Systèmes d'exploitation

TP 1 - Le shell Unix

2023-02-13

1 Pour débuter

Le shell est un programme dans lequel les utilisateurs peuvent taper des commandes pour invoquer des programmes plus ou moins compliqués. Maîtriser le shell nécessite d'apprendre quelques commandes comme un nouveau vocabulaire dans une langue que vous étudiez. Cependant, à la différence d'une langue parlée, un petit nombre de "mots" (c'est-à-dire de commandes) vous mènera loin, et nous allons couvrir ces quelques éléments essentiels aujourd'hui.

Commençons par **lancer un terminal** en cliquant sur l'icône dédiée dans l'interface graphique. Vous devriez alors voir apparaître dans la fenêtre du terminal un invite de commande sous la forme `$` ou `%`.

Vérifions que notre shell est bien Bash. Écrivons une première commande et validons avec la touche `Entrée` pour l'éxecuter :

```
$ echo $SHELL
```

`$SHELL` est une variable qui indique votre shell actuel. `/bin/bash` devrait s'afficher si vous êtes sur Ubuntu. Autrement, si vous êtes sur un autre système d'exploitation, comme macOS, et si `/bin/bash` ne s'affiche pas, lancez le shell Bash en exécutant la commande `bash`.

Exécutons à présent une seconde commande.

```
$ echo Bonjour
```

"Bonjour" devrait s'afficher sur le terminal car la commande `echo` affiche des chaînes de caractères à l'écran.

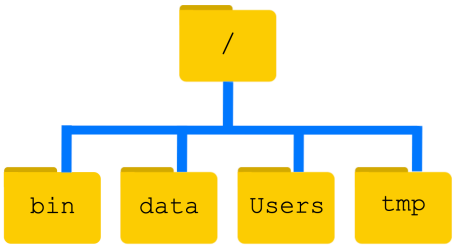
Quelques raccourcis utiles

- `TAB` fournit une auto-completion de la commande en train d'être tapée.
- La touche ↑ permet de retrouver la commande précédente et de remonter dans l'historique des commandes. De façon similaire ↓ permet de descendre dans l'historique des commandes.
- `CTRL` + `A` renvoie au début de la ligne courante.
- `CTRL` + `E` renvoie à la fin de la ligne courante.
- `CTRL` + `W` supprime le mot situé avant le curseur.
- `CTRL` + `U` supprime le contenu de la ligne situé avant le curseur.

2 Naviguer dans les fichiers et les répertoires

Le système de fichiers est la partie du système d'exploitation responsable de la gestion des fichiers et des répertoires. Il organise les données en fichiers, qui contiennent des informations, et en répertoires (également appelés "dossiers"), qui contiennent des fichiers ou d'autres répertoires.

Le système de fichiers forme une arborescence :



Au sommet se trouve le **répertoire racine** qui contient tout le reste. On y fait référence en utilisant un slash `/` seul.

À l'intérieur de ce répertoire se trouvent plusieurs autres répertoires : `bin` (où sont stockés certains programmes), `Users` (où se trouvent les répertoires personnels des utilisateurs), `tmp` (pour les fichiers temporaires), etc.

Plusieurs commandes sont fréquemment utilisées pour créer, inspecter, renommer et supprimer des fichiers et des répertoires.

Tout d'abord, déterminons où nous nous trouvons en lançant une commande appelée `pwd` (qui signifie "*print working directory*"). Les répertoires sont comme des lieux, à tout moment lorsque nous utilisons l'interpréteur de commandes, nous nous trouvons à un endroit précis appelé **répertoire de travail actuel** (*current working directory*). Les commandes lisent et écrivent la plupart du temps des fichiers dans le répertoire de travail actuel, il est donc important de savoir où l'on se trouve avant de lancer une commande.

