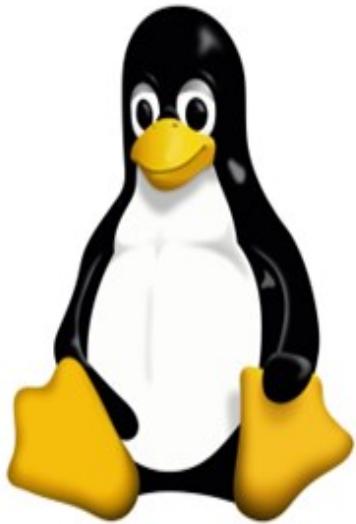


Cyber Public School



**LINUX
COMMANDS**

LINUX



COMMANDS

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.

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.

Introduction To Linux And Shells

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.

Introduction To Linux And Shells

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.

Introduction To Linux And Shells

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.

Introduction To Linux And Shells

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:

```
LS(1)                               BSD General Commands Manual      LS(1)

NAME
  ls -- list directory contents

SYNOPSIS
  ls [-ABCDEFGHJKLMNOPRSTUW@abcdefghijklmnopqrstuvwxyz%] [file ...]

DESCRIPTION
  For each operand that names a file of a type other than directory,
  ls displays its name as well as any requested, associated informa-
  tion.  For each operand that names a file of type directory, ls dis-
  plays the names of files contained within that directory, as well as
  any requested, associated information.

  If no operands are given, the contents of the current directory are
  displayed.  If more than one operand is given, non-directory oper-
  ands are displayed first; directory and non-directory operands are
  sorted separately and in lexicographical order.

  The following options are available:

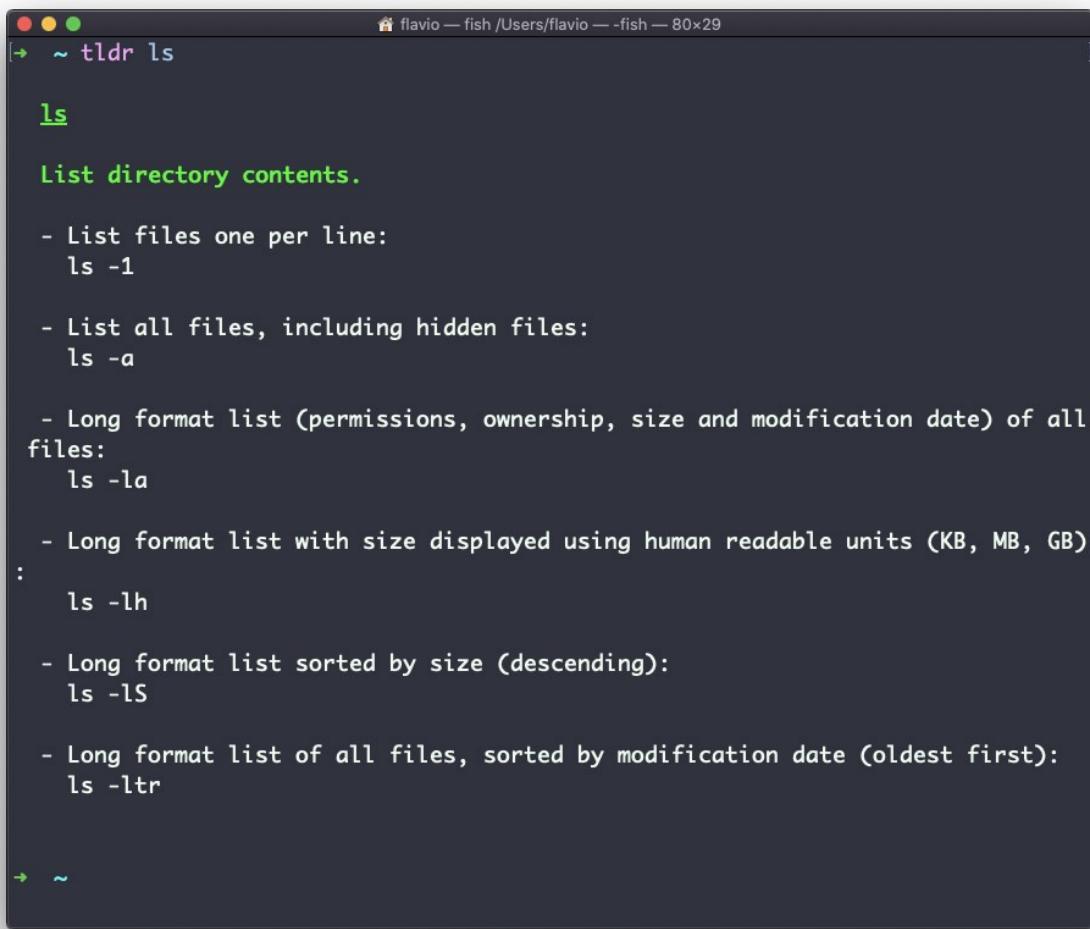
  -@      Display extended attribute keys and sizes in long (-l) out-
          put.

  -1      (The numeric digit ``one''.)  Force output to be one entry
  :
```

This is a man (from manual) page. Man pages are an essential tool to learn, as a developer. They contain so much information that sometimes it's almost too much. The above screenshot is just 1 of 14 screens of explanation for the ls command.

Man

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:



A screenshot of a macOS terminal window titled "flavio — fish /Users/flavio — -fish — 80x29". The command `tldr ls` has been run, and the output is displayed. The output starts with the word `ls` in green, followed by a description: "List directory contents.". Below this, a series of bullet points provide examples of how to use the `ls` command:

- List files one per line:
`ls -1`
- List all files, including hidden files:
`ls -a`
- Long format list (permissions, ownership, size and modification date) of all files:
`ls -la`
- Long format list with size displayed using human readable units (KB, MB, GB):
:
`ls -lh`
- Long format list sorted by size (descending):
`ls -lS`
- Long format list of all files, sorted by modification date (oldest first):
`ls -ltr`

The terminal prompt "`~`" is visible at the bottom.

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
```

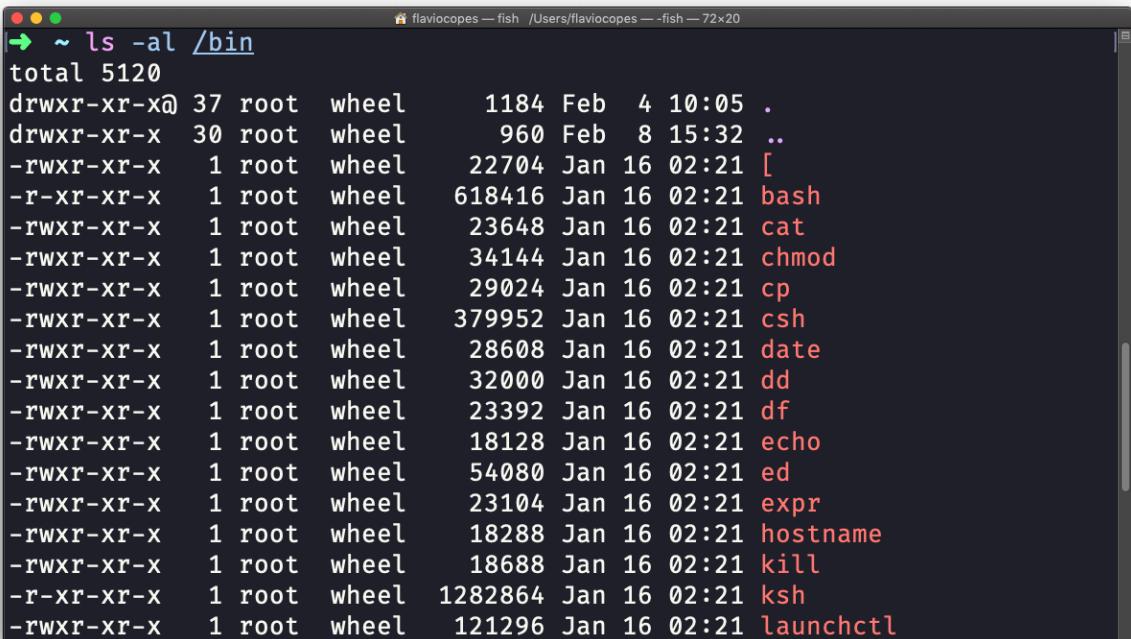


```
flaviocopes — fish /Users/flaviocopes — -fish — 72x9
→ ~ ls /bin
[      csh      ed      launchctl  mv      rmdir      tcsh
bash    date    expr      link      pax      sh      test
cat      dd      hostname   ln      ps      sleep      unlink
chmod   df      kill      ls      pwd      stty      wait4path
cp      echo     ksh      mkdir      rm      sync      zsh
→ ~
```

ls

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

```
ls -al /bin
```

A screenshot of a macOS terminal window titled "flaviocopes — fish — /Users/flaviocopes — fish — 72x20". The command "ls -al /bin" is run, and the output shows a detailed listing of files in the /bin directory. The output includes file permissions (e.g., drwxr-xr-x), link counts (e.g., 37, 30), owners (root), groups (wheel), sizes (e.g., 1184, 960), modification dates (e.g., Feb 4, 2016, 15:32), and file names (e.g., ., .., bash, cat, chmod, cp, csh, date, dd, df, echo, ed, expr, hostname, kill, ksh, launchctl).

```
ls -al /bin
total 5120
drwxr-xr-x@ 37 root wheel      1184 Feb  4 10:05 .
drwxr-xr-x  30 root wheel      960  Feb  8 15:32 ..
-rw xr-xr-x  1 root wheel    22704 Jan 16 02:21 [
-rw xr-xr-x  1 root wheel   618416 Jan 16 02:21 bash
-rw xr-xr-x  1 root wheel   23648 Jan 16 02:21 cat
-rw xr-xr-x  1 root wheel   34144 Jan 16 02:21 chmod
-rw xr-xr-x  1 root wheel   29024 Jan 16 02:21 cp
-rw xr-xr-x  1 root wheel  379952 Jan 16 02:21 csh
-rw xr-xr-x  1 root wheel   28608 Jan 16 02:21 date
-rw xr-xr-x  1 root wheel   32000 Jan 16 02:21 dd
-rw xr-xr-x  1 root wheel   23392 Jan 16 02:21 df
-rw xr-xr-x  1 root wheel   18128 Jan 16 02:21 echo
-rw xr-xr-x  1 root wheel   54080 Jan 16 02:21 ed
-rw xr-xr-x  1 root wheel   23104 Jan 16 02:21 expr
-rw xr-xr-x  1 root wheel   18288 Jan 16 02:21 hostname
-rw xr-xr-x  1 root wheel   18688 Jan 16 02:21 kill
-rw xr-xr-x  1 root wheel 1282864 Jan 16 02:21 ksh
-rw xr-xr-x  1 root wheel 121296 Jan 16 02:21 launchctl
```

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

ls

- 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

Once you have a folder, you can move into it using the cd command. cd

means change directory. You invoke it specifying a folder to move into. You

can specify a folder name, or an entire path.

Example:

```
mkdir fruits  
cd fruits
```

Now you are into the fruits folder.

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

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

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

You can use it to form a path:

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

Cd

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

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

```
cd /etc
```

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

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.

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.

```
touch pear  
mv pear new_pear
```

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

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

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

Cp

You can copy a file using the cp command:

```
touch test  
cp apple another_apple
```

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

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

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

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
```

Find

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.

In

The In 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

```
In <original> <link>
```

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

```
In recipes.txt newrecipes.txt
```

In

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

```
flavio — fish /Users/flavio — -fish — 63x10
|→ ~ ls -al newrecipes.txt
-rw-r--r-- 1 flavio  staff  8 Sep  2 11:25 newrecipes.txt
|→ ~
```

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.

```
flavio — fish /Users/flavio — -fish — 49x9
|→ ~ ln recipes.txt newrecipes.txt
|→ ~ cat newrecipes.txt
recipes
|→ ~ rm recipes.txt
|→ ~ cat newrecipes.txt
recipes
|→ ~
```

In

Soft links

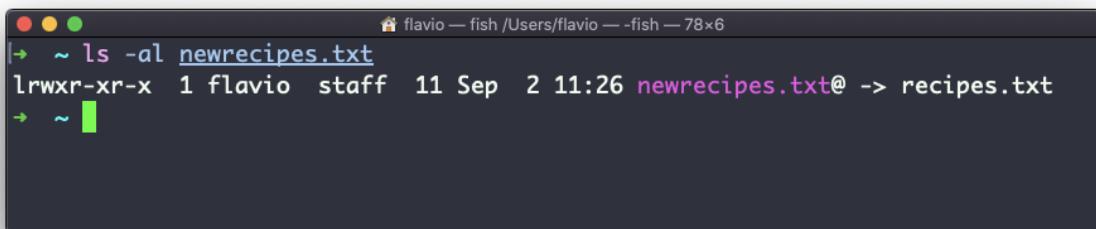
Soft links are different. They are more powerful as you can link to other filesystems and to directories, but when the original is removed, the link will be broken.²⁴ You create soft links using the -s option of ln :

```
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:



```
flavio — fish /Users/flavio — -fish — 78x6
ls -al newrecipes.txt
lrwxr-xr-x 1 flavio  staff  11 Sep  2 11:26 newrecipes.txt@ -> recipes.txt
```

In

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:

```
flavio — fish /Users/flavio — -fish — 63x10
|~ ln -s recipes.txt newrecipes.txt
|~ cat newrecipes.txt
recipes
|~ rm recipes.txt
|~ cat newrecipes.txt
cat: newrecipes.txt: No such file or directory
|~
```

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
```

Gzip

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 - 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
```

Gzip

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



A screenshot of a macOS terminal window. The title bar shows "flavio — fish /Users/flavio — -fish — 58x5". The command entered is `gzip -kv wget-log`. The output shows "wget-log: 49.7% -- replaced with wget-log.gz". The prompt "`~`" is visible at the bottom.

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
```

Tar

You can also just list the files contained in an archive:



```
flavio — fish /Users/flavio — -fish — 55x5
|~ tar -tf archive.tar
file1
file2
~ |
```

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:

```
bash-3.2$ ls
words.txt
bash-3.2$
```

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 .

```
bash-3.2$ ls
words.txt
bash-3.2$ ls -al
total 0
drwxr-xr-x  3 flavi  staff   96 Sep  3 15:20 .
drwxr-xr-x+ 55 flavi  staff  1760 Sep  3 15:20 ..
-rw-r--r--  1 flavi  staff     0 Sep  3 15:20 words.txt
bash-3.2$
```

