GUIDE COMPLET · NIVEAU DÉBUTANT

Le Terminal pour les débutants

Comprendre et maîtriser Bash, Shell & Zsh
en partant de zéro

Terminal Shell Bash Zsh

```
vous@ordinateur:~$ echo "Bienvenue dans la ligne de commande !"
Bienvenue dans la ligne de commande !
vous@ordinateur:~$ _
```

Un cours pédagogique, structuré et illustré · Du premier clic au premier script

Table des matières

CHAPITRE 1

Comprendre les concepts de base

Avant de taper la moindre commande, prenons cinq minutes pour démêler quatre mots qu'on confond souvent : terminal, shell, Bash et Zsh. Une fois cette image claire en tête, tout le reste devient beaucoup plus simple.

Une analogie pour tout comprendre

Imaginez que vous voulez parler à votre ordinateur. Habituellement, vous le faites avec la souris : vous cliquez sur des icônes, vous glissez des fichiers. C'est ce qu'on appelle l'**interface graphique** (les fenêtres, les boutons…). Mais il existe une autre façon de discuter avec la machine : en lui **écrivant des ordres au clavier**. C'est la **ligne de commande**.

Pour cette conversation écrite, trois éléments entrent en jeu. Reprenons l'image d'une conversation téléphonique :

Élément	Rôle	Analogie
Le terminal	La fenêtre dans laquelle vous tapez et lisez le texte	Le téléphone (l'appareil par lequel passe la voix)
Le shell	Le programme qui *comprend* vos ordres et les exécute	L'interprète qui traduit ce que vous dites
Bash / Zsh	Deux interprètes différents, avec leurs accents et leurs habitudes	Deux traducteurs : l'un classique, l'autre plus moderne

Le terminal : la fenêtre

Le **terminal** (aussi appelé « émulateur de terminal » ou « console ») est simplement une **application**. Quand vous l'ouvrez, vous obtenez une fenêtre, souvent à fond sombre, où du texte s'affiche. Le terminal ne fait *rien* de lui-même : il se contente d'afficher ce que vous tapez et de montrer les réponses. C'est une vitre entre vous et le shell.

> **À savoir**
>
> Sur macOS, l'application s'appelle « Terminal ». Sur Linux on trouve « GNOME Terminal », « Konsole », etc. Sur Windows, il y a « Windows Terminal » et « PowerShell ». Ce sont toutes des fenêtres dans lesquelles tourne un shell.

Le shell : le cerveau qui comprend vos ordres

Le **shell** (« coquille » en anglais) est le vrai programme intelligent. Quand vous écrivez `ls` et que vous appuyez sur Entrée, c'est le shell qui :

1. lit votre texte,
2. comprend que vous voulez lister des fichiers,
3. demande au système d'exploitation de le faire,
4. et renvoie le résultat au terminal pour affichage.

Le mot « shell » vient de l'idée d'une *coquille* autour du noyau (« kernel ») du système : c'est l'enveloppe avec laquelle l'humain dialogue, plutôt que de parler directement au cœur de la machine.

Bash et Zsh : deux shells parmi d'autres

Il n'existe pas un seul shell, mais plusieurs. Ce sont des programmes concurrents qui font globalement la même chose, avec des différences de confort et de fonctionnalités.

Shell	En bref
`sh`	L'ancêtre (Bourne shell), minimaliste. Encore présent partout pour la compatibilité.
`bash`	**Bourne Again Shell** : le plus répandu, standard sur la plupart des Linux. C'est notre shell de référence dans ce cours.
`zsh`	**Z shell** : très proche de Bash mais plus moderne et personnalisable. Shell par défaut de macOS depuis 2019.
`fish`	Encore plus convivial pour les débutants, mais moins « standard ».

> **À retenir**
>
> Le **terminal** est la fenêtre. Le **shell** est le programme qui interprète vos commandes à l'intérieur. **Bash** et **Zsh** sont deux shells différents. **99 % de ce que vous allez apprendre fonctionne à l'identique dans Bash et dans Zsh** — vous n'avez donc pas à choisir tout de suite.

Pourquoi apprendre la ligne de commande ?

On pourrait croire que c'est dépassé. C'est tout le contraire. Voici ce que la ligne de commande vous apporte :

- **Rapidité.** Renommer 300 fichiers prend une seconde au lieu d'une heure de clics.
- **Puissance.** Certaines opérations n'existent tout simplement pas dans l'interface graphique.
- **Automatisation.** Une tâche répétitive peut être enregistrée dans un *script* et relancée à volonté (chapitre 10).
- **Universalité.** Les serveurs, les conteneurs, le cloud n'ont souvent *aucune* interface graphique : la ligne de commande est la seule porte d'entrée.

- **Incontournable pour le développement.** Git, Docker, Node, Python… presque tous les outils modernes se pilotent au terminal.

Astuce de débutant

N'essayez pas de tout mémoriser. Personne ne connaît toutes les commandes par cœur. L'objectif est de comprendre la *logique* : une fois qu'elle est acquise, une recherche rapide suffit à retrouver l'option précise dont vous avez besoin.

Vérifiez votre compréhension

1. Dans l'analogie du téléphone, qui joue le rôle de « l'interprète » ?
2. Bash et Zsh sont-ils des terminaux ou des shells ?
3. Citez deux avantages concrets de la ligne de commande par rapport à la souris.

CHAPITRE 2

Premiers pas avec le terminal

Place à la pratique. Nous allons ouvrir le terminal, comprendre ce que veut dire la petite ligne qui s'affiche, taper nos toutes premières commandes, et surtout apprendre le réflexe le plus important : savoir demander de l'aide.

Ouvrir le terminal

Système	Comment ouvrir le terminal
Linux	Raccourci `Ctrl + Alt + T`, ou cherchez « Terminal » dans le menu des applications.
macOS	Ouvrez `Spotlight` (`Cmd + Espace`), tapez « Terminal », validez.
Windows	Installez « Windows Terminal » (recommandé), ou activez WSL pour disposer d'un vrai Linux. PowerShell existe aussi mais utilise une autre syntaxe.

> **Windows : un cas un peu à part**
>
> Windows utilise nativement *PowerShell* et l'ancien *cmd*, dont les commandes diffèrent de Bash. Pour suivre ce cours dans les meilleures conditions, installez le **WSL** (« Windows Subsystem for Linux ») : vous obtiendrez un vrai environnement Linux avec Bash, à l'intérieur de Windows.

L'invite de commande (le « prompt »)

Une fois le terminal ouvert, vous voyez une ligne qui ressemble à ceci :

```
marie@portable:~/Documents$
```

Cette ligne s'appelle l'**invite de commande** ou *prompt*. Elle attend que vous tapiez quelque chose. Décortiquons-la :

- `marie` — votre nom d'utilisateur ;
- `portable` — le nom de la machine (après le `@`) ;
- `~/Documents` — le dossier dans lequel vous vous trouvez actuellement (le `~` signifie « mon dossier personnel », nous y reviendrons) ;
- `$` — le symbole qui marque la fin du prompt. Tout ce que vous tapez vient après.

