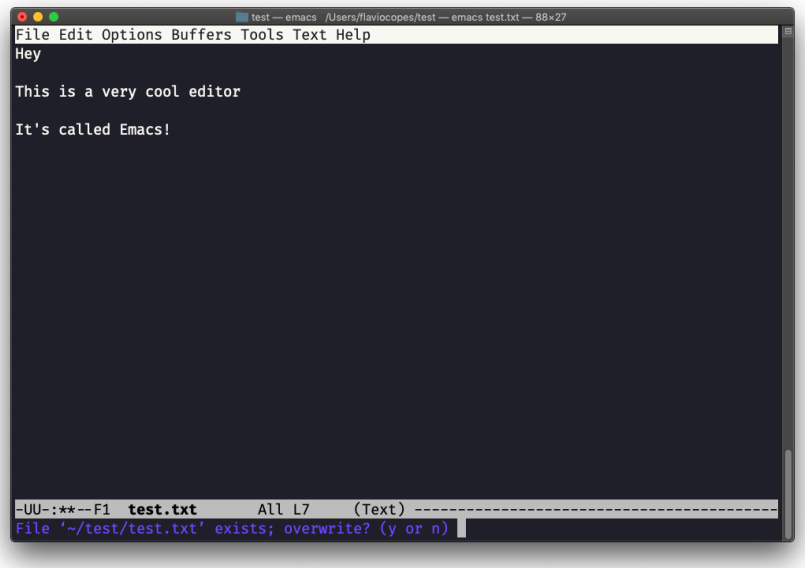
You can also edit an existing file calling `emacs <filename>`:



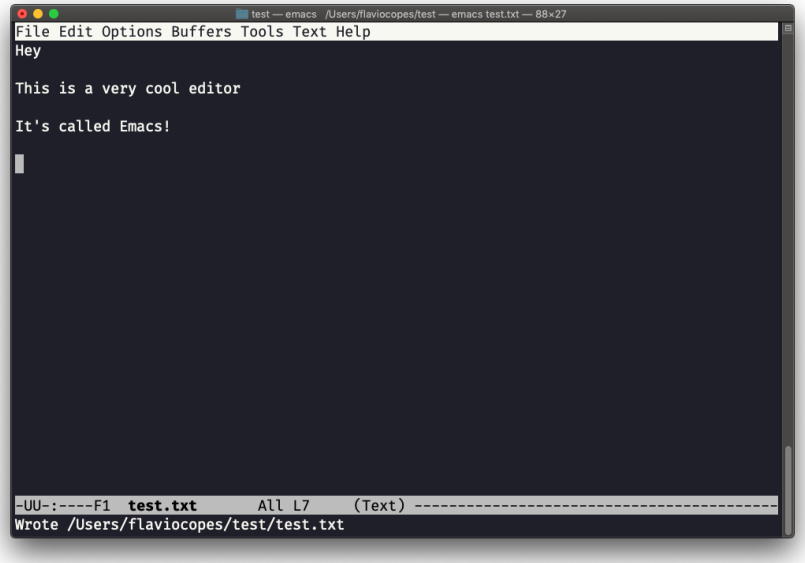

You can start editing and once you are done, press `ctrl-x` followed by `ctrl-w`. You confirm the folder:




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.

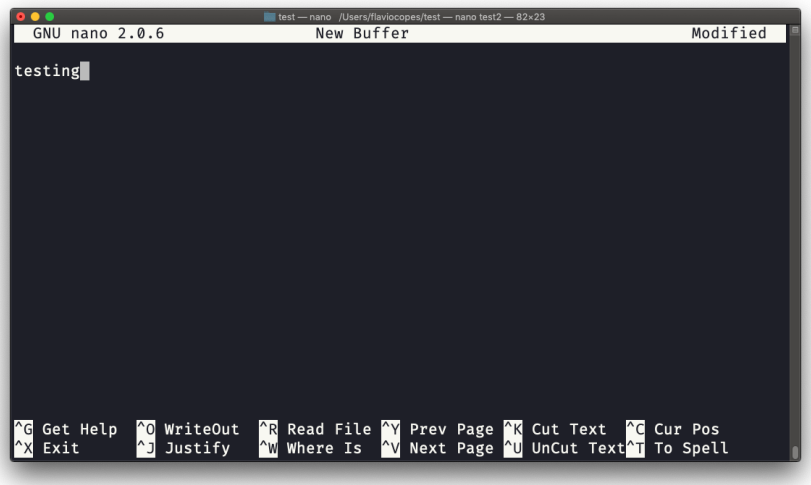
# nano

`nano` is a beginner friendly editor.