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

Alias

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:

```
bash-3.2$ alias ll='ls -al'  
bash-3.2$ ll  
total 0  
drwxr-xr-x  3 flavio  staff    96 Sep  3 15:20 .  
drwxr-xr-x+ 55 flavio  staff   1760 Sep  3 15:20 ..  
-rw-r--r--  1 flavio  staff      0 Sep  3 15:20 words.txt  
bash-3.2$
```

Now calling alias without any option will list the aliases defined:

```
bash-3.2$ alias  
alias ll='ls -al'  
bash-3.2$
```

Alias

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
```

Cat

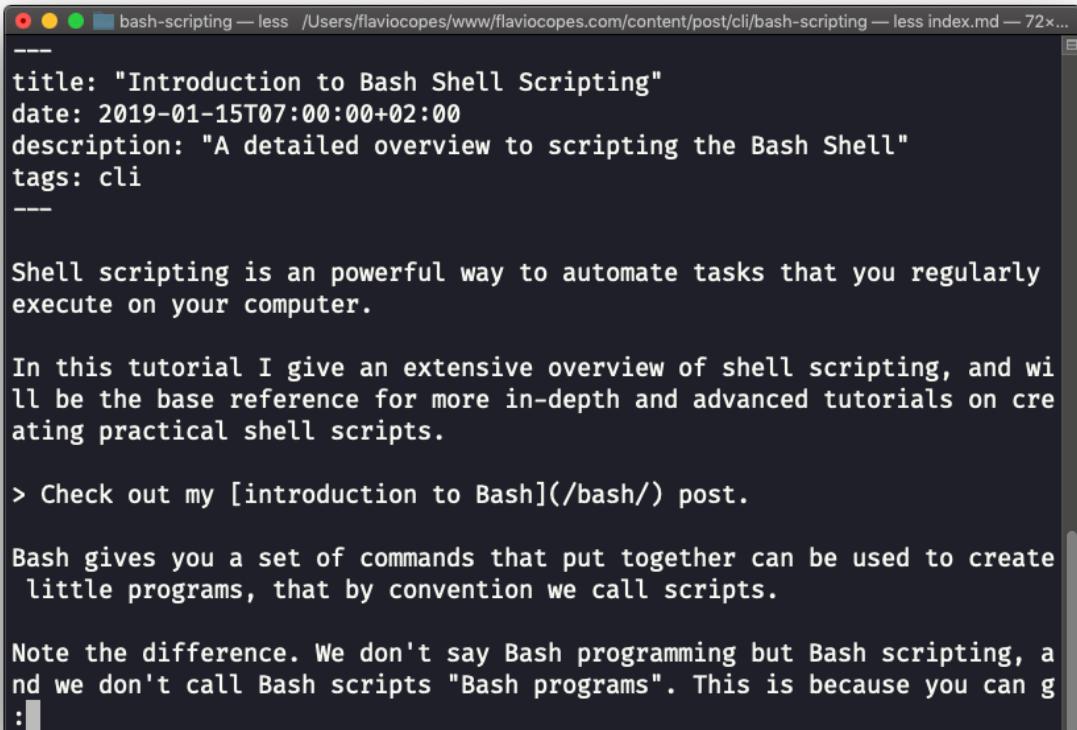
When watching source code files it's great to see the line numbers, and you can have cat print them using the -n option:

```
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 .



A screenshot of a terminal window titled "bash-scripting — less /Users/flaviocopes/www/flaviocopes.com/content/post/cli/bash-scripting — less index.md — 72x...". The window displays the contents of a file named "index.md" using the "less" command. The content includes metadata at the top:

```
title: "Introduction to Bash Shell Scripting"
date: 2019-01-15T07:00:00+02:00
description: "A detailed overview to scripting the Bash Shell"
tags: cli
```

Followed by a section of text:

```
Shell scripting is an powerful way to automate tasks that you regularly execute on your computer.
```

Then a note:

```
In this tutorial I give an extensive overview of shell scripting, and will be the base reference for more in-depth and advanced tutorials on creating practical shell scripts.
```

A link:

```
> Check out my [introduction to Bash](/bash/) post.
```

Another note:

```
Bash gives you a set of commands that put together can be used to create little programs, that by convention we call scripts.
```

A final note:

```
Note the difference. We don't say Bash programming but Bash scripting, and we don't call Bash scripts "Bash programs". This is because you can g
```

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

Less

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 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
```

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

```
tail -n +10
```

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
```

Wc

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 occurrences of the document.getElementById line in the index.md file:

```
grep document.getElementById index.md
```

```
flavio — bash /Users/flavio — bash — 77x7
bash-3.2$ grep document.getElementById index.md
document.getElementById('button').addEventListener('click', () => {
  document.getElementById('button').addEventListener('click', () => {
bash-3.2$
```

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

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

```
flavio — bash /Users/flavio — bash — 77x7
bash-3.2$ grep -n document.getElementById index.md
60:document.getElementById('button').addEventListener('click', () => {
128:  document.getElementById('button').addEventListener('click', () => {
bash-3.2$
```

Grep

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
```

```
flavio — bash /Users/flavio — bash — 76x16
bash-3.2$ grep -nC 2 document.getElementById index.md
58-
59-```js
60:document.getElementById('button').addEventListener('click', () => {
61-   //item clicked
62-})
-- 
126-```js
127>window.addEventListener('load', () => {
128:   document.getElementById('button').addEventListener('click', () => {
129-     setTimeout(() => {
130-       items.forEach(item => {
bash-3.2$
```

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
```

Grep

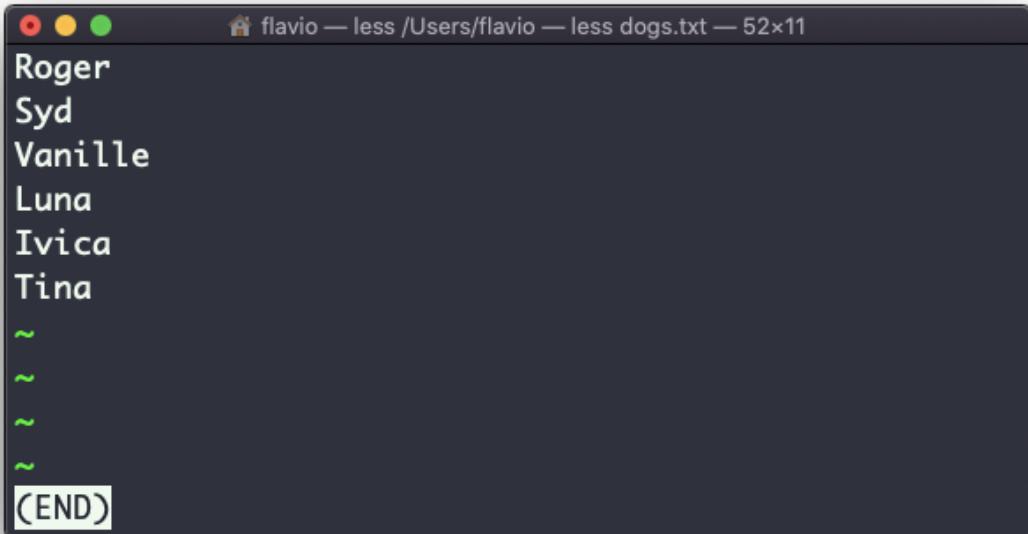
```
flavio — bash /Users/flavio — bash — 77x7
bash-3.2$ less index.md | grep -n document.getElementById
60:document.getElementById('button').addEventListener('click', () => {
128:   document.getElementById('button').addEventListener('click', () => {
bash-3.2$
```

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:

```
flavio — bash /Users/flavio — bash — 66x12
bash-3.2$ ls -al
total 120
drwxr-xr-x    4 flavio  staff   128 Apr  3 08:15 .
drwxr-xr-x  138 flavio  staff  4416 Aug 22 14:58 ..
-rw-r--r--    1 flavio  staff  49556 Apr  3 09:37 banner.png
-rw-r--r--    1 flavio  staff   5659 Feb 11  2020 index.md
bash-3.2$ ls -al | grep -v index.md
total 120
drwxr-xr-x    4 flavio  staff   128 Apr  3 08:15 .
drwxr-xr-x  138 flavio  staff  4416 Aug 22 14:58 ..
-rw-r--r--    1 flavio  staff  49556 Apr  3 09:37 banner.png
bash-3.2$
```

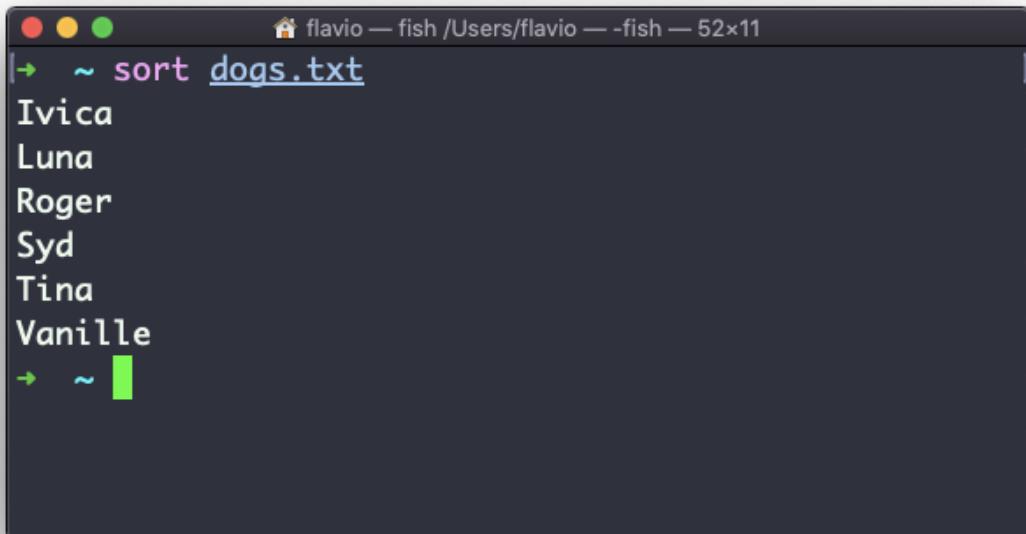
Sort

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



```
Roger
Syd
Vanille
Luna
Ivica
Tina
~  
~  
~  
~  
(END)
```

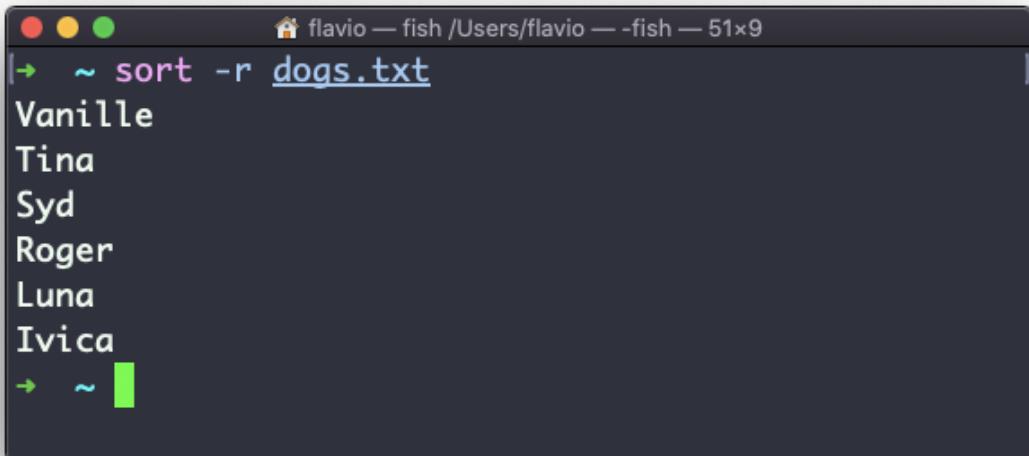
This list is unordered. The sort command helps us sorting them by name:



```
~ sort dogs.txt
Ivica
Luna
Roger
Syd
Tina
Vanille
~
```

Sort

Use the r option to reverse the order:



```
fladio — fish /Users/fladio — -fish — 51x9
~ sort -r dogs.txt
Vanille
Tina
Syd
Roger
Luna
Ivica
~ ~ |
```

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:



```
fladio — nano /Users/fladio — nano dogs.txt — 60x15
GNU nano 2.0.6          File: dogs.txt          Modified

Roger
Syd
Vanille
Luna
Ivica
Tina
Roger
Syd

[ Cancelled ]
^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit      ^J Justify ^W Where ^V Next Page ^U Uncut ^T To SpellCheck
```

Sort

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



A screenshot of a macOS terminal window titled "flavio — fish /Users/flavio — -fish — 40x8". The command entered is `sort -u dogs.txt`. The output shows the names of the dogs listed vertically: Ivica, Luna, Roger, Syd, Tina, and Vanille. The terminal has a dark theme with red, yellow, and green window controls.

```
flavio — fish /Users/flavio — -fish — 40x8
sort -u dogs.txt
Ivica
Luna
Roger
Syd
Tina
Vanille
```

sort does not just work 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

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

```
flavio — man /Users/flavio — less + man sort — 115x56

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:

Uniq

```
flavio — fish /Users/flavio — -fish — 40x17
|~ cat dogs.txt
Roger
Syd
Vanille
Luna
Ivica
Tina
Roger
Syd
|~ sort dogs.txt | uniq
Ivica
Luna
Roger
Syd
Tina
Vanille
|~
```

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

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

Uniq

```
flavio — fish /Users/flavio — -fish — 40x15
~ cat dogs.txt
Roger
Syd
Vanille
Luna
Ivica
Tina
Roger
Syd
~ sort dogs.txt | uniq -u
Ivica
Luna
Tina
Vanille
~
```

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

Uniq

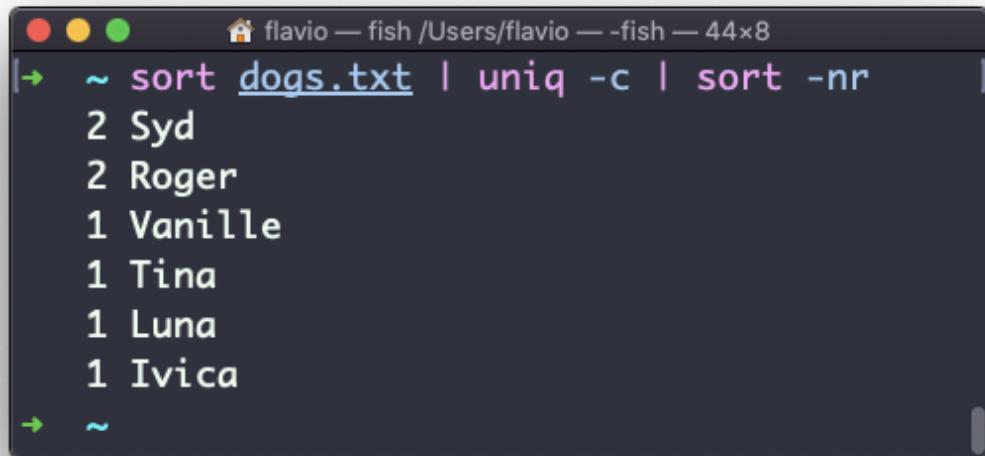
```
flavio — fish /Users/flavio — -fish — 40x17
|~ cat dogs.txt
Roger
Syd
Vanille
Luna
Ivica
Tina
Roger
Syd
|~ sort dogs.txt | uniq -c
 1 Ivica
 1 Luna
 2 Roger
 2 Syd
 1 Tina
 1 Vanille
|~
```

Use the special combination:

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

Uniq

to then sort those lines by most frequent:



```
flavio — fish /Users/flavio — -fish — 44x8
|~ sort dogs.txt | uniq -c | sort -nr
2 Syd
2 Roger
1 Vanille
1 Tina
1 Luna
1 Ivica
|~
```

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:

```
flavio — fish /Users/flavio — -fish — 50x8
|~ cat dogs.txt
Roger
Syd
|~ cat moredogs.txt
Roger
Syd
Vanille
~
```

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

```
flavio — fish /Users/flavio — -fish — 50x5
|~ diff dogs.txt moredogs.txt
2a3
> Vanille
~
```

diff

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

```
flavio — fish /Users/flavio — -fish — 50x5
~ diff moredogs.txt dogs.txt
3d2
< Vanille
→ ~
```

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

```
flavio — fish /Users/flavio — -fish — 86x9
~ diff -y dogs.txt moredogs.txt
Roger
Syd
→ ~ | Roger
Syd
> Vanille
```

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:

```
flavio — fish /Users/flavio — -fish — 67x8
~ diff -u dogs.txt moredogs.txt
--- dogs.txt    2020-09-07 08:54:56.000000000 +0200
+++ moredogs.txt      2020-09-07 08:55:09.000000000 +0200
@@ -1,2 +1,3 @@
  Roger
  Syd
+Vanille
→ ~ |
```

diff

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

```
testing ls dir1
dogs.txt
testing ls dir2
dogs.txt
testing diff -u dir1 dir2
diff -u dir1/dogs.txt dir2/dogs.txt
--- dir1/dogs.txt      2020-09-07 08:54:56.000000000 +0200
+++ dir2/dogs.txt      2020-09-07 08:55:09.000000000 +0200
@@ -1,2 +1,3 @@
    Roger
    Syd
+Vanille
→ testing
```

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

```
testing diff -rq dir1 dir2
Files dir1/dogs.txt and dir2/dogs.txt differ
→ testing
```

diff

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



The image shows a terminal window with the title bar "flavio — man /Users/flavio — less + man diff — 115x55". The window contains the man page for the "diff" command. The page is titled "DIFF(1)" and includes sections for NAME, SYNOPSIS, and DESCRIPTION. The DESCRIPTION section lists numerous options with their descriptions, such as -i for ignoring case differences, --ignore-case for ignoring case differences in file contents, and --label for using a label instead of a file name.

```
flavio — man /Users/flavio — less + man diff — 115x55
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"
```



A screenshot of a macOS terminal window titled "flavio — bash /Users/flavio — bash — 61x5". The window contains the following text:

```
bash-3.2$ echo "The path variable is $PATH"
The path variable is /usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/Apple/usr/bin
bash-3.2$
```

Echo

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

```
flavio — bash /Users/flavio — bash — 61x5
bash-3.2$ echo The cost is $5
The cost is
bash-3.2$ echo The cost is \$5
The cost is $5
bash-3.2$
```

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 ~
```

Echo

```
flavio — bash /Users/flavio — bash — 61x5
[bash-3.2$ echo ~
/Users/flavio
bash-3.2$ ]
```

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

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

```
flavio — bash /Users/flavio — bash — 61x6
[bash-3.2$ echo $(ls -al)
total 8 drwxr-xr-x 4 flavio staff 128 Sep 3 15:43 .
drwxr-xr-x+ 55 flavio staff 1760 Sep 3 15:20 ..
-rw-r--r-- 1 flavio staff 6 Sep 3 15:43 output.txt
-rw-r--r-- 1 flavio staff 0 Sep 3 15:20 words.txt
bash-3.2$ ]
```

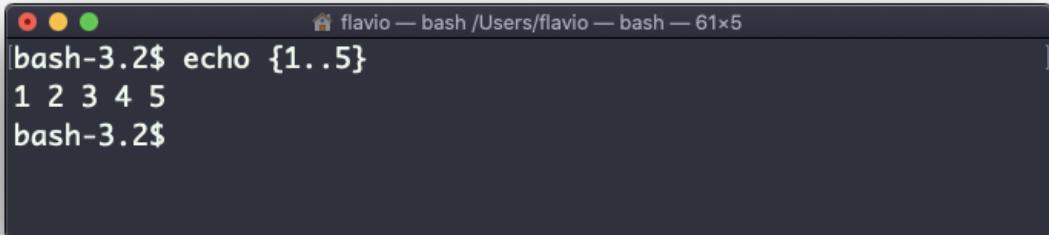
Note that whitespace is not preserved by default. You need to wrap the command in double quotes to do so:

```
flavio — bash /Users/flavio — bash — 61x7
[bash-3.2$ echo "$(ls -al)"
total 8
drwxr-xr-x 4 flavio staff 128 Sep 3 15:43 .
drwxr-xr-x+ 55 flavio staff 1760 Sep 3 15:20 ..
-rw-r--r-- 1 flavio staff 6 Sep 3 15:43 output.txt
-rw-r--r-- 1 flavio staff 0 Sep 3 15:20 words.txt
bash-3.2$ ]
```

Echo

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

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



```
flavio — bash /Users/flavio — bash — 61x5
bash-3.2$ echo {1..5}
1 2 3 4 5
bash-3.2$
```

A screenshot of a macOS terminal window titled "flavio — bash /Users/flavio — bash — 61x5". The window shows the command "echo {1..5}" being run, which outputs the numbers 1, 2, 3, 4, and 5 on separate lines. The terminal has a dark background with light-colored text.

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:



```
flavio — fish /Users/flavio — -fish — 65x7
|~ sudo touch test.txt
|~ echo test >> test.txt
<W> fish: An error occurred while redirecting file 'test.txt'
open: Permission denied
|~
```

You can use chown to transfer the ownership to you:

Chown

```
flavio — fish /Users/flavio — -fish — 65x7
~ sudo touch test.txt
~ echo test >> test.txt
<W> fish: An error occurred while redirecting file 'test.txt'
open: Permission denied
~ sudo chown flavio test.txt
~ echo test >> test.txt
~
```

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:

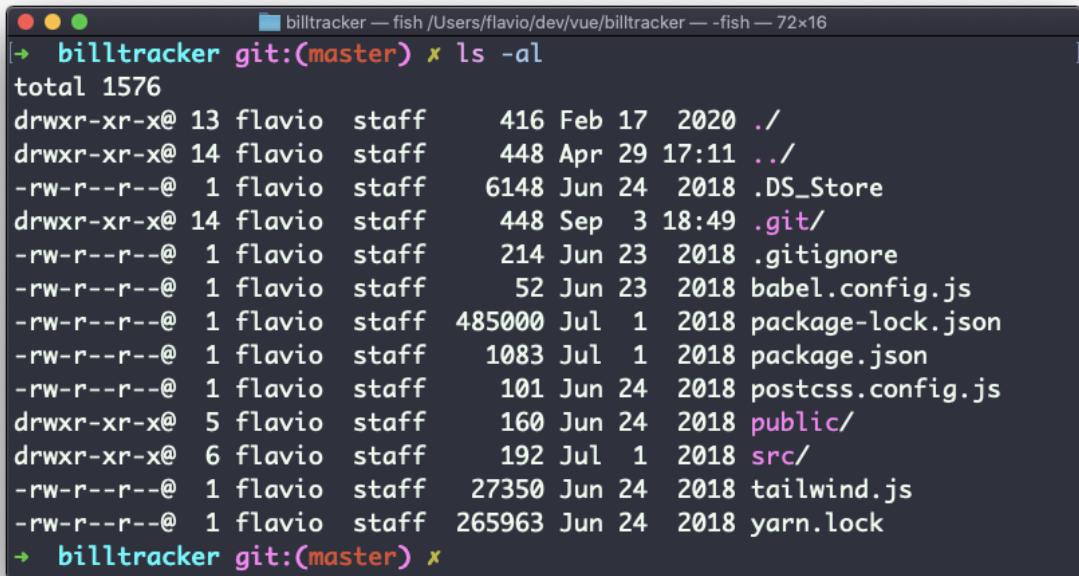
```
chown flavio:users test.txt
```

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.



```
billtracker — fish /Users/flavio/dev/vue/billtracker — -fish — 72x16
[→ billtracker git:(master) x ls -al
total 1576
drwxr-xr-x@ 13 flavio  staff      416 Feb 17  2020 .
drwxr-xr-x@ 14 flavio  staff      448 Apr 29 17:11 ..
-rw-r--r--@  1 flavio  staff    6148 Jun 24  2018 .DS_Store
drwxr-xr-x@ 14 flavio  staff      448 Sep  3 18:49 .git/
-rw-r--r--@  1 flavio  staff     214 Jun 23  2018 .gitignore
-rw-r--r--@  1 flavio  staff      52 Jun 23  2018 babel.config.js
-rw-r--r--@  1 flavio  staff   485000 Jul  1  2018 package-lock.json
-rw-r--r--@  1 flavio  staff     1083 Jul  1  2018 package.json
-rw-r--r--@  1 flavio  staff     101 Jun 24  2018 postcss.config.js
drwxr-xr-x@  5 flavio  staff     160 Jun 24  2018 public/
drwxr-xr-x@  6 flavio  staff     192 Jul  1  2018 src/
-rw-r--r--@  1 flavio  staff   27350 Jun 24  2018 tailwind.js
-rw-r--r--@  1 flavio  staff  265963 Jun 24  2018 yarn.lock
→ billtracker git:(master) x
```

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

Chmod

- The third set represents the permissions of the everyone else

Those sets are composed by 3 values. rwx means that specific person 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).

Chmod

All followed by the file or folder name.

Here are some examples:

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

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

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

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

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

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

Chmod

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:

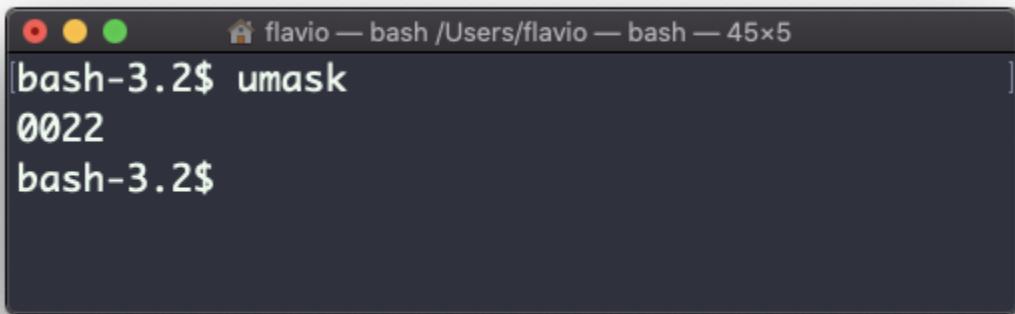
```
chmod 777 filename  
chmod 755 filename  
chmod 644 filename
```

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 :

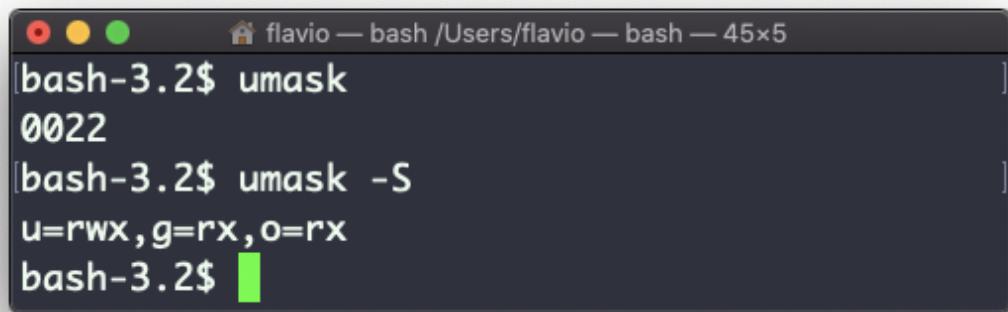


```
flavio — bash /Users/flavio — bash — 45x5
bash-3.2$ umask
0022
bash-3.2$
```

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

Another common value is 0002 .

