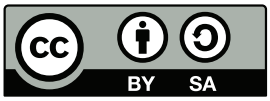
La ligne de commande

Robert Alessi

14 septembre 2018



Sommaire

1 Introduction

La *ligne de commande* est avant tout une interface de communication avec l'ordinateur, tout comme le sont les interfaces graphiques auxquelles nous sommes habitués. Les interfaces graphiques telles que le *Bureau* sous Windows ou bien le *Finder* sous Mac OS sont en réalité comme une couche qui se superpose à la ligne de commande, qui continue toujours à exister. Tout en masquant à l'utilisateur la ligne de commande, elles traduisent néanmoins en ligne de commande les opérations qui sont faites à l'aide de la souris.

Exemple Supposons que l'on veuille créer un répertoire [1] nommé *travail* sur le bureau, puis déplacer dans ce réperoire un fichier *exemple.pdf* que l'on vient de créer. À l'aide de la souris, on doit à peu près effectuer les opérations suivantes :

1. faire un simple-clic du bouton droit de la souris, et choisir Dossier > Nouveau dans le menu contextuel qui s'affiche ;
2. dans la fenêtre qui s'affiche, saisir le nom du nouveau répertoire : `travail`, puis cliquer sur le bouton Ok ;
3. dernière opération, le déplacement du fichier *exemple.pdf* : à l'aide du bouton gauche de la souris, faire glisser le fichier *exemple.pdf* sur l'icône du répertoire *travail*, et relâcher le bouton à ce moment.

Pour réaliser les mêmes opérations à la ligne de commande, il aurait fallu saisir les lignes suivantes :

1. création du dossier *travail* :
   `mkdir travail`
2. déplacement du fichier *exemple.pdf* dans le dossier travail :
   `mv exemple.pdf travail`

Commentaire Il faut bien comprendre que le rôle de l'interface graphique n'est que de traduire en lignes de commande les opérations que nous effectuons à l'aide de la souris. Reprenons à présent les deux dernières lignes de commande pour mieux les comprendre :

1. dans « `mkdir travail` », `mkdir` est le nom d'un programme fait pour créer des répertoires ; `mkdir` est en effet pour l'anglais *make directory*. Quant à *travail*, c'est tout simplement le nom du répertoire qu'on veut faire créer par le programme `mkdir`. La terminologie est la suivante : `mkdir` est le nom du programme, et `travail` est *l'argument* que l'on passe à `mkdir`. Remarquez que l'on doit séparer l'argument du nom du programme par un espace. Pour terminer, on appuie sur la touche *Entrée* pour commander l'exécution du programme.

1. C'est ansi que l'on appellera ce qui, sous Windows et Mac OS X, se nomme *dossier*.

2. dans « `mv exemple.pdf travail` », le nom du programme est `mv`, pour l'anglais *move* ; sa fonction est de déplacer des fichiers ou des répertoires. Comme son comportement, par rapport au programme `mkdir`, est différent, il accepte non pas un, mais deux arguments, chacun séparé par des espaces. Observez de nouveau cette ligne de commande : tandis que le premier argument est le nom du fichier que l'on souhaite déplacer, le deuxième est le nom du répertoire de destination de ce fichier. Pour terminer, de la même manière que précédemment, on appuie sur la touche *Entrée* pour commander le déplacement du fichier.

Par cet exemple, on espère faire comprendre que si la syntaxe de la ligne de commande peut paraître au premier abord difficile à maîtriser, elle permet aussi, par sa sobriété même, de réaliser de manière bien plus rapide et bien plus sûre les opérations que l'on fait à l'aide de la souris. En voici les principales raisons :

— l'interface graphique est une surcouche logicielle ; elle ralentit donc le système d'exploitation ;
— l'interface graphique, comme tout logiciel très complexe, comporte des erreurs de programmation. Ces *bugs* peuvent aller jusqu'à bloquer complètement le système d'exploitation ;
— à l'aide de l'interface graphique, on ne peut réaliser que les équivalents en ligne de commande qui ont été prévus par les programmeurs. En se privant de la ligne de commande, l'utilisateur se prive donc aussi de pouvoir réaliser les opérations qui ont été laissées de côté [2] ;
— les lignes de commande peuvent être chaînées. Ainsi, par la simple ligne `mkdir travail ; mv exemple.pdf travail` on peut réaliser en une seule fois toutes les opérations décrites précédemment. Il suffit, comme on le voit ici, de séparer les commandes par un point-virgule `;` ;
— les lignes de commande acceptent des caractères appelés *jokers* à l'aide desquels on peut déclencher des opérations complexes, portant sur un très grand nombre de fichiers. Par exemple, le caractère `*` peut se substituer à n'importe quelle chaîne de caractères. Ainsi, pour reprendre ce qui précède, la commande `mv *.pdf travail` aura pour effet de déplacer automatiquement tous les fichiers au format `PDF` dans le répertoire *travail*.