`pwd` vous indique où vous êtes :

```
$ pwd
```

Nous pouvons voir ce qu'il y a dans le répertoire courant en exécutant la commande `ls`:

```
$ ls
```

Nous pouvons rendre sa sortie plus compréhensible en utilisant l'option `-F` qui indique à `ls` de classer la sortie en ajoutant un marqueur aux noms de fichiers et de répertoires pour indiquer ce qu'ils sont :

- un slash `/` indique qu'il s'agit d'un répertoire
- `@` indique un lien
- `*` indique un exécutable

```
$ ls -F
```

> **Effacer le terminal**
>
> Si votre écran est trop encombré, vous pouvez effacer votre terminal à l'aide de la commande `clear` ou des touches `CTRL` + `L`. Vous pouvez toujours accéder aux commandes précédentes en utilisant les touches ↑ et ↓.

2.1 Obtenir de l'aide

`ls` a beaucoup d'autres options. Il existe deux manières courantes pour savoir comment s'utilise une commande et quelles options elle accepte. En fonction de votre environnement (Linux ou macOS), vous constaterez peut-être qu'une seule de ces méthodes fonctionne.

1. On peut lire son manuel avec la commande `man`.

```
$ man ls
```

2. Nous pouvons passer l'option `--help` à la commande.

```
$ ls --help
```

2.2 La commande `man`

```
$ man ls
```

Cette commande transforme votre terminal en une page avec une description de la commande `ls` et de ses options.

Pour naviguer dans les pages `man`, vous pouvez utiliser les touches `↑` et `↓` pour vous déplacer ligne par ligne, ou `B` et `Espace` pour sauter d'une page entière vers le haut et vers le bas.

Pour rechercher un caractère ou un mot dans les pages `man`, utilisez `/` suivi du caractère ou du mot recherché. Parfois, une recherche donne plusieurs résultats. Si c'est le cas, vous pouvez passer d'un résultat à l'autre en utilisant la touche `N` (pour avancer) et la combinaison de touches `MAJ` + `N` (pour reculer).

Pour quitter les pages `man`, appuyez sur la touche `Q`.

Explorer plus d'options de la commande `ls`

Vous pouvez également utiliser deux options en même temps. Que fait la commande `ls` lorsqu'elle est utilisée avec l'option `-l` ? Qu'en est-il si vous utilisez à la fois l'option `-l` et l'option `-h` ?

Liste dans l'ordre chronologique inverse

Par défaut, `ls` affiche le contenu d'un répertoire par ordre alphabétique de nom. La commande `ls -t` affiche les éléments par date de dernière modification au lieu de l'ordre alphabétique. La commande `ls -r` affiche le contenu d'un répertoire dans l'ordre inverse.

Quel fichier s'affiche en dernier lorsque vous combinez les options `-t` et `-r` ? Vous devrez peut-être utiliser l'option `-l` pour voir les dernières dates modifiées.

2.3 Lister le contenu d'autres répertoires

Non seulement nous pouvons utiliser `ls` pour le répertoire de travail actuel, mais nous pouvons également l'utiliser pour afficher le contenu d'un répertoire différent.

Examinons notre répertoire `Desktop` en exécutant la commande suivante :

```
$ ls -F Desktop
```

Il s'agit de la commande `ls` avec l'option `-F` et l'argument `Desktop`. L'argument `Desktop` indique à `ls` que nous voulons une liste du contenu d'un autre répertoire que notre répertoire de travail actuel.

L'affichage doit être une liste de tous les fichiers et sous-répertoires de votre répertoire `Desktop`. Jetez un oeil à votre bureau dans l'interface graphique pour confirmer que c'est bien le cas.

2.4 Changer de répertoire de travail

Nous pouvons changer notre répertoire de travail pour un répertoire différent, de sorte que nous ne soyons plus situés dans notre répertoire personnel.

La commande pour changer de répertoire de travail actuel est `cd` suivie d'un nom de répertoire pour changer notre répertoire de travail. `cd` signifie « *change directory* ». La commande `cd` revient à double-cliquer sur un dossier dans une interface graphique pour accéder à un dossier.