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



```
flavio — bash /Users/flavio — bash — 45x5
bash-3.2$ umask
0022
bash-3.2$ umask -S
u=rwx,g=rx,o=rx
bash-3.2$
```

Umask

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

Other users belonging to the same group (g) have read and executionpermission, 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
```

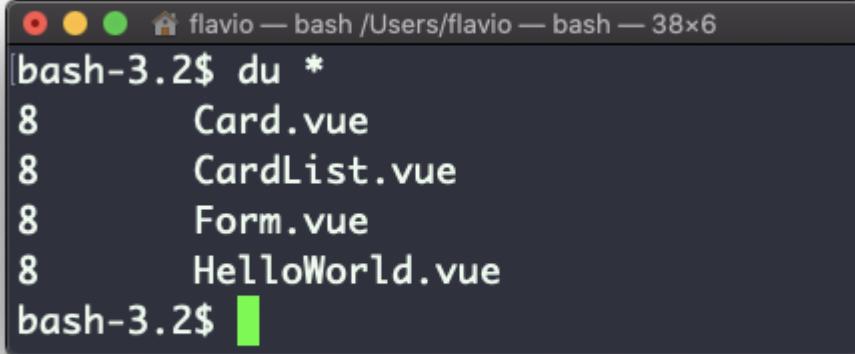


```
flavio — bash /Users/flavio — bash — 38x5
[bash-3.2$ du
32 .
bash-3.2$ ]
```

A screenshot of a macOS terminal window titled "flavio — bash /Users/flavio — bash — 38x5". The command "du" is run, resulting in the output "32 ." which indicates the total size of the current directory in bytes.

The 32 number here is a value expressed in bytes.

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



```
flavio — bash /Users/flavio — bash — 38x6
[bash-3.2$ du *
8 Card.vue
8 CardList.vue
8 Form.vue
8 HelloWorld.vue
bash-3.2$ ]
```

A screenshot of a macOS terminal window titled "flavio — bash /Users/flavio — bash — 38x6". The command "du *" is run, listing the size of individual files: Card.vue, CardList.vue, Form.vue, and HelloWorld.vue, all with a size of 8 bytes.

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:

```
flavio — bash /Users/flavio — bash — 58x10
[bash-3.2$ du -h vuehandbook
4.0K    vuehandbook/12-vue-watchers
4.0K    vuehandbook/07-vue-single-file-components
4.0K    vuehandbook/19-vue-components-communication
84K     vuehandbook/20-vuex
48K     vuehandbook/21-bonus-vue-router
12K     vuehandbook/01-vue-introduction
120K    vuehandbook/02-vue-first-app
8.0K    vuehandbook/06-vue-components
536K   vuehandbook/04-vue-devtools
```

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

```
flavio — bash /Users/flavio — bash — 65x14
[bash-3.2$ du -ah vuehandbook
4.0K    vuehandbook/12-vue-watchers/index.md
4.0K    vuehandbook/12-vue-watchers
4.0K    vuehandbook/07-vue-single-file-components/index.md
4.0K    vuehandbook/07-vue-single-file-components
4.0K    vuehandbook/19-vue-components-communication/index.md
4.0K    vuehandbook/19-vue-components-communication
36K     vuehandbook/20-vuex/vuex-store.png
12K     vuehandbook/20-vuex/index.md
36K     vuehandbook/20-vuex/codesandbox.png
84K     vuehandbook/20-vuex
12K     vuehandbook/.DS_Store
32K     vuehandbook/21-bonus-vue-router/banner.jpg
16K     vuehandbook/21-bonus-vue-router/index.md
```

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

```
du -h | sort -nr
```

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

```
flavio — bash /Users/flavio — bash — 63x13
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:

```
flavio — fish /Users/flavio — fish — 106x9
↳ ~ df
Filesystem 512-blocks Used Available Capacity iused ifree %iused Mounted on
/dev/disk1s1 976490576 21974760 487735816 5% 488418 4881964462 0% /
devfs 375 375 0 100% 649 0 100% /dev
/dev/disk1s2 976490576 454785000 487735816 49% 4287975 4878164905 0% /System/Volumes/Data
/dev/disk1s5 976490576 10485848 487735816 3% 5 4882452875 0% /private/var/vm
map auto_home 0 0 0 100% 0 0 100% /System/Volumes/Data/home
↳ ~
```

Using the -h option (df -h) will show those values in a human-readable format:

```
flavio — fish /Users/flavio — fish — 100x9
↳ ~ df -h
Filesystem Size Used Avail Capacity iused ifree %iused Mounted on
/dev/disk1s1 466Gi 10Gi 233Gi 5% 488418 4881964462 0% /
devfs 188Ki 188Ki 0Bi 100% 649 0 100% /dev
/dev/disk1s2 466Gi 217Gi 233Gi 49% 4287984 4878164896 0% /System/Volumes/Data
/dev/disk1s5 466Gi 5.0Gi 233Gi 3% 5 4882452875 0% /private/var/vm
map auto_home 0Bi 0Bi 0Bi 100% 0 0 100% /System/Volumes/Data/home
↳ ~
```

You can also specify a file or directory name to get information about the specific volume it lives on:

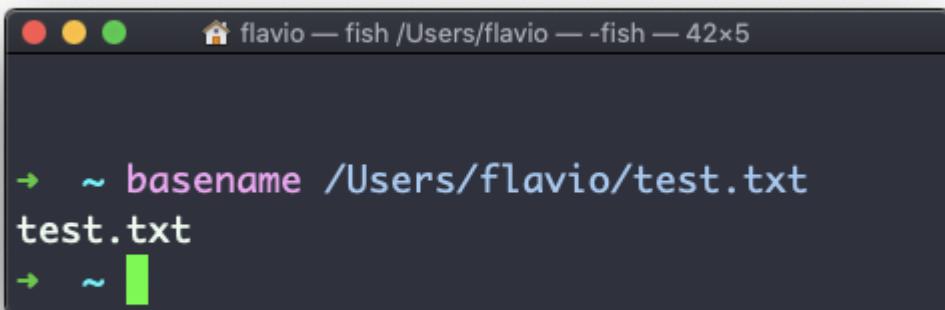
```
flavio — fish /Users/flavio — fish — 109x6
↳ ~ df dev
Filesystem 512-blocks Used Available Capacity iused ifree %iused Mounted on
/dev/disk1s2 976490576 454788176 487732640 49% 4287987 4878164893 0% /System/Volumes/Data
↳ ~
```

Basename

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

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

will return the test.txt string:



A screenshot of a macOS terminal window titled "flavio — fish /Users/flavio — -fish — 42x5". The command "basename /Users/flavio/test.txt" is entered, and the output "test.txt" is displayed below it.

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

If you run basename on a path string that points to a directory, you will get the last segment of the path. In this example, /Users/flavio is a directory:



A screenshot of a macOS terminal window titled "flavio — fish /Users/flavio — -fish — 42x5". The command "basename /Users/flavio/" is entered twice, and the output "flavio" is displayed twice below it.

```
→ ~ basename /Users/flavio/
flavio
→ ~ basename /Users/flavio/
flavio
→ ~
```

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:



A screenshot of a macOS terminal window. The title bar shows the user 'flavio' and the application 'fish'. The window contains the following text:

```
flavio — fish /Users/flavio — -fish — 44x5
[→ ~ dirname /Users/flavio/test.txt ]
/Users/flavio
→ ~
```

Ps

Your computer is running, at all times, tons of different processes. You can inspect them all using the ps command:



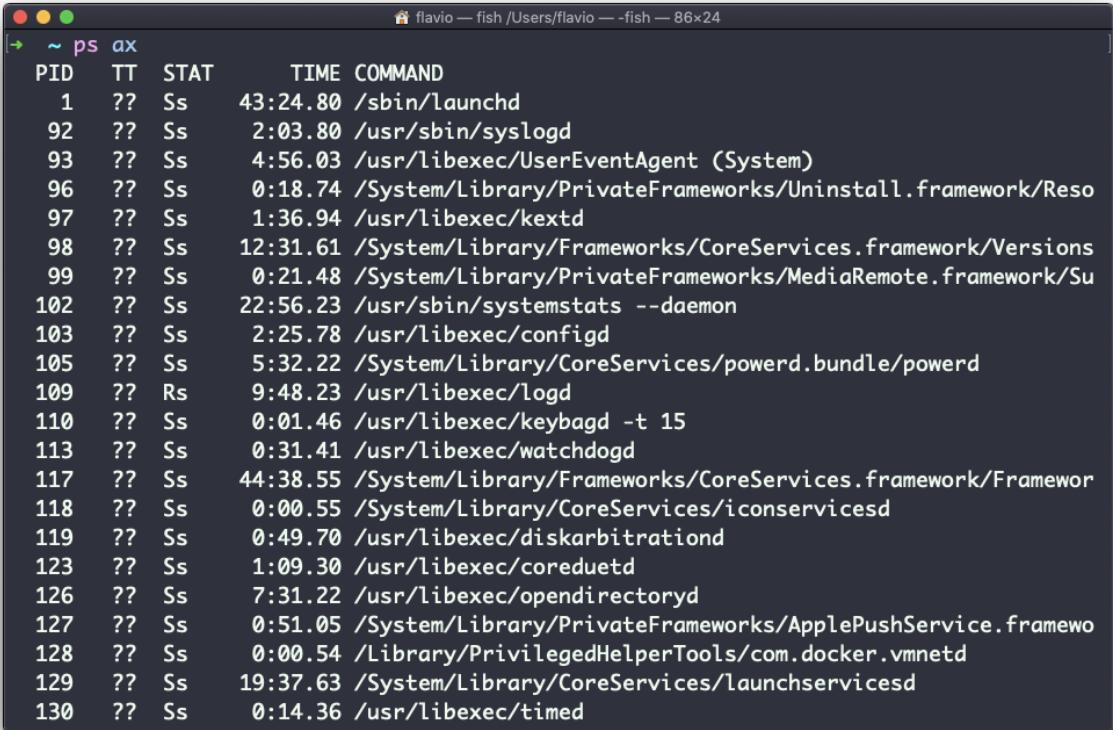
```
flavio — fish /Users/flavio — -fish — 63x12
[~] ~ ps
PID TTY      TIME CMD
59474 ttys000  0:00.13 /usr/local/bin/fish -l
  941 ttys002  0:00.61 /usr/local/bin/fish -l
  71366 ttys004 0:01.08 -fish
  4216 ttys009  0:00.31 /usr/local/bin/fish -l
  68714 ttys009 0:20.82 hugo serve
  5088 ttys013  0:02.89 -fish
[~]
```

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 :

Ps



```
flavio — fish /Users/flavio — -fish — 86x24
~ ps ax
PID  TT  STAT      TIME COMMAND
  1  ??  Ss   43:24.80 /sbin/launchd
  92  ??  Ss   2:03.80 /usr/sbin/syslogd
  93  ??  Ss   4:56.03 /usr/libexec/UserEventAgent (System)
  96  ??  Ss   0:18.74 /System/Library/PrivateFrameworks/Uninstall.framework/Reso
  97  ??  Ss   1:36.94 /usr/libexec/kextd
  98  ??  Ss   12:31.61 /System/Library/Frameworks/CoreServices.framework/Versions
  99  ??  Ss   0:21.48 /System/Library/PrivateFrameworks/MediaRemote.framework/Su
 102  ??  Ss   22:56.23 /usr/sbin/systemstats --daemon
 103  ??  Ss   2:25.78 /usr/libexec/configd
 105  ??  Ss   5:32.22 /System/Library/CoreServices/powerd.bundle/powerd
 109  ??  Rs   9:48.23 /usr/libexec/logd
 110  ??  Ss   0:01.46 /usr/libexec/keybagd -t 15
 113  ??  Ss   0:31.41 /usr/libexec/watchdogd
 117  ??  Ss   44:38.55 /System/Library/Frameworks/CoreServices.framework/Frameworks/IconServices.framework/IconServicesd
 118  ??  Ss   0:00.55 /System/Library/CoreServices/iconservicesd
 119  ??  Ss   0:49.70 /usr/libexec/diskarbitrationd
 123  ??  Ss   1:09.30 /usr/libexec/coreduetd
 126  ??  Ss   7:31.22 /usr/libexec/opendirectoryd
 127  ??  Ss   0:51.05 /System/Library/PrivateFrameworks/ApplePushService.framework/ApplePushService
 128  ??  Ss   0:00.54 /Library/PrivilegedHelperTools/com.docker.vmnetd
 129  ??  Ss   19:37.63 /System/Library/CoreServices/launchservicesd
 130  ??  Ss   0:14.36 /usr/libexec/timed
```

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:

Ps

```
flavio — fish /Users/flavio — -fish — 84x21
[~] ~ ps axww
  PID  TT  STAT      TIME COMMAND
    1  ??  Ss      43:25.81 /sbin/launchd
    92  ??  Ss      2:03.82 /usr/sbin/syslogd
    93  ??  Ss      4:56.05 /usr/libexec/UserEventAgent (System)
    96  ??  Ss      0:18.74 /System/Library/PrivateFrameworks/Uninstall.framework/Resources/uninstallld
    97  ??  Ss      1:36.94 /usr/libexec/kextd
    98  ??  Ss     12:31.92 /System/Library/Frameworks/CoreServices.framework/Versions/A/Frameworks/FSEvents.framework/Versions/A/Support/fsevents
    99  ??  Ss      0:21.48 /System/Library/PrivateFrameworks/MediaRemote.framework/Support/mediaremoted
   102  ??  Ss     22:56.23 /usr/sbin/systemstats --daemon
   103  ??  Ss      2:25.80 /usr/libexec/configd
   105  ??  Ss      5:32.38 /System/Library/CoreServices/powerd.bundle/powerd
   109  ??  Ss      9:48.48 /usr/libexec/logd
   110  ??  Ss      0:01.46 /usr/libexec/keybagd -t 15
   113  ??  Ss      0:31.42 /usr/libexec/watchdogd
   117  ??  Ss     44:38.88 /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Support/mds
   118  ??  Ss      0:00.55 /System/Library/CoreServices/iconservicesd
```

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"
```

Ps

The columns returned by ps represent some key information. The first information is PID , the process ID. This is key when you want to reference this process in another command, for example to kill it.

Then we have TT that tells us the terminal id used.
Then STAT tells us the state of the process:

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

Ps

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

It's common to have + which indicates the process is in the foreground in its terminal. s means the process is a session leader. TIME tells us how long the process has been running.

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:

```
flavio — top /Users/flavio — top — 80x24
Processes: 574 total, 2 running, 572 sleeping, 3807 threads          11:39:53
Load Avg: 1.24, 1.93, 2.23 CPU usage: 4.61% user, 3.68% sys, 91.69% idle
SharedLibs: 234M resident, 62M data, 16M linkededit.
MemRegions: 369828 total, 6215M resident, 163M private, 1603M shared.
PhysMem: 14G used (3684M wired), 1742M unused.
VM: 3080G vsize, 1991M framework vsize, 316459873(128) swapins, 322451178(0) swa
Networks: packets: 41309179/63G in, 44958830/24G out.
Disks: 68270763/1950G read, 43422129/1680G written.

