Mise à niveau : Shell Unix

Révision et mise à niveau

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

2026-11-01

Table des matières

0.1 Objectifs pédagogiques . . . 2
0.1.a Message de bienvenue . . . 3
1 BLOC 1 : Fondamentaux . . . 3
1.1 Parcours pédagogiques . . . 3
1.2 ◯ Minimal . . . 3
1.3 ◯ Approfondi . . . 3
1.4 ◯ Avancé . . . 3
1.5 Introduction et navigation . . . 3
1.5.a Qu'est-ce qu'un shell ? . . . 3
1.5.b Pourquoi le shell pour un informaticien ? . . . 4
1.5.c Anatomie d'une commande . . . 4
1.5.d Commandes de navigation essentielles . . . 4
1.5.e Chemins absolus vs relatifs . . . 4
1.5.f Illustration . . . 5
1.6 Manipulation de fichiers . . . 6
1.6.a Commandes de manipulation essentielles . . . 7
1.6.b Options importantes . . . 7
1.6.c Illustration . . . 7
2 BLOC 2 : Visualisation et redirection . . . 9
2.1 Parcours pédagogiques . . . 9
2.2 ◯ Minimal . . . 9
2.3 ◯ Approfondi . . . 9
2.4 ◯ Avancé . . . 10
2.5 Lire et afficher des fichiers . . . 10
2.5.a Commandes de visualisation . . . 10
2.5.b Redirections . . . 10

2.5.c Parcours 11
2.5.d Illustration 11
2.6 Premières recherches 12
2.6.a Exemples pratiques d'expressions régulières utiles : 13
2.6.b Illustration 13
3 BLOC 3 : Traitement de texte avancé 15
3.1 Parcours pédagogiques — Bloc 3 : Traitement de texte avancé 15
3.2 ◯ Minimal 15
3.3 ◯ Approfondi 16
3.4 ◯ Avancé 16
3.5 3.1 Extraction avec `cut` 16
3.5.a Parcours 16
3.5.b Illustration et pratique 16
3.6 3.2 Transformation avec `tr` 18
3.7 Édition avec `sed` 19
4 BLOC 4 : Pipelines et composition 21
4.1 Parcours pédagogiques — Bloc 4 : Pipelines et composition 21
4.2 ◯ Minimal 21
4.3 ◯ Approfondi 21
4.4 ◯ Avancé 21
4.5 Philosophie Unix et pipes 21
5 BLOC 5 : Notions complémentaires 23
5.1 Parcours pédagogiques — Bloc 5 : Notions complémentaires 23
5.2 ◯ Minimal 23
5.3 ◯ Approfondi 23
5.4 ◯ Avancé 23
5.5 Permissions et processus 23
5.6 5.2 Variables et scripting 25

0.1 Objectifs pédagogiques

À l'issue de ce TP, vous serez capable de :

- Naviguer efficacement dans l'arborescence Unix
- Manipuler fichiers et répertoires en ligne de commande
- Rechercher et filtrer des données avec grep
- Extraire et transformer du texte avec cut, tr, sed
- Composer des pipelines pour traiter des données
- Créer des scripts shell simples pour automatiser vos tâches

0.1.a Message de bienvenue

Les blocs grisés comme celui-ci contiennent des commandes Shell (bash). Vous pouvez les exécuter directement dans un terminal pour tester les commandes expliquées dans ce TP.

```
echo "Bienvenue dans le TP Shell Unix : Maîtrise des fondamentaux !"
echo "Durée : 3 heures | Format : Explication → Illustration → Pratique"
```

```
Bienvenue dans le TP Shell Unix : Maîtrise des fondamentaux !
Durée : 3 heures | Format : Explication → Illustration → Pratique
```

1 BLOC 1 : Fondamentaux

1.1 Parcours pédagogiques

1.2 ◯ Minimal

Compétence	Commandes essentielles
Navigation	`pwd`, `cd`, `ls -lah`, chemins, wildcards `*.txt`
Fichiers	`touch`, `mkdir -p`, `cp -r`, `mv`, `rm -i`

1.3 ◯ Approfondi

Compétence	Commandes essentielles
Navigation	`find`, `pushd/popd`, `alias ll='ls -lah'`, fonctions shell
Fichiers	`mkdir -p {a,b}`, `cp -a`, `rsync -av`, `rename`

1.4 ◯ Avancé

Compétence	Outils et techniques
Navigation	`fzf`, `fd`, `zoxide`, `CDPATH`, symbolic links
Fichiers	`find -exec`, `rsync --exclude`, `inotifywait`, scripts

1.5 Introduction et navigation

1.5.a Qu'est-ce qu'un shell ?