Le $ et le #

Un prompt qui se termine par `$` signifie que vous êtes un utilisateur normal. S'il se termine par `#`, vous êtes *administrateur* (root) : redoublez alors de prudence, car vous pouvez tout modifier sur la machine. Dans ce cours, le `$` en début de ligne sert juste à indiquer « voici une commande à taper » — vous ne tapez jamais le `$` lui-même.

Anatomie d'une commande

Presque toutes les commandes suivent le même schéma :

```
# commande  [options]  [arguments]
$ ls -l /etc
```

- **La commande** (`ls`) : l'action à effectuer.
- **Les options** (`-l`) : elles modifient le comportement. Elles commencent par un tiret `-` (forme courte) ou deux tirets `--` (forme longue, ex. `--all`).
- **Les arguments** (`/etc`) : ce sur quoi la commande agit (un fichier, un dossier, du texte…).

Astuce

Les options courtes se combinent souvent : `ls -l -a` peut s'écrire `ls -la` . Pratique et très courant.

Vos toutes premières commandes

Essayez-les une par une. Tapez la commande, appuyez sur **Entrée**, observez le résultat.

```
TERMINAL
$ echo Bonjour
Bonjour

$ whoami      # affiche votre nom d'utilisateur
marie

$ date        # affiche la date et l'heure
vendredi 29 mai 2026, 14:07:33 (UTC+2)

$ pwd         # affiche le dossier courant (Print Working Directory)
/home/marie

$ cal         # affiche un petit calendrier du mois
```

La commande `echo` « fait écho » : elle réaffiche simplement ce que vous lui donnez. C'est la commande la plus simple qui soit, et on l'utilise tout le temps dans les scripts pour afficher des messages.

Attention à la casse

Le terminal distingue les majuscules des minuscules. `echo` fonctionne, mais `Echo` ou `ECHO` donneront une erreur « command not found ». De manière générale, tout est sensible à la casse : un dossier `Photos` est différent d'un dossier `photos`.

Le réflexe n°1 : demander de l'aide

Vous ne retiendrez jamais toutes les options d'une commande, et ce n'est pas grave. Voici trois façons d'obtenir de l'aide sans quitter le terminal :

```
$ ls --help      # résumé rapide des options (le plus pratique)
$ man ls         # manuel complet et détaillé (on quitte avec la touche q)
$ type ls        # indique ce qu'est réellement "ls"
```

La commande `man` (pour « manual ») ouvre un manuel que l'on parcourt avec les flèches ou la barre Espace, et que l'on quitte en appuyant sur la touche **q**. C'est dense mais c'est la documentation officielle.

Astuce moderne

L'outil `tldr` (à installer) donne des exemples concrets et lisibles, bien plus digestes que `man` pour un débutant. `tldr tar` vous montre directement les usages les plus fréquents de la commande.

À vous de jouer

1. Affichez votre nom d'utilisateur, puis la date du jour.
2. Faites afficher la phrase `J'apprends le terminal` avec `echo`.
3. Ouvrez le manuel de la commande `echo` avec `man echo`, puis quittez-le.
4. Quelle est la différence entre une *option* et un *argument* ?

CHAPITRE 3

Naviguer dans le système de fichiers

Au terminal, vous vous déplacez de dossier en dossier sans jamais voir vos fichiers à l'écran comme dans un explorateur. Ce chapitre vous apprend à savoir où vous êtes, à regarder autour de vous, et à vous déplacer. C'est la base de tout le reste.

L'arborescence des fichiers

Sur Linux et macOS, tous les fichiers sont organisés comme un **arbre** renversé. Tout part d'un unique point de départ appelé la **racine**, notée `/`. À partir de là, les dossiers contiennent d'autres dossiers, qui contiennent des fichiers, et ainsi de suite.

```
/                    ← la racine, le point de départ de tout
├── home/
│   └── marie/       ← votre dossier personnel
│       ├── Documents/
│       ├── Images/
│       └── notes.txt
├── etc/             ← fichiers de configuration du système
└── usr/             ← programmes installés
```

> **Le dossier personnel et le ~**
>
> Votre **dossier personnel** (par exemple `/home/marie` sur Linux, `/Users/marie` sur macOS) est l'endroit où vivent vos fichiers. Il a un raccourci magique : le caractère `~` (le « tilde »). Partout, `~` veut dire « mon dossier personnel ». Ainsi `~/Documents` est un raccourci pour `/home/marie/Documents`.

Les trois commandes de navigation

Commande	Signifie	Rôle
`pwd`	Print Working Directory	« Où suis-je ? » — affiche le dossier courant
`ls`	List	« Que vois-je ? » — liste le contenu du dossier
`cd`	Change Directory	« J'y vais » — change de dossier

pwd — savoir où l'on est

```
$ pwd
/home/marie
```

ls — regarder le contenu

```
$ ls
Documents  Images  Musique  notes.txt

$ ls -l          # format long : détails (taille, date, permissions)
drwxr-xr-x  2 marie marie 4096 mai 28 09:12 Documents
-rw-r--r--  1 marie marie  214 mai 29 14:03 notes.txt

$ ls -a          # affiche aussi les fichiers cachés (qui commencent par un point)
$ ls -lh         # tailles "humaines" : Ko, Mo au lieu d'octets bruts
```

> **Les fichiers cachés**
>
> Tout fichier ou dossier dont le nom commence par un point (ex. `.bashrc`) est **caché** : `ls` ne le montre pas par défaut. Il faut `ls -a` pour les voir. Ces fichiers servent souvent à la configuration ; nous les retrouverons au chapitre 8.

cd — se déplacer

```
$ cd Documents     # entre dans le dossier Documents
$ cd ..            # remonte d'un niveau (dossier parent)
$ cd ~             # retourne directement au dossier personnel
$ cd               # (sans rien) fait la même chose : retour à la maison
$ cd -             # revient au dossier précédent
$ cd /etc          # va directement dans /etc
```

Chemins absolus et chemins relatifs

C'est **le** concept clé de ce chapitre. Il existe deux façons de désigner un fichier ou un dossier.

Type de chemin	Définition	Exemple
Absolu	Part toujours de la racine `/`. Il est valable partout, peu importe où vous êtes.	`/home/marie/Documents/cv.txt`
Relatif	Part de l'endroit où vous êtes *maintenant*. Il dépend du dossier courant.	`Documents/cv.txt`

Pour les chemins relatifs, deux raccourcis sont indispensables :

- `.` (un point) = « le dossier où je suis actuellement » ;
- `..` (deux points) = « le dossier juste au-dessus » (le parent).

```
# Je suis dans /home/marie
$ cd Documents/Projets   # relatif : je descends
```

```
$ cd ../../Images        # remonte deux fois, puis entre dans Images
$ cd /home/marie/Images  # absolu : même résultat, où qu'on soit
```

À retenir

Un chemin qui commence par `/` est **absolu**. Tout le reste est **relatif** au dossier courant. En cas de doute, `pwd` vous dit toujours où vous êtes.