PID  COMMAND      %CPU TIME      #TH  #WQ  #PORT MEM    PURG   CMPRS  PGRP
229  WindowServer 16.1 13:11:15 10   4    6607+ 845M+ 1216K- 173M   229
1185 com.docker.h 7.6  15:21:25 18   0    42    2374M  0B     316M   1162
82661 top          7.4  00:01.00 1/1  0    27+   8352K+ 0B     0B     82661
381  Terminal      6.0  04:31.43 11   4    436   105M   14M    28M-   381
1152 1Password     5.6  02:00:08 8    2    2473- 530M- 244K   349M   1152
0   kernel_task    3.9  06:15:45 262/12 0    0    510M+  0B     0B     0
82622 Google Chrom 2.8  00:07.20 16   1    209   81M    4096B  0B     38992
377  Bear          2.1  41:08.16 12   5    1760- 557M- 13M-   460M-   377
98108 Books         2.1  38:16.24 6    1    822   302M  0B     286M   98108
347  cloudd        2.0  10:10.00 16   5    785+  31M+  384K+ 12M-   347
149  hidd          1.9  45:27.12 7    2    560   7208K  0B     2496K  149
50060 Music         1.5  45:57.57 22   3    684   201M+ 460K   77M    50060
148  bluetoothd    1.5  44:49.50 3    1    801   18M   0B     7520K  148
1154 Rectangle      1.3  07:31.75 3    1    210-  52M-  0B     45M    1154
```

top

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, howmany 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 constantlyupdated.

By default, as you can see from the %CPU column highlighted, they aresorted 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>
```

kill

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

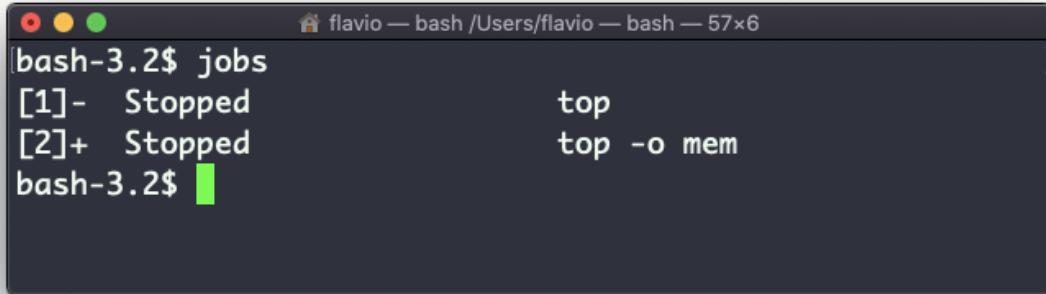
When we run a command in Linux / macOS, we can set it to run in the background using the & symbol after the command. For example we can run top in the background:

```
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 jobnumber: 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:



```
flavio — bash /Users/flavio — bash — 57x6
[bash-3.2$ jobs
[1]-  Stopped                  top
[2]+  Stopped                  top -o mem
bash-3.2$ ]
```

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

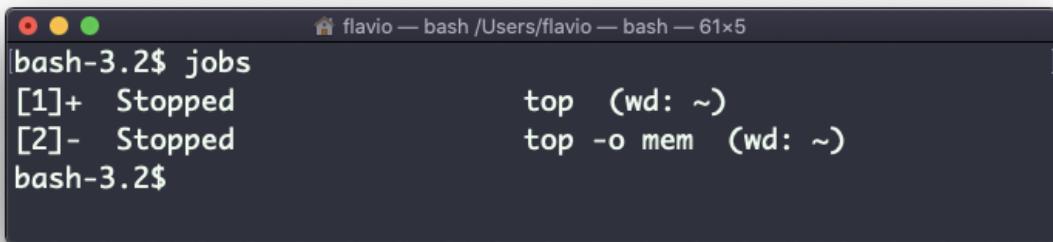
Running jobs -l will also print the process id of each job.

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:



A screenshot of a macOS terminal window titled "flavio — bash /Users/flavio — bash — 61x5". The window shows the following output:

```
bash-3.2$ jobs
[1]+  Stopped                  top  (wd: ~)
[2]-  Stopped                  top -o mem  (wd: ~)
bash-3.2$
```

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.

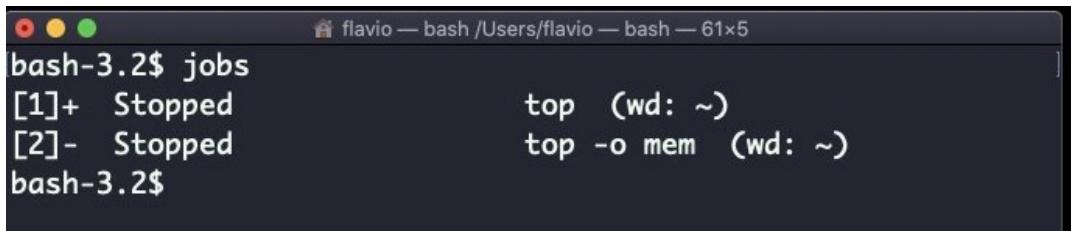
When a command is running in the background, because you started it with & at the end (example: top & or because you put it in the background with the bg command, you can put it to the foreground using fg .

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.



```
flavio — bash /Users/flavio — bash — 61x5
bash-3.2$ jobs
[1]+  Stopped                  top  (wd: ~)
[2]-  Stopped                  top -o mem  (wd: ~)
bash-3.2$
```

Running fg 2 will resume job #2:

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	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:

```
flavio — bash /Users/flavio — bash — 61x19
[bash-3.2$ type ls
ls is hashed (/bin/ls)
[bash-3.2$ type cat
cat is /bin/cat
[bash-3.2$ type pwd
pwd is a shell builtin
[bash-3.2$ type ll
ll is aliased to `ls -al'
[bash-3.2$ type top
top is hashed (/usr/bin/top)
[bash-3.2$ type jobs
jobs is a shell builtin
[bash-3.2$ type bg
bg is a shell builtin
[bash-3.2$ type kill
kill is a shell builtin
[bash-3.2$ type top
top is hashed (/usr/bin/top)
bash-3.2$
```

Type

This is Zsh:

```
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 ~ %
```

Type

This is Fish:

```
[~] flazio — fish /Users/flazio — -fish — 80x39
[~] ~ 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

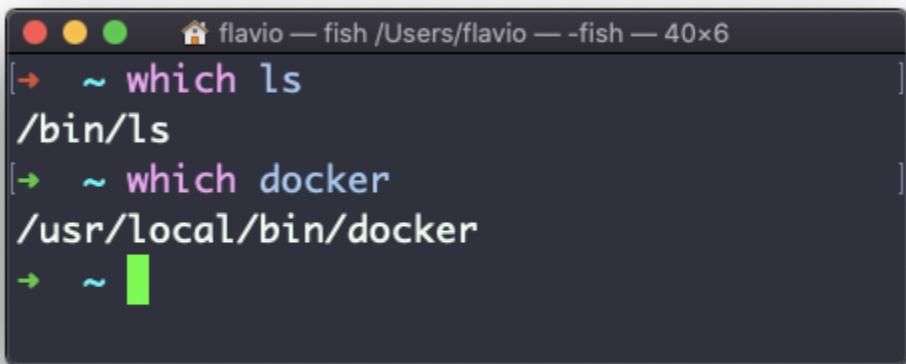
is aliasing to. You can see the ll alias, in the case of Bash and Zsh, but

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 shellpath, 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:



```
[~] flazio — fish /Users/flazio — -fish — 40x6
[→ ~ which ls
/bin/ls
[→ ~ which docker
/usr/local/bin/docker
→ ~ |
```

A screenshot of a macOS terminal window titled "flazio — fish /Users/flazio — -fish — 40x6". The window shows the output of the "which" command. It first finds "ls" at "/bin/ls" and then "docker" at "/usr/local/bin/docker". The cursor is currently at the end of the "docker" command line.

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 standardinput 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 :



```
testing — fish /Users/flavio/testing — -fish — 62x7
[→ testing ls
file1      file2      file3      todelete.txt
[→ testing cat todelete.txt
file1
file3
→ testing ]
```

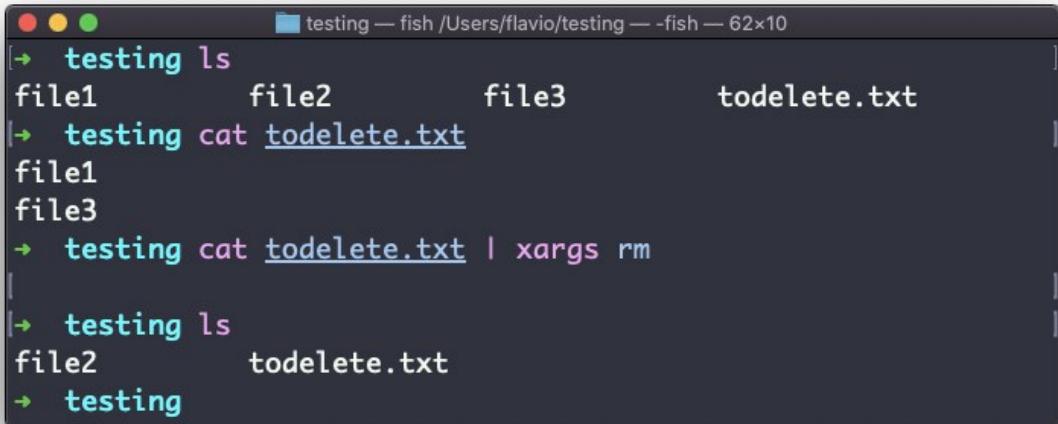
Xargs

We will channel the output of cat to delete.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 screenshot shows a terminal window titled "testing — fish /Users/flavio/testing — -fish — 62x10". The session history is as follows:

- ls
- cat todelete.txt
- file1
file3
- cat todelete.txt | xargs rm
- ls
- file2 todelete.txt

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:

Xargs



```
testing — cat /Users/flavio/testing — xargs -p rm — 44x5
|→ testing cat todelete.txt | xargs -p rm
rm file1 file3?...
```

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`:



```
testing — cat /Users/flavio/testing — xargs -p -n1 rm — 55x5
|→ testing cat todelete.txt | xargs -p -n1 rm
rm file1?...
```

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

One of them is to run multiple commands:

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



```
testing — cat /Users/flavio/testing — xargs -p -I % sh -c 'ls %; rm %' — 67x5
|→ testing cat todelete.txt | xargs -p -I % sh -c 'ls %; rm %'
sh -c ls file1; rm file1?...
```

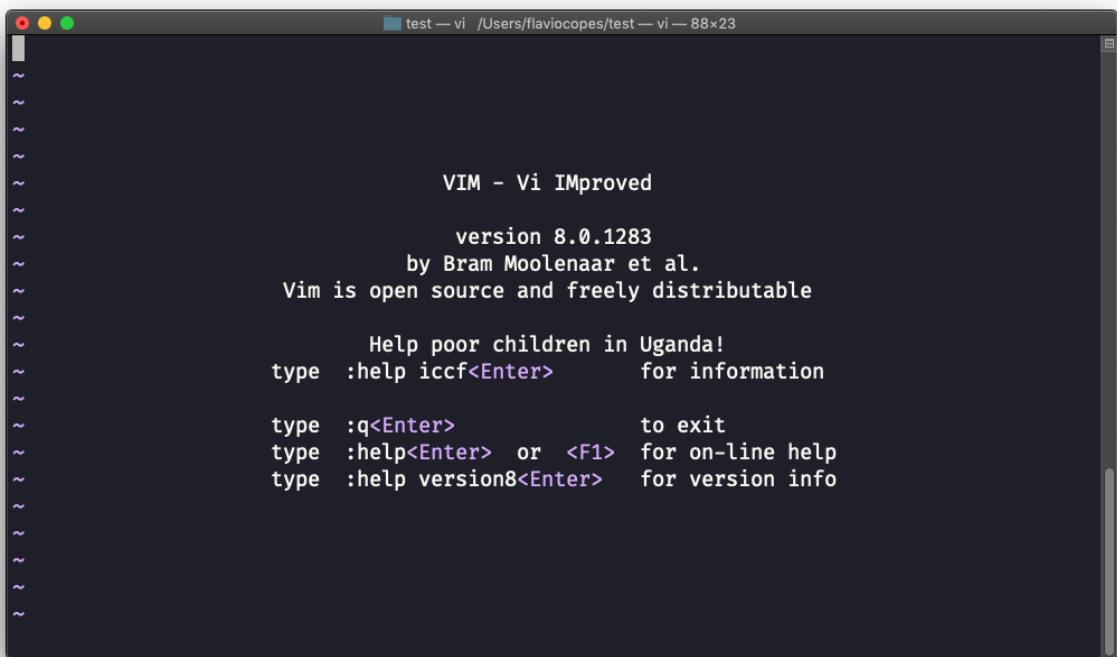
You can swap the `%` symbol I used above with anything else, it's a variable .

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 improved.

You start it by running vi on the command line.



A screenshot of a Mac OS X terminal window titled "test — vi /Users/flaviocopes/test — vi — 88x23". The window displays the Vim startup screen, which includes the title "VIM - Vi IMproved", the version "version 8.0.1283", the author "by Bram Moolenaar et al.", the fact that "Vim is open source and freely distributable", and several help and usage instructions. The terminal has a dark background with light-colored text.

```
VIM - Vi IMproved
version 8.0.1283
by Bram Moolenaar et al.
Vim is open source and freely distributable

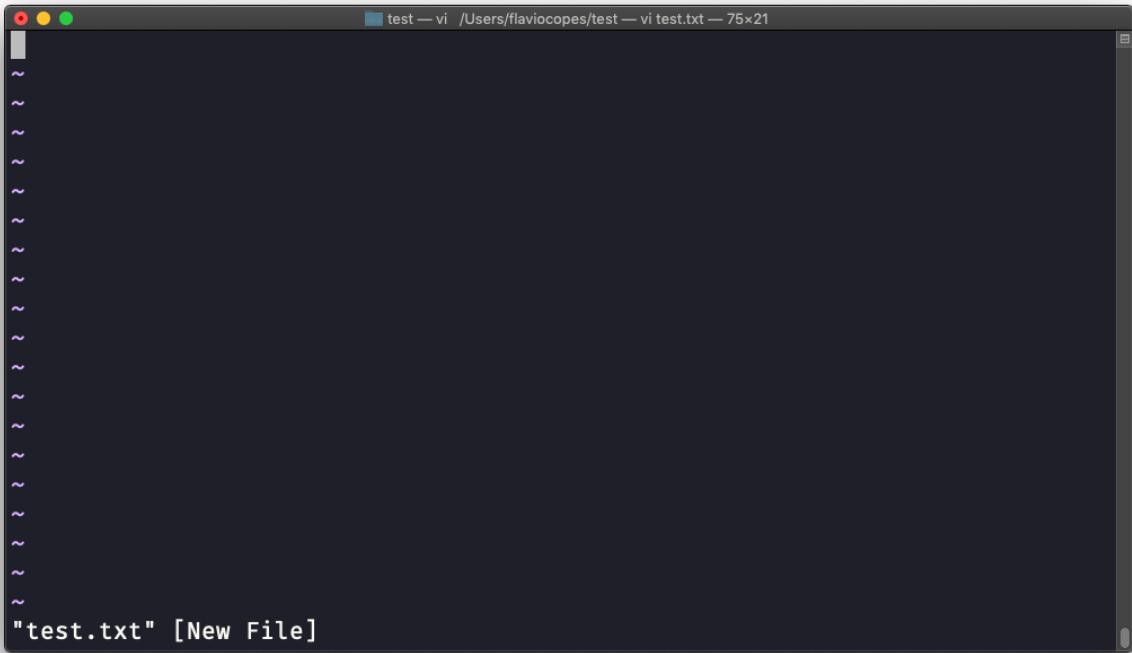
      Help poor children in Uganda!
type :help iccf<Enter>      for information

type :q<Enter>          to exit
type :help<Enter> or <F1> for on-line help
type :help version8<Enter> for version info
```

