Git Cheat Sheet

TerminalVibes — The Terminal for Vibe Coders · neovand.github.io/terminalvibes

Some of these commands have a blank to fill in. Two notations mark one:

<file> Your own word goes here — cat <file> means cat notes.txt. Never type the brackets; < is a real operator in the shell.

PID · NAME · URL The same blank in the capitals man pages use. Swap in the actual number, name, or address.

Orientation

pwd

Print the directory you are standing in

The full absolute path from / down to here. Half of all terminal confusion is being in the wrong folder — when anything misbehaves, pwd first.

ls

List the files and folders here

Names only by default. Give it a path (ls src) to peek inside a folder without moving into it.

ls -a

List everything, including hidden dotfiles

Files whose names start with a dot (.bashrc, .git, .env) are hidden by default — -a reveals them, plus the special . and .. entries.

ls -l

Long listing: permissions, link count, owner, group, size, date

One file per line with the full story. The two names in the middle are the owner and its group — vibe staff means owner vibe, group staff. Single-letter flags cluster: ls -la is ls -l -a.

whoami

Print your username

A quick identity check — most useful after sudo, over ssh, or when a script behaves as if it belongs to someone else.

clear

Wipe the screen (Ctrl+L does the same)

Only clears the view — nothing is undone or lost, and you can still scroll up. On Windows cmd the equivalent is cls, but in WSL and Git Bash clear works as shown.

man *<command>*

Open the full manual page for a command

Angle brackets mean "put your own word here" — you type man ls, never the brackets. Scroll with arrows or space, / searches (press Enter to run the search), q quits. A command can have pages in several numbered sections: man 5 crontab asks for the file-format one. Git Bash on Windows ships without man — use <command> --help there.

***<command>* --help**

Quick usage summary and flag list

In a usage line, [square brackets] mean optional, ALL-CAPS words are placeholders, and ... means "as many as you like". A comma joins two spellings of one flag: -a, --all. A flag letter belongs to its command, not to the terminal — -r is recursive to cp and reverse to sort — so look it up rather than guessing. Great habit: run it on any command an AI suggests.

Navigation

cd *<dir>*

Move into a directory

Takes a relative path (cd projects) or an absolute one (cd /usr/local). Press TAB while typing — completion prevents the typos that cause most cd failures.

cd ..

Go up one level to the parent directory

Chain it to climb faster: cd ../.. goes up two levels in one hop.

cd

Jump home from anywhere (same as cd ~)

Home is /Users/you on macOS and /home/you on Linux. In WSL your Windows files live under /mnt/c — your Linux home is separate.

cd -

Bounce back to the previous directory

Toggles between your last two locations — perfect for hopping between a project and a log folder.

cd /

Go to the root of the filesystem

Everything on the machine lives somewhere beneath /. Look around with ls, but be careful running commands from here.

cd ~/projects

Go to a path spelled from your home folder

~ expands to your home directory, so this works no matter where you currently stand — the most reliable way to write personal paths.

Files & Folders

mkdir *<name>*

Create a new directory

Errors if it already exists — harmless, but mkdir -p stays quiet instead.

mkdir -p src/app/utils

Create nested directories in one go

-p is for parents: it makes every missing folder along the path and never complains if some already exist — the standard way to build a project skeleton.

touch *<file>*

Create an empty file (or update its timestamp)

If the file exists, only its modified time changes — the contents are untouched. Safe to run on anything.

cp *<src>* *<dest>*

Copy a file

If <dest> is a folder the copy lands inside it; if it is a filename you get a copy under that name. Overwrites silently — cp -i asks first.

cp -r *<dir>* *<dest>*

Copy a folder and everything inside it

Plain cp refuses directories; -r (recursive) descends into them. The classic quick backup: cp -r project project-backup.

mv *<src>* *<dest>*

Move a file — or rename it (same command)

mv old.txt new.txt renames; mv old.txt archive/ moves. It silently overwrites an existing destination — mv -i asks before clobbering.

rm *<file>*

Delete a file — permanently, no trash can

There is no undo and no recycle bin. Before deleting, ls the exact name you are about to remove; after, only backups or your editor's local history can help.

rm -r *<dir>*

Delete a folder and everything inside it

Recursive and permanent. Slow down when you see -rf: -f silences every warning. Wrong folder + wildcard + rm -rf is the classic terminal disaster.

rmdir *<dir>*

Delete a directory only if it is empty

The cautious cousin of rm -r — it refuses if anything is inside, which makes it a safe way to clean up folder skeletons.

open .