Changons maintenant le répertoire de travail pour l'emplacement `Desktop`.

```
$ cd Desktop
```

Il existe un raccourci dans le shell pour remonter d'un niveau de répertoire :

```
$ cd ..
```

`..` est un nom de répertoire spécial signifiant "le répertoire contenant celui-ci", ou plus succinctement, **le parent du répertoire courant**.

Le répertoire spécial `..` n'apparaît pas lorsque nous exécutons `ls`. Si nous voulons l'afficher, nous pouvons ajouter l'option `-a` à ls -F :

```
ls -F -a
```

`-a` signifie "afficher tout" (y compris les fichiers cachés) ; cela oblige `ls` à nous montrer les noms des fichiers et de répertoires commençant par ., tels que `...`

Comme vous pouvez le voir, un autre répertoire spécial qui s'appelle simplement `.` s'affiche. Il désigne simplement « le répertoire de travail actuel ». Il peut sembler redondant d'avoir un nom pour cela, mais nous en verrons bientôt quelques utilisations.

Fichiers cachés

En plus des répertoires cachés `..` et `.`, vous devriez également voir un fichier appelé `.bash_profile`. Ce fichier contient généralement les paramètres de configuration du shell.

Vous devriez également voir d'autres fichiers et répertoires commençant par `..` Ce sont généralement des fichiers et des répertoires utilisés pour configurer différents programmes sur votre ordinateur. Le préfixe `.` permet d'éviter que ces fichiers de configuration n'encombrent le terminal lorsque la commande `ls` standard est utilisée.

`cd` sans répertoire

Déplacez vous plusieurs fois dans l'arborescence du système de fichiers en utilisant la commande `cd`. Maintenant, que se passe-t-il si vous tapez `cd` tout seul, sans donner de répertoire ?

Astuce: la commande `pwd` vous permettrant de déterminer le répertoire courant.

`cd` sans argument vous ramènera à votre répertoire personnel, ce qui est très bien si vous vous êtes perdu dans le système de fichiers.

2.5 Chemins absolus

Jusqu'à présent, lors de la spécification de noms de répertoires, ou même d'un chemin de répertoire (comme ci-dessus), nous avons utilisé des **chemins relatifs**. Lorsque vous utilisez un chemin relatif avec une commande telle que `ls` ou `cd`, cet emplacement est déterminé à partir du répertoir de travail actuel, plutôt qu'à partir de la racine du système de fichiers.

Cependant, il est possible de spécifier le **chemin absolu** vers un répertoire en incluant son chemin complet à partir du répertoire racine, qui est indiqué par une barre oblique au début. Le slash initial indique à l'ordinateur de suivre le chemin à partir de la racine du système de fichiers, de sorte qu'il se réfère toujours à exactement un répertoire, peu importe où nous nous trouvons lorsque nous exécutons la commande.

Par exemple, pour se déplacer dans le répertoire du bureau depuis n'importe quel emplacement, on pourra utiliser :

```
$ cd /Users/<nom de l'utilisateur>/Desktop
```

Chemin ~

Le shell interprète un caractère tilde (`~`) au début d'un chemin comme signifiant « le répertoire personnel de l'utilisateur actuel ». Par exemple, si le répertoire personnel de Jean est `/Users/jean`, alors `~/data` est équivalent à `/Users/jean/data`. Cela ne fonctionne que s'il s'agit du premier caractère du chemin.

`cd -`

Un autre raccourci est le caractère `-` (tiret). La commande `cd -` nous ramène dans le répertoire précédent dans lequel nous nous trouvions. Cela est plus rapide que d'avoir à se souvenir, puis à taper, le chemin complet.

C'est un moyen très efficace d'aller et venir entre deux répertoires : si vous exécutez `cd -` deux fois, vous vous retrouvez dans le répertoire de départ.