Run it using `nano <filename>`.

You can directly type characters into the file without worrying about modes.

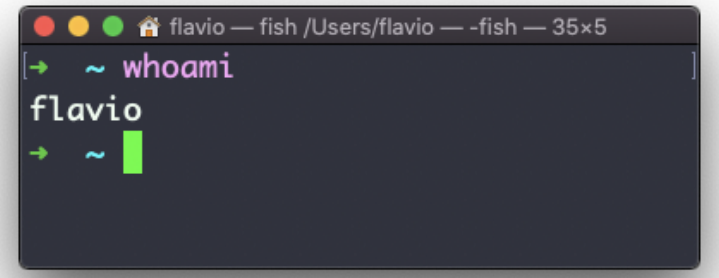
You can quit without editing using `ctrl-X`. If you edited the file buffer, the editor will ask you for confirmation and you can save the edits, or discard them. The help at the bottom shows you the keyboard commands that let you work with the file:




`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.

# whoami

Type `whoami` to print the user name currently logged in to the terminal session:

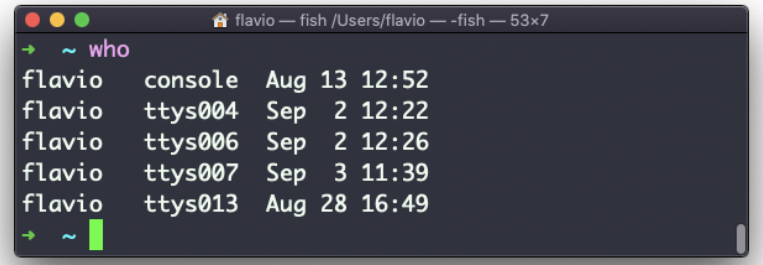



> Note: this is different from the `who am i` command, which prints more information

# who

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

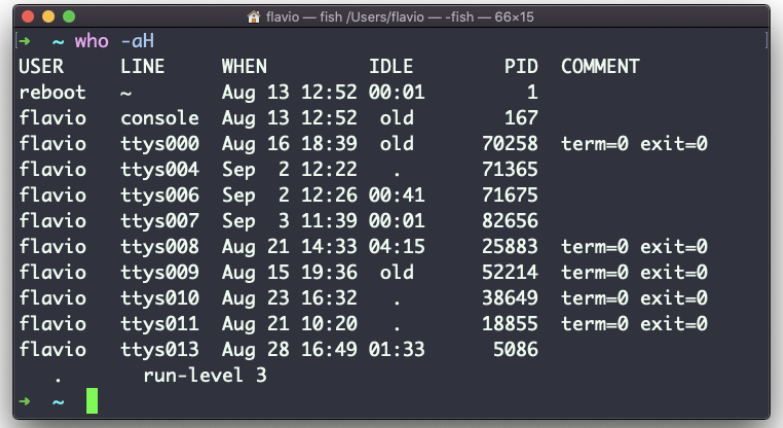
Unless you're using a server multiple people have access to, chances are you will be the only user logged in, multiple times:



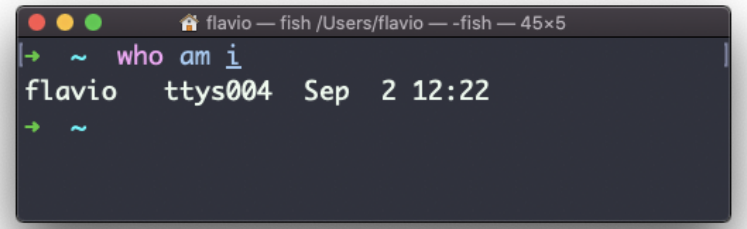

Why multiple times? Because each shell opened will count as an access.

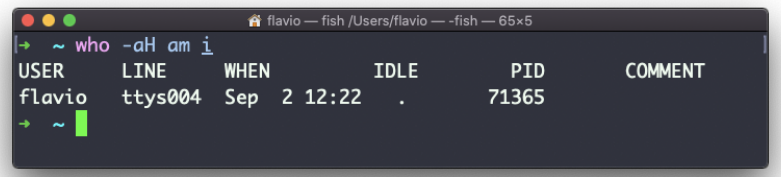
You can see the name of the terminal used, and the time/day the session was started.