En d'autres termes, en passant par la ligne de commande, l'utilisateur gagne en sécurité, en rapidité et en maîtrise du système ce qu'il perd en ergonomie.

1.1 Lancement du terminal

1. Sous Linux, il suffit de rechercher dans le menu une application nommée *terminal* ou *xterm* dans les *outils système*.
2. Sous MacOs, l'application s'appelle *terminal*. Elle donne accès à un nombre limité de commandes.

2. C'est d'ailleurs ainsi, bien souvent, que les techniciens compétents dépannent les ordinateurs : en réalisant des commandes auxquelles l'interface graphique ne permet pas d'accéder.

3. Sous Windows, il faut installer *Cygwin* qui est disponible à l'adresse suivante : https://cygwin.com. Pour savoir comment faire :
   a) https://x.cygwin.com/docs/ug/setup.html (en anglais)
   b) http://migale.jouy.inra.fr/?q=fr/cygwin-install (en français : dans ce document, il faut cependant sauter le point 10a)

2 Chemins d'accès et premières commandes

Les commandes portent le plus souvent sur des fichiers. Il est donc important, pour savoir où se trouvent les fichiers que vous voulez traiter, de connaître leur *chemin d'accès*.

2.1 Chemins relatifs et absolus

Le home directory Dans les systèmes Linux, tous vos fichiers se trouvent dans votre répertoire personnel, appelé le *home directory*. Le nom de votre répertoire personnel est le même que celui de l'identifiant sous lequel vous vous êtes connecté. Par ailleurs, tous les répertoires des différents utilisateurs sont situés à la racine du disque dur dans un répertoire fondamental appelé `home`.

Supposons que votre identifiant soit `jacques` ; votre répertoire personnel sera donc :

```
/home/jacques
```

Obervez attentivement cette ligne. Vous remarquez que les noms des répertoires sont séparés par le caractère `/`. Cela veut dire que le signe `/` est utilisé pour indiquer que l'on passe d'un répertoire donné à l'un de ses sous-répertoires. Dans notre exemple, le répertoire `jacques` est donc inclus dans le répertoire `home`.

Remarquez encore le `/` qui est placé *devant* `home` : comme il n'est lui-même précédé de rien, il indique que le répertoire `home` est placé *à la racine du disque dur*.

Definition : chemins absolus, chemins relatifs Un chemin d'accès est dit *absolu* quand il est donné à partir de la racine du disque dur. Il est *relatif* quand il est donné à partir de tout autre endroit du disque dur. Soit par exemple le répertoire `travail` créé par l'utilisateur `jacques` dans son répertoire personnel. À partir de ce répertoire, le chemin d'accès absolu sera

```
/home/jacques/travail/
```

tandis que le chemin relatif sera

```
travail/
```

Corrolaire : tout chemin d'accès absolu commence nécessairement par le caractère `/` ; quand ce n'est pas le cas, le chemin d'accès est nécessairement relatif.

Conventions Il existe un grand nombre de raccourcis ou de signes conventionnels qui sont utilisés dans la ligne de commande. On en retiendra trois pour le moment :

— *home directory* : depuis tout endroit du disque dur, tout utilisateur peut accéder à son répertoire personnel par le raccourci

```
~/
```

Ainsi, pour l'utilisateur `jacques`, `~/travail` est l'équivalent de `/home/jacques/travail`.

— répertoire parent : quel que soit le répetoire dans lequel on se trouve, la séquence `..` désigne le *répertoire parent*, c'est-à-dire le répertoire qui le contient, ou bien qui est situé au niveau supérieur dans l'arborescence du disque dur. Par exemple, à partir du répertoire `/home/jacques/travail`, `..` désigne le répertoire `/home/jacques`.

— répertoire courant : Quant au signe « `.` », il désigne tout simplement le répertoire dans lequel on se trouve.