La touche Tab : votre meilleure amie

Commencez à taper un nom de dossier, puis appuyez sur la touche **Tab** : le terminal complète automatiquement le nom. Cela évite les fautes de frappe et fait gagner un temps fou. Appuyez deux fois sur Tab pour voir toutes les possibilités.

À vous de jouer

1. Affichez votre position avec `pwd`, puis listez son contenu en format long.
2. Entrez dans votre dossier `Documents`, puis revenez à la maison de deux façons différentes.
3. Affichez les fichiers cachés de votre dossier personnel.
4. En partant de `/home/marie/Documents`, écrivez le chemin *relatif* pour atteindre `/home/marie/Images`.

CHAPITRE 4

Créer, copier, déplacer et supprimer

Vous savez vous déplacer ; il est temps d'agir. Ce chapitre couvre les manipulations de base : créer des dossiers et des fichiers, les copier, les renommer, les déplacer et les supprimer. Toutes ces actions que vous faisiez à la souris, en quelques lettres.

Créer : mkdir et touch

mkdir — créer un dossier

```
$ mkdir mon_projet            # crée un dossier
$ mkdir -p site/css/images     # -p crée toute l'arborescence d'un coup
```

L'option `-p` (« parents ») est très pratique : elle crée tous les dossiers intermédiaires nécessaires, sans erreur s'ils existent déjà.

touch — créer un fichier vide

```
$ touch notes.txt             # crée un fichier vide nommé notes.txt
$ touch a.txt b.txt c.txt     # crée plusieurs fichiers à la fois
```

Les noms avec espaces

Les espaces sont mal vus au terminal car ils séparent les arguments. `touch mon fichier.txt` crée *deux* fichiers ! Pour un nom contenant un espace, mettez-le entre guillemets : `touch "mon fichier.txt"`, ou remplacez l'espace par un underscore : `mon_fichier.txt`. La seconde solution est vivement recommandée.

Copier : cp

La syntaxe est toujours `cp source destination`.

```
$ cp notes.txt sauvegarde.txt      # copie un fichier sous un nouveau nom
$ cp notes.txt Documents/          # copie dans un autre dossier
$ cp -r site/ site_backup/         # -r pour copier un dossier entier (récursif)
```

Pour copier un **dossier** et tout son contenu, l'option `-r` (récursif) est obligatoire.

Déplacer et renommer : mv

Surprise : il n'y a pas de commande « renommer ». Au terminal, **renommer = déplacer vers un nouveau nom**. La commande `mv` (move) fait les deux.

```
$ mv notes.txt idees.txt           # RENOMME notes.txt en idees.txt
$ mv idees.txt Documents/           # DÉPLACE le fichier dans Documents
$ mv idees.txt Documents/notes.txt  # déplace ET renomme en une fois
```

Comment distinguer renommer et déplacer ?

Si la destination est un *nom de fichier*, vous renommez. Si la destination est un *dossier existant*, vous déplacez. `mv` devine selon ce que vous lui donnez.

Supprimer : rm et rmdir

```
$ rm notes.txt              # supprime un fichier
$ rm a.txt b.txt            # supprime plusieurs fichiers
$ rmdir vieux_dossier       # supprime un dossier VIDE uniquement
$ rm -r mon_projet          # supprime un dossier ET tout son contenu
$ rm -i notes.txt           # -i demande confirmation avant de supprimer
```

DANGER : rm est définitif

Au terminal, **il n'y a pas de corbeille**. Ce que `rm` efface est perdu pour de bon, immédiatement, sans confirmation. La commande la plus redoutée est `rm -rf` : elle supprime tout, sans rien demander. Ne tapez **jamais** `rm -rf /` ni une commande `rm -rf` dont vous ne comprenez pas chaque mot. Prenez l'habitude de vérifier avec `ls` avant de supprimer, et utilisez `rm -i` quand vous hésitez.

Filet de sécurité

Vous pouvez installer l'outil `trash-cli` : la commande `trash` envoie alors les fichiers dans la corbeille au lieu de les détruire. Bien plus rassurant que `rm` au quotidien.

Un exemple complet

Mettons tout bout à bout : créons un petit projet, organisons-le, faisons une sauvegarde.

```
TERMINAL
$ mkdir -p blog/articles          # crée le dossier et son sous-dossier
$ cd blog
$ touch index.html articles/jour1.txt
$ ls -R                          # -R liste aussi le contenu des sous-dossiers
.:
articles  index.html
```

```
./articles:
jour1.txt
$ cp -r articles articles_backup  # sauvegarde du dossier
$ mv index.html accueil.html      # on renomme la page d'accueil
```

À vous de jouer

1. Créez un dossier `entrainement`, entrez dedans, et créez-y trois fichiers vides.
2. Copiez l'un d'eux sous un nouveau nom, puis renommez un autre fichier.
3. Créez d'un seul coup l'arborescence `photos/2026/janvier`.
4. Supprimez un fichier avec demande de confirmation, puis supprimez le dossier `entrainement` en entier.

CHAPITRE 5

Lire et éditer le contenu des fichiers

Savoir manipuler des fichiers, c'est bien ; voir et modifier ce qu'ils contiennent, c'est mieux. Ce chapitre présente les commandes pour afficher du texte, un éditeur simple pour le modifier, et une notion puissante : les redirections.

Afficher le contenu d'un fichier

Commande	Usage idéal
`cat`	Afficher d'un coup tout un fichier (court).
`less`	Lire un fichier long, page par page, sans tout déverser à l'écran.
`head`	Voir seulement le début (10 premières lignes par défaut).
`tail`	Voir seulement la fin (10 dernières lignes par défaut).

```
$ cat notes.txt            # affiche tout le fichier
$ less rapport.log         # navigation : flèches, Espace, et q pour quitter
$ head -n 5 notes.txt      # les 5 premières lignes
$ tail -n 20 rapport.log   # les 20 dernières lignes
$ tail -f rapport.log      # -f "suit" le fichier en direct (très utile pour les logs)
```

Astuce indispensable : tail -f

L'option `-f` (« follow ») affiche les nouvelles lignes *au fur et à mesure* qu'elles s'ajoutent. C'est l'outil de référence pour surveiller un fichier journal pendant qu'un programme tourne. On l'arrête avec `Ctrl + C`.

Éditer un fichier : nano

Pour modifier du texte directement dans le terminal, le plus simple pour débuter est **nano**. Il s'ouvre, vous tapez comme dans un bloc-notes, et les raccourcis sont affichés en bas de l'écran.

```
$ nano notes.txt           # ouvre (ou crée) le fichier dans l'éditeur
```

Dans nano, le symbole `^` signifie la touche **Ctrl**. Les raccourcis essentiels :

Raccourci	Action
`Ctrl + O`	Enregistrer (« Write Out »), puis Entrée pour confirmer le nom
`Ctrl + X`	Quitter (il propose d'enregistrer si besoin)
`Ctrl + K`	Couper la ligne courante
`Ctrl + W`	Rechercher du texte

Et vim / emacs ?