Le **shell** est un interpréteur de commandes qui permet d'interagir avec le système. Les shells les plus courants sont :

- `bash` (Bourne Again Shell) - le plus répandu
- `zsh` (Z Shell) - par défaut sur macOS récents

- `sh` (Bourne Shell) - le shell historique

Différence Terminal vs Shell :

- Le **terminal** est l'interface graphique (fenêtre) où vous tapez des commandes
- Le **shell** est le programme qui interprète vos commandes

1.5.b Pourquoi le shell pour un informaticien ?

- **Interactivité** : exécution rapide de commandes
- **Automatisation** : scripts pour tâches répétitives
- **Serveurs de calcul** : HPC, clusters sans interface graphique
- **Pipelines de données** : prétraitement avant analyse
- **Reproductibilité** : traçabilité et versionnage des opérations (y compris dans des conteneurs type docker)

1.5.c Anatomie d'une commande

Une commande shell se compose généralement d'un verbe (la commande), d'options (modificateurs) et d'arguments (cibles).

```
commande -options arguments
```

Exemple : `ls -lah /home`

- `ls` : la commande (lister)
- `-lah` : options
  - `-`: format court des options (`-l`, `-a`, `-h`) avec possibilité de les combiner
  - `--all --human-readable -l` : format long des options
- `/home` : argument (chemin)

1.5.d Commandes de navigation essentielles

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`

1.5.e Chemins absolus vs relatifs

- Le système de fichiers est organisé en une arborescence avec la racine `/`.
  - Les fichiers peuvent être des répertoires ou des fichiers ordinaires.
  - Les répertoires peuvent contenir d'autres répertoires (sous-répertoires).
  - Chaque utilisateur a un répertoire personnel, souvent `/home/username`.
- Un chemin **Absolu** : commence par `/` → `/home/user/documents`

- Un chemin **Relatif** : par rapport au répertoire courant → `../data` ou `./script.sh`
  - `.` : répertoire courant
  - `..` : répertoire parent
  - `~` : raccourci vers votre répertoire personnel

1.5.f Illustration

Tester les commandes de navigation de base :

```
pwd
```

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

```
ls -lah
```

```
total 8.0K
drwxr-xr-x 2 root root 4.0K Nov  1 16:40 .
drwxr-xr-x 5 root root 4.0K Nov  1 16:40 ..
-rw-r--r-- 1 root root    0 Nov  1 16:40 test.txt
```

Plusieurs commandes de shell peuvent être combinées sur une seule ligne en les séparant par des `&&` (exécute la commande suivante seulement si la précédente réussit) ou des `;` (exécute toutes les commandes successivement) ou par des `||` (exécute la commande suivante seulement si la précédente échoue).

```
cd /tmp && pwd
```

```
/tmp
```

```
cd ~ && pwd
```

```
/home/jovyan
```

ℹ️ Exercice : Navigation et exploration

Objectif : Créer une arborescence de projet et s'y déplacer

1. Créer l'arborescence suivante :

```
projet/
├── data/
├── scripts/
└── results/
```

2. Se déplacer dans `data/` et créer 3 fichiers vides
3. Lister le contenu de `projet/` depuis `data/` **sans changer de répertoire**
4. Revenir à votre répertoire personnel en **une seule commande**

Questions de réflexion :

- Quelle différence y a-t-il entre `cd ..` et `cd ../..` ?
- Pourquoi utiliser `mkdir -p` plutôt que `mkdir` simple ?
- `man mkdir`, la commande `man` affiche le manuel d'une commande.

```
#| code-fold: true
#| code-summary: "Solution"
mkdir -p projet/{data,scripts,results}
cd projet/data
touch file1.txt file2.txt file3.txt
echo "Contenu de projet/ depuis data/ :"
ls ..
cd ~
```

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

il est aussi possible de créer l'arborescence en une seule commande `{` et `}`, permettant de grouper des parties communes.

1.6 Manipulation de fichiers

Le shell offre de nombreuses commandes pour manipuler fichiers et répertoires.

1.6.a Commandes de manipulation essentielles

Commande	Description	Exemple
`touch`	Créer un fichier vide, modifier la date	`touch fichier.txt`
`mkdir`	Créer un répertoire	`mkdir dossier`
`cp`	Copier fichiers/répertoires	`cp source destination`
`mv`	Déplacer/renommer	`mv ancien nouveau`
`rm`	Supprimer	`rm fichier.txt`

1.6.b Options importantes

- `-r` : récursif (pour les répertoires)
- `-f` : force (pas de confirmation)
- `-v` : verbose (affiche les opérations)
- `-i` : interactif (demande confirmation)
- `-p` : parents (crée les répertoires intermédiaires)

⚠ **ATTENTION :** `rm` supprime définitivement les fichiers ! Toujours vérifier avant.