2.2 Premières commandes

pwd Signifie *print working directory*. Cette commande vous retourne tout simplement le chemin d'accès absolu du répertoire dans lequel vous vous trouvez. Très utile pour ne pas se perdre ! Exemple :

```
[robert@kiddo ~]$ pwd
/home/robert
```

La séquence `[robert@kiddo ~]$` est *l'invite de commande* (anglais *prompt*). C'est à la suite de cette invite que l'on entre les commandes. Nous y reviendrons. Observez pour le moment quelques unes des informations données par cette invite : l'utilisateur `robert` est connecté sur l'ordinateur `kiddo` ; ensuite, le signe `~` indique qu'il se trouve dans son *home directory*, ce que retourne en effet la commande `pwd` qui a été entrée ici. Enfin, le signe `$` indique que l'utilisateur `robert` n'est pas l'administrateur du système. En effet, le *prompt* de l'administrateur du système, que l'on appelle *root*, se termine par le signe `#`. Exemple :

```
[root@kiddo ~]# pwd
/root
[root@kiddo ~]#
```

Remarquez que le *home directory* de l'utilisateur `root` n'est pas situé dans le répertoire `/home` qui est réservé aux utilisateurs non privilégiés.

mv Signifie *move*. Cette commande déplace les fichiers d'un endroit vers un autre. La syntaxe est la suivante :

```
mv␣<source>␣<destination>
```

Par convention, le signe ␣ marque l'espace.

Exemple : déplacement du fichier `trachiniennes.pdf` dans le répertoire `travail` :

```
[robert@kiddo ~]$ mv trachiniennes.pdf travail/
```

Déplacement du fichier `trachiniennes.pdf` depuis le répertoire `travail` vers le répertoire courant (désigné par le raccourci `.`) :

```
[robert@kiddo ~]$ mv travail/trachiniennes.pdf .
```

Déplacement avec indication des chemins absolus :

```
[robert@kiddo ~]$ mv /home/robert/trachiniennes.pdf
↪  /home/robert/travail/
```

Utilisation de raccourcis :

```
[robert@kiddo ~]$ mv ~/trachiniennes.pdf ~/travail/
```

cp Signifie *copy*. Cette commande copie des fichiers depuis un endroit vers un autre. La syntaxe est comparable à celle de la séquence `mv`.

```
cp␣<source>␣<destination>
```

Exemple : copie du fichier `trachiniennes.pdf` dans le répertoire `travail` :

```
[robert@kiddo ~]$ cp trachiniennes.pdf travail/
```

Copie du fichier `trachiniennes.pdf` depuis le répertoire `travail` vers le répertoire courant (désigné par le raccourci `.`) :

```
[robert@kiddo ~]$ cp travail/trachiniennes.pdf .
```

Copie avec indication des chemins absolus :

```
[robert@kiddo ~]$ cp /home/robert/trachiniennes.pdf
↪  /home/robert/travail/
```

Utilisation de raccourcis :

```
[robert@kiddo ~]$ cp ~/trachiniennes.pdf ~/travail/
```

cd Signifie *change directory*. Permet de changer de répertoire courant, par exemple pour travailler sur les fichiers d'un répertoire différent de son *home directory*. La syntaxe est la suivante :

```
cd␣<chemin_d'accès_du_nouveau_répertoire>
```

Exemple : changement vers le répertoire `/usr/bin` :

```
[robert@kiddo ~]$ cd /usr/bin
[robert@kiddo /usr/bin]$
```

Remarquez le changement de l'invite après l'exécution de la commande. L'invite nous donne l'indication du nouveau répertoire.

Confirmation par la commande `pwd` :

```
[robert@kiddo /usr/bin]$ pwd
/usr/bin
[robert@kiddo /usr/bin]$
```

NB : la commande `cd` seule fait revenir l'utilisateur directement dans son *home directory*.

ls Signifie *list*. Affiche à l'écran tous les fichiers et les répertoires contenus dans un répertoire donné. Si on ne précise pas le répertoire dont il faut lister les fichiers, la commande liste les fichiers du répertoire courant. Exemple : on vérifie que le fichier `trachiniennes.pdf` se trouve bien dans le répertoire travail :

```
[robert@kiddo ~]$ ls travail/
trachiniennes.pdf
```

Comme on le voit, la commande retourne le nom du seul fichier qui se trouve dans le répertoire `travail`.
La commande `ls` est l'une des plus importantes ; elle admet de nombreuses options que nous détaillerons plus loin dans ce cours.