Exercice: chemins absolus ou relatifs

À partir de `/Users/amanda/data`, laquelle des commandes suivantes Amanda pourrait-elle utiliser pour accéder à son répertoire personnel, c'est -à-dire `/Users/amanda` ?

1. `cd .`
2. `cd /`
3. `cd /home/amanda`
4. `cd ../..`
5. `cd ~`
6. `cd home`
7. `cd ~/data/..`
8. `cd`

9. `cd ..`

3 Syntaxe générale d'une commande shell

Nous rencontré des commandes, des options et des arguments, intéressons nous maintenant en détails à la structure de ces commandes et à leurs composants.

Considérons la commande ci-dessous comme un exemple général de commande :

```
$ ls -F /
```

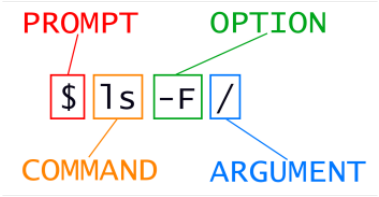


Figure 1: Syntaxe d'une commande shell

`ls` est la **commande**, avec une **option** `-F` et un **argument** `/` .

Certaines options commencent par un seul tiret (`-`) d'autres par deux tirets (`--`). Les options modifient le comportement d'une commande.

Les arguments indiquent à la commande sur quoi opérer (par exemple, les fichiers et les répertoires).

Une commande peut être appelée avec plus d'une option et plus d'un argument, mais une commande ne nécessite pas forcément un argument ou une option.

> *flags*
>
> On appelle parfois les options *switchs* ou *flags* , en particulier pour les options qui ne prennent aucun argument.

Chaque partie est séparée par des espaces et la capitalisation est importante.

4 Travailler avec des fichiers et des répertoires

4.1 Créer un répertoire

Créons un nouveau répertoire appelé `tp1` à l'aide de la commande `mkdir` (qui n'a pas de sortie) :

```
$ mkdir tp1
```

`mkdir` signifie "*make directory*". Comme `tp1` est un chemin relatif (il n'a pas de slash au début), le nouveau répertoire est créé dans le répertoire de travail actuel. Vérifions que c'est bien le cas avec :

```
$ ls -F
```

Notez que mkdir ne se limite pas à la création de répertoires uniques un à la fois :

```
$ mkdir donnees resultats
```

Créer un répertoire avec des sous-répertoires

`mkdir` peut également créer un répertoire avec des sous-répertoires imbriqués en une seule opération. Par exemple, créer en une seule commande l'arborescence `projet/donnees`. Utilisez la commande `man` pour déterminer quelle option doit être utilisée pour réaliser cette opération.

Quels noms pour les fichiers et les répertoires ?

Certains noms de fichiers et de répertoires peuvent vous compliquer la vie lorsque vous travaillez sur la ligne de commande. Nous fournissons ici quelques conseils utiles pour les noms de vos fichiers et répertoires.

- **N'utilisez pas d'espaces**.

  Les espaces peuvent rendre un nom plus significatif, mais comme les espaces sont utilisés pour séparer les arguments sur la ligne de commande, il est préférable de les éviter dans les noms de fichiers et de répertoires. Vous pouvez utiliser `-` ou à la `_` place.

- **Ne commencez pas le nom par `-` (tiret)**.

  Les commandes traitent les noms commençant par -comme des options.

- **Si vous devez faire référence à des noms de fichiers ou de répertoires contenant des espaces ou d'autres caractères spéciaux, vous devez entourer le nom de guillemets (" ")**.

4.2 Créer un fichier texte

Changeons notre répertoire de travail en `tp1`, puis exécutons un éditeur de texte appelé **Nano** pour créer un fichier appelé `brouillon.txt` :

```
$ cd tp1
$ nano brouillon.txt
```

Tapons le texte suivant :

L'optimisation prématurée est la racine de tous les maux.