1.6.c Illustration

```
# Créer un répertoire
mkdir test_dir

# Créer des fichiers ou changer leur date à maintenant
touch file1.txt file2.txt

# Lister le contenu
ls -l
```

```
total 8
-rw-r--r-- 1 root root    0 Nov  1 16:40 file1.txt
-rw-r--r-- 1 root root    0 Nov  1 16:40 file2.txt
drwxr-xr-x 5 root root 4096 Nov  1 16:40 projet
-rw-r--r-- 1 root root    0 Nov  1 16:40 test.txt
drwxr-xr-x 2 root root 4096 Nov  1 16:40 test_dir
```

```
# Copier un fichier
cp file1.txt file1_backup.txt

# Déplacer un fichier
```

```
mv file2.txt test_dir/

# Lister le contenu récursivement
ls -R

# Lister sous forme d'arborescence
tree .
```

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

./projet:
data  results  scripts

./projet/data:
file1.txt  file2.txt  file3.txt

./projet/results:

./projet/scripts:

./test_dir:
file2.txt
.
├── file1.txt
├── file1_backup.txt
├── projet
│   ├── data
│   │   ├── file1.txt
│   │   ├── file2.txt
│   │   └── file3.txt
│   ├── results
│   └── scripts
├── test.txt
└── test_dir
    └── file2.txt

6 directories, 7 files
```

ℹ️ Exercice : Organisation de fichiers**

Objectif : Manipuler fichiers et répertoires

1. Dans `projet/data/`, créer 5 fichiers vides nommés `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` en le renommant `data3_clean.csv`
4. Supprimer `data5.csv` en demandant une confirmation avant la suppression

```
cd projet/data
touch data1.csv data2.csv data3.csv data4.csv data5.csv
mkdir raw
mv data1.csv data2.csv raw/
mkdir processed
cp data3.csv processed/data3_clean.csv
# rm -i data5.csv
```

2 BLOC 2 : Visualisation et redirection

2.1 Parcours pédagogiques

2.2 ◯ Minimal

Thème	Compétences clés
Visualisation	`cat fichier.txt`, `head -n 10`, `tail -n 5`, `wc -l` (lignes), `-w` (mots), `-c` (octets)
Redirections	`>` (écraser), `>>` (ajouter), `<` (entrée depuis fichier)
Recherche	`grep "motif" fichier.txt`, `-i` (insensible casse), `-v` (inverser), `-n` (numéros), `-c` (compter)

2.3 ◯ Approfondi

Thème	Compétences clés
Visualisation	`less` (navigation), `head`/`tail` avancés, `tail -f` (suivi temps réel)
Redirections	`2>` (erreurs), `2>&1` (fusion stdout/stderr), `&>` (tout rediriger)
Recherche	`grep -r "TODO" src/`, `-A 3 -B 2 -C 5` (contexte), regex de base `^$.*[]`, `grep -E` (regex étendues `+?|{}`)

2.4 ◯ Avancé

Thème	Compétences clés
Visualisation	Combinaisons complexes avec pipes, traitement de gros fichiers streaming
Redirections	Here-documents `<< EOF`, process substitution `<()`, pipelines avancés
Recherche	`rg` (ripgrep), `ag` (silver searcher), `grep -P` (PCRE), backreferences `\1`, extraction structurée (IPs, emails)

2.5 Lire et afficher des fichiers

Il existe plusieurs commandes pour afficher le contenu d'un fichier texte dans le terminal.

2.5.a Commandes de visualisation

Commande	Usage	Quand l'utiliser
`cat`	Affiche tout le contenu	Petits fichiers
`less`	Navigation interactive	Gros fichiers
`head`	Premières lignes	Aperçu rapide
`tail`	Dernières lignes	Logs, suivi
`wc`	Compte lignes/mots/caractères	Statistiques

2.5.b Redirections

En shell, les flux d'entrée/sortie sont des concepts clés :

- `stdin` (0) : entrée standard (clavier par défaut)
- `stdout` (1) : sortie standard (écran par défaut)
- `stderr` (2) : sortie d'erreur (écran par défaut)

Les redirections permettent de rediriger ces flux vers des fichiers ou d'autres commandes.

Opérateur	Action	Exemple
`>`	Écraser le fichier	`echo "texte" > file.txt`
`>>`	Ajouter à la fin	`echo "suite" >> file.txt`
`<`	Entrée depuis fichier	`wc -l < file.txt`
`2>`	Rediriger erreurs	`commande 2> errors.log`

2.5.c Parcours

- `[Minimal]` : Lire fichier, redirection simple `>` et `>>`
- `[Approfondi]` : `head`, `tail -f`, redirections `<` et `2>`
- `[Avancé]` : Combiner pipes et redirections pour traiter gros fichiers

2.5.d Illustration

