# Fiche 1 — Navigation et manipulation de fichiers

## Révision et mise à niveau

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-10

## Table des matières

1 Objectifs pédagogiques ..... 1
2 Préparation de l'environnement de travail ..... 2
3 Le répertoire courant : `pwd` et `ls` ..... 3
3.1 Commandes de navigation ..... 3
3.2 Enchaîner les commandes ..... 4
4 Créer un fichier : `touch` et les fichiers cachés ..... 4
5 Copier un fichier : `cp` ..... 5
6 Renommer / déplacer un fichier : `mv` ..... 5
7 Supprimer un fichier : `rm` ..... 7
8 Rechercher des fichiers : `find` ..... 7
9 Auto-vérification ..... 10
10 Ce qu'il faut retenir ..... 10

## 1 Objectifs pédagogiques

**À l'issue de cette fiche, vous serez capable de :**

- vous situer dans l'arborescence Unix avec `pwd` ;
- changer de répertoire avec `cd`, en chemins absolus et relatifs ;
- inspecter un répertoire avec `ls` (options `-l`, `-a`, `-h`, `-R`) ;
- créer un fichier avec `touch` et comprendre les fichiers cachés ;
- copier un fichier avec `cp` ;
- renommer ou déplacer un fichier avec `mv` ;
- supprimer un fichier avec `rm`, en mesurant le risque ;
- rechercher des fichiers avec `find`.

> **i** Comment lire ce support
>
> Les blocs grisés contiennent des commandes **bash** exécutées lors de la publication du document : vous voyez à la fois la commande et son résultat. Vous pouvez les recopier telles quelles dans un terminal. Chaque fiche travaille dans son propre répertoire, ce qui rend les exercices reproductibles : vous pouvez relancer une cellule aussi souvent que nécessaire — **tant qu'aucune cellule n'a affiché `ERREUR :` depuis la préparation**. Passé ce point il n'y a plus de relance isolée sûre : relancer la cellule de préparation — elle `rm -rf` le répertoire de la fiche, donc tout ce qui a été produit depuis est effacé — **puis toutes les cellules qui suivent, dans l'ordre**.

## 2 Préparation de l'environnement de travail

Toutes les manipulations de cette fiche se déroulent dans un répertoire dédié, créé et nettoyé par la cellule ci-dessous. Le `rm -rf` est **borné à ce sous-répertoire** ; l'écriture `"${WORK:?}"` interdit toute suppression si la variable était vide.

Si une cellule plus bas affiche un message commençant par `ERREUR :`, sachez ce qui s'est passé : la garde fait `exit 1`, ce qui **termine le shell de la cellule** ; le moteur de calcul en redémarre un neuf, et `${WORK}` comme le répertoire courant sont donc perdus. La cellule en échec est malgré tout marquée comme exécutée, pas comme fausse. Relancer la préparation puis **toutes** les cellules qui suivent, dans l'ordre, est la seule reprise sûre — la préparation `rm -rf` le répertoire au passage. Dans cette fiche les messages varient selon la cellule (par exemple `ERREUR : setup`, `répertoire de travail absent`, `retour … impossible`) ; c'est le préfixe `ERREUR :` qu'il faut surveiller, pas une phrase particulière. D'autres fiches peuvent aussi afficher un message qui vient du shell lui-même : la reprise sûre y est la même.

```
WORK="$HOME/work/assignments/shell_tp/fiche01"
rm -rf "${WORK:?}" && mkdir -p "${WORK}" && cd "${WORK}" || { echo "ERREUR :
setup"; exit 1; }
```

L'arborescence de projet utilisée par les exercices est créée immédiatement : `mkdir -p` fabrique les répertoires intermédiaires manquants et n'échoue pas si le chemin existe déjà, ce qui rend la cellule idempotente.

```
cd "${WORK}" || { echo "ERREUR : répertoire de travail absent — exécuter la
cellule de préparation"; exit 1; }
mkdir -p projet/{data,scripts,results}
```

# 3 Le répertoire courant : `pwd` et `ls`

Le **shell** est l'interpréteur de commandes qui exécute ce que vous tapez. Le terminal est la fenêtre ; le shell (`bash` ici) est le programme qui interprète la ligne de commande.

