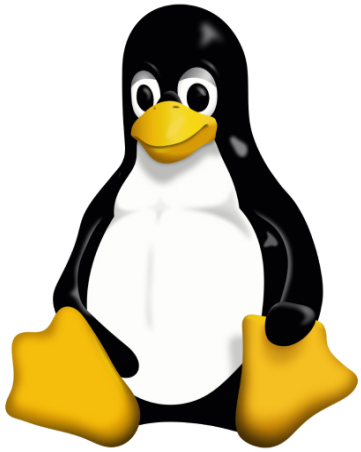
THE LINUX



COMMANDS HANDBOOK

Flavio Copes

Table of Contents

uniq

diff

echo

chown

chmod

umask

du

df

basename

dirname

ps

top

kill

killall

jobs

bg

fg

type

which

nohup

xargs

vim

emacs

nano

whoami

who

su

sudo

Preface

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

I find this approach gives a well-rounded overview.

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

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

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

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

You can reach me on Twitter @flaviocopes.

Enjoy!

Introduction to Linux and shells

Linux is an operating system, like macOS or Windows.

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

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

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

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

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

Linux is the ultimate freedom.

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

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

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

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

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

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

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

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

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

I don't have a Linux computer.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

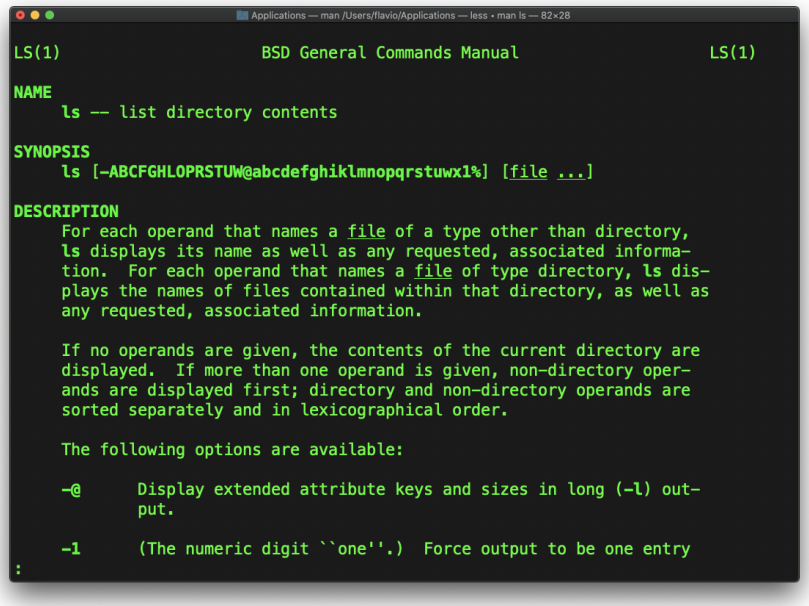
Each single shell has its own unique features and advanced usage, but they all share a common functionality: they can let you execute programs, and they can be programmed.

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

man

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

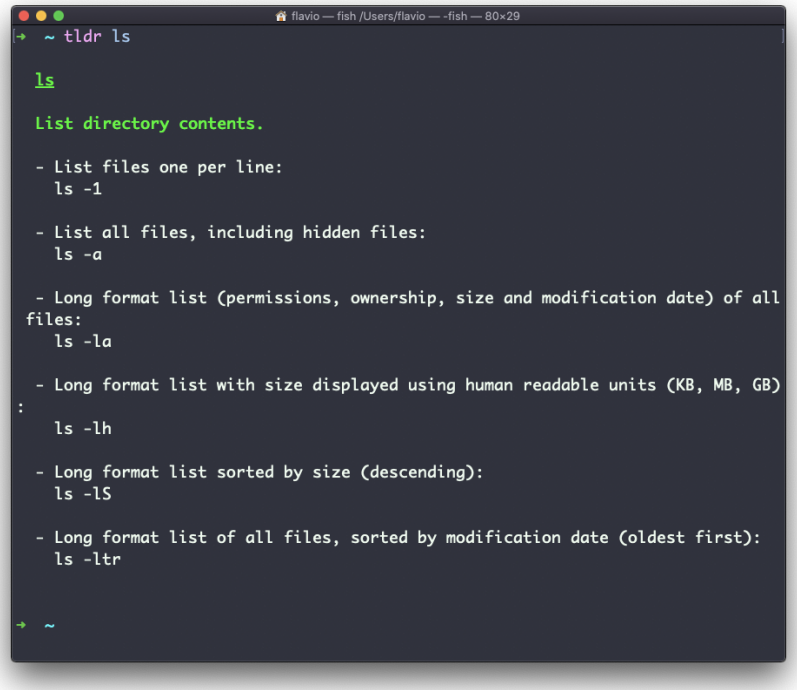
Every time I don't know how to use a command, I type `man <command>` to get the manual:



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

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

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



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

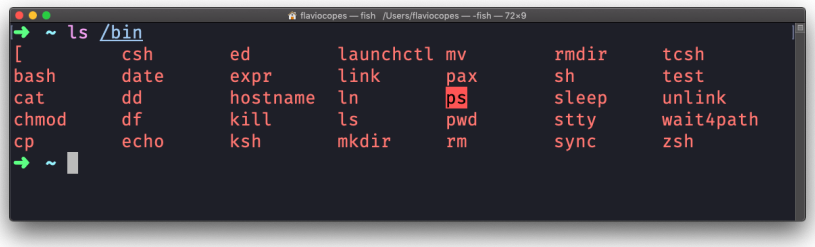
ls

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

```
ls
```

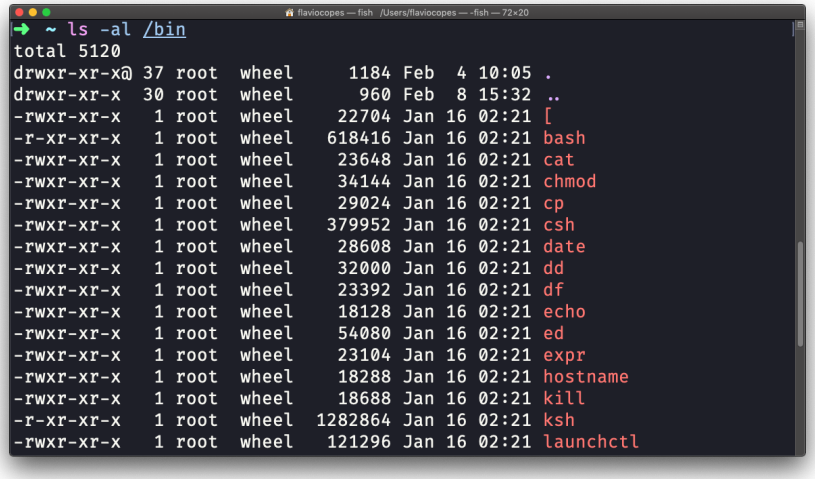
If you add a folder name or path, it will print that folder contents:

```
ls /bin
```



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

```
ls -al /bin
```



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

You have, from left to right:

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

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

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

cd

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

Example:

```
mkdir fruits
cd fruits
```

Now you are into the `fruits` folder.

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

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

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

You can use it to form a path:

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

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

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

```
cd /etc
```

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

pwd

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

```
pwd
```

It will print the current folder path.

mkdir

You create folders using the `mkdir` command:

```
mkdir fruits
```

You can create multiple folders with one command:

```
mkdir dogs cars
```

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

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

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

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

rmdir

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

```
mkdir fruits
rmdir fruits
```

You can also delete multiple folders at once:

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

The folder you delete must be empty.

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

```
rm -rf fruits cars
```

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

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

mv

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

```
touch test
mv pear new_pear
```

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

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

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

cp

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

```
touch test
cp apple another_apple
```

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

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

open

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

```
open <filename>
```

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

```
open <directory name>
```

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

```
open .
```

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

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

```
open <application name>
```

touch

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

```
touch apple
```

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

find

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

Let's learn it by example.