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:



# su

While you're logged in to the terminal shell with one user, you might have the need to switch to another user.

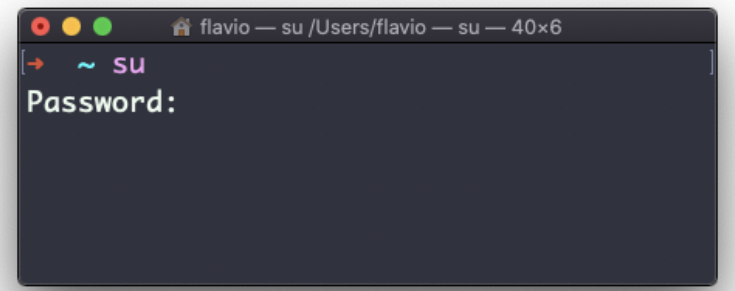
For example you're logged in as root to perform some maintenance, but then you want to switch to a user account.

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.

# sudo

`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).

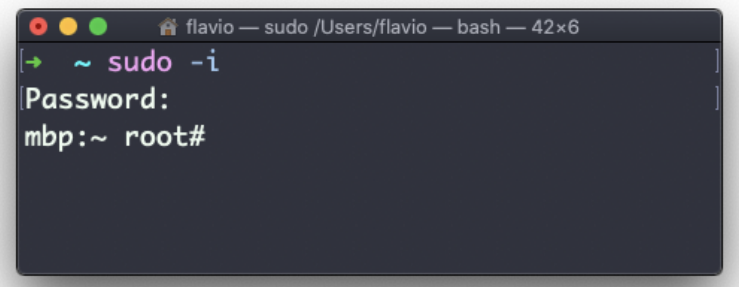
The permissions are highly configurable, which is great especially in a multi-user server environment, and some users can be granted access to running specific commands through `sudo`.

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
```

# passwd

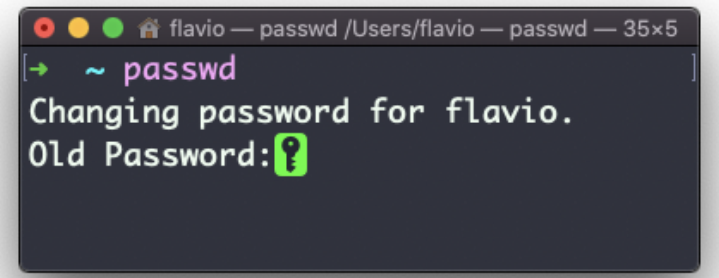
Users in Linux have a password assigned. You can change the password using the `passwd` command.

There are two situations here.

The first is when you want to change your password. In this case you type:

```
passwd
```

and an interactive prompt will ask you for the old password, then it will ask you for the new one:




When you're `root` (or have superuser privileges) you can set the username of which you want to change the password:

```
passwd <username> <new password>
```

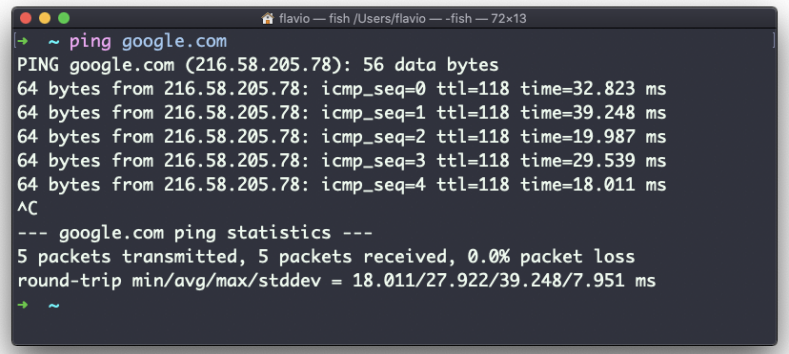
In this case you don't need to enter the old one.

# ping

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`:



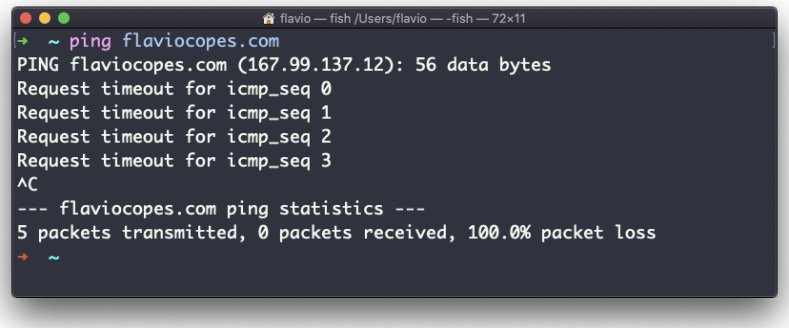

The commands sends a request to the server, and the server returns a response.

`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.

# traceroute

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?

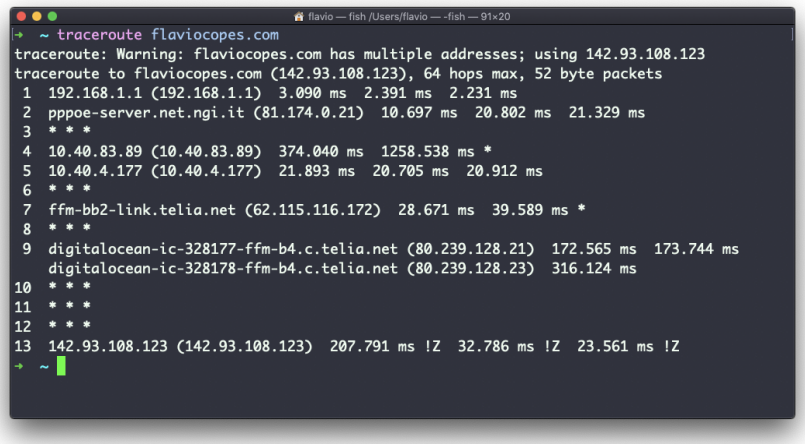
The `traceroute` command is made for this.

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`:

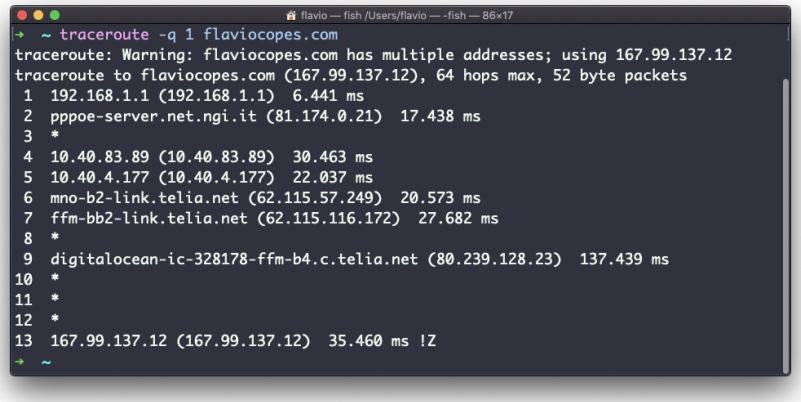


Not every router travelled returns us information. In this case, `traceroute` prints `* * *`. Otherwise, we can see the hostname, the IP address, and some performance indicator.

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
```



# clear

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:

> Note: this command has a handy shortcut: `ctrl-L`

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.

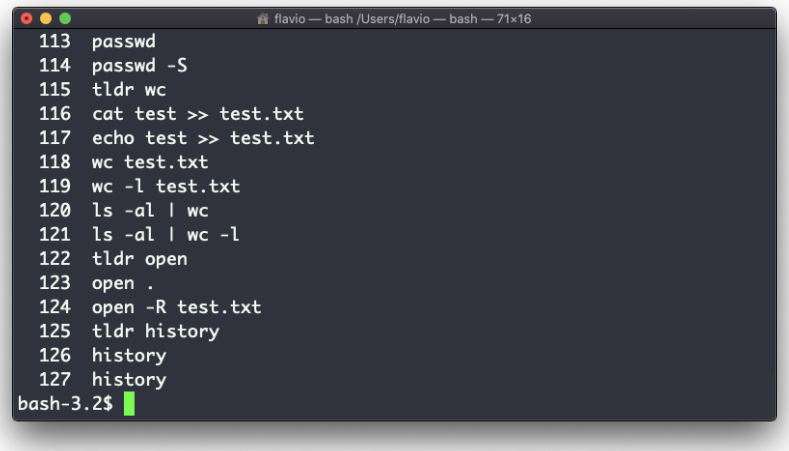
# history

Every time we run a command, that's memorized in the history.