Vous entendrez parler de `vim` et `emacs`, des éditeurs très puissants mais à la prise en main déroutante. Le piège classique du débutant : ouvrir `vim` par accident et ne plus savoir en sortir. La solution : tapez `Échap` puis `:q!` et Entrée pour quitter sans rien enregistrer. Pour le quotidien, restez sur `nano`.

Les redirections : envoyer la sortie ailleurs

Par défaut, une commande affiche son résultat à l'écran. Les **redirections** permettent d'envoyer ce résultat dans un fichier à la place. C'est l'une des idées les plus puissantes du shell.

Symbole	Effet
`>`	Écrit le résultat dans un fichier (**écrase** le contenu existant)
`>>`	Ajoute le résultat **à la fin** du fichier (sans rien effacer)
`<`	Lit l'entrée depuis un fichier au lieu du clavier

```
$ echo "Première ligne" > journal.txt   # crée le fichier avec ce contenu
$ echo "Deuxième ligne" >> journal.txt  # ajoute sans écraser
$ cat journal.txt
Première ligne
Deuxième ligne
$ date >> journal.txt                   # on peut rediriger n'importe quelle commande
```

Le piège du simple chevron

Le `>` **écrase tout** le contenu existant, sans avertissement. Si vous voulez juste ajouter quelque chose, c'est `>>` qu'il faut utiliser. Une confusion entre les deux est une cause classique de perte de données.

À retenir

`cat` pour les petits fichiers, `less` pour les longs, `head` / `tail` pour les extrémités, `nano` pour éditer. `>` écrase, `>>` ajoute. Ces redirections deviendront le ciment de tout ce qui suit.

À vous de jouer

1. Créez avec `echo` et `>` un fichier `liste.txt` contenant un fruit, puis ajoutez deux autres fruits avec `>>`.
2. Affichez son contenu avec `cat`, puis seulement la première ligne avec `head`.
3. Ouvrez `liste.txt` dans `nano`, ajoutez une ligne, enregistrez et quittez.
4. Que se passe-t-il si vous refaites `echo "kiwi" > liste.txt` ? Et avec `>>` ?

CHAPITRE 6

Rechercher, filtrer et enchaîner

Voici le chapitre où le terminal montre sa vraie puissance. Vous allez apprendre à retrouver des fichiers, à fouiller dans du texte, et surtout à brancher les commandes les unes aux autres avec le pipe. C'est la philosophie d'Unix : de petits outils simples, combinés, qui font des choses extraordinaires.

Retrouver des fichiers : find

`find` parcourt l'arborescence à la recherche de fichiers selon des critères.

```
$ find . -name "*.txt"          # tous les .txt à partir du dossier courant
$ find . -name "rapport*"        # tous ceux qui commencent par "rapport"
$ find . -type d                 # uniquement les dossiers (d = directory)
$ find . -size +1M               # les fichiers de plus de 1 mégaoctet
```

Fouiller dans du texte : grep

`grep` cherche un mot ou un motif **à l'intérieur** des fichiers et affiche les lignes qui correspondent. C'est probablement la commande de recherche la plus utilisée au monde.

```
$ grep "erreur" rapport.log      # lignes contenant "erreur"
$ grep -i "erreur" rapport.log   # -i ignore la casse (Erreur, ERREUR...)
$ grep -n "TODO" code.py        # -n affiche les numéros de ligne
$ grep -r "motdepasse" .         # -r cherche dans tous les fichiers, récursivement
$ grep -c "erreur" rapport.log   # -c compte le nombre de lignes correspondantes
```

Le pipe | : le concept central

Le **pipe** (la barre verticale `|`, obtenue avec `AltGr + 6` sur un clavier français, ou `Maj + \` ailleurs) connecte deux commandes : il prend la *sortie* de la première et la donne en *entrée* à la seconde. C'est un tuyau entre programmes.

> **L'image du tuyau**
>
> Pensez à une chaîne de montage. `commande1 | commande2 | commande3` : le résultat de la première coule dans la deuxième, qui le transforme et le passe à la troisième. Chaque outil fait une seule chose, mais bien.

```
$ ls -l | grep ".txt"           # liste les fichiers, puis ne garde que les .txt
$ cat rapport.log | grep "erreur" | head  # erreurs, puis les 10 premières
$ history | grep "git"          # retrouve vos anciennes commandes git
```

Les petits outils à connecter

Commande	Rôle
`wc`	Compte les lignes, mots et caractères (`wc -l` = nombre de lignes)
`sort`	Trie les lignes par ordre alphabétique ou numérique (`sort -n`)
`uniq`	Supprime les doublons *consécutifs* (souvent précédé de `sort`)
`cut`	Extrait des colonnes d'un texte
`tr`	Remplace ou supprime des caractères

Des combinaisons concrètes

TERMINAL

```
# Combien de fichiers .jpg dans ce dossier ?
$ ls *.jpg | wc -l
42

# Lister les mots uniques d'un fichier, triés
$ cat texte.txt | tr ' ' '\n' | sort | uniq

# Top 5 des commandes que vous tapez le plus souvent
$ history | awk '{print $2}' | sort | uniq -c | sort -nr | head -5
```

La philosophie Unix

« Écris des programmes qui font une seule chose et la font bien ; écris des programmes qui coopèrent. » Vous ne mémorisez pas des centaines de commandes complexes : vous assemblez quelques outils simples comme des briques LEGO. C'est tout l'art de la ligne de commande.

À retenir

`find` cherche des *fichiers*, `grep` cherche *dans* les fichiers. Le pipe `|` branche les commandes entre elles. Maîtriser `grep` + `|` change radicalement votre efficacité.

À vous de jouer

1. Trouvez tous les fichiers `.txt` de votre dossier personnel avec `find`.
2. Dans un fichier texte de votre choix, cherchez un mot avec `grep` en affichant les numéros de ligne.
3. Comptez le nombre de fichiers dans un dossier en combinant `ls` et `wc -l`.
4. Affichez la liste de vos dossiers triée par ordre alphabétique avec `ls | sort`.

CHAPITRE 7

Permissions, utilisateurs et sudo

Linux et macOS sont des systèmes multi-utilisateurs où chaque fichier appartient à quelqu'un et possède des droits d'accès. Comprendre ce système de permissions vous évitera bien des messages « Permission denied » mystérieux — et vous fera manipuler `sudo` avec la prudence qu'il mérite.

Lire les permissions

Reprenons la sortie de `ls -l` et regardons sa première colonne :

```
$ ls -l
-rw-r--r--  1 marie  staff  1024 mai 29 14:00 notes.txt
drwxr-xr-x  3 marie  staff  4096 mai 28 09:00 Documents
```

La suite de dix caractères au début (`-rw-r--r--`) décrit les permissions. Décomposons-la :

Position	Exemple	Signification
1er caractère	`-` ou `d`	Type : `-` = fichier, `d` = dossier (directory)
Caractères 2 à 4	`rw-`	Droits du **propriétaire** (user)
Caractères 5 à 7	`r--`	Droits du **groupe**
Caractères 8 à 10	`r--`	Droits des **autres** (tout le monde)

Chaque groupe de trois utilise les mêmes trois lettres :

- `r` = **read** (lire) ;
- `w` = **write** (écrire / modifier) ;
- `x` = **execute** (exécuter, pour un programme ; ou « entrer » pour un dossier).