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

```
vi test.txt
```

Vim



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 appears at the bottom of the editor:

Vim

```
-- INSERT --
```

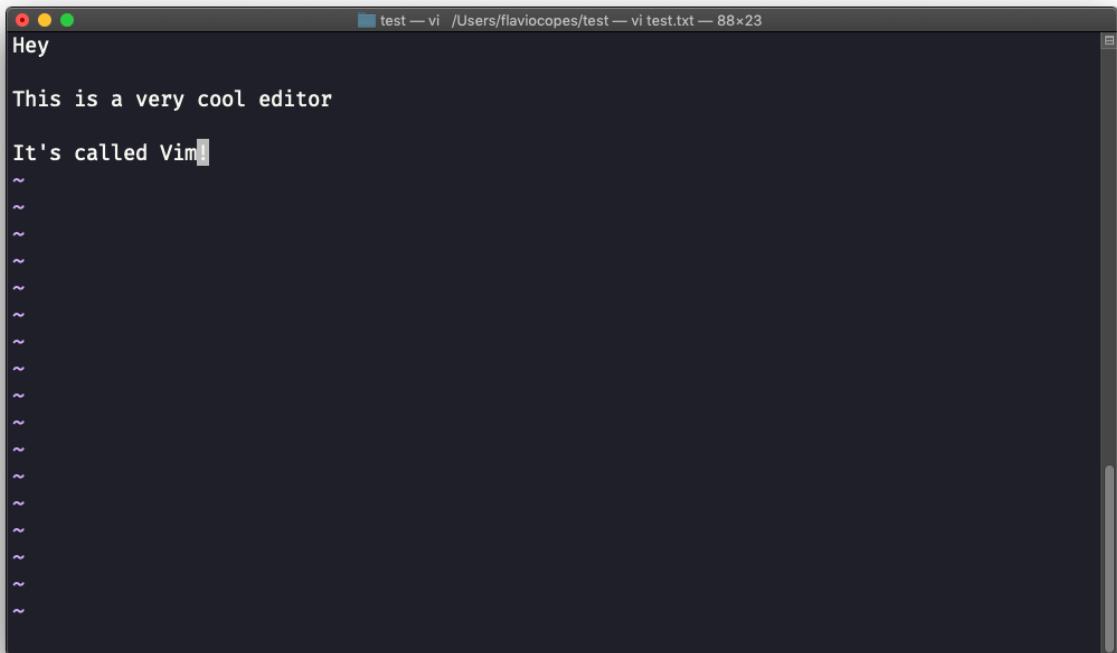
Now you can start typing and filling the screen with the file contents:

```
Hey
This is a very cool editor
It's called Vim!
```

```
-- INSERT --
```

Vim

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**.



A screenshot of a Mac OS X terminal window titled "test — vi /Users/flaviocopes/test — vi test.txt — 88x23". The window contains the following text:

```
Hey
This is a very cool editor
It's called Vim!
```

The text "It's called Vim!" is currently being typed, with the cursor visible at the end of "Vim!". The terminal window has a dark background and light-colored text.

Vim

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.

Vim

- 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 :

```
File Edit Options Buffers Tools Help
Welcome to GNU Emacs, a part of the GNU operating system.

Get help      C-h (Hold down CTRL and press h)
Emacs manual  C-h r      Browse manuals  C-h i
Emacs tutorial  C-h t      Undo changes    C-x u
Buy manuals   C-h RET    Exit Emacs      C-x C-c
Activate menubar M-
('C' means use the CTRL key. 'M-' means use the Meta (or Alt) key.
If you have no Meta key, you may instead type ESC followed by the character.)
Useful tasks:
Visit New File          Open Home Directory
Customize Startup        Open *scratch* buffer

GNU Emacs 26.1 (build 1, x86_64-apple-darwin18.2.0)
of 2019-01-16
Copyright (C) 2018 Free Software Foundation, Inc.

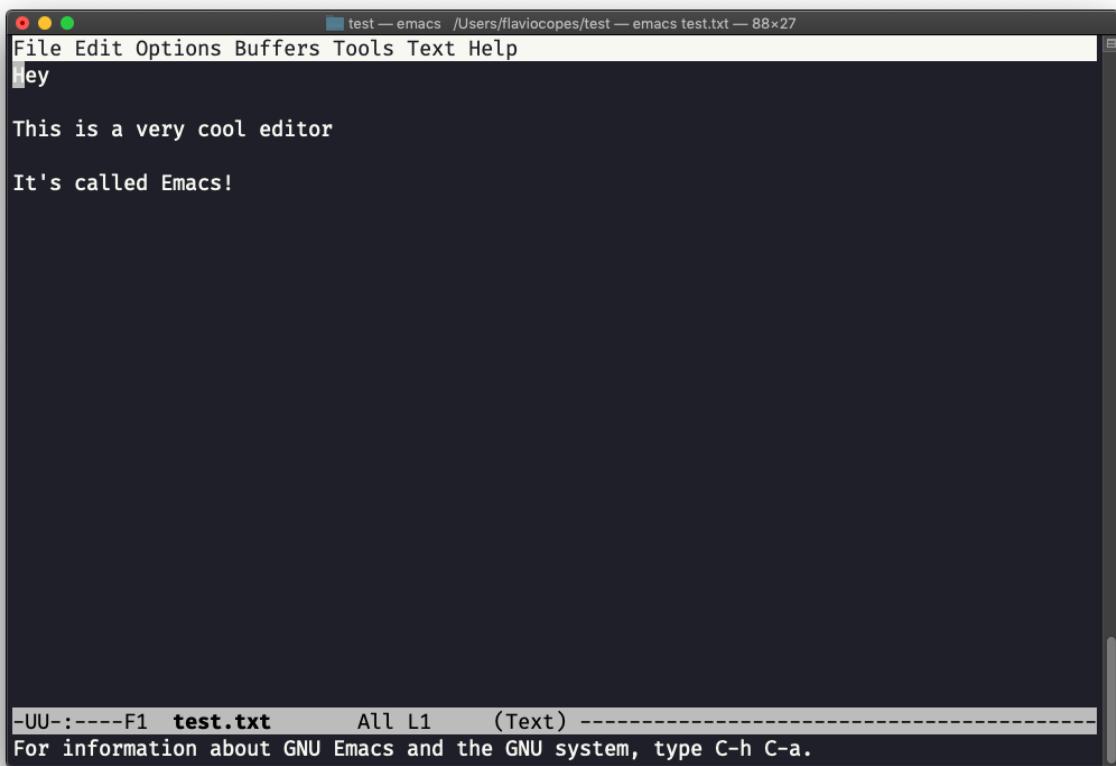
GNU Emacs comes with ABSOLUTELY NO WARRANTY; type C-h C-w for full details.
Emacs is Free Software--Free as in Freedom--so you can redistribute copies
of Emacs and modify it; type C-h C-c to see the conditions.
Type C-h C-o for information on getting the latest version.

-UUU:%%--F1  *GNU Emacs*  All L1  (Fundamental) -----
For information about GNU Emacs and the GNU system, type C-h C-a.
```

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 :



emacs

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

```
File Edit Options Buffers Tools Minibuf Help
Hey
This is a very cool editor
It's called Emacs!

-UU-:**--F1  test.txt      All L7      (Text) -----
Write file: ~/test/
```

and Emacs tell you the file exists, asking you if it should overwrite it:

emacs

```
File Edit Options Buffers Tools Text Help
Hey
This is a very cool editor
It's called Emacs!
-UU-:***--F1 test.txt      All L7      (Text) -----
File '~/test/test.txt' exists; overwrite? (y or n) [ ]
```

Answer **y** , and you get a confirmation of success:

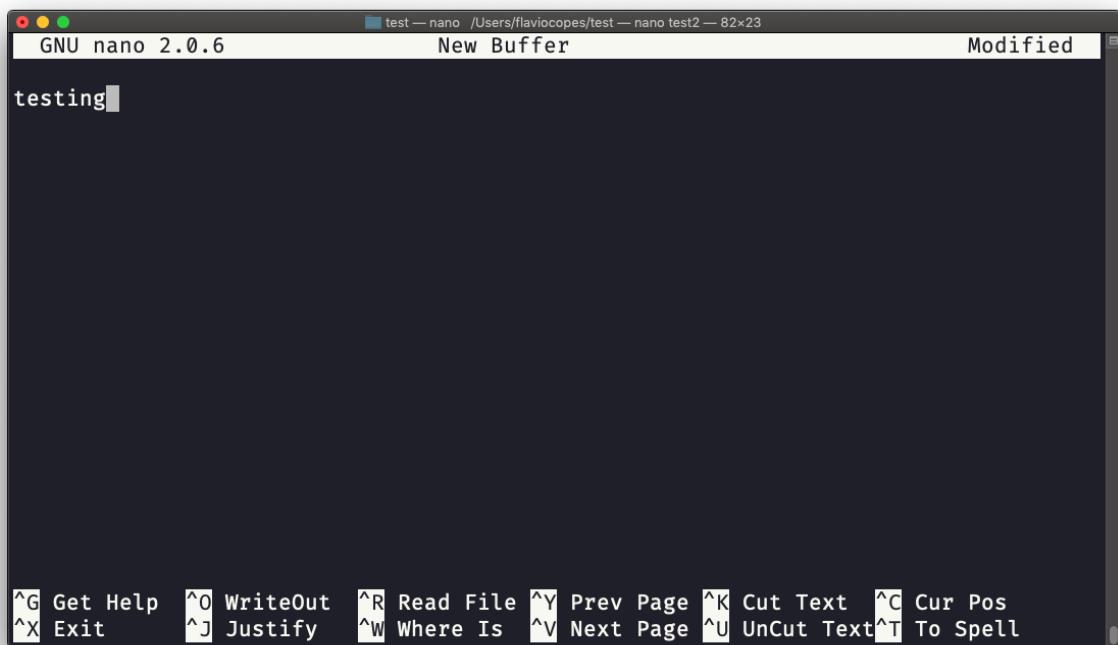
emacs

```
File Edit Options Buffers Tools Text Help
Hey
This is a very cool editor
It's called Emacs!
[empty line]
-UUU-----F1 test.txt      All L7      (Text) -----
Wrote /Users/flaviocopes/test/test.txt
```

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

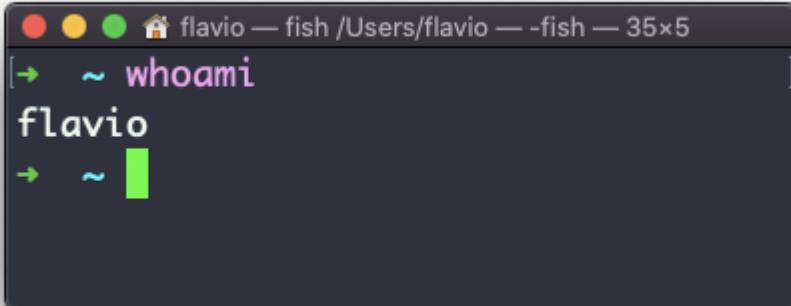
nano is a beginner friendly editor. Run it using nano . You can directly type characters into the file without worrying about modes. You can quit without editing using ctrl-X . If you edited the file buffer, the editor will ask you for confirmation and you can save the edits, or discard them. The help 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:



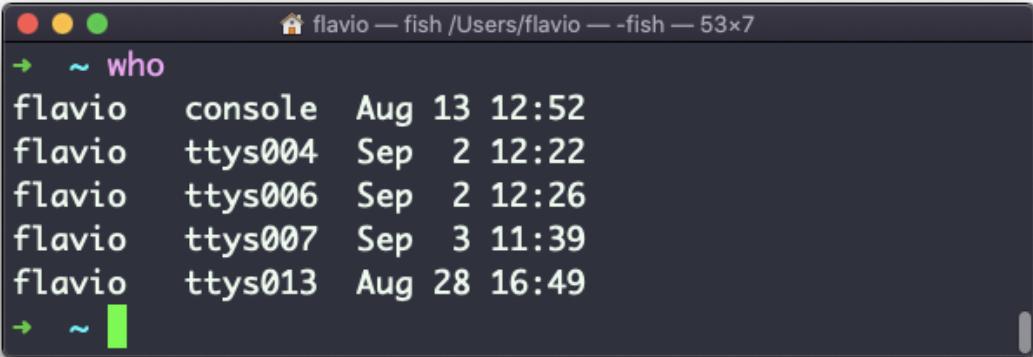
A screenshot of a terminal window titled "flavio — fish /Users/flavio — -fish — 35x5". The window shows the command "whoami" being typed and its output "flavio". The terminal has a dark background with light-colored text and icons.

```
[~] whoami
flavio
[~]
```

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:



```
flavio — fish /Users/flavio — -fish — 53x7
~ who
flavio  console Aug 13 12:52
flavio  ttys004 Sep  2 12:22
flavio  ttys006 Sep  2 12:26
flavio  ttys007 Sep  3 11:39
flavio  ttys013 Aug 28 16:49
~ |
```

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:

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:

```
flavio — fish /Users/flavio — -fish — 66x15
[→ ~ who -aH
USER    LINE    WHEN      IDLE      PID  COMMENT
reboot  ~        Aug 13 12:52 00:01      1
flavio   console  Aug 13 12:52  old      167
flavio   ttys000  Aug 16 18:39  old      70258 term=0 exit=0
flavio   ttys004  Sep  2 12:22   .       71365
flavio   ttys006  Sep  2 12:26  00:41      71675
flavio   ttys007  Sep  3 11:39  00:01      82656
flavio   ttys008  Aug 21 14:33  04:15      25883 term=0 exit=0
flavio   ttys009  Aug 15 19:36  old      52214 term=0 exit=0
flavio   ttys010  Aug 23 16:32   .       38649 term=0 exit=0
flavio   ttys011  Aug 21 10:20   .       18855 term=0 exit=0
flavio   ttys013  Aug 28 16:49  01:33      5086
.          run-level 3
→ ~ |
```

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

```
flavio — fish /Users/flavio — -fish — 45x5
[→ ~ who am i
flavio  ttys004  Sep  2 12:22
→ ~ |
```

```
flavio — fish /Users/flavio — -fish — 65x5
[→ ~ who -aH am i
USER    LINE    WHEN      IDLE      PID  COMMENT
flavio  ttys004  Sep  2 12:22   .       71365
→ ~ |
```

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).