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

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

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

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

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

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

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

You can search under multiple root trees:

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

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

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

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

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

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

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

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

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

Search files edited more than 3 days ago

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

Search files edited in the last 24 hours

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

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

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

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

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

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

ln

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

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

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

Hard links

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

A hard link is created using

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

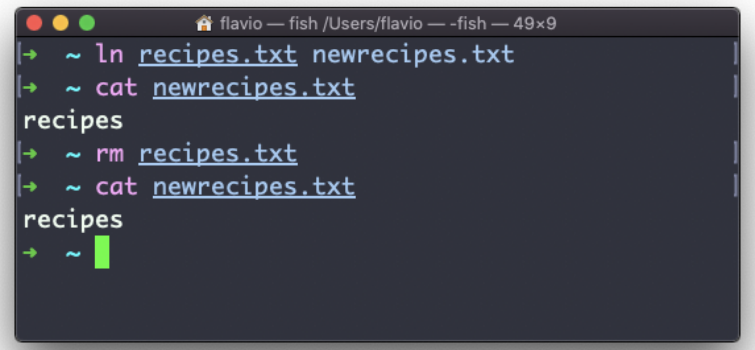
For example, say you have a file called recipes.txt. You can create a hard link to it using:

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

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

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

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



Soft links

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

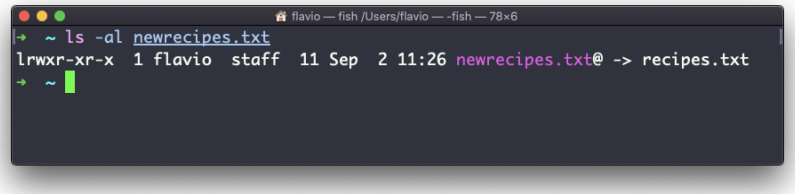
You create soft links using the `-s` option of `ln`:

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

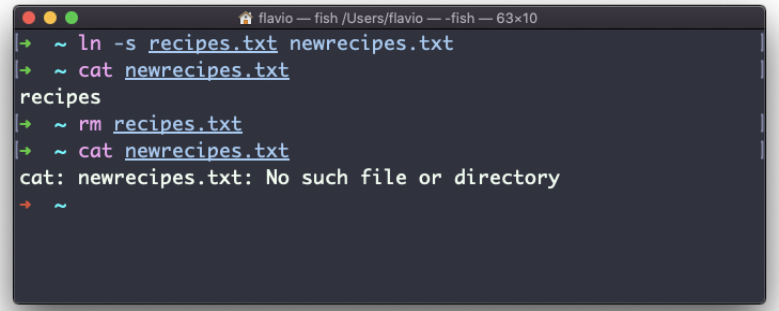
For example, say you have a file called recipes.txt. You can create a soft link to it using:

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

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



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



gzip

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

Here's the simplest usage:

```
gzip filename
```

This will compress the file, and append a `.gz` extension to it. The original file is deleted. To prevent this, you can use the `-c` option and use output redirection to write the output to the `filename.gz` file:

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

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

Or you can use the `-k` option:

```
gzip -k filename
```

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

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

```
gzip -1 filename
```

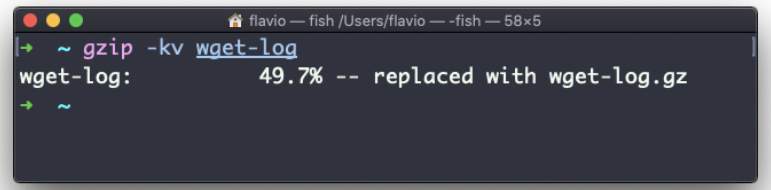
You can compress multiple files by listing them:

```
gzip filename1 filename2
```

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

```
gzip -r a_folder
```

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



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

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

gunzip

The `gunzip` command is basically equivalent to the `gzip` command, except the `-d` option is always enabled by default.

The command can be invoked in this way:

```
gunzip filename.gz
```

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

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

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

tar

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

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

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

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

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

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

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

> the `x` option stands for *extract*

and to extract them to a specific directory, use:

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

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

`tar` is often used to create a **compressed archive**, gzipping the archive.

This is done using the `z` option:

```
tar -czf archive.tar.gz file1 file2
```

This is just like creating a tar archive, and then running `gzip` on it.