Un tiret `-` à la place d'une lettre signifie que le droit est absent. Ainsi `-rw-r--r--` se lit : « fichier ordinaire ; le propriétaire peut lire et écrire ; le groupe et les autres peuvent seulement lire ».

Modifier les permissions : chmod

`chmod` (« change mode ») modifie les droits. Deux notations existent ; la **symbolique** est la plus lisible pour débuter.

Notation symbolique

```
$ chmod +x script.sh       # rend le fichier exécutable (pour tous)
$ chmod u+x script.sh      # exécutable seulement pour le propriétaire (u = user)
$ chmod g-w notes.txt      # retire au groupe le droit d'écriture
$ chmod o-r secret.txt     # les "autres" (o) ne peuvent plus lire
```

On combine une cible (`u` user, `g` group, `o` others, `a` all), un signe (`+` ajoute, `-` retire) et un droit (`r`, `w`, `x`).

Notation numérique (octale)

Chaque droit a une valeur : `r`=4, `w`=2, `x`=1. On les additionne pour chaque catégorie.

Chiffre	Droits	Détail
`7`	`rwx`	4+2+1 — tout
`6`	`rw-`	4+2 — lire et écrire
`5`	`r-x`	4+1 — lire et exécuter
`4`	`r--`	4 — lire seulement
`0`	`---`	aucun droit

```
$ chmod 644 notes.txt      # rw- r-- r--  (config classique d'un fichier)
$ chmod 755 script.sh     # rwx r-x r-x  (config classique d'un programme)
$ chmod 700 prive/        # rwx --- ---  (accessible à vous seul)
```

Changer le propriétaire : chown

```
$ sudo chown marie fichier.txt        # change le propriétaire
$ sudo chown marie:equipe fichier.txt  # propriétaire ET groupe
```

sudo : le pouvoir de l'administrateur

Certaines actions (installer un logiciel, modifier un fichier système) exigent les droits d'**administrateur**, appelé *root* ou *superutilisateur*. La commande `sudo` (« superuser do ») exécute **une seule commande** avec ces droits, après vous avoir demandé votre mot de passe.

```
$ sudo apt update           # met à jour la liste des paquets (Debian/Ubuntu)
[sudo] Mot de passe de marie :
```

Avec sudo, plus de garde-fou

En tant que root, le système ne vous protège plus de vous-même. Une faute de frappe peut rendre la machine inutilisable. Règle d'or : **n'utilisez `sudo` que lorsque c'est nécessaire**, et ne lancez jamais une commande `sudo` copiée d'Internet sans la comprendre entièrement. Méfiez-vous tout particulièrement de `sudo rm -rf`.

À retenir

Les permissions se lisent en trois blocs : propriétaire, groupe, autres — chacun avec `r`, `w`, `x`. `chmod` les modifie, `chown` change le propriétaire, et `sudo` octroie temporairement les pleins pouvoirs… à manier avec respect.

À vous de jouer

1. Affichez les permissions d'un de vos fichiers avec `ls -l` et traduisez-les en mots.
2. Créez un fichier, retirez aux « autres » le droit de lecture, puis vérifiez.
3. Que vaut `755` en lettres ? Et `644` ?
4. Pourquoi vaut-il mieux éviter de tout faire en `sudo` ?

CHAPITRE 8

Variables, environnement et personnalisation

Le shell n'est pas figé : il a une mémoire (les variables), un environnement qui influence le comportement des programmes, et des fichiers de configuration que vous pouvez modeler à votre goût. Ce chapitre vous montre comment rendre le terminal vraiment vôtre.

Les variables

Une **variable** est une boîte étiquetée qui contient une valeur. On la crée avec un signe `=` (sans espace autour !) et on lit son contenu en mettant un `$` devant son nom.

```
$ prenom="Marie"          # création (attention : PAS d'espace autour du =)
$ echo $prenom             # lecture avec le $
Marie
$ echo "Bonjour $prenom"
Bonjour Marie
```

Le piège des espaces

`prenom = "Marie"` (avec espaces) ne fonctionne **pas** : le shell croit que vous lancez une commande nommée `prenom`. C'est `prenom="Marie"`, collé, qu'il faut écrire.

Variables shell et variables d'environnement

Il existe deux familles :

- les **variables shell** : elles n'existent que dans votre session actuelle ;
- les **variables d'environnement** : elles sont « exportées » et transmises aux programmes que vous lancez.

Pour transformer une variable en variable d'environnement, on utilise `export` :

```
$ export EDITOR=nano       # dit aux programmes d'utiliser nano comme éditeur
$ env                      # affiche toutes les variables d'environnement
$ echo $HOME               # le chemin de votre dossier personnel
$ echo $USER               # votre nom d'utilisateur
```

Variable	Contient
`$HOME`	Le chemin de votre dossier personnel
`$USER`	Votre nom d'utilisateur
`$PWD`	Le dossier courant
`$SHELL`	Le shell par défaut de votre compte
`$PATH`	La liste des dossiers où le shell cherche les programmes

Le PATH : comment le shell trouve les commandes

Quand vous tapez `ls`, comment le shell sait-il où se trouve le programme `ls` ? Il regarde dans une liste de dossiers : la variable `$PATH`, dont les dossiers sont séparés par des `:`.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
$ which ls              # dans lequel de ces dossiers se trouve "ls" ?
/bin/ls
```

> **« command not found »**
>
> Ce message signifie souvent que le programme n'est pas installé, ou qu'il n'est pas dans un dossier listé par `$PATH`. Pour ajouter un dossier au PATH : `export PATH="$HOME/mes_outils:$PATH"`. On place le nouveau dossier devant pour qu'il soit prioritaire.

Les alias : créer ses propres raccourcis

Un **alias** est un surnom pour une commande (souvent longue). Très pratique pour les commandes que vous tapez souvent.

```
$ alias ll="ls -lah"          # désormais "ll" lance "ls -lah"
$ alias ..="cd .."            # remonter d'un dossier en tapant ".."
$ alias gs="git status"
```

Les fichiers de configuration

Problème : variables, exports et alias définis dans la session **disparaissent dès que vous fermez le terminal**. Pour qu'ils soient permanents, on les écrit dans un **fichier de configuration**, lu automatiquement à chaque ouverture d'un terminal.

Fichier	Pour quel shell
`~/.bashrc`	Bash (lu à chaque nouveau terminal interactif)
`~/.bash_profile`	Bash (lu à la connexion ; souvent il appelle `.bashrc`)
`~/.zshrc`	Zsh (l'équivalent, pour le shell Z)
`~/.profile`	Réglages communs, indépendants du shell