Open the current folder in Finder (macOS)

Linux: xdg-open . — WSL: explorer.exe . (note the dot). Handy bridge back to the GUI.

Viewing Files

```
cat <file>
```

Print a whole file to the screen

Best for short files. cat a huge or binary file and the screen floods — reach for less instead, and reset if the display garbles.

```
less <file>
```

Page through a big file comfortably

Space or arrows scroll, / searches forward, n repeats the search, q quits. man uses the same pager, so these keys work there too.

```
head <file>
```

Show the first 10 lines

head -n 25 shows 25. Perfect for checking what kind of file you are dealing with.

```
tail <file>
```

Show the last 10 lines

The end of a log file is usually where the newest — and most interesting — entries are. tail -5 is shorthand for tail -n 5: a bare number here is a count, unlike the 9 in kill -9, which is a signal number.

```
tail -f server.log
```

Follow a file live as new lines arrive

The standard way to watch a running server write its log. It never exits on its own — Ctrl+C stops watching.

```
wc -l <file>
```

Count the lines in a file

wc alone prints lines, words and bytes. Counting is the quickest sanity check that a pipeline or filter did what you expected.

```
file <file>
```

Ask what kind of file this is

Reads the contents, not the extension — it will tell you a ".txt" is really a PNG. Useful before you cat something questionable.

Text & Pipes

```
<command> | <command>
```

Pipe: feed one command's output into the next

The superpower. Small tools compose into big answers: history | grep ssh finds every ssh command you have ever run.

```
<command> > out.txt
```

Redirect output into a file — OVERWRITES it

> truncates the file to zero before writing. If the file mattered, it is already gone — reach for >> when in doubt.

```
<command> >> out.txt
```

Append output to the end of a file

Adds without destroying — the safe default for collecting results, building logs, or writing a script line by line with echo.

```
<command> 2> errors.txt
```

Capture error output separately

Every command has three channels: stdin (0), stdout (1) and stderr (2). 2> catches just the errors; 2>&1 sends stream 2 wherever stream 1 currently points — the &1 makes it a reference to a stream rather than a file named 1.

```
sort <file>
```

Sort lines alphabetically

sort -n sorts numerically (so 9 comes before 10), sort -r reverses. Sorting is also the required warm-up for uniq.

```
uniq -c
```

Collapse repeated adjacent lines and count them

-c prefixes each surviving line with how many it stood for — not grep's -c, which prints one total. Only removes duplicates that sit next to each other, which is why the recipe is always sort first, then uniq.

```
cut -d, -f1
```

Take one column from delimited text

-d sets the delimiter (a comma here), -f picks the field. Both are flags that take a value, and cut lets you jam it on: -f2 is -f 2. cut -d: -f1 /etc/passwd lists every username.

```
sort | uniq -c | sort -rn
```

The classic frequency pipeline: count and rank

Sort to group, count the groups, sort the counts biggest-first. Top IPs in a log, most-used commands in history — same three steps every time.

Searching

```
grep "<text>" <file>
```

Print the lines that contain the text

Quote the pattern so spaces and special characters survive the trip to grep. Its patterns are regular expressions, not globs: here * means "zero or more of the thing before it", . matches any single character, and ^ and $ anchor to the start and end of a line. So ERR* matches ER followed by any number of Rs — not "starts with ERR".

```
grep -i "<text>" <file>
```

Search ignoring upper/lower case

ERROR, Error and error all match. Add -n to show line numbers with each hit.

```
grep -rn "<text>" .
```

Search every file under this folder, recursively

-r descends into subdirectories, -n prints file and line numbers — the terminal's project-wide find-in-files.

```
grep -v "<text>"
```

Keep the lines that do NOT match

The filter-out flag: cat server.log | grep -v DEBUG hides the noise so the signal stands out.

```
grep -c "<text>" <file>
```

Count matching lines instead of printing them

How many errors today? grep -c ERROR server.log answers with one number.

```
find . -name "*.md"
```

Find files by name pattern, anywhere below here

find searches file NAMES; grep searches file CONTENTS. Quote the pattern so the shell hands the wildcard to find instead of expanding it early. This is also the answer when a glob is not enough: a glob never crosses a /, so *.tmp only matches this folder — find . -name "*.tmp" reaches the whole tree.

```
find . -type d -name "node_modules"
```

Find directories by name

-type d matches directories only, -type f files only. Combine with -name for precise hunts.