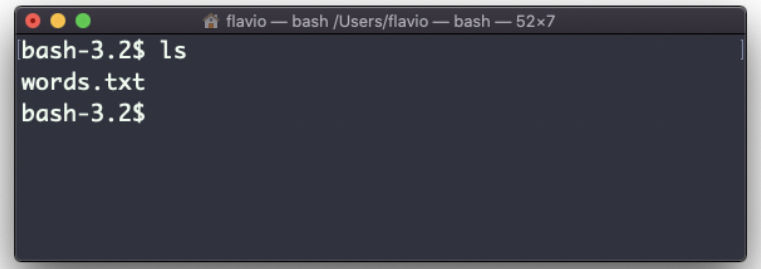
To unarchive a gzipped archive, you can use `gunzip`, or `gzip -d`, and then unarchive it, but `tar -xf` will recognize it's a gzipped archive, and do it for you:

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

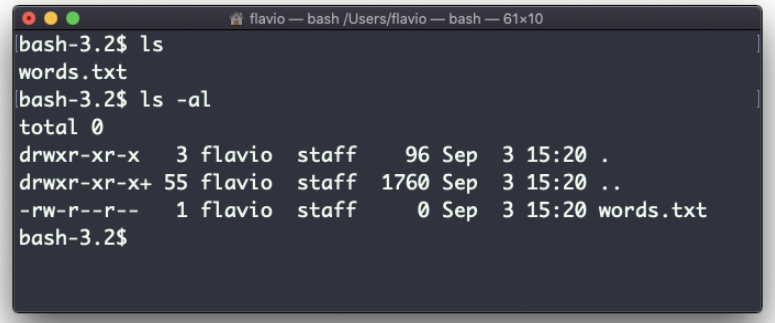
alias

It's common to always run a program with a set of options you like using.

For example, take the `ls` command. By default it prints very little information:



while using the `-al` option it will print something more useful, including the file modification date, the size, the owner, and the permissions, also listing hidden files (files starting with a `.`:

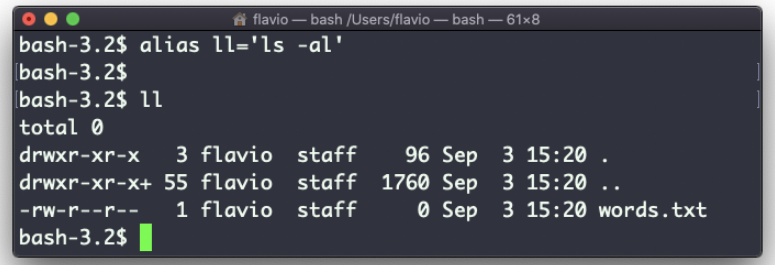


You can create a new command, for example I like to call it `ll`, that is an alias to `ls -al`.

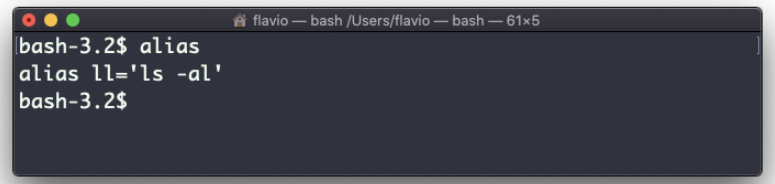
You do it in this way:

```
alias ll='ls -al'
```

Once you do, you can call `ll` just like it was a regular UNIX command:



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



The alias will work until the terminal session is closed.

To make it permanent, you need to add it to the shell configuration, which could be `~/.bashrc` or `~/.profile` or `~/.bash_profile` if you use the Bash shell, depending on the use case.