```
# Créer un fichier texte
echo "Ligne 1" > fichier.txt
# Ajouter des lignes
echo "Ligne 2" >> fichier.txt
echo "Ligne 3" >> fichier.txt
# Afficher le contenu
cat fichier.txt
```

```
Ligne 1
Ligne 2
Ligne 3
```

Le shell propose aussi des boucles comme `for`. La forme basique est :

```
for variable in liste; do
  commandes
done
```

On peut utiliser cette boucle pour créer plusieurs lignes dans un fichier :

```
for i in {1..10}; do echo "Ligne $i" >> demo.txt; done
#Afficher les 3 premières lignes
head -n 3 demo.txt
#Afficher les 3 dernières lignes
tail -n 3 demo.txt
```

```
Ligne 1
Ligne 2
Ligne 3
Ligne 8
Ligne 9
Ligne 10
```

```
# Compter le nombre de lignes d'un fichier
wc -l < demo.txt
```

```
10
```

> ℹ️ Exercice : Création de données**
>
> **Objectif :** Manipuler fichiers texte et redirections
>
> 1. Créer un fichier `notes.txt` contenant les notes de 15 étudiants (nom, note) au format `EtudiantX,note` (X de 1 à 15, note de 15.5).
> 2. Afficher les 3 premières lignes de `notes.txt`
> 3. Ajouter 5 nouvelles notes pour des étudiants supplémentaires (16 à 20) avec une note de 16.0
> 4. Compter le nombre total de lignes dans `notes.txt` et enregistrer le résultat dans `stats.txt`
>
> ```
> for i in {1..15}; do echo "Etudiant$i,15.5" >> notes.txt; done
> head -n 3 notes.txt
> for i in {16..20}; do echo "Etudiant$i,16.0" >> notes.txt; done
> wc -l notes.txt > stats.txt
> cat stats.txt
> ```
>
> ```
> Etudiant1,15.5
> Etudiant2,15.5
> Etudiant3,15.5
> 20 notes.txt
> ```

2.6 Premières recherches

En plus des commandes internes au shell, Unix propose des outils puissants pour rechercher et filtrer du texte. Les fichiers sont traités comme des flux de texte (ligne par ligne), ce qui permet d'appliquer des commandes de recherche et de filtrage.

2.6.0.a `grep` - Rechercher des motifs

La commande `grep` (Global Regular Expression Print) permet de rechercher des lignes correspondant à un motif donné. Le motif peut être une chaîne simple ou une expression régulière.

```
grep "motif" fichier.txt
```

La syntaxe de base des expressions régulières et des extensions courantes :

- Caractères et classes
  - `.` : correspond à n'importe quel caractère (sauf saut de ligne par défaut)

    - `[abc]` : l'un des caractères `a` ou `b` ou `c`
    - `[a-z]` : intervalle (ici toute lettre minuscule)
    - `[^0-9]` : négation — tout caractère sauf chiffres
    - `\d`, `\w`, `\s` : digit, word, space (selon moteur : PCRE/grep -P)
- Quantificateurs
    - `*` : 0 ou plus
    - `+` : 1 ou plus
    - `?` : 0 ou 1
    - `{m}`, `{m,}`, `{m,n}` : répétitions exactes / au moins / intervalle
    - quantificateurs « paresseux » `*?`, `+?`, `{m,n}?` (PCRE) pour minimiser la correspondance
- Groupes, références et alternatives
    - `(expr)` : groupe capturant — permet `\1`, `\2` en remplacements
    - `(?:expr)` : groupe non capturant (pas de backreference)
    - `a|b` : alternative (a ou b)
- Ancres et assertions
    - `^` : début de ligne
    - `$` : fin de ligne
    - `\b` : bord de mot (word boundary)
    - Lookaround (assertions sans consommer les caractères) — PCRE / grep -P / perl:
        - `(?=expr)` : lookahead positif
        - `(?!expr)` : lookahead négatif
        - `(?⩽expr)` : lookbehind positif
        - `(?<!expr)` : lookbehind négatif

2.6.a Exemples pratiques d'expressions régulières utiles :

- Code postal français : `^[0-9]{5}$`
- Numéro de téléphone français : `^0[1-9]( [0-9]{2}){4}$`
- Adresses e-mail (simplifié) : `[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,6}`
- Date ISO (YYYY-MM-DD) : `^[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$`

2.6.b Illustration

La commande `cat > app.log << 'EOF'` crée (ou écrase) le fichier `app.log` et y écrit le texte jusqu'à la ligne contenant uniquement `EOF`.

"EOF" dans l'exemple ci-dessous est une convention pour indiquer la fin du texte à insérer dans le fichier. Tout ce qui est entre `<< 'EOF'` et `EOF` sera écrit dans le fichier `app.log`. Cela s'appelle une redirection de type « here document ».