```
# On ajoute nos personnalisations à la fin du fichier
$ nano ~/.bashrc
# ... on y écrit nos alias et exports, on enregistre ...
$ source ~/.bashrc          # recharge le fichier SANS rouvrir le terminal
```

La commande source

Après avoir modifié `.bashrc` ou `.zshrc`, vos changements ne s'appliquent pas tout de suite. `source ~/.bashrc` recharge le fichier dans la session courante. C'est le réflexe à avoir à chaque modification.

À retenir

Les variables (`nom=valeur`, lecture avec `$`) stockent des valeurs ; `export` les rend disponibles aux programmes ; `$PATH` indique où chercher les commandes ; les alias créent des raccourcis. Pour rendre tout cela permanent : on l'écrit dans `~/.bashrc` (ou `~/.zshrc`) puis on fait `source`.

À vous de jouer

1. Créez une variable `ville` contenant le nom de votre ville et affichez « J'habite à … ».
2. Affichez le contenu de `$HOME`, `$USER` et `$PATH`.
3. Créez un alias `ll` pour `ls -lah`, puis testez-le.
4. Ajoutez cet alias à votre `~/.bashrc` (ou `~/.zshrc`) et rechargez avec `source`.

CHAPITRE 9

Bash ou Zsh ? Passer à Zsh

Vous avez tout appris en pensant « Bash », mais presque tout fonctionne aussi en Zsh. Ce chapitre compare les deux concrètement, explique pourquoi tant de gens passent à Zsh, et vous guide pas à pas — sans rien casser.

Quel shell utilisez-vous en ce moment ?

```
$ echo $SHELL          # votre shell par défaut
/bin/bash
$ echo $0              # le shell actuellement en cours
```

Bash et Zsh sont cousins

Zsh a été conçu pour rester largement compatible avec Bash. La grande majorité des commandes, des scripts et de la syntaxe sont identiques. Passer de l'un à l'autre ne vous oblige donc **pas** à tout réapprendre — vous gagnez surtout du confort.

Ce que Zsh apporte de plus

Fonctionnalité	Bash	Zsh
Autocomplétion (Tab)	Basique	Très riche : options, descriptions, navigation au clavier
Correction de fautes de frappe	Non	Propose « vouliez-vous dire… ? »
Recherche dans l'historique	Correcte	Plus intelligente, par préfixe
Personnalisation du prompt	Possible mais verbeux	Très souple, avec thèmes
Écosystème de plugins	Limité	Énorme (via Oh My Zsh)
Jokers (globbing) avancés	Standards	Plus puissants (ex. `**` récursif natif)

Recommandation pratique

Pour **écrire des scripts** portables, restez sur Bash (présent partout). Pour votre **usage interactif quotidien**, Zsh + Oh My Zsh offre une expérience plus agréable. Beaucoup de gens combinent les deux : scripts en Bash, terminal de tous les jours en Zsh.

Installer et activer Zsh

```
# Vérifier s'il est déjà installé
$ zsh --version

# L'installer si besoin (Debian / Ubuntu)
$ sudo apt install zsh

# Le définir comme shell par défaut
$ chsh -s $(which zsh)

# Fermez puis rouvrez le terminal pour que le changement prenne effet
```

> **Sur macOS**
>
> Depuis macOS Catalina (2019), Zsh est déjà le shell par défaut : vous n'avez probablement rien à installer.

Oh My Zsh : la touche finale

Oh My Zsh est un cadre de configuration très populaire qui transforme Zsh en quelques secondes : jolis thèmes, autocomplétion enrichie, raccourcis Git, et des centaines de plugins.

```
# Installation (nécessite curl et git)
$ sh -c "$(curl -fsSL https://raw.githubusercontent.com/ohmyzsh/ohmyzsh/master/tools/install.sh)"
```

Ensuite, tout se règle dans `~/.zshrc` :

```
# Dans ~/.zshrc
ZSH_THEME="agnoster"                    # choix du thème visuel
plugins=(git docker npm sudo z)         # plugins activés
```

Plugin populaire	Ce qu'il fait
`git`	Raccourcis et infos Git directement dans le prompt
`z`	Sauter vers un dossier fréquent en tapant juste une partie de son nom
`zsh-autosuggestions`	Suggère la fin de la commande d'après votre historique (en gris)
`zsh-syntax-highlighting`	Colore vos commandes : vert si valide, rouge si erreur de frappe

Prudence avec les scripts d'installation

Installer un logiciel en exécutant un script téléchargé sur Internet (`curl ... | sh`) est courant, mais ce n'est pas anodin : vous lui faites confiance aveuglément. N'utilisez cette méthode qu'avec des sources officielles et reconnues, comme le dépôt officiel d'Oh My Zsh.

À retenir

Bash et Zsh sont largement compatibles. Zsh brille pour l'usage interactif (autocomplétion, thèmes, plugins), surtout avec Oh My Zsh. Pour les scripts portables, Bash reste la valeur sûre. Passer à Zsh ne casse rien : vos commandes habituelles continuent de fonctionner.

À vous de jouer

1. Quel shell est votre shell par défaut ? Vérifiez avec `echo $SHELL`.
2. Vérifiez si Zsh est installé avec `zsh --version`.
3. Citez deux fonctionnalités de Zsh absentes (ou plus faibles) dans Bash.
4. Pourquoi peut-on préférer Bash pour écrire des scripts destinés à d'autres machines ?

CHAPITRE 10

Écrire son premier script

*Tout ce que vous tapez ligne par ligne, vous pouvez l'enregistrer dans un fichier et le rejouer d'un seul coup : c'est un **script**. C'est ici que la ligne de commande devient un véritable outil d'automatisation. Nous allons construire, brique par brique, des scripts qui réfléchissent et se répètent.*

Qu'est-ce qu'un script ?

Un script shell est simplement un fichier texte contenant une suite de commandes, exécutées de haut en bas. Par convention, on lui donne l'extension `.sh` (mais ce n'est pas obligatoire).

Le shebang : la première ligne magique

Tout script commence par une ligne spéciale, le **shebang**, qui indique quel interpréteur doit lire le fichier :

```
#!/bin/bash     ← cette ligne dit "exécute-moi avec Bash"
```

Sans elle, le système ne sait pas forcément avec quel shell lancer votre fichier. Pour un script Zsh, on écrirait `#!/bin/zsh`.

Créer et lancer un premier script

FICHIER : bonjour.sh

```
#!/bin/bash
echo "Bonjour, bienvenue dans mon premier script !"
echo "Nous sommes le $(date)"
```

`$(date)` exécute la commande `date` et insère son résultat : c'est la **substitution de commande**. Maintenant, rendons le script exécutable et lançons-le :

```
$ chmod +x bonjour.sh      # on le rend exécutable (rappel du chapitre 7)
$ ./bonjour.sh             # on le lance (le ./ signifie "ici, dans ce dossier")
Bonjour, bienvenue dans mon premier script !
Nous sommes le vendredi 29 mai 2026, 14:30
```

Pourquoi ./ devant le nom ?

Le shell ne cherche les programmes que dans les dossiers du `$PATH` (chapitre 8), et le dossier courant n'en fait pas partie, par sécurité. Le préfixe `./` dit explicitement « lance le fichier qui est ici, dans le dossier courant ».

Les variables dans un script

```
#!/bin/bash
nom="Marie"
age=30
echo "Je m'appelle $nom et j'ai $age ans."
```

Lire ce que l'utilisateur saisit : read

```
#!/bin/bash
echo "Comment t'appelles-tu ?"
read nom
echo "Enchanté, $nom !"
```

Les conditions : if

Un script peut prendre des décisions. La structure `if … then … else … fi` (noter le `fi` qui ferme le bloc, « if » à l'envers) teste une condition.