Be careful with quotes if you have variables in the command: using double quotes the variable is resolved at definition time, using single quotes it's resolved at invocation time. Those 2 are different:

```
alias lsthis="ls $PWD"
alias lscurrent='ls $PWD'
```

$PWD refers to the current folder the shell is into. If you now navigate away to a new folder, `lscurrent` lists the files in the new folder, `lsthis` still lists the files in the folder you were when you defined the alias.

cat

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

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

```
cat file
```

You can print the content of multiple files:

```
cat file1 file2
```

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

```
cat file1 file2 > file3
```

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

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

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

```
cat -n file1
```

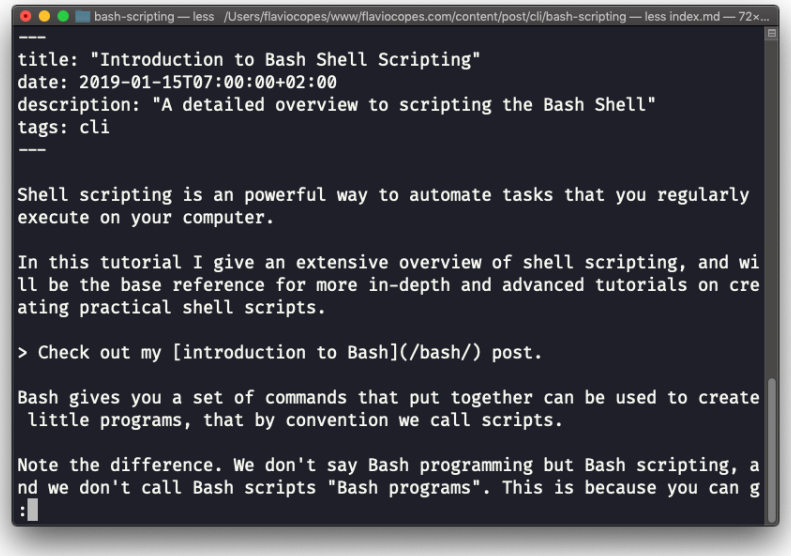
You can only add a number to non-blank lines using `-b`, or you can also remove all the multiple empty lines using `-s`.

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

less

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

Usage: `less <filename>`.



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

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

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

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

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

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

tail

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

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

To exit, press `ctrl-C`.

You can print the last 10 lines in a file:

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

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

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

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

wc

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

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

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

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

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

We can tell it to just count the lines:

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

or just the words:

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

or just the bytes:

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

Bytes in ASCII charsets equate to characters, but with non-ASCII charsets, the number of characters might differ because some characters might take multiple bytes, for example this happens in Unicode.

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

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

grep

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

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

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

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

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

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

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

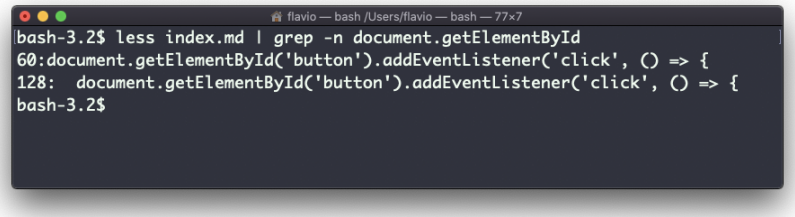
One very useful thing is to tell grep to print 2 lines before, and 2 lines after the matched line, to give us more context. That's done using the `-C` option, which accepts a number of lines:

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

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

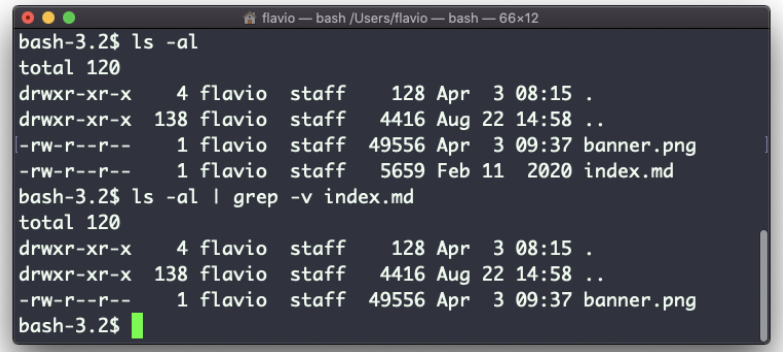
As mentioned, you can use grep to filter the output of another command. We can replicate the same functionality as above using:

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



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

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

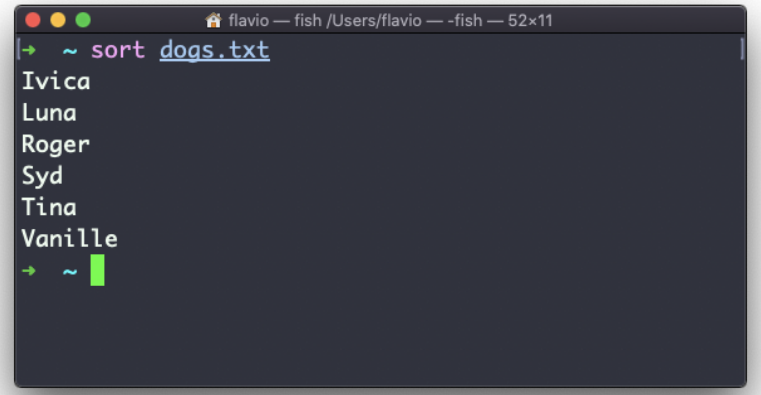


sort

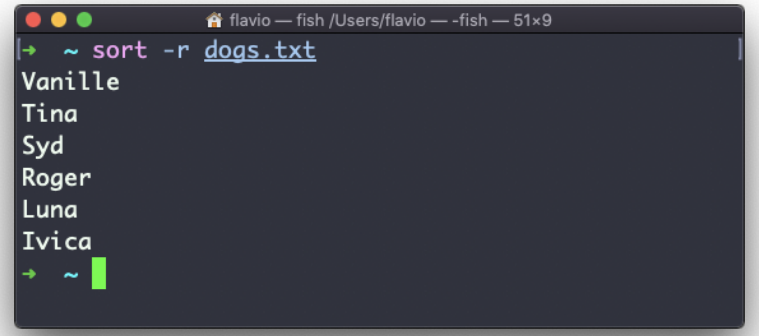
Suppose you have a text file which contains the names of dogs:

This list is unordered.

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

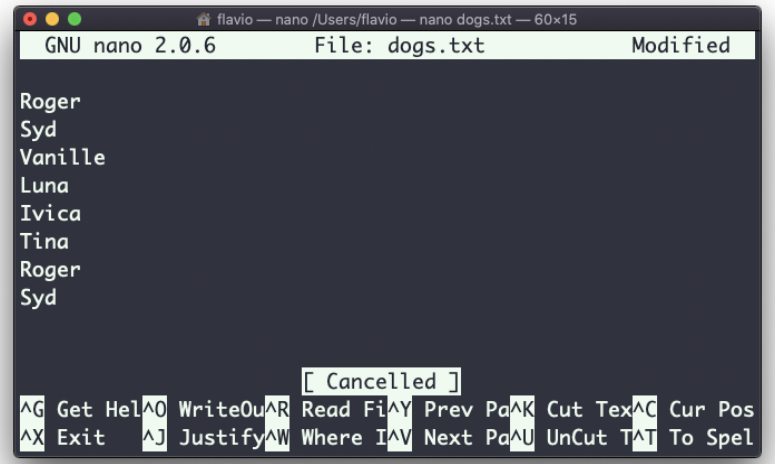


Use the `r` option to reverse the order:

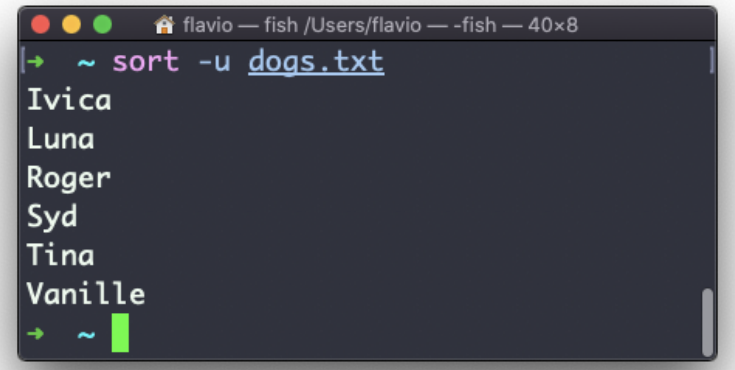


Sorting by default is case sensitive, and alphabetic. Use the `--ignore-case` option to sort case insensitive, and the `-n` option to sort using a numeric order.

If the file contains duplicate lines:



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



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

```
ls | sort
```

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


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