Le système de fichiers est une arborescence de racine `/`. Deux formes de chemins coexistent :

- un chemin **absolu** commence par `/` → `/home/user/documents` ;
- un chemin **relatif** est interprété depuis le répertoire courant → `../data`, `./script.sh`, avec `.` le répertoire courant, `..` le parent et `~` le répertoire personnel.

## 3.1 Commandes de navigation

| Commande | Description | Exemple |
|---|---|---|
| `pwd` | Print Working Directory — affiche le répertoire courant | `pwd` |
| `ls` | List — liste le contenu d'un répertoire | `ls -la` |
| `cd` | Change Directory — change de répertoire | `cd /tmp` |

Vérifions d'abord où nous nous trouvons : la commande `pwd` est le contrôle de base de toute navigation.

```
pwd
```

```
/home/jovyan/work/assignments/shell_tp/fiche01
```

`ls` sans option liste les noms ; l'option `-l` ajoute le détail (permissions, propriétaire, taille, date), `-a` les fichiers cachés, `-h` les tailles lisibles, `-R` la récursion.

```
ls -lah
```

```
total 12K
drwxr-xr-x 3 root root 4.0K Nov 10 18:44 .
drwxr-xr-x 3 root root 4.0K Nov 10 18:44 ..
drwxr-xr-x 5 root root 4.0K Nov 10 18:44 projet
```

```
ls -l
```

```
total 4
drwxr-xr-x 5 root root 4096 Nov 10 18:44 projet
```

### 3.2 Enchaîner les commandes

Plusieurs commandes peuvent tenir sur une ligne, séparées par `&&` (la suite n'est exécutée que si la commande précédente réussit), `;` (exécution inconditionnelle) ou `||` (la suite n'est exécutée que si la précédente échoue).

```
cd /tmp && pwd
cd "${WORK}" || { echo "ERREUR : retour au répertoire de travail impossible";
exit 1; }
```

```
/tmp
```

```
cd ~ && pwd
cd "${WORK}" || { echo "ERREUR : retour au répertoire de travail impossible";
exit 1; }
```

```
/home/jovyan
```

## 4 Créer un fichier : `touch` et les fichiers cachés

`touch` crée un fichier vide si celui-ci n'existe pas, sinon il met à jour ses horodatages.

```
mkdir -p test_dir
touch file1.txt file2.txt
ls -l
```

```
total 8
-rw-r--r-- 1 root root    0 Nov 10 18:44 file1.txt
-rw-r--r-- 1 root root    0 Nov 10 18:44 file2.txt
drwxr-xr-x 5 root root 4096 Nov 10 18:44 projet
drwxr-xr-x 2 root root 4096 Nov 10 18:44 test_dir
```

Un fichier dont le nom commence par `.` est **masqué** : il n'apparaît qu'avec `ls -a`. Les redirections `>` (écrase) et `>>` (ajoute) permettent d'en écrire le contenu. On **amorce toujours** avec `>` avant tout `>>`, afin que la cellule reste idempotente.

```
echo "set number" > .fiche01.cfg
echo "syntax on" >> .fiche01.cfg
ls -lah
cat .fiche01.cfg 2>/dev/null || echo "…(fichier vide ou absent)"
```

```
total 20K
drwxr-xr-x 4 root root 4.0K Nov 10 18:44 .
drwxr-xr-x 3 root root 4.0K Nov 10 18:44 ..
-rw-r--r-- 1 root root   21 Nov 10 18:44 .fiche01.cfg
-rw-r--r-- 1 root root    0 Nov 10 18:44 file1.txt
-rw-r--r-- 1 root root    0 Nov 10 18:44 file2.txt
drwxr-xr-x 5 root root 4.0K Nov 10 18:44 projet
drwxr-xr-x 2 root root 4.0K Nov 10 18:44 test_dir
set number
syntax on
```

> ⚠️ Avertissement
>
> `cat` lit un fichier. Redirigée vers `/dev/null` avec un message de repli, elle n'interrompt jamais une séquence : c'est la garde à adopter dès qu'un fichier peut être absent.