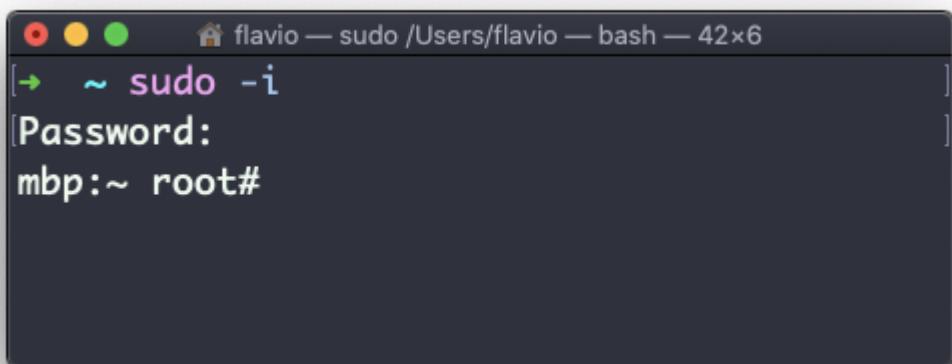
The permissions are highly configurable, which is great especially in a multiuser server environment, and some users can be granted access to running specific commands through sudo .

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:



A screenshot of a macOS terminal window titled "flavio — sudo /Users/flavio — bash — 42x6". The window shows the following text:
[~] ~ sudo -i
[Password:]
mbp:~ root#

Sudo

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
```

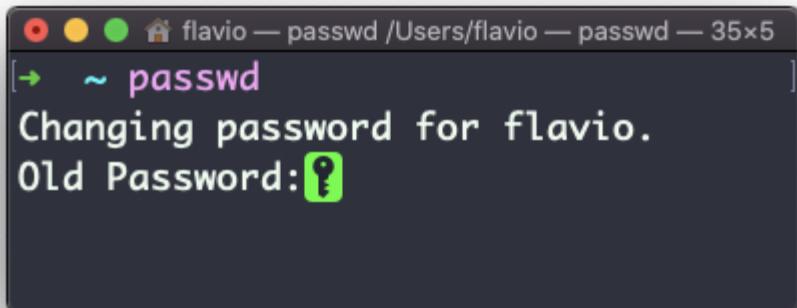
Password

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:



```
flavio — passwd /Users/flavio — passwd — 35x5
~ passwd
Changing password for flavio.
Old Password: ?
```

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>
```

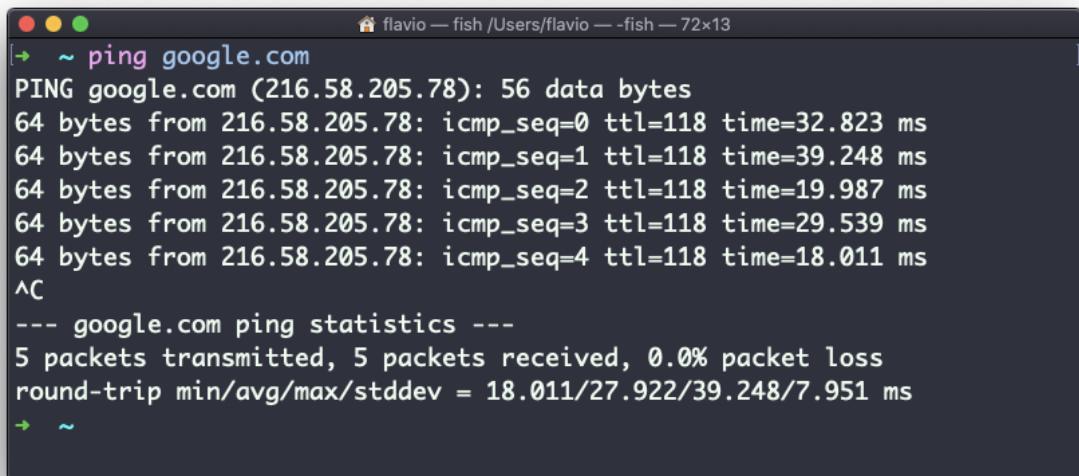
In this case you don't need to enter the old one.

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 :



```
[~] flavio — fish /Users/flavio — -fish — 72x13
[~] ~ ping google.com
PING google.com (216.58.205.78): 56 data bytes
64 bytes from 216.58.205.78: icmp_seq=0 ttl=118 time=32.823 ms
64 bytes from 216.58.205.78: icmp_seq=1 ttl=118 time=39.248 ms
64 bytes from 216.58.205.78: icmp_seq=2 ttl=118 time=19.987 ms
64 bytes from 216.58.205.78: icmp_seq=3 ttl=118 time=29.539 ms
64 bytes from 216.58.205.78: icmp_seq=4 ttl=118 time=18.011 ms
^C
--- google.com ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 18.011/27.922/39.248/7.951 ms
[~]
```

Ping

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:

```
flavio — fish /Users/flavio — -fish — 72x11
[~] ~ ping flaviocopes.com
PING flaviocopes.com (167.99.137.12): 56 data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
Request timeout for icmp_seq 2
Request timeout for icmp_seq 3
^C
--- flaviocopes.com ping statistics ---
5 packets transmitted, 0 packets received, 100.0% packet loss
[~]
```

Ping

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 nearer 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 homerouter, 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 :

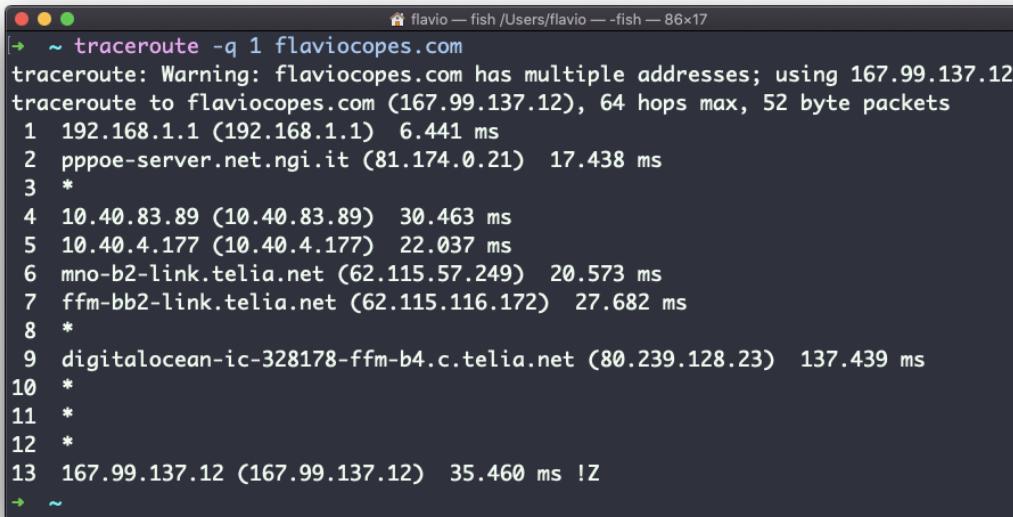
```
flavio — fish /Users/flavio — -fish — 91x20
[~] ~ traceroute flaviocopes.com
traceroute: Warning: flaviocopes.com has multiple addresses; using 142.93.108.123
traceroute to flaviocopes.com (142.93.108.123), 64 hops max, 52 byte packets
 1  192.168.1.1 (192.168.1.1)  3.090 ms  2.391 ms  2.231 ms
 2  pppoe-server.net.ngi.it (81.174.0.21)  10.697 ms  20.802 ms  21.329 ms
 3  * * *
 4  10.40.83.89 (10.40.83.89)  374.040 ms  1258.538 ms *
 5  10.40.4.177 (10.40.4.177)  21.893 ms  20.705 ms  20.912 ms
 6  * * *
 7  ffm-bb2-link.telia.net (62.115.116.172)  28.671 ms  39.589 ms *
 8  * * *
 9  digitalocean-ic-328177-ffm-b4.c.telia.net (80.239.128.21)  172.565 ms  173.744 ms
   digitalocean-ic-328178-ffm-b4.c.telia.net (80.239.128.23)  316.124 ms
10  * * *
11  * * *
12  * * *
13  142.93.108.123 (142.93.108.123)  207.791 ms !Z  32.786 ms !Z  23.561 ms !Z
[~] ~
```

Traceroute

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
```

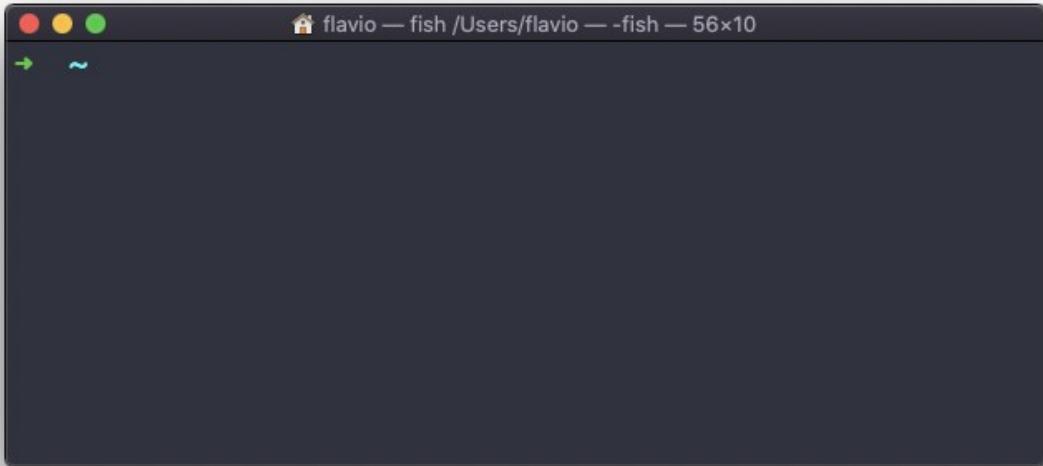


```
flavio — fish /Users/flavio — -fish — 86x17
[+ ~ traceroute -q 1 flaviocopes.com
traceroute: Warning: flaviocopes.com has multiple addresses; using 167.99.137.12
traceroute to flaviocopes.com (167.99.137.12), 64 hops max, 52 byte packets
 1  192.168.1.1 (192.168.1.1)  6.441 ms
 2  pppoe-server.net.ngi.it (81.174.0.21)  17.438 ms
 3  *
 4  10.40.83.89 (10.40.83.89)  30.463 ms
 5  10.40.4.177 (10.40.4.177)  22.037 ms
 6  mno-b2-link.telia.net (62.115.57.249)  20.573 ms
 7  ffm-bb2-link.telia.net (62.115.116.172)  27.682 ms
 8  *
 9  digitalocean-ic-328178-ffm-b4.c.telia.net (80.239.128.23)  137.439 ms
10  *
11  *
12  *
13  167.99.137.12 (167.99.137.12)  35.460 ms !Z
```

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.

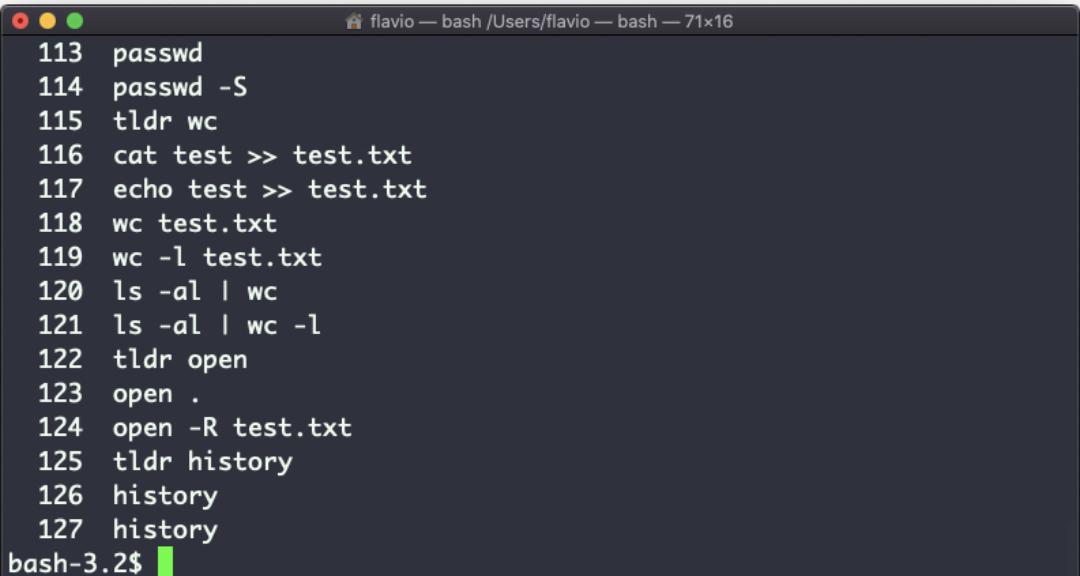
So you might want to use **clear -x** instead, which still clears the screen, but lets you go back to see the previous work by scrolling up.

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:



A screenshot of a macOS terminal window titled "flavio — bash /Users/flavio — bash — 71x16". The window displays a list of command history entries, each preceded by a number (113 to 127). The commands listed are: passwd, passwd -S, tldr wc, cat test >> test.txt, echo test >> test.txt, wc test.txt, wc -l test.txt, ls -al | wc, ls -al | wc -l, tldr open, open ., open -R test.txt, tldr history, history, and history again. The prompt at the bottom is "bash-3.2\$".

```
113 passwd
114 passwd -S
115 tldr wc
116 cat test >> test.txt
117 echo test >> test.txt
118 wc test.txt
119 wc -l test.txt
120 ls -al | wc
121 ls -al | wc -l
122 tldr open
123 open .
124 open -R test.txt
125 tldr history
126 history
127 history
bash-3.2$
```

You can use the syntax ! 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
```

History



```
flavio — bash /Users/flavio — bash — 71x8
bash-3.2$ history | grep docker
  7 git clone https://github.com/docker/getting-started.git
  8 git clone https://github.com/docker/getting-started.git
 14 docker container stop $(docker container ls)
 15 docker container stop $(docker container ls)
 16 docker container stop $(docker container ls -aq)
130 history | grep docker
bash-3.2$
```

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 :

```
flavio — bash /Users/flavio — bash — 61x5
[bash-3.2$ echo $TEST
test
bash-3.2$
```

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

```
flavio — bash /Users/flavio — nano script.sh — 61x12
GNU nano 2.0.6          File: script.sh          Modified

#!/bin/bash
echo $TEST
```

[Read 2 lines]
^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^V Next Page ^U Uncut TAT To Spell

Export

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":

Export

```
flavio — bash /Users/flavio — bash — 61x6
|bash-3.2$ export TEST="test"
|bash-3.2$ ./script.sh
test
bash-3.2$
```

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.