2.3 Options

On a donné plus haut l'exemple de la commande `ls` qui affiche à l'écran tous les fichiers contenus dans un répertoire donné. Voici ce que retourne cette commande lancée sur notre dépôt Git :

```
[robert@kiddo courses]$ ls
fichiers ls-R makefile _preamble.tex README.md README.pdf
↪  README.tex texfiles
```

Mais on peut souhaiter recueillir davantage d'informations. Par exemple, dans la liste ci-dessus, on ne peut pas distinguer les fichiers des répertoires. Heureusement, les commandes peuvent recevoir des *options*. Celles-ci sont de deux types :

1. Les options « longues », qui sont préfixées par `--` et suivies de noms entiers.
2. Les options « courtes », qui sont préfixées par `-` et suivies d'abréviations.

Voici donc ce que donne la même commande `ls`, suivie de l'option `-l` pour « use a long listing format » et de l'option `--color` :

```
1  [robert@kiddo courses]$ ls -l --color
2  total 56
3  drwxr-xr-x 3 robert robert  4096 12 sept. 21:52 fichiers
4  -rw-r--r-- 1 robert robert    88 12 sept. 20:57 ls-R
5  -rw-r--r-- 1 robert robert   627 12 sept. 15:11 makefile
6  -rw-r--r-- 1 robert robert   558 12 sept. 11:20 _preamble.tex
7  -rw-r--r-- 1 robert robert  1254 12 sept. 20:57 README.md
8  -rw-r--r-- 1 robert robert 27345 12 sept. 20:58 README.pdf
9  -rw-r--r-- 1 robert robert  1570 12 sept. 12:13 README.tex
10 -rw-r--r-- 1 robert robert    49 12 sept. 20:57 texfiles
```

Remarque L'option `--color` permet de distinguer facilement les fichiers et les répertoires.

Commentaire L'option `-l`, « long listing format », affiche d'abord sur la première ligne la somme des *file system blocks* occupés par les fichiers qui sont listés[1]. Les fichiers et les répertoires sont ensuite donnés dans les lignes suivantes. Prenons comme exemple la ligne 4 ci-dessus :

```
-rw-r--r-- 1 robert robert    88 12 sept. 20:57 ls-R
```

Il faut l'analyser en dix parties, de la façon suivante :

1	2	3	4	5	6	7	8	9	10
-	rw-	r--	r--	1	robert	robert	88	12 sept. 20:57	ls-R

1. Le *file system block* est la plus petite unité d'écriture possible sur un sytème de fichiers donné.

Voici une analyse simplifiée de cette ligne. Retenez que d'autres valeurs que celles qui sont commentées ci-dessous sont possibles.

1. Peut avoir les valeurs suivantes :
   — `-` pour les fichiers ;
   — `d` pour les répertoires ;
   — `l` pour les liens.
2. Permissions données au propriétaire. Il y a trois types de permissions que vous devez connaître ici :
   — `-` : aucune permission ;
   — `r` : permission en lecture ;
   — `w` : permission en écriture ;
   — `x` : permission en exécution.
   La première position représente les droits en *lecture* (valeurs possibles : `-` ou `r`) ; La deuxième position représente les droits en *écriture* (valeurs possibles : `-` ou `w`) ; La troisième position représente les droits en *exécution* (valeurs possibles : `-` ou `x`).
3. Permission de *groupe*. Les groupes peuvent réunir plusieurs utilisateurs. Par exemple, on peut créer un groupe *travail* et y mettre plusieurs utilisateurs qui auront ainsi des permissions communes sur les fichiers et les répertoires.
4. Permissions données à tout le monde.
5. Le nombre de liens sur le fichier ou le répertoire listé. Un fichier a en principe au moins un lien tandis qu'un répertoire en a au moins deux car le système considère que le répertoire parent et le répertoire courant (`..` et `.`, voir *supra* section 2.1 page 8) sont des liens.
6. Le nom du propriétaire du fichier.
7. Le nom du groupe dont les membres peuvent avoir des permissions sur le fichier.
8. La taille du fichier mesurée en *bytes*[2].
9. La date à laquelle le fichier a été créé ou modifié pour la dernière fois.
10. Le nom du fichier.

2. Le *byte* est une séquence de 8 *bits* traitée comme une seule unité d'information. Le *bit* de données peut avoir deux valeurs : 0 ou 1, ce nous qui rappelle que les ordinateurs sont des machines électriques dans lesquelles le courant peut passer (1) ou ne pas passer (0). L'unité conventionnelle du *bit* est b tandis que l'unité du *byte* est B. Il ne faut pas les confondre.

3 Bash, le shell

Ce que vous montre le terminal, à savoir l'invite de commande ou *prompt* en anglais, s'appelle le *shell*. Il y a plusieurs types de *shells*, mais le plus connu s'appelle *bash* [1]. La section 2.2 page 8 vous a appris une première série de commandes.