Text Surgery

```
sed 's/old/new/g' FILE
```

Replace old with new on every line

Read it as: s(ubstitute) / find this / replace with this / g = every match on the line. Prints the result — the file itself is untouched. Any delimiter works: s|/usr|/opt| saves escaping slashes.

```
sed '/DEBUG/d' FILE
```

Drop every line matching a pattern

Addresses pick lines, d deletes them. Also by position: sed 3d (line 3), sed 2,5d (a range), sed $d (the last line).

```
sed -n '40,55p' FILE
```

Print only lines 40–55

-n silences the default output, p prints the selected lines — the precision way to peek at the middle of a huge log. Regex addresses work too: sed -n /ERROR/p.

```
sed -i.bak 's/old/new/g' FILE
```

Edit the file in place — keeping a .bak backup

The house rule: never bare -i. The suffix saves each original as file.bak first — instant diff, instant undo (mv file.bak file). It is also the portable spelling: on macOS bare -i swallows the next argument as the backup suffix and fails with an error pointing nowhere near the real problem. When an agent proposes bare -i, amend it.

```
awk '{print $2}' FILE
```

Print the second column of every line

Fields split on runs of spaces — perfect for ragged, column-aligned output. $0 is the whole line; commas join fields with a space: awk '{print $1, $3}'. The single quotes are load-bearing: $2 here is awk's own language, not a shell variable, and unquoted the shell would swallow it and the braces first.

```
awk -F, '{print $2}' FILE
```

Same, but split on commas (CSV — comma-separated values)

-F sets the field separator. For clean single-character delimiters cut -d, -f2 does the same job; awk wins when spacing is messy.

```
awk '/error/ {print $1}' FILE
```

Pull a field from matching lines only

A /pattern/ guard filters which lines the action runs on — grep and column-pull in one step.

Processes & Ports

```
ps aux
```

List every running process

Bare ps shows only this terminal's handful; aux widens it to the whole machine — a all users, u the readable columns, x including processes with no terminal, which is where servers hide. They take no dash: ps uses an older option style. PID is the number you pass to kill, %CPU is measured against one core (so 340% is three cores, not an error), COMMAND is the interpreter's name — a JavaScript server shows as node. Your real output has more columns than the course shows.

```
pgrep NAME
```

Print the PIDs of processes matching a name

The short way to do ps aux | grep NAME. Exits 1 with no output when nothing matches.

```
kill PID
```

Ask a process to stop (SIGTERM)

A signal is a short fixed message the kernel delivers to a running process. They all start with SIG: SIGTERM is "terminate", and bare kill is kill -15, its number. A polite request — the program can save, clean up and exit, or catch the signal and ignore it, and then you escalate.

```
kill -9 PID
```

Force a process to stop (SIGKILL)

The 9 is a signal number, not a count — signal 9 is SIGKILL. Cannot be caught or refused, and skips all cleanup: unsaved work is lost and the lock file it was holding stays behind, so the next copy finds it and refuses to start. Last resort, not first.

```
lsof -i :3000
```

Find what is holding a port

lsof is "list open files"; -i narrows it to network connections. A port is a numbered door and only one program can hold one at a time — which is the whole of "address already in use". The only column you need is PID; (LISTEN) confirms that process is sitting at the door. Silence means the port is free.

```
command &
```

Run it in the background, keep your prompt

The shell answers with [1] and a PID. jobs lists what is backstage, fg %1 brings job 1 forward, kill %1 stops it. The job is tied to this shell: close the window and it goes too, so do not background a long build and walk away. Not to be confused with && (run on success) or 2>&1 (a stream reference).

```
Ctrl+Z then bg
```

Suspend the foreground job, resume it in the background

The rescue for when you started something long and forgot the &. Ctrl+Z hands your prompt back but stops the job dead — jobs shows it as Stopped, and no work happens until bg restarts it backstage (or fg brings it back).