Cronjob

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:



A screenshot of a macOS terminal window titled "flavio — fish /Users/flavio — -fish — 62x7". The window shows the command "crontab -l" being run, followed by the message "crontab: no crontab for flavio". The terminal has a dark theme with red, yellow, and green window controls.

```
[→ ~ crontab -l
crontab: no crontab for flavio
→ ~ ]
```

Cronjob

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/>

Cronjob

Crontab Generator - Generate

crontab-generator.org

Crontab Generator

Get frustrated with Cron on your server? Try our [Webcron Service](#).

If you want to periodically perform a task (e.g. sending Emails, backing up database, doing regular maintenance, etc.) at specified times and dates, there are two ways to set scheduled tasks:

Method 1: Use our [online cron job service](#) that will save you a headache.
Method 2: Use Cron available in Unix/Linux systems.

If you go with method 2, the following generator can help you produce a crontab syntax that you can copy & paste to your crontab file (You can open the file by using command `crontab -e`). Below the generated crontab syntax, a list of run times will be displayed too. The predictions will help you ensure that you set the time and date right.

Complete the following form to generate a crontab line

Ctrl-click (or command-click on the Mac) to select multiple entries

Minutes	Hours	Days
<input checked="" type="radio"/> Every Minute	<input checked="" type="radio"/> Every Hour	<input checked="" type="radio"/> Every Day
<input type="radio"/> Even Minutes	<input type="radio"/> Even Hours	<input type="radio"/> Even Days
<input type="radio"/> Odd Minutes	<input type="radio"/> Odd Hours	<input type="radio"/> Odd Days
<input type="radio"/> Every 5 Minutes	<input type="radio"/> Every 6 Hours	<input type="radio"/> Every 5 Days
<input type="radio"/> Every 15 Minutes	<input type="radio"/> Every 12 Hours	<input type="radio"/> Every 10 Days
<input type="radio"/> Every 30 Minutes		<input type="radio"/> Every Half Month
0	Midnight	1
1	1am	2
2	2am	3
3	3am	4
4	4am	5
5	5am	6
6	6am	7
7	7am	8
8	8am	9
9	9am	10

Months	Weekday
<input checked="" type="radio"/> Every Month	<input checked="" type="radio"/> Every Weekday
<input type="radio"/> Even Months	<input type="radio"/> Monday-Friday
<input type="radio"/> Odd Months	<input type="radio"/> Weekend Days
<input type="radio"/> Every 4 Months	
<input type="radio"/> Every Half Year	
Jan	Sun
Feb	Mon
Mar	Tue
Apr	Wed
May	Thu
Jun	Fri
Jul	Sat
Aug	
Sep	
Oct	

Command To Execute

Command Examples:
Execute PHP script:
`/usr/bin/php /home/username/public_html/cron.php`

Cronjob

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

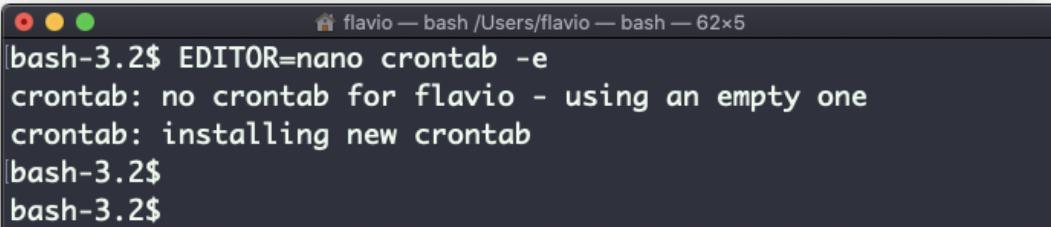
I chose to run a script located in /Users/flavio/test.sh every 12 hours. This is the crontab line I need to run:

```
* */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:



A screenshot of a terminal window titled "flavio — bash /Users/flavio — bash — 62x5". The user runs the command "EDITOR=nano crontab -e". The terminal outputs: "crontab: no crontab for flavio - using an empty one" and "crontab: installing new crontab". The user then exits with "ctrl-X" and saves with "y". The prompt "bash-3.2\$" appears twice at the bottom.

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

```
crontab -l
```

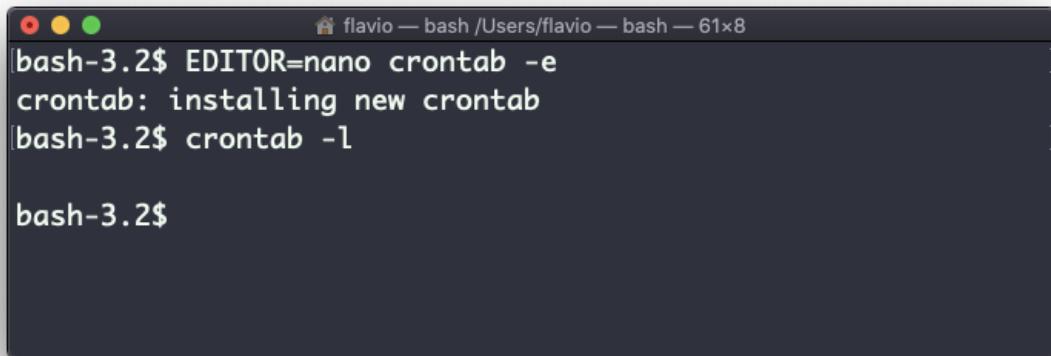
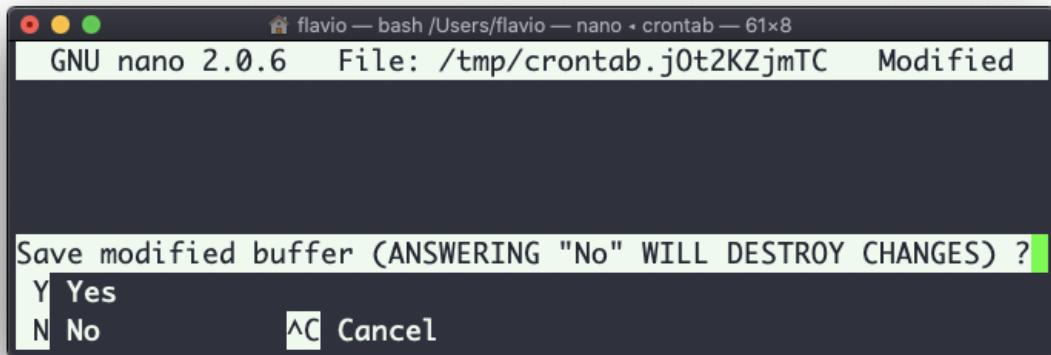
Cronjob



```
flavio — bash /Users/flavio — bash — 61x8
bash-3.2$ crontab -l
* */12 * * * /Users/flavio/test.sh >/dev/null 2>&1

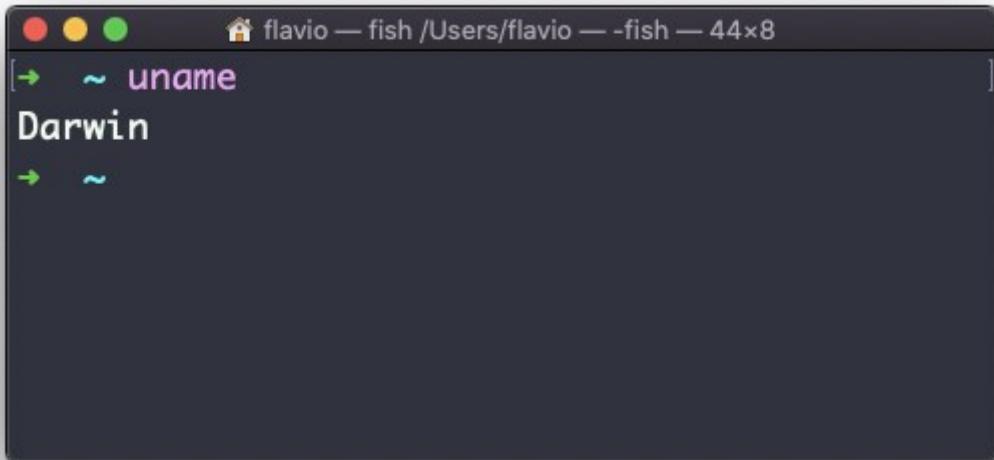
bash-3.2$
```

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:



```
flavio — fish /Users/flavio — -fish — 44x8
[→ ~ uname
Darwin
[→ ~
```

A screenshot of a macOS terminal window titled "flavio — fish /Users/flavio — -fish — 44x8". The window shows the command "uname" being run, followed by the output "Darwin". The terminal has a dark background with light-colored text.

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):



```
flavio — fish /Users/flavio — -fish — 44x8
[→ ~ uname -mp
x86_64 i386
[→ ~ |
```

A screenshot of a macOS terminal window titled "flavio — fish /Users/flavio — -fish — 44x8". The window shows the command "uname -mp" being run, followed by the output "x86_64 i386". The terminal has a dark background with light-colored text.

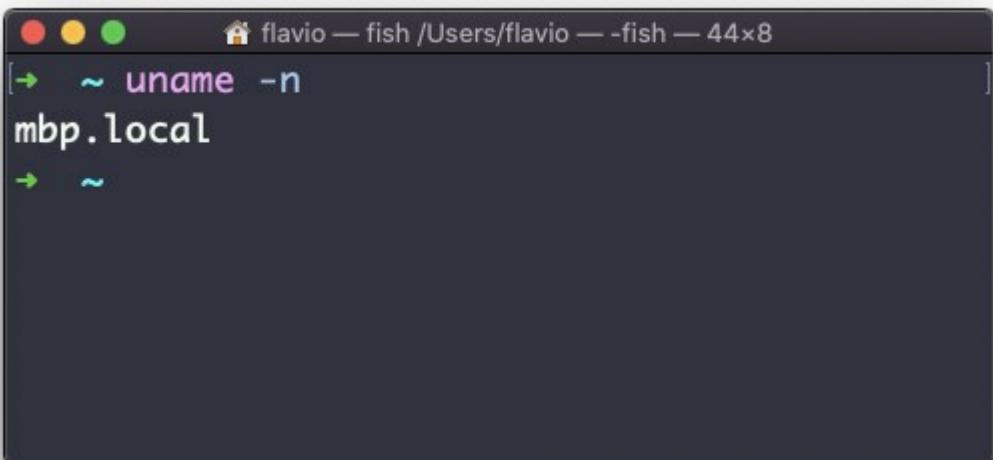
Uname

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



```
[~ flazio — fish /Users/flazio — -fish — 44x8
[→ ~ uname -srv
Darwin 19.6.0 Darwin Kernel Version 19.6.0:
Thu Jun 18 20:49:00 PDT 2020; root:xnu-6153.
141.1~1/RELEASE_X86_64
→ ~ ]
```

The n option prints the node network name:



```
[~ flazio — fish /Users/flazio — -fish — 44x8
[→ ~ uname -n
mbp.local
→ ~ ]
```

Uname

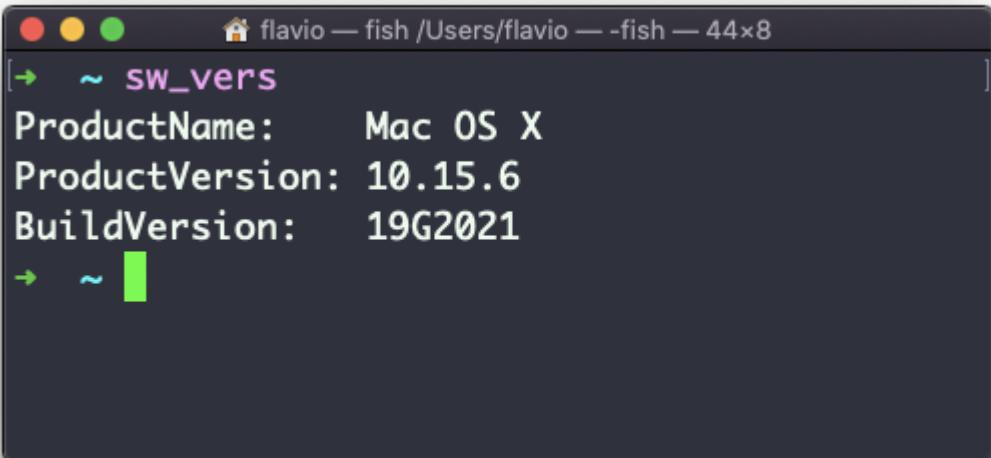
The a option prints all the information available:



```
[~] flazio — fish /Users/flazio — -fish — 44x8
[~] ~ uname -a
Darwin mbp.local 19.6.0 Darwin Kernel Version 19.6.0: Thu Jun 18 20:49:00 PDT 2020; root :xnu-6153.141.1~1/RELEASE_X86_64 x86_64
[~] ~
```

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"



```
[~] flazio — fish /Users/flazio — -fish — 44x8
[~] ~ sw_vers
ProductName: Mac OS X
ProductVersion: 10.15.6
BuildVersion: 19G2021
[~] ~
```

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
```

env

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
```

env

Removing the -i option will make PATH available again inside the program:

A screenshot of a macOS terminal window titled "flavio — fish /Users/flavio — -fish — 73x5". The window shows the command "env NAME=flavio node app.js" being run. The output displays the environment variable "NAME=flavio" followed by the system's PATH: "/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/Apple/usr/bin".

```
flavio
~ env NAME=flavio node app.js
flavio
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/Apple/usr/bin
~
```

The env command can also be used to print out all the environment variables, if ran with no options:

```
env
```

it will return a list of the environment variables set, for example:

```
HOME=/Users/flavio
LOGNAME=flavio
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/Apple/usr/bin
PWD=/Users/flavio
SHELL=/usr/local/bin/fish
```

You can also make a variable inaccessible inside the program you run, using the `-u` option, for example this code removes the `HOME` variable from the command environment:

```
env -u HOME node app.js
```

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:/Library/Apple/usr/bin
PWD=/Users/flavio
SHELL=/usr/local/bin/fish
```

with a few more lines, usually.

You can append a variable name as a parameter, to only show that variable value:

```
printenv PATH
```

Printenv

```
flavio — fish /Users/flavio — -fish — 46x5
[→ ~ printenv PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/
Library/Apple/usr/bin
→ ~ ]
```

Conclusion

Thanks a lot for reading this book.

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

Contacts us

<https://cyberpublicschool.com/>

<https://www.instagram.com/cyberpublicschool/>

**Phone no.: +91 9631750498 India
+61 424866396 Australia**



Our Successful Oscp Student.

<https://cyberpublicschool.com/>