Les commandes portent sur les *fichiers*. Avant de continuer, il faut savoir que sous Linux *tout est fichier* : un fichier texte est un fichier, mais un répertoire aussi, de même que le clavier, l'écran et tous les périphériques de l'ordinateur sont des fichiers. Ainsi, un programme vidéo qui joue un film ne fait pas autre chose que copier des séquences vidéo vers le fichier qui désigne l'écran et des séquences son vers le fichier qui désigne la carte son.

La deuxième chose à savoir est que Linux est un système dit *sans extension*. L'*extension* est une séquence de 1 à 4 caractères placée après un point dans un nom de fichier. Elle permet de connaître quel est le type de chaque fichier. Par exemple, c'est par l'extension que l'on saura que `fichier.png` est un fichier image.

À la différence d'autres systèmes informatiques, Linux ne fait aucun cas de l'extension mais regarde directement à l'intérieur de chaque fichier pour en déterminer le type.

file La commande `file` permet de tout savoir sur les types de fichiers. Nous pouvons la lancer sur la racine de notre dépôt Git :

```
1 [robert@kiddo courses]$ file *
2 fichiers:      directory
3 ls-R:          ASCII text
4 makefile:      makefile script, ASCII text
5 _preamble.tex: LaTeX 2e document, ASCII text
6 README.md:     UTF-8 Unicode text
7 README.pdf:    PDF document, version 1.5
8 README.tex:    LaTeX 2e document, UTF-8 Unicode text
9 texfiles:      ASCII text
```

Remarques :

1. Les éléments des lignes 2, 3, 4 et 9 n'ont pas d'extension mais Linux détermine leur type de façon précise.
2. À la suite de la commande `file`, le caractère `*` a une signification spéciale : il désigne toute séquence formée par zéro ou plus de caractères : on l'a utilisé ici pour lister tous les fichiers du répertoire courant (v. *infra* section 3.2 page 16 pour plus de détails).

1. Pour *bourne again shell*.

Minuscules et majuscules À savoir également : Linux est sensible à la casse dans les noms des fichiers. Ainsi, `fichier.txt` et `Fichier.txt` sont deux fichiers différents.

Espaces Soit la commande suivante :

```
[robert@kiddo courses]$ ls -l README.md README.pdf
-rw-r--r-- 1 robert robert  1254 13 sept. 08:14 README.md
-rw-r--r-- 1 robert robert 27345 13 sept. 08:15 README.pdf
```

Comme on l'a vu plus haut (section 2.3 page 10), le *shell* n'a pas de mal à distinguer l'*option courte* des *arguments* car à la différence des arguments, l'option est préfixée par le signe `-`. La commande est donc interprétée de la façon suivante : « veuillez donner des informations au *format long* sur les deux fichiers `README.md` et `README.pdf` ».

C'est donc l'*espace* qui sert à délimiter les options et les arguments dans les commandes du *shell*. Techniquement, comme l'espace est interprété comme un délimiteur, on dit que c'est un *caractère actif* du *shell*. Ainsi, dans une ligne de commande, un fichier nommé `photos de vacances.zip` sera interprété comme une suite distincte de trois arguments :

1. `photos`
2. `de`
3. `vacances.zip`

Et le *shell* ne pourra pas le trouver. Il y a deux solutions possibles :

1. Placer le nom du fichier entre guillemets simples :

   ```
   [robert@kiddo courses]$ ls 'photos de vacances.zip'
   'photos de vacances.zip'
   ```

2. Préfixer les espaces par ce qu'on appelle l'*escape character* qui est le *backslash* (\)

   ```
   [robert@kiddo courses]$ ls photos\ de\ vacances.zip
   'photos de vacances.zip'
   ```

   Le rôle du caractère d'échappement est en effet d'annuler la signification particulière du caractère qui le suit. Or dans le *shell*, l'espace est un *caractère actif* puisqu'il est le délimiteur entre les commandes, les options et les arguments.