Network & Secrets

```
curl localhost:3000/health
```

Ask a server directly and print its reply

The one-line way to check "is it actually running?" instead of trusting a claim. "Connection refused" means nothing is listening — check with lsof -i :3000.

```
curl -s -o out.json URL
```

Save the reply to a file, quietly

-o writes the body to a file instead of the screen; -s hides the progress meter, which you always want inside scripts and pipelines. -I asks for just the headers.

```
curl -s URL | jq -r .field
```

Fetch JSON and pull one value out of it

A jq filter is a path: . is everything, .latest is a key, .server.port goes deeper, .items[0] indexes a list. -r drops the quotes so the value can feed the next command.

```
echo 'KEY=value' > .env && chmod 600 .env
```

Put a secret in a file only you can read

A .env file is plain text: NAME=value lines, one per key ("env" is short for environment). Never type a key directly into a command — your shell history keeps it. Load it with source .env and use "$KEY", and add .env to .gitignore. Note .gitignore only stops git starting to track a file: if the key is already committed, revoke and reissue it in the provider's website.

```
ssh user@host
```

Open a shell on another machine

Every command in this course works there too. The prompt changes to show the other machine — check it before running anything destructive. exit comes home.

The Toolshed

```
brew install NAME
```

Install a tool (macOS)

A package manager installs other programs: it fetches the right build and drops it where $PATH already looks — which is all "installed" means. Check with which NAME. Linux: sudo apt install NAME (apt writes to folders that belong to the machine, so it needs sudo; brew does not). Windows: winget install NAME. npm i -g NAME is a different kind of thing — a per-language manager that can only install JavaScript tools. Homebrew is not on a stock Mac: install it once from the instructions at brew.sh.

```
tar -tzf archive.tar.gz
```

List what is inside an archive without unpacking

Peek first: an archive decides where its own files land, and a badly built one scatters them across your folder. unzip -l does the same for .zip.

```
tar -xzf archive.tar.gz
```

Extract a .tar.gz archive

x extract, z un-gzip, f this file. Swap x for c to create (tar -czf backup.tar.gz notes/) or t to list. The letters are separate instructions, not one magic word.

```
ln -s target linkname
```

Create a symlink — a signpost, not a copy

-s is symbolic: a tiny file whose whole content is a path to somewhere else. That -> arrow (and the leading l) in ls -l is a symlink. A relative target is worked out from the link's own folder, so moving the link breaks it. Deleting the link leaves the original; deleting the original leaves a broken link.

```
du -sh *
```

Size of every folder here, biggest hog obvious

-s summarises: one total per item instead of a line per file. -h makes the numbers human — K, M, G, each rung about a thousand of the one below. Measure before you delete, the same discipline as ls before rm. df -h shows how full the whole disk is.

Permissions

```
ls -l
```

Read the permission string on every file

The 10-character prefix decodes as type + rwx for user, group, other: -rwxr-xr-- means you can do anything, your group can read/run, others read only. The type character is - for a file, d for a directory, l for a symlink.

```
chmod +x <script>
```

Make a file executable

chmod takes a small grammar: + adds a permission, - removes it, and u/g/o/a pick the audience. A bare +x means a+x — all three audiences at once. The fix for "Permission denied" on your own script; after it, run the script with ./script.sh.

```
chmod 755 <file>
```

Recipe: you rwx, everyone else read/run

The standard mode for scripts and directories. Each digit is one audience: user, group, other. On a directory x does not mean "run" — it means you may go inside and reach what is in there.

```
chmod 644 <file>
```

Recipe: you read/write, everyone else read-only

The standard mode for ordinary files — nobody can execute it, others cannot change it.

```
sudo <command>
```

Run one command as the administrator (root)

Respect it: sudo bypasses every safety rail. Never sudo a command you do not understand — especially one an AI wrote. Git Bash has no sudo; WSL does.

```
sudo !!
```

Re-run the previous command with sudo

!! expands to your last command — the classic move after “Permission denied” on something you meant to run as root.