```
#!/bin/bash
echo "Entre un nombre :"
read n
if [ "$n" -gt 10 ]; then
    echo "$n est plus grand que 10"
else
    echo "$n est inférieur ou égal à 10"
fi
```

Les comparaisons numériques utilisent des codes à deux lettres :

Test	Signifie	Test	Signifie
`-eq`	égal à	`-ne`	différent de
`-gt`	strictement supérieur	`-lt`	strictement inférieur
`-ge`	supérieur ou égal	`-le`	inférieur ou égal

Les espaces dans les crochets

La syntaxe `[ "$n" -gt 10 ]` est très pointilleuse : il faut un espace après `[` et avant `]`. `[$n -gt 10]` provoquera une erreur. Pensez aussi à mettre vos variables entre guillemets, comme `"$n"`, pour éviter les surprises quand elles sont vides.

Les boucles

La boucle for

```
#!/bin/bash
for fruit in pomme banane cerise; do
    echo "J'aime la $fruit"
done
```

Très utile pour traiter des fichiers en lot :

```
#!/bin/bash
for fichier in *.txt; do
    echo "Traitement de $fichier"
    cp "$fichier" "$fichier.backup"
done
```

La boucle while

```
#!/bin/bash
compteur=1
while [ "$compteur" -le 5 ]; do
    echo "Tour numéro $compteur"
    compteur=$((compteur + 1))
done
```

Notez `$((…))` : c'est ainsi que Bash effectue des calculs arithmétiques.

Les arguments : passer des valeurs au script

On peut donner des informations au script au moment de le lancer. Elles sont récupérées via `$1`, `$2`, etc. (`$0` est le nom du script lui-même).

FICHIER : salut.sh

```
#!/bin/bash
echo "Bonjour $1, tu as $2 ans !"
```

```
$ ./salut.sh Marie 30
Bonjour Marie, tu as 30 ans !
```

Variable	Contient
`$0`	Le nom du script
`$1, $2, …`	Le premier, le deuxième argument…
`$#`	Le nombre d'arguments reçus
`$@`	Tous les arguments d'un coup

Un script utile et complet

Un petit script de sauvegarde qui regroupe tout ce que nous avons vu :

FICHIER : sauvegarde.sh

```
#!/bin/bash
# Sauvegarde un dossier dans une archive datée

if [ $# -eq 0 ]; then
    echo "Usage : ./sauvegarde.sh "
    exit 1
fi

dossier="$1"
date_jour=$(date +%Y-%m-%d)
archive="sauvegarde-$date_jour.tar.gz"

echo "Sauvegarde de $dossier en cours..."
tar -czf "$archive" "$dossier"
echo "Terminé ! Archive créée : $archive"
```

Bonnes habitudes de scripting

Commentez vos scripts avec des lignes débutant par `#`. Donnez des noms de variables clairs. Testez sur des fichiers sans importance avant le vrai usage. Et n'oubliez pas `exit 1` pour signaler une erreur (par convention, `0` = succès, tout le reste = problème).

À retenir

Un script = un fichier de commandes commençant par `#!/bin/bash`, rendu exécutable avec `chmod +x` et lancé par `./script.sh`. Avec les variables, `read`, `if`, les boucles `for` / `while` et les arguments `$1 $2`, vous disposez déjà de tout pour automatiser des tâches réelles.

À vous de jouer

1. Écrivez un script qui demande votre prénom et vous salue.
2. Écrivez un script qui affiche les nombres de 1 à 10 avec une boucle `for`.
3. Écrivez un script qui reçoit un nombre en argument et dit s'il est positif ou négatif.
4. Améliorez le script de sauvegarde pour qu'il vérifie d'abord que le dossier existe.

CHAPITRE 11

Astuces, raccourcis et bonnes pratiques

Ce qui sépare un débutant d'un utilisateur à l'aise, ce n'est pas le nombre de commandes connues, mais une poignée de réflexes qui font gagner un temps fou. Voici les plus utiles, à adopter dès maintenant.

Les raccourcis clavier essentiels

Raccourci	Action
`Tab`	Complète automatiquement noms de fichiers et commandes
`↑` / `↓`	Parcourt les commandes précédentes
`Ctrl + C`	Interrompt la commande en cours
`Ctrl + L`	Efface l'écran (comme la commande `clear`)
`Ctrl + A` / `Ctrl + E`	Aller au début / à la fin de la ligne
`Ctrl + U` / `Ctrl + K`	Effacer du curseur jusqu'au début / jusqu'à la fin
`Ctrl + R`	Recherche dans l'historique (très puissant, voir plus bas)
`Ctrl + D`	Fermer le terminal (équivaut à `exit`)

Le réflexe Ctrl + C

Une commande tourne en boucle, ou vous êtes coincé ? `Ctrl + C` l'arrête net. C'est votre bouton « stop » universel ; gardez-le en tête.

L'historique des commandes

```
$ history              # affiche toutes vos commandes passées, numérotées
$ history | grep apt   # retrouve vos commandes contenant "apt"
$ !!                   # relance la dernière commande
$ sudo !!              # relance la dernière commande avec sudo (très pratique !)
$ !42                  # relance la commande n°42 de l'historique
```

Ctrl + R, le chercheur d'historique

Appuyez sur `Ctrl + R` et commencez à taper quelques lettres d'une commande passée : le terminal la retrouve instantanément. Appuyez à nouveau sur `Ctrl + R` pour remonter aux occurrences plus anciennes. C'est l'astuce qui change tout au quotidien.

Les jokers (wildcards)

Les **jokers** permettent de désigner plusieurs fichiers d'un coup selon un motif.

Joker	Signifie	Exemple
`*`	N'importe quelle suite de caractères	`*.txt` → tous les fichiers .txt
`?`	Un seul caractère quelconque	`photo?.jpg` → photo1.jpg, photoA.jpg…
`[ ]`	Un caractère parmi un ensemble	`fichier[12].txt` → fichier1.txt, fichier2.txt

```
$ ls *.jpg              # toutes les images .jpg
$ rm rapport_2023*      # tout ce qui commence par rapport_2023
$ cp Images/*.png Backup/ # copie tous les .png dans Backup
```

Jokers et rm : danger

Combiner `*` avec `rm` est risqué. Avant un `rm motif*`, faites d'abord `ls motif*` pour voir *exactement* ce qui sera supprimé. Cette simple précaution évite des catastrophes.

Enchaîner les commandes

Opérateur	Effet
`;`	Lance les commandes l'une après l'autre, quoi qu'il arrive
`&&`	Lance la suivante **seulement si** la précédente a réussi
`||`	Lance la suivante **seulement si** la précédente a échoué