   On comprendra qu'il vaut mieux éviter d'utiliser les espaces dans les noms des fichiers. À la place des espaces, on utilisera le caractère de soulignement ou *underscore* : `photos_de_vacances.zip`

Fichiers cachés Tout fichier dont le nom commence par un point (`.`) est considéré comme un fichier caché. Le plus souvent, les fichiers cachés contiennent des paramètres de configuration. La commande `ls` est capable de les afficher si on lui passe l'option `-a` pour *all*. Appliquons cette option sur notre dépôt Git :

```
1  [robert@kiddo courses]$ ls -la
2  total 72
3  drwxr-xr-x 4 robert robert  4096 13 sept. 12:44 .
4  drwxr-xr-x 6 robert robert  4096 10 sept. 11:04 ..
5  drwxr-xr-x 3 robert robert  4096 13 sept. 12:44 fichiers
6  drwxr-xr-x 8 robert robert  4096 13 sept. 11:29 .git
7  -rw-r--r-- 1 robert robert    19  1 août   2017 .gitignore
8  -rw-r--r-- 1 robert robert    88 13 sept. 08:14 ls-R
9  -rw-r--r-- 1 robert robert   627 12 sept. 15:11 makefile
10 -rw-r--r-- 1 robert robert   588 13 sept. 11:29 _preamble.tex
11 -rw-r--r-- 1 robert robert  1254 13 sept. 08:14 README.md
12 -rw-r--r-- 1 robert robert 27345 13 sept. 08:15 README.pdf
13 -rw-r--r-- 1 robert robert  1570 12 sept. 12:13 README.tex
14 -rw-r--r-- 1 robert robert    49 13 sept. 08:14 texfiles
```

Nous voyons ainsi apparaître à la ligne 6 un répertoire caché et à la ligne 7 un fichier caché.

3.1 Commandes courantes

Ces commandes s'ajoutent à celles qui sont décrites plus haut (section 2.2 page 8).

mkdir Sert à créer un nouveau répertoire. L'option `-p` peut-être utilisée si l'on veut créer d'un coup un répertoire et un ou plusieurs sous-répertoires. L'option `-v`, pour *verbose*, demande aussi à `mkdir` de retourner un message de confirmation :

```
[robert@kiddo courses]$ mkdir -pv sandbox/robert
mkdir: création du répertoire 'sandbox'
mkdir: création du répertoire 'sandbox/robert'
```

touch Sert à créer un fichier vide dont le nom est passé en argument. Cette commande sert également à modifier les métadonnées de temps associées aux fichiers (date de création et/ou de modification).

L'exemple suivant montre comment créer un nouveau dossier dans lequel on crée également un fichier vide `fichier.txt`. Ensuite, on utilise la commande `mv` pour *déplacer* ce fichier vers un autre fichier `fichier-mk2.txt` au même endroit. Le résultat de cette action particulière, le *déplacement au même endroit*, est tout simplement de renommer le fichier. Enfin, la commande `ls -l` sert de moyen de contrôle :

```
[robert@kiddo courses]$ mkdir -pv sandbox
mkdir: création du répertoire 'sandbox'
[robert@kiddo courses]$ touch sandbox/fichier.txt
```

```
[robert@kiddo courses]$ mv sandbox/fichier.txt                    ⤾
↪  sandbox/fichier-mk2.txt
[robert@kiddo courses]$ ls -l sandbox/
total 0
-rw-r--r-- 1 robert robert 0 13 sept. 15:51 fichier-mk2.txt
```

3.1.1 Commandes destructives

rm Pour *remove*. Il suffit de passer en argument à cette commande ce que l'on souhaite détruire. Par défaut, cette commande ne détruit pas les répertoires. Elle accepte une série d'options dont voici les plus importantes :

-i demande une confirmation à chaque opération de destruction.

-f pour *force* ; fait le contraire de `-i`.

-r pour *recursive* ; détruit les répertoires et leur contenu.

Cette commande doit être exécutée avec précaution car il n'y a aucun retour en arrière possible.

Par exemple, soit le répertoire `sandbox`. La commande :

```
[robert@kiddo courses]$ rm -rf sandbox
```

détruira sans aucune demande de confirmation à la fois le répertoire `sandbox` et tout ce qu'il contient, fichiers, répertoires et sous-répertoires.

Pour donner une idée de la puissance de destruction de cette commande, la ligne suivante :

```
[robert@kiddo courses]$ rm -rf *
```

détruira absolument tout sans demande de confirmation pour ne laisser que les fichiers cachés du répertoire courant dont le nom commence par un point.