You can display all the history using:

```
history
```

This shows the history with numbers:

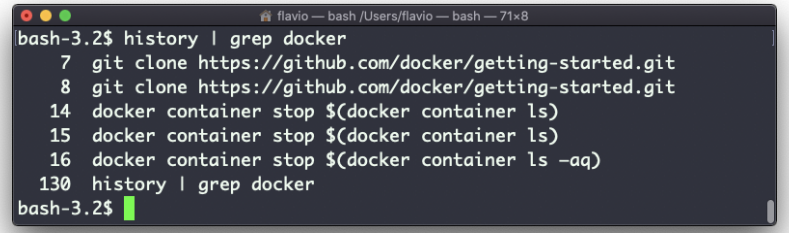



You can use the syntax `!<command number>` to repeat a command stored in the history, in the above example typing `!121` will repeat the `ls -al | wc -l` command.

Typically the last 500 commands are stored in the history.

You can combine this with `grep` to find a command you ran:

```
history | grep docker
```




To clear the history, run `history -c`

# export

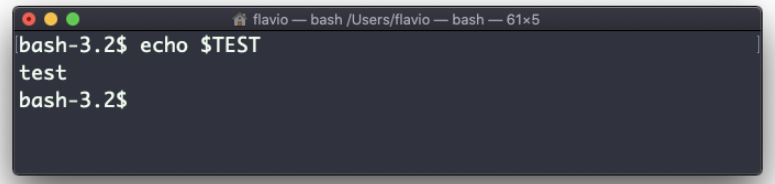
The `export` command is used to export variables to child processes.

What does this mean?

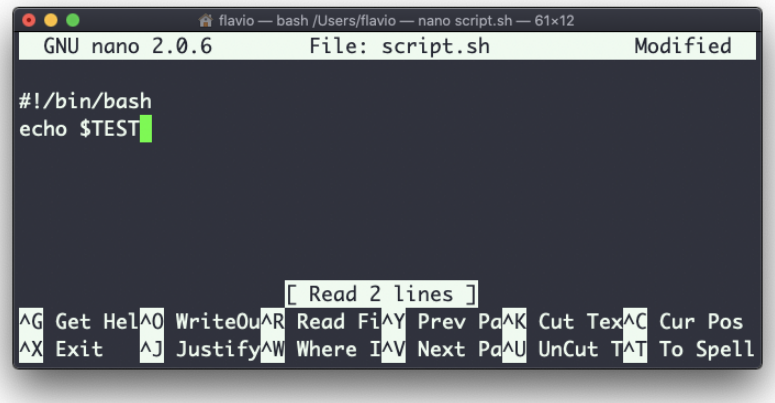
Suppose you have a variable TEST defined in this way:

```
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.

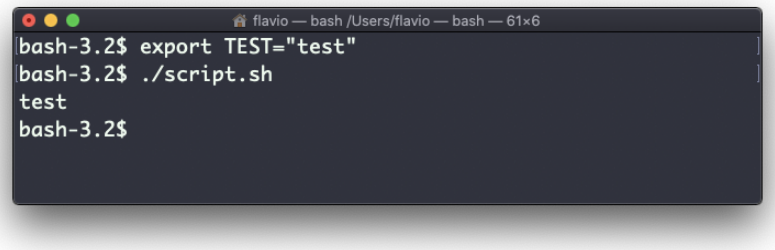
To make the variable available there we need to define `TEST` not in this way:

```
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.

# crontab

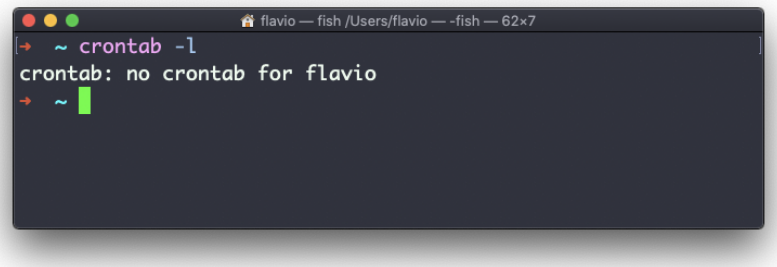
Cron jobs are jobs that are scheduled to run at specific intervals. You might have a command perform something every hour, or every day, or every 2 weeks. Or on weekends. They are very powerful, especially on servers to perform maintenance and automations.