```
$ mkdir projet; cd projet            # crée puis entre, dans tous les cas
$ cd projet && ls                    # ne liste QUE si le cd a réussi
$ cd projet || echo "Dossier introuvable"  # message si échec
```

Le && est votre ami

`&&` est idéal pour chaîner des étapes dépendantes : `make && ./mon_programme` ne lance le programme que si la compilation a réussi. C'est plus sûr qu'un simple `;`.

Bonnes pratiques générales

- **Tab partout :** complétez toujours plutôt que de taper en entier. Moins de fautes, plus de vitesse.
- **Vérifiez avant de détruire :** un `ls` avant un `rm` n'a jamais fait de mal.
- **Lisez les messages d'erreur :** ils disent presque toujours ce qui ne va pas. Ne les ignorez pas.
- **Évitez les espaces** dans les noms de fichiers : préférez `les_underscores` ou le-tiret.

- **Ne lancez pas en sudo** ce que vous n'avez pas compris, surtout copié d'Internet.
- **Sauvegardez avant les grosses opérations :** une copie de secours coûte peu et sauve beaucoup.

À retenir

Tab pour compléter, ↑ pour rappeler, Ctrl+R pour chercher, Ctrl+C pour arrêter. Les jokers (`*`, `?`) traitent les fichiers en masse — avec prudence. `&&` enchaîne en cas de succès. Ces réflexes valent dix nouvelles commandes.

À vous de jouer

1. Utilisez `Ctrl + R` pour retrouver une commande tapée plus tôt.
2. Listez tous les fichiers commençant par la lettre « a » avec un joker.
3. Écrivez une seule ligne qui crée un dossier `test` et y entre uniquement si la création a réussi.
4. Relancez votre dernière commande avec `!!`.

CHAPITRE 12

Antisèche & aller plus loin

Ce dernier chapitre rassemble tout en un coup d'œil : un mémo des commandes à garder sous la main, les erreurs de débutant à éviter, des pistes pour progresser, et un glossaire des termes rencontrés.

Antisèche des commandes

Navigation

Commande	Action
`pwd`	Affiche le dossier courant
`ls -lah`	Liste détaillée, fichiers cachés, tailles lisibles
`cd dossier` / `cd ..` / `cd ~`	Entrer / remonter / revenir à la maison

Fichiers et dossiers

Commande	Action
`mkdir -p a/b/c`	Créer une arborescence de dossiers
`touch fichier`	Créer un fichier vide
`cp -r src dest`	Copier (dossier avec `-r`)
`mv ancien nouveau`	Déplacer ou renommer
`rm -i fichier`	Supprimer (avec confirmation)

Contenu et recherche

Commande	Action
`cat` / `less` / `head` / `tail -f`	Afficher tout / paginer / début / suivre la fin
`grep -rin "mot" .`	Chercher du texte (récursif, sans casse, numéroté)
`find . -name "*.txt"`	Trouver des fichiers
`cmd1 | cmd2`	Brancher la sortie de l'une sur l'entrée de l'autre
`cmd > fic` / `cmd >> fic`	Rediriger (écraser / ajouter)

Système

Commande	Action
`chmod +x` / `chmod 755`	Modifier les permissions
`sudo cmd`	Exécuter en administrateur
`echo $PATH` / `which cmd`	Voir le PATH / localiser un programme
`man cmd` / `cmd --help`	Obtenir de l'aide
`history` / `Ctrl+R`	Historique des commandes

Les erreurs de débutant à éviter

Erreur	Conséquence	Le bon réflexe
Confondre `>` et `>>`	Fichier écrasé	`>>` pour ajouter, `>` pour (ré)écrire
Espace autour du `=`	La variable n'est pas créée	`nom="valeur"` collé
`rm` sans vérifier	Perte définitive (pas de corbeille)	`ls` d'abord, ou `rm -i`
Tout faire en `sudo`	Risque pour le système	`sudo` uniquement si nécessaire
Ignorer les messages d'erreur	On reste bloqué	Les lire : ils expliquent le problème
Bloqué dans `vim`	Panique	`Échap` puis `:q!` et Entrée

Une feuille de route pour progresser

1. **Pratiquez au quotidien :** forcez-vous à utiliser le terminal pour de petites tâches réelles (déplacer des fichiers, chercher du texte).
2. **Apprenez Git en ligne de commande :** c'est l'outil incontournable du versionnage de code et un excellent terrain d'entraînement.
3. **Automatisez une vraie tâche :** transformez une corvée répétitive en script (sauvegarde, renommage en masse…).
4. **Découvrez les outils de connexion à distance :** `ssh` pour piloter un serveur, `scp` / `rsync` pour transférer des fichiers.
5. **Explorez le traitement de texte avancé :** `sed` et `awk` ouvrent un monde de possibilités.
6. **Personnalisez votre environnement :** peaufinez votre `.zshrc`, ajoutez des alias, essayez des plugins.

Où chercher de l'aide

La documentation intégrée (`man`, `--help`, l'outil `tldr`) répond à la plupart des questions. Pour le reste, les communautés en ligne (forums, sites de questions-réponses techniques) regorgent de réponses à des problèmes que d'autres ont déjà rencontrés. Apprendre à bien formuler sa recherche fait partie du métier.

Glossaire

Terme	Définition
Terminal	L'application (la fenêtre) dans laquelle on tape des commandes.
Shell	Le programme qui interprète et exécute les commandes (Bash, Zsh…).
Bash / Zsh	Deux shells répandus ; Zsh est plus moderne et personnalisable.
Prompt	L'invite de commande qui attend votre saisie (se termine par `$` ou `#`).
Argument	Ce sur quoi agit une commande (fichier, dossier, texte).
Option / flag	Modificateur du comportement d'une commande (ex. `-l`, `--all`).
Chemin absolu	Chemin partant de la racine `/`, valable partout.
Chemin relatif	Chemin partant du dossier courant.
Racine (root)	Le sommet de l'arborescence, noté `/` (à ne pas confondre avec l'utilisateur root).
Pipe (`|`)	Tuyau qui relie la sortie d'une commande à l'entrée d'une autre.
Redirection	Envoi de la sortie d'une commande vers un fichier (`>`, `>>`).
Joker (wildcard)	Caractère (`*`, `?`) désignant plusieurs fichiers par motif.
Variable d'environnement	Valeur nommée transmise aux programmes (ex. `$PATH`, `$HOME`).
PATH	Liste des dossiers où le shell cherche les programmes.
Alias	Surnom pour une commande plus longue.
Script	Fichier contenant une suite de commandes à exécuter.
Shebang	Première ligne d'un script (`#!/bin/bash`) indiquant l'interpréteur.
sudo	Exécute une commande avec les droits d'administrateur.
Permissions (rwx)	Droits de lecture, écriture, exécution sur un fichier.

Le mot de la fin

Vous avez maintenant toutes les bases pour utiliser le terminal avec confiance : naviguer, manipuler des fichiers, rechercher, gérer les permissions, personnaliser votre environnement et écrire vos premiers scripts. La maîtrise vient avec la pratique : ouvrez un terminal, essayez, trompez-vous, recommencez. Chaque commande tapée vous rend un peu plus à l'aise. Bon voyage dans la ligne de commande !

— Fin du cours —