```
# Créer un fichier de log exemple

cat > app.log << 'EOF'
INFO: Application démarrée
ERROR: Connexion échouée
WARNING: Mémoire faible
INFO: Traitement en cours
ERROR: Fichier introuvable
DEBUG: Variable x = 10
INFO: Database connectée
EOF
```

```
cat app.log
grep "ERROR" app.log
grep -ci "warning" app.log
grep -v "DEBUG" app.log
```

```
INFO: Application démarrée
ERROR: Connexion échouée
WARNING: Mémoire faible
INFO: Traitement en cours
ERROR: Fichier introuvable
DEBUG: Variable x = 10
INFO: Database connectée
ERROR: Connexion échouée
ERROR: Fichier introuvable
1
INFO: Application démarrée
ERROR: Connexion échouée
WARNING: Mémoire faible
INFO: Traitement en cours
ERROR: Fichier introuvable
INFO: Database connectée
```

ℹ️ Exercice : Recherche basique**

Objectif : Utiliser `grep` pour analyser un fichier de log 1. Afficher toutes les lignes contenant « ERROR » dans `app.log` 2. Compter le nombre de lignes contenant « WARNING » (insensible à la casse) 3. Afficher toutes les lignes ne contenant pas « INFO » 4. Afficher les lignes contenant « database » avec leur numéro de ligne (insensible à la casse)

```
#| code-fold: true
#| code-summary: "Solution"
grep "ERROR" app.log
grep -c "WARNING" app.log
grep -v "INFO" app.log
grep -in "database" app.log
```

```
ERROR: Connexion échouée
ERROR: Fichier introuvable
1
ERROR: Connexion échouée
WARNING: Mémoire faible
ERROR: Fichier introuvable
DEBUG: Variable x = 10
7:INFO: Database connectée
```

3 BLOC 3 : Traitement de texte avancé

3.1 Parcours pédagogiques — Bloc 3 : Traitement de texte avancé

3.2 ◯ Minimal

Thème	Compétences clés
Extraction (`cut`)	`cut -d',' -f1`, `cut -f2`, extraction simple de colonnes
Transformation (`tr`)	Remplacement simple `tr ';' ','`, casse `tr 'a-z' 'A-Z'`
Édition (`sed`)	Substitution simple `sed 's/a/b/'`, suppression `sed '/motif/d'`
Flux texte	Entrée standard `< fichier`, redirection `>`

3.3 ◯ Approfondi

Thème	Compétences clés
Extraction (`cut`)	Colonnes multiples `-f1,3`, caractères `-c1-10`
Transformation (`tr`)	Suppression `tr -d`, compression `tr -s`
Édition (`sed`)	Substitution globale `s///g`, expressions multiples `-e`
Pipelines	`cat | cut | tr | sed`, traitement en chaîne
Formats	Fichiers CSV simples, séparateurs personnalisés

3.4 ◯ Avancé

Thème	Compétences clés
Extraction conditionnelle	`grep | cut`, filtrage avancé
Transformation complexe	Pipelines `tr + sed` pour nettoyage massif
Regex avancées	Groupes `\(\)`, backreferences `\1`, ancrages `^ $`
Traitement streaming	Gros fichiers, scripts shell
Ouverture	Introduction conceptuelle à `awk`

3.5 3.1 Extraction avec `cut`

`cut` permet d'extraire des colonnes de fichiers structurés.

3.5.a Parcours

- `[Minimal]` : cut -d',' -f1
- `[Approfondi]` : cut plusieurs colonnes, caractères
- `[Avancé]` : combiner cut avec grep et pipes pour extraction conditionnelle

3.5.b Illustration et pratique

```
cat > etudiants.csv << 'EOF'
nom,prenom,note
Dupont,Jean,15
Martin,Marie,17
Bernard,Luc,12
EOF
```

```
echo "Champs nom,prenom,note dans etudiants.csv :"
cut -d',' -f1,3 etudiants.csv

echo "Premiers 6 caractères de chaque ligne :"
cut -c1-6 etudiants.csv
```

```
Champs nom,prenom,note dans etudiants.csv :
nom,note
Dupont,15
Martin,17
Bernard,12
Premiers 6 caractères de chaque ligne :
nom,pr
Dupont
Martin
Bernar
```

ℹ️ Exercice : Extraction de colonnes

Objectif : Utiliser `cut` pour extraire des données spécifiques

Dans le fichier `etudiants.csv` précedemment créé, effectuer les extractions demandées :

1. Extraire uniquement les notes des étudiants
2. Extraire les noms et prenoms des étudiants