Environment

```
echo $HOME
```

Print the value of a variable

$NAME expands to the variable’s value before echo even runs. Try $USER, $SHELL, $PWD — the shell keeps its state in variables.

```
echo $PATH
```

The list of folders searched for commands

Colon-separated, checked left to right. “command not found” means the tool is not in any of these folders — or not installed at all.

```
which <command>
```

Print the full path of what would actually run

Answers “which python am I really running?” If which finds nothing, PATH is why your command is not found.

```
export VAR=value
```

Set a variable for this session and its children

Lasts until the terminal closes. To make it permanent, add the export line to ~/.bashrc or ~/.zshrc.

```
env
```

List every environment variable

The full picture of what programs inherit from your shell. Pipe it through grep to find one: env | grep PATH.

```
alias gs='git status'
```

Create a shortcut command

Type gs, get git status. Aliases live only in this session until you add them to your shell config file.

```
source ~/.bashrc
```

Reload your shell config without reopening the terminal

Running a file the ordinary way starts a child shell that exits and takes its changes with it; source runs the lines in the shell you are sitting in, so they stick. That is why you source a config instead of running it — and why a script full of cd never seems to move you. zsh users (macOS default): source ~/.zshrc.

Scripting

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

Shebang: the required first line of a bash script

Tells the system which program runs the file. env is itself a program: handed a name, it looks that name up on PATH, which is how the line finds bash wherever this machine keeps it. A script is just saved commands — everything you type interactively works inside one.

```
chmod +x script.sh
```

Make the script runnable

One-time step per script. Without it you get "Permission denied" even on a file you wrote yourself.

```
./script.sh
```

Run a script from the current directory

The ./ is required — the current directory is deliberately NOT on PATH, so you must point at the script explicitly.

```
bash script.sh
```

Run a script without the executable bit

Hands the file straight to bash, skipping chmod. Handy for quick tests of a script you just wrote.

```
echo "$1"
```

Use the first argument passed to your script

$1, $2, … hold the arguments; $@ is all of them. Quote them so arguments containing spaces stay whole.

```
echo $?
```

Exit code of the last command — 0 means success

Every command reports success (0) or failure (anything else) as it finishes. Which non-zero number it picks is that command’s own business and means nothing outside it — only "zero or not zero" is portable. Scripts, && and || all decide from this number.

```
<command> && <command>
```

Run the second only if the first SUCCEEDS

npm test && npm run deploy deploys only on green tests. Beware the classic horror: if cd fails, the next command runs in the wrong folder — && stops that.

```
<command> || <command>
```

Run the second only if the first FAILS

The fallback operator: npm test || echo "tests failed" — the right-hand side is the plan B. Careful chaining all three: a && b || c runs left to right with neither operator outranking the other, so if b fails the || branch fires even though a succeeded. It is not the one-branch-or-the-other shape it looks like.

```
<command> ; <command>
```

Run both, one after the other, regardless

No safety logic at all — the second runs even if the first exploded. Prefer && when the second step depends on the first.

Panic Button

```
Ctrl+C
```

Cancel the running command — or discard a typed line

Typed a scary command? Just don’t press Enter — Ctrl+C throws the line away unrun. Nothing you type executes until you press Enter.

```
q
```

Quit a full-screen pager (less, man, git log)

A “frozen” terminal is very often just a pager waiting politely. If q does nothing, try Ctrl+C first, then q.

```
lsof -i :3000
```

"Address already in use" — find and stop the squatter

Prints the process holding the port, with its PID. kill that PID, then start your server again. Silence from lsof means the port is actually free.

```
:q!
```

Trapped in vim? Press Esc, then type :q! and Enter

Quits without saving (:wq saves instead). Git and other tools open vim as an editor without warning — this is the universal exit. The nano equivalent is Ctrl+X.

```
Ctrl+D
```

Close the shell (same as typing exit)

Sends “end of input”. If a program is waiting for input you did not intend to give, Ctrl+D on an empty line often releases it.

```
reset
```

Fix a garbled, glitchy terminal display

After cat-ing a binary file the screen can fill with junk — reset repaints everything. It takes a second; your session survives.

```
echo rm *.log
```

Preview what a wildcard will hit — before deleting

echo expands the glob harmlessly and shows the exact file list rm would receive. Make this reflex and rm loses most of its teeth.

```
rm
```

Deleted a file? There is no undo — but check your editor

rm skips the trash entirely. VS Code’s Timeline view keeps local history of files you edited, and if the project uses git, a committed file can be restored.

```
history
```

What did I actually run? Read the record

Numbered list of your recent commands — the first step of any post-incident investigation, and Ctrl+R searches it interactively.