## 5 Copier un fichier : `cp`

`cp source destination` duplique un fichier. Vers un répertoire, on indique le répertoire de destination ; vers un autre nom, on indique le nouveau chemin complet.

| Option | Signification |
|---|---|
| `-r` | récursif — copie aussi les répertoires |
| `-v` | verbeux — affiche chaque opération |
| `-p` | préserve les métadonnées (propriétaire, dates) |

```
cp file1.txt file1_backup.txt
ls -l file1.txt file1_backup.txt
```

```
-rw-r--r-- 1 root root 0 Nov 10 18:44 file1.txt
-rw-r--r-- 1 root root 0 Nov 10 18:44 file1_backup.txt
```

## 6 Renommer / déplacer un fichier : `mv`

`mv` déplace une source vers une destination. Si les deux chemins désignent le même répertoire, l'effet est un **renommage**.

```
mv file2.txt test_dir/
ls -R
tree .
```

```
.:
file1.txt  file1_backup.txt  projet  test_dir

./projet:
data  results  scripts

./projet/data:

./projet/results:

./projet/scripts:

./test_dir:
file2.txt
.
├── file1.txt
├── file1_backup.txt
├── projet
│   ├── data
│   ├── results
│   └── scripts
└── test_dir
    └── file2.txt

6 directories, 3 files
```

> ⚠️ Avertissement
>
> Après un `mv`, le chemin d'origine est périmé : avant d'enchaîner une commande sur `file2.txt`, on repart du répertoire courant (`pwd`, `ls`) ou du nouveau chemin — ne pas réutiliser par réflexe l'ancien chemin.

## 7 Supprimer un fichier : `rm`

> ⚠ Avertissement
>
> `rm` supprime **définitivement** : il n'y a pas de corbeille en shell. Avant toute suppression, on vérifie le contenu visé avec `ls`, et on peut exiger une confirmation avec l'option `-i`.

| Option | Signification |
|---|---|
| `-r` | récursif — supprime un répertoire et son contenu |
| `-f` | force — aucune confirmation, aucune erreur si absent |
| `-i` | interactif — demande confirmation pour chaque cible |

```
rm file1_backup.txt
ls -l file1_backup.txt 2>/dev/null || echo "file1_backup.txt a bien été supprimé"
```

```
file1_backup.txt a bien été supprimé
```

## 8 Rechercher des fichiers : `find`

`find <racine> <critères>` parcourt récursivement une arborescence. Contrairement à `ls -R`, il filtre sur le nom, le type ou la date, et il affiche des chemins complets.

| Critère | Signification |
|---|---|
| `-name "*.txt"` | motif sur le nom (les `*` sont entre guillemets) |
| `-type d` | ne retenir que les répertoires |
| `-type f` | ne retenir que les fichiers ordinaires |

```
find . -name "*.txt"
find projet -type d
```

```
./test_dir/file2.txt
./file1.txt
projet
projet/scripts
projet/data
projet/results
```

### Exercice 1 — Navigation et exploration

**Objectif :** créer une arborescence de projet et s'y déplacer.

1. Créer l'arborescence `projet/` contenant `data/`, `scripts/` et `results/`.
2. Se déplacer dans `data/` et y créer trois fichiers vides.
3. Lister le contenu de `projet/` **depuis** `data/`, sans changer de répertoire.
4. Revenir au répertoire personnel en **une seule** commande.

**Questions de réflexion :**

1. Quelle différence y a-t-il entre `cd ..` et `cd ../..` ?
2. Pourquoi utiliser `mkdir -p` plutôt que `mkdir` simple ?
3. Comment consulter le manuel de la commande `mkdir` depuis le terminal ?

Contrôlez votre position avant de commencer :

```
pwd
```

```
/home/jovyan/work/assignments/shell_tp/fiche01
```

```
cd "${WORK}" || { echo "ERREUR : répertoire de travail absent — exécuter la cellule de préparation"; exit 1; }
mkdir -p projet/{data,scripts,results}
cd projet/data || { echo "ERREUR : répertoire projet/data absent"; exit 1; }
touch file1.txt file2.txt file3.txt
echo "Contenu de projet/ depuis data/ :"
ls ..
cd ~ && pwd
```

```
Contenu de projet/ depuis data/ :
data  results  scripts
/home/jovyan
```