```
echo "Extraire uniquement les notes :"
cut -d',' -f3 etudiants.csv
echo "Extraire les noms et prénoms :"
cut -d',' -f1,2 etudiants.csv
```

```
Extraire uniquement les notes :
note
15
17
12
Extraire les noms et prénoms :
nom,prenom
Dupont,Jean
Martin,Marie
Bernard,Luc
```

3.6 3.2 Transformation avec `tr`

`tr` (translate) permet de transformer des caractères dans un flux texte.

- `[Minimal]` : tr simple (a→A, ;→,) : `tr`tr “;” “,”`
- `[Approfondi]` : supprimer, compresser espaces : `tr -d '0-9'`, `tr -s ' '`
- `[Avancé]` : pipelines pour nettoyage de gros fichiers

tr lit depuis l’entrée standard et écrit sur la sortie standard. On peut donc l’utiliser en combinaison avec des redirections ou des pipes ().

ℹ️ Exercice : Nettoyage de données**

Dans la chaine de caractères suivante, effectuer les transformations demandées :

```
echo "  Alice;  Bob;  Charlie123;   David  " > raw_data.txt
```

- Remplacer `;` par `,`
- Convertir minuscules en majuscules
- Supprimer les chiffres
- Compresser les espaces multiples en un seul espace

```
echo "Remplacer ';' par ',' :"
tr ';' ',' < raw_data.txt
echo "Remplacer les minuscules par des majuscules :"
tr 'a-z' 'A-Z' < raw_data.txt
echo "Supprimer les chiffres :"
tr -d '0-9' < raw_data.txt
echo "Compresser les espaces multiples en un seul espace :"
tr -s ' ' < raw_data.txt
```

```
Remplacer ';' par ',' :
  Alice,  Bob,  Charlie123,   David
Remplacer les minuscules par des majuscules :
  ALICE;  BOB;  CHARLIE123;   DAVID
Supprimer les chiffres :
  Alice;  Bob;  Charlie;   David
Compresser les espaces multiples en un seul espace :
 Alice; Bob; Charlie123; David
```

3.7 Édition avec `sed`

La commande `sed` (stream editor) permet de faire des substitutions et transformations avancées sur des flux texte. Elle peut agit comme uen editeur de texte en ligne de commande.

La syntaxe de base pour une substitution est :

```
sed 's/motif/remplacement/options' fichier.txt
```

- `[Minimal]` : substitution simple, suppression de ligne : `s/ancien/nouveau/`, `sed '/motif/d'`
- `[Approfondi]` : global, multiple expressions : `s/ancien/nouveau/g`, `-e 's/a/A/' -e 's/b/B/'`
- `[Avancé]` : pipelines sed + tr + grep pour transformation complexe (cf. pipe ci-dessous)

ℹ️ Exercice : Transformations complexes**

Dans le fichier `products.txt` suivant, effectuer les transformations demandées :

```
cat > products.txt << 'EOF'
001;Laptop;999.99
002;Smartphone;499.49
003;Tablet;299.29
EOF
```

En utilisant `sed` et/ou `tr`, réaliser les opérations suivantes :

- Remplacer `;` par `,`
- Masquer les ID produits (ex: 001 → XXX)
- Convertir les prix en euros (ex: 999.99 → 999,99 €)

```
# Remplacer les points par des virgules dans les prix
tr ';' ',' < products.txt
# Masquer les ID produits
sed 's/^[0-9]\{3\}/XXX/' products.txt
# convertir les prix sans unité en euros
sed 's/\([0-9]\+\)\.\([0-9]\+\)/\1,\2 €/' products.txt
```