3.2 Wildcards

Les *wildcards*, ou « métacaractères » sont des caractères ou des séquences de caractères qui servent à représenter des séries de caractères. On les utilise pour construire des schémas de remplacement. Voici quels sont les plus utilisés :

— `*` représente zéro ou plus de caractères.
— `?` représente un caractère unique.
— `[]` représente une série de caractères. Par exemple, `[QGH]` représente l'une des trois lettres majuscules Q, G ou H. Ainsi, la commande :

```
ls [QGH]*
```

retournera tous les noms de fichiers qui commencent par l'une de ces trois lettres. Pour représenter une *série continue de caractères*, on peut utiliser le trait d'union. Par exemple, `[a-z]` représente toutes les lettres minuscules non accentuées de l'alphabet ; et `[A-Za-z]` toutes les lettres non accentuées, majuscules ou minuscules.

3.3 Redirection et chaînage

Nous avons vu jusqu'ici que les commandes renvoient normalement leur résultat sur le terminal lui-même. On peut cependant rediriger ce que les commandes renvoient vers un fichier à l'aide des *opérateurs de redirection*. Trois d'entre eux sont utiles à connaître :

1. `>` redirige vers un nouveau fichier. Si le fichier n'existe pas, il est créé. S'il existe, il sera écrasé et remplacé par un nouveau fichier du même nom.
2. `>>` fait la même chose que `>`, mais *ajoute* le résultat au fichier si celui-ci existe.
3. `<` lit le contenu du fichier dont le nom suit et le passe en argument à la commande qui précède pour traitement.

Dans l'exemple qui suit, on demande à la commande `ls -l` de rediriger son résultat vers un fichier `all-files.txt`. On s'assure que ce fichier a bien été créé, puis on demande à la commande `cat` d'en afficher le contenu au terminal. Les trois commandes sont entrées aux lignes 1, 2 et 4 :

```
1  [robert@kiddo courses]$ ls -l > all-files.txt
2  [robert@kiddo courses]$ ls
3  all-files.txt  fichiers  ls-R  makefile  _preamble.tex  README.md  ↩
   ↪  README.tex
4  [robert@kiddo courses]$ cat all-files.txt
5  total 24
6  drwxr-xr-x 3 robert robert 4096 13 sept. 17:15 fichiers
7  -rw-r--r-- 1 robert robert   88 13 sept. 13:34 ls-R
8  -rw-r--r-- 1 robert robert  627 12 sept. 15:11 makefile
9  -rw-r--r-- 1 robert robert  668 13 sept. 15:26 _preamble.tex
10 -rw-r--r-- 1 robert robert 1254 13 sept. 13:34 README.md
11 -rw-r--r-- 1 robert robert 1570 12 sept. 12:13 README.tex
```

L'exemple suivant est plus subtil. Il fait appel à une commande, `wc -l` qui compte les lignes des fichiers [2] :

```
1  [robert@kiddo courses]$ wc -l makefile
2  21 makefile
3  [robert@kiddo courses]$ wc -l < makefile
```

2. Voir plus loin page 19.

```
4  21
```

Comme on le voit, la commande entrée à la ligne 1 renvoie deux informations : le nombre de lignes du fichier (21) et le nom du fichier lui-même. Quant à la commande de la ligne 3, elle utilise l'opérateur de redirection `<` qui a pour effet de passer en argument à la commande non pas un nom de fichier à traiter, mais son contenu seul, lu puis redirigé. C'est une variante anonyme de la commande de la ligne 1.

Chaînage En anglais, *piping*. Comme nous l'avons vu, la redirection permet d'envoyer des résultats vers des fichiers ou bien des contenus de fichiers vers des commandes. Le *chaînage* consiste à faire passer un résultat fourni par une commande à une autre commande censée en fournir un nouveau traitement. L'opérateur de chaînage est le caractère *pipe* (`|`).

Avant d'aller plus loin, étudions rapidement deux nouvelles commandes qui servent à filtrer le contenu des fichiers.

head `head -<num> fichier` affiche au terminal les `<num>` premières lignes d'un fichier. Sans l'option `-<num>`, les 10 premières lignes sont affichées. Exemple :

```
[robert@kiddo courses]$ head -3 ls-R
./fichiers/01-ligne-de-commande.tex
./makefile
./_preamble.tex
```

tail `tail -<num> fichier` affiche au terminal les `<num>` dernières lignes d'un fichier. Sans l'option `-<num>`, les 10 dernières lignes sont affichées. Exemple :

```
[robert@kiddo courses]$ tail -3 ls-R
./_preamble.tex
./README.md
./README.tex
```

En outre, on peut passer à `tail` l'option `-n +<num>` qui affiche tout un fichier jusqu'à la dernière ligne, *mais en commençant à partir de la ligne* `<num>`. L'exemple suivant en montre une application directe.