La solution ci-dessus finit par un `cd ~` ; la cellule qui suit ramène le shell dans `${WORK}`, le répertoire où les exercices suivants attendent d'être exécutés.

```
cd "${WORK}" || { echo "ERREUR : retour au répertoire de travail impossible"; exit 1; }
```

ℹ Exercice 2 — Mettre en pratique la navigation

**Objectif :** organiser l'arborescence du projet en vous déplaçant et en déplaçant des fichiers.

1. Dans `projet/data/`, créer cinq fichiers vides `data1.csv` à `data5.csv`.
2. Créer un sous-répertoire `raw/` et y déplacer `data1.csv` et `data2.csv`.
3. Créer un sous-répertoire `processed/` et y copier `data3.csv` sous le nom `data3_clean.csv`.
4. Supprimer `data5.csv` **en demandant une confirmation** avant la suppression.

```
cd "${WORK}/projet/data" || { echo "ERREUR : projet/data absent — exécuter d'abord la solution de l'exercice 1"; exit 1; }
touch data1.csv data2.csv data3.csv data4.csv data5.csv
mkdir -p raw
mv data1.csv data2.csv raw/
mkdir -p processed
cp data3.csv processed/data3_clean.csv
ls -R ..
# Étape 4 — interactif : à taper dans un terminal, jamais dans une cellule exécutée.
# rm -i data5.csv
```

```
..:
data  results  scripts

../data:
data3.csv  data5.csv  file2.txt  processed
data4.csv  file1.txt  file3.txt  raw

../data/processed:
data3_clean.csv

../data/raw:
data1.csv  data2.csv

../results:

../scripts:
```

## 9 Auto-vérification

Six vérifications. Chacune porte sur un fait **démontré par une cellule exécutée** de cette fiche ; elles supposent que vous avez exécuté toutes les cellules précédentes, y compris les solutions des deux exercices.

```
cd "${WORK}" || { echo "ERREUR : dossier absent ${WORK}"; exit 1; }
total=0
ok=0
check() {
  total=$((total + 1))
  if [ "$2" = "$3" ]; then
    ok=$((ok + 1))
    echo "OK   $1 : $2"
  else
    echo "KO   $1 : obtenu [$2], attendu [$3]"
  fi
}
check "dossier de travail" "$([ -d "${WORK}" ] && echo oui || echo non)" "oui"
check "trois dossiers dans projet" "$(ls -1 "${WORK}/projet" | wc -l | tr -d '
')" "3"
check "config ecree puis completee" "$(wc -l < "${WORK}/.fiche01.cfg" | tr -d '
')" "2"
check "file2 deplace dans test_dir" "$([ -f "${WORK}/test_dir/file2.txt" ] &&
echo oui || echo non)" "oui"
check "copie propre dans processed" "$([ -f "${WORK}/projet/data/processed/
data3_clean.csv" ] && echo oui || echo non)" "oui"
check "backup supprime" "$([ -f "${WORK}/file1_backup.txt" ] && echo oui || echo
non)" "non"
echo "OK : ${ok} / ${total}"
```

Trois vérifications ne correspondent à aucun énoncé d'exercice : `config ecree puis completee` vient de la démonstration sur `>` puis `>>`, `file2 deplace dans test_dir` de la démonstration sur `mv`, et `backup supprime` de la démonstration sur `rm`. Elles contrôlent donc que vous avez **exécuté toutes les cellules** de la fiche, solutions comprises, et pas seulement répondu aux exercices. Les plus utiles restent celles qui constatent l'**effet cumulé** d'un `mv`, d'un `cp` et d'un `rm`.

## 10 Ce qu'il faut retenir

- `pwd` avant tout : on ne manipule jamais un répertoire qu'on n'a pas vérifié.
- Les chemins relatifs (`.` , `..`, `~`) rendent les commandes courtes mais dépendantes de votre position — d'où le réflexe `pwd`.
- `mkdir -p`, `touch`, `cp` vers un nom explicite et l'amorçage `>` avant `>>` rendent une manipulation **reproductible**.

- `rm` est irréversible : `ls` d'abord, `-i` en cas de doute, jamais de `rm -rf` sur un chemin non contrôlé.
- `find` remplace `ls -R` dès qu'il faut filtrer.