```
001,Laptop,999.99
002,Smartphone,499.49
003,Tablet,299.29
XXX;Laptop;999.99
XXX;Smartphone;499.49
XXX;Tablet;299.29
001;Laptop;999,99 €
002;Smartphone;499,49 €
003;Tablet;299,29 €
```

La commande `awk` est un langage de traitement de texte puissant pour des transformations plus complexes basées sur des motifs et des champs. Cependant, son apprentissage dépasse le cadre de ce TP.

4 BLOC 4 : Pipelines et composition

4.1 Parcours pédagogiques — Bloc 4 : Pipelines et composition

4.2 ◯ Minimal

Thème	Compétences clés
Pipes	Pipe simple `|`, enchaînement de deux commandes
Filtrage	`grep` en pipeline
Analyse simple	`wc -l` pour compter
Logs	Lecture et filtrage basique de fichiers de logs

4.3 ◯ Approfondi

Thème	Compétences clés
Pipelines multiples	`grep | wc`, `cut | sort`
Tri	`sort`, options de tri
Agrégation	`uniq`, `uniq -c`
Analyse de logs	Extraction de champs, statistiques simples

4.4 ◯ Avancé

Thème	Compétences clés
Tri avancé	Tri multi-colonnes `sort -k`
Pipelines conditionnels	Combinaisons complexes avec `grep`, `sed`
Performance	Traitement streaming de gros fichiers
Chaînes complexes	Pipelines réutilisables dans scripts

4.5 Philosophie Unix et pipes

Le shell Unix permet de chaîner des commandes via des **pipes** (`|`), où la sortie standard d'une commande devient l'entrée standard de la suivante.

- `[Minimal]` : pipe simple `|`
- `[Approfondi]` : sort, uniq, wc en pipeline
- `[Avancé]` : pipelines conditionnels, tri multi-colonnes

ℹ️ Exercice : Premiers pipelines**

Dans le fichier `webserver.log` suivant, effectuer les analyses demandées :

- Créer le fichier `webserver.log` avec des entrées simulées
- Afficher les lignes contenant « 404 » en utilisant un pipe avec `grep` et `cat`
- Compter le nombre de lignes contenant « 404 » avec un pipe entre `grep` et `wc -l`
- Lister les codes HTTP uniques présents dans le log avec `cut`, `sort` et `uniq`

```
cat > webserver.log << 'EOF'
127.0.0.1 - - [2026-01-31 10:00:01] "GET /index.html HTTP/1.1" 200 1024
192.168.0.5 - - [2026-01-31 10:01:12] "GET /missing.png HTTP/1.1" 404 512
10.0.0.2 - - [2026-01-31 10:02:34] "POST /api/upload HTTP/1.1" 500 2048
127.0.0.1 - - [2026-01-31 10:03:45] "GET /about HTTP/1.1" 200 2048
203.0.113.4 - - [2026-01-31 10:04:56] "GET /page HTTP/1.1" 404 256
EOF
```

```
#| code-fold: true
#| code-summary: "Solution"

# Afficher les lignes contenant "404"
cat webserver.log | grep "404"
# Compter le nombre de lignes contenant "404"
grep "404" webserver.log | wc -l
# Lister les codes HTTP uniques
cut -d' ' -f9 webserver.log | sort | uniq
```

```
192.168.0.5 - - [2026-01-31 10:01:12] "GET /missing.png HTTP/1.1" 404 512
203.0.113.4 - - [2026-01-31 10:04:56] "GET /page HTTP/1.1" 404 256
2
200
404
500
```

5 BLOC 5 : Notions complémentaires

5.1 Parcours pédagogiques — Bloc 5 : Notions complémentaires

5.2 ◯ Minimal

Thème	Compétences clés
Permissions	`ls -l`, `chmod +x`
Processus	Avant-plan / arrière-plan
Scripts	Script simple, `echo`, variables
Exécution	Lancer un script

5.3 ◯ Approfondi

Thème	Compétences clés
Permissions	Notation numérique `chmod 755`, `644`
Processus	`ps`, `top`, `jobs`, `&`
Scripting	Arguments `$1`, `$@`, boucles `for`
Variables	Export, guillemets simples/doubles

5.4 ◯ Avancé

Thème	Compétences clés
Gestion processus	Background, `kill`, détachement
Scripts robustes	Vérification arguments, codes de retour
Automatisation	Scripts avec `find`, `wc`, pipes
Environnement	`source`, variables persistantes

5.5 Permissions et processus

Sous unix les droits d'accès aux fichiers et répertoires sont gérés via un système de permissions.

Chaque fichier/répertoire a des permissions pour trois catégories d'utilisateurs :

- Propriétaire (user)
- Groupe (group)
- Autres (others)

Les permissions possibles sont :

- Lecture (r)
- Écriture (w)
- Exécution (x) - pour les fichiers, permet d'exécuter un script ou un programme ; pour les répertoires, permet d'accéder au contenu.

Les permissions sont affichées avec la commande `ls -l` :

```
-rwxr-xr-- 1 user group 1234 Jan 31 12:00 script.sh
drwxr-xr-- 1 user group 4096 Jan 31 12:00 mon_dossier
```

Les trois premiers caractères `rwx` sont les permissions du propriétaire, les trois suivants `r-x` celles du groupe, et les trois derniers `r--` celles des autres. Le premier caractère indique le type (fichier, répertoire, lien).

Il est possible de modifier les permissions avec la commande `chmod` (change mode). Soit en notation symbolique :