Cet exemple reprend des commandes connues. Supposons que l'on veuille connaître simplement le nombre de fichiers du notre dépôt Git. Nous savons produire une liste à l'aide de la commande `ls -l`. Nous savons également que la commande `wc -l` compte les lignes. Cependant, la première ligne retournée par la commande `ls -l`, qui donne la somme des *file system blocks* occupés par le contenu du répertoire, doit être exclue du compte (voir *supra*, page 11). C'est ici qu'intervient la commande, `tail`, qui retourne les dernières lignes d'un fichier. Avec l'option `-n +2`, la première ligne sera ignorée :

```
[robert@kiddo courses]$ ls -l | tail -n +2 | wc -l
6
```

3.4 Filtrage

Comme leur nom l'indique, les commandes de filtrage servent à mettre en forme des fichiers texte tout en sélectionnant certaines parties de leur contenu.

Nous en avons étudié deux plus haut (page précédente) :

1. `head` qui sélectionne les premières lignes d'un fichier.
2. `tail` qui sélectionne les dernières lignes d'un fichier.

On ajoutera ici les commandes suivantes :

cat Affiche au terminal tout le contenu d'un fichier :

```
[robert@kiddo courses]$ cat ls-R
./fichiers/01-ligne-de-commande.tex
./makefile
./_preamble.tex
./README.md
./README.tex
```

wc Pour *word count*. Cette commande a été utilisée plus haut une fois avec l'option `-l` pour compter les lignes d'un fichier (page 17). Utilisée sans option, elle retourne le nombre de lignes (`-l`), de mots (`-w`) et de caractères (`-m`) d'un fichier :

```
[robert@kiddo courses]$ wc makefile
 21 114 627 makefile
```

cut Permet de mettre en forme des données. Prenons l'exemple du fichier suivant : `etudiants.txt`

```
Fonsec Sophie 123456 sophie.fonsec@quelquepart.net
Pédot Hector 456789 hector.pedot@ailleurs.org
```

Il contient sur chaque ligne un nom, un prénom, un matricule et une adresse email. Nous souhaitons collecter simplement les données suivantes :

1. Nom
2. Prénom
3. email

La commande `cut` peut être utilisée à cet effet avec les deux options suivantes :

1. `-d` indiquer par quel caractère les données sont délimitées (ici un espace).
2. `-f` (pour *field* en anglais) pour indiquer quelles données doivent être sélectionnées (ici, les éléments 1, puis 2, puis 4).

```
[robert@kiddo courses]$ cut -d ' ' -f 2,1,4 etudiants.txt
Fonsec Sophie sophie.fonsec@quelquepart.net
Pédot Hector hector.pedot@ailleurs.org
```

Mais comment faire pour modifier l'ordre des données et les mettre en forme de façon à placer le prénom avant le nom et avoir les adresses email entre crochets pointus ? Comment faire aussi pour récupérer les données dans un tableur ?

awk Ce programme accessible à la ligne de commande permet d'effectuer ce travail facilement. Il sélectionne les données dans l'ordre que l'on souhaite à l'aide de variables : `$1`, `$2`, `$3`, &c. Il effectue ensuite ce qu'on appelle des *actions*, lesquelles sont spécifiées entre accolades `{}`. Nous allons ici utiliser l'action `print`. Voici la commande :

```
[robert@kiddo courses]$ awk '{print $2 ";" $1 ";<" $4 ">"}'
↪  etudiants.txt
Sophie;Fonsec;<sophie.fonsec@quelquepart.net>
Hector;Pédot;<hector.pedot@ailleurs.org>
```

Commentaire

1. La totalité de l'argument passé à `awk` a été placée entre guillemets simples `' '`. On renvoie sur ce point à la règle posée page 14.
2. Entre les accolades, l'instruction `print` accomplit successivement les tâches suivantes :
   a) Impression du champ 2.
   b) Impression de la chaîne littérale `;` placée entre guillemets[3].
   c) Impression du champ 1.
   d) Impression de la chaîne littérale `;<` placée entre guillemets.
   e) Impression du champ 4.
   f) Impression de la chaîne littérale `>` placée entre guillemets.

Pour terminer, il suffit de renommer le fichier `etudiants.txt` en `etudiants.csv` par la commande :

```
mv etudiants.txt etudiants.csv
```

et de l'ouvrir dans LibreOffice Calc.

3. On n'aurait pas pu employer ici les guillemets simples car le premier guillemet simple aurait évidemment été compris comme le guillemet fermant celui qui se trouve juste avant l'accolade ouvrante.