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
```

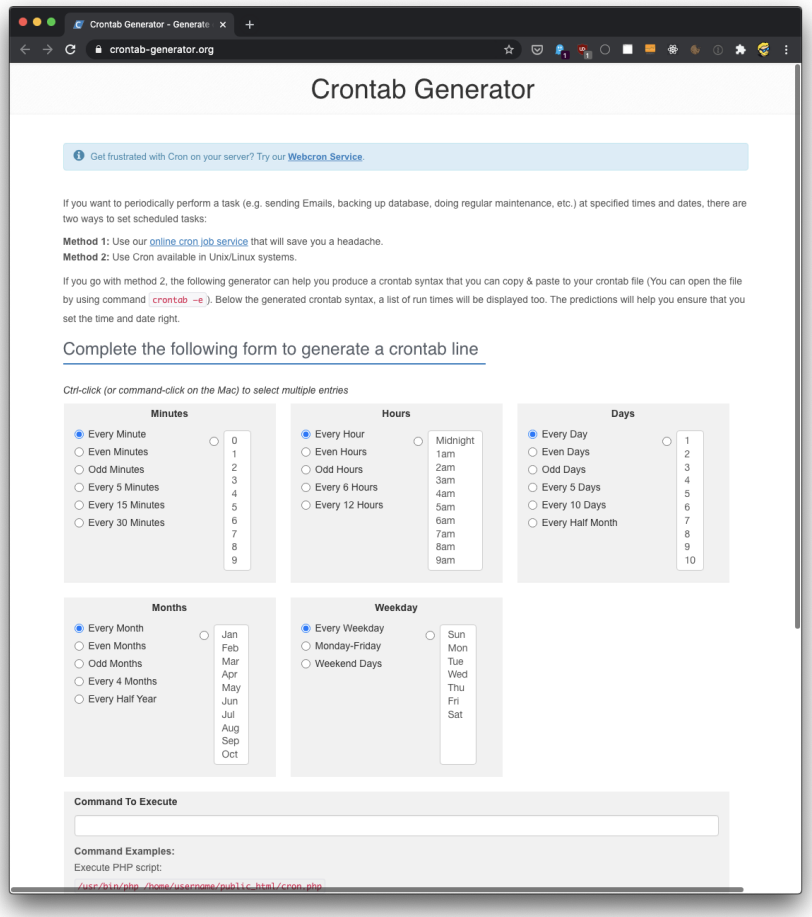
to edit the cron jobs, and add new ones.

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/




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:

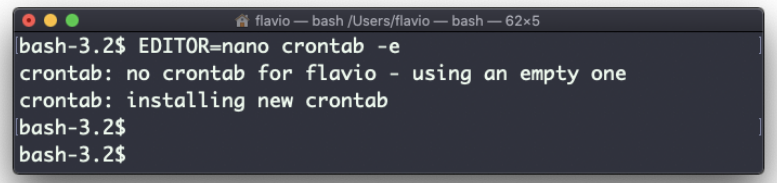
```
* */12 * * * /Users/flavio/test.sh >/dev/null 2>&1
```

I run `crontab -e`:

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

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

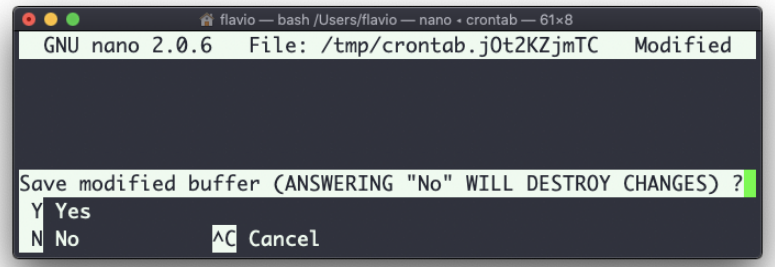
If all goes well, the cron job is set up:

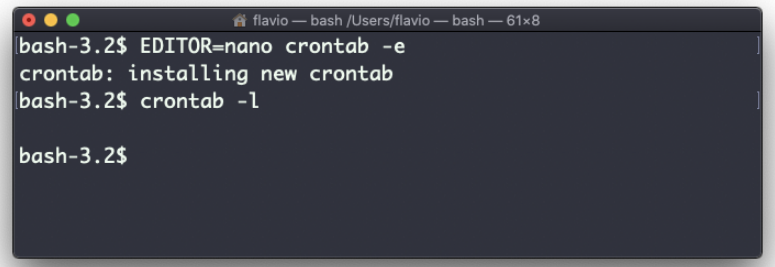



Once this is done, you can see the list of active cron jobs by running:

```
crontab -l
```

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



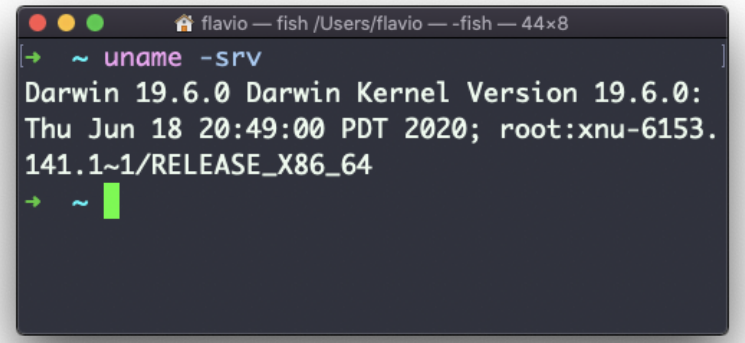



# uname

Calling `uname` without any options will return the Operating System codename:

The `m` option shows the hardware name ( `x86_64` in this example) and the `p` option prints the processor architecture name ( `i386` in this example):

The `s` option prints the Operating System name. `r` prints the release, `v` prints the version:

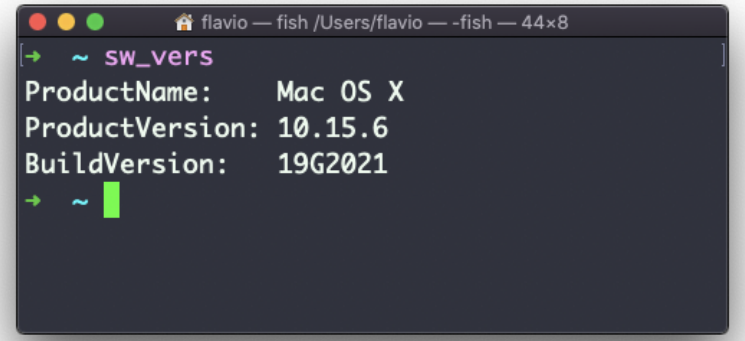



The `n` option prints the node network name:

The `a` option prints all the information available:

On macOS you can also use the `sw_vers` command to print more information about the macOS Operating System. Note that this differs from the Darwin (the Kernel) version, which above is `19.6.0` .

> Darwin is the name of the kernel of macOS. The kernel is the "core" of the Operating System, while the Operating System as a whole is called macOS. In Linux, Linux is the kernel, GNU/Linux would be the Operating System name, although we all refer to it as "Linux"



# env

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:/l
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
```

# printenv

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.

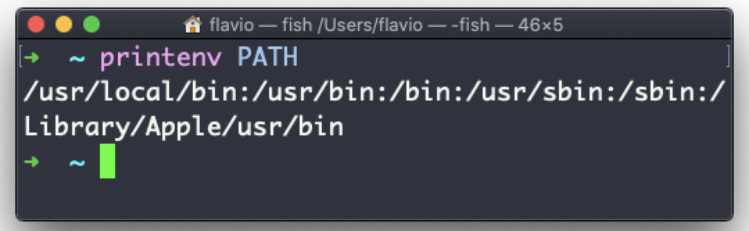
You can print them all to the terminal using the `printenv` command. The output will be something like this:

```
HOME=/Users/flavio
LOGNAME=flavio
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/l
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
```



# Conclusion

Thanks a lot for reading this book.

I hope it will inspire you to know more about Linux and its capabilities.

To learn more, check out my blog flaviocopes.com.

Send any feedback, errata or opinions at hey@flaviocopes.com