```
chmod u+x script.sh  # ajoute la permission d'exécution pour le propriétaire
chmod u+x script.sh  # ajoute la permission d'exécution pour le propriétaire
chmod g-w mon_dossier  # retire la permission d'écriture pour le groupe
```

Il est aussi possible d'utiliser la notation numérique dans laquelle chaque groupe permission (rwx) est vue comme une valeur binaire de 3 bits qui peut être convertie en un chiffre octal (0-7 : r=4, w=2, x=1) :

```
chmod 755 script.sh   # rwx r-x r-x
chmod 644 fichier.txt # rw- r-- r--
```

Les processus sont des programmes en cours d'exécution. Chaque processus a un PID (Process IDentifier) unique. Il a aussi des informations associées comme l'utilisateur, l'état, l'utilisation CPU/mémoire. Une processus peut être en avant-plan (foreground) ou en arrière-plan (background). En avant -plan, il bloque le terminal jusqu'à sa fin. En arrière-plan, il permet de continuer à utiliser le terminal. Lorsque le terminal est fermé, les processus en avant-plan sont terminés. Ceux en arrière-plan s'arrêtent aussi sauf s'ils sont détachés (avec `nohup` ou `disown` ou les outils `screen` ou `tmux`).

Les commandes utiles pour gérer les processus sont : | Commande | Description | Exemple | | ——— | ——————————- | ——————— | | `ps` | Affiche les processus en cours | `ps aux` | `top` | Affiche les processus en temps réel | `top` | | `kill` | Termine un processus par son PID | `kill 1234` | `&` | Lance un processus en arrière-plan | `./script.sh &` | `jobs` | Liste les processus en arrière-plan | `jobs` |

- `[Minimal]` : chmod +x, ls -l

- `[Approfondi]` : notation numérique, processus ps, top
- `[Avancé]` : scripts lancés en background, gestion erreurs

5.6 5.2 Variables et scripting

Cette partie n'est pas exhaustive mais donne les bases pour commencer à écrire des scripts shell.

Le shell permet de créer des scripts pour automatiser des tâches répétitives. Un script est un fichier texte executable contenant une série de commandes shell. Un script commence généralement par une ligne shebang (`#!/bin/bash`) pour indiquer quel interpréteur utiliser.

Les variables en shell sont créées sans déclaration préalable. On les assigne avec `=` sans espaces. Pour accéder à la valeur d'une variable, on utilise le préfixe `$`. On peut aussi utiliser les accolades `{}` pour délimiter le nom de la variable. Les variables sont typiquement des chaînes de caractères utilisées dans les commandes.

En shell, les guillemets simples (`'`) préservent la valeur littérale, tandis que les guillemets doubles (`"`) permettent l'expansion des variables.

Les scripts peuvent accepter des arguments positionnels, accessibles via `$1`, `$2`, etc. Le nombre total d'arguments est dans `$#`, et tous les arguments dans `$@`.

```
# Exemple de script simple
cat > hello.sh << 'EOF'
#!/bin/bash
# Script de salutation
NAME=$1
echo "Bonjour, $NAME !"
echo 'Exemple avec guillemets simples : $NAME'
EOF

chmod +x hello.sh
```

Pour qu'une variable soit accessible dans les sous-processus, elle doit être exportée avec `export`. La commande `source` ou `.` permet d'exécuter un script dans le contexte actuel du shell, important pour charger des variables d'environnement.

Un exemple plus complet :

```
cat > sum.sh << 'EOF'
#!/bin/bash

# Script pour afficher le maximum des paramètres
```

```
# Le shell offre des structures de contrôle comme les conditions et les boucles.
if [ $# -eq 0 ]; then
  echo "Usage: $0 nombre1 nombre2 ..."
  exit 1
fi

MAX=$1
for NUM in "$@"; do
  if [ "$NUM" -gt "$MAX" ]; then
    MAX=$NUM
  fi
done
echo "Le maximum est : $MAX"

export MAX

EOF

MAX=0
echo "Variable MAX avant source : $MAX"
chmod +x sum.sh
./sum.sh 10 25 7 42 3
echo "Variable MAX sans source : $MAX"
source sum.sh 10 25 7 42 3
echo "Variable MAX après source : $MAX"
```

```
Variable MAX avant source : 0
Le maximum est : 42
Variable MAX sans source : 0
Le maximum est : 42
Variable MAX après source : 42
```

- `[Minimal]` : variables simples, echo
- `[Approfondi]` : arguments, boucles
- `[Avancé]` : scripts robustes avec vérification d'arguments, find, wc