Une fois que nous sommes satisfaits de notre texte, nous pouvons appuyer sur `Ctrl + O` pour écrire les données sur le disque (on nous demandera dans quel fichier nous voulons enregistrer cela : appuyez sur `Entrée` pour accepter la valeur par défaut suggérée de `brouillon.txt`).

Une fois notre fichier enregistré, nous pouvons utiliser `Ctrl + X` pour quitter l'éditeur et revenir au shell.

touch

Nous avons vu comment créer des fichiers texte à l'aide de l'éditeur nano. Maintenant, essayez la commande suivante :

```
$ touch mon_fichier.txt
```

1. Qu'a fait la commande `touch` ? Lorsque vous regardez votre répertoire actuel à l'aide de l'explorateur de fichiers de l'interface graphique, le fichier apparaît-il ?
2. Utilisez `ls -l` pour inspecter les fichiers. Quelle est la taille de `mon_fichier.txt` ?
3. Quand voudriez-vous créer un fichier de cette façon ?

Extension des fichiers

Vous avez peut-être remarqué que tous les fichiers sont nommés "quelque chose point extension", et que jusque là nous avons utilisé l'extension `.txt`. Il s'agit simplement d'une convention : nous pouvons appeler un fichier `brouillon` ou presque n'importe quoi d'autre. Cependant, la plupart des gens utilisent des noms en deux parties pour les aider (ainsi que leurs programmes) à distinguer les différents types de fichiers. La deuxième partie d'un tel nom est appelée l'**extension du nom de fichier** et indique le type de données que contient le fichier : `.txt` signale un fichier texte brut, `.pdf` indique un document PDF, etc.

Il ne s'agit que d'une convention, bien qu'elle soit importante. Les fichiers contiennent des octets : c'est à nous et à nos programmes d'interpréter ces octets. Nommer une image PNG d'une baleine sous le nom de `baleine.mp3` ne la transforme pas par magie en un enregistrement de chant de baleine, même si cela peut inciter le système d'exploitation à essayer de l'ouvrir avec un lecteur de musique lorsque quelqu'un double-clique dessus.

4.3 Déplacer des fichiers et des répertoires

Nous souhaitons désormais changez le nom du fichier `brouillon.txt` pour un nom plus informatif. Pour cela, utilisons la commande `mv`, qui est l'abréviation de "*move*" :

```
$ mv brouillon.txt citation.txt
```

Le premier argument indique à `mv` ce que nous "déplaçons", tandis que le second indique où cela doit aller. Dans ce cas, nous passons `brouillon.txt` et `citation.txt`, ce qui a le même effet que de renommer le fichier.

> **`mv -i`**
>
> Il faut être prudent lors de la spécification du nom du fichier cible, car `mv` écrasera silencieusement tout fichier existant portant le même nom, ce qui pourrait entraîner une perte de données. Une option supplémentaire, `mv -i` (ou `mv --interactive`), peut être utilisée pour demander une confirmation avant d'écraser.
> Notez que cela `mv` fonctionne également sur les répertoires.

Déplaçons maintenant le fichier `citation.txt` dans un répertoire `mes_citations` créé pour l'occasion.

```
mkdir mes_citations
mv citation.txt mes_citations
```

Nous utilisons `mv` à nouveau, mais cette fois nous utilisons le nom d'un répertoire comme deuxième argument pour indiquer à `mv`que nous voulons conserver le nom du fichier mais placer le fichier à un nouvel emplacement.

4.4 Copier des fichiers et des répertoires

La commande `cp` fonctionne à peu près comme `mv`, sauf qu'elle copie un fichier au lieu de le déplacer.

Réalisons une copie du fichier `mes_citations/citation.txt` :

```
$ cp mes_citations/citation.txt citation_copie.txt
```

Nous pouvons vérifier que la copie a bien été créée grâce à `ls`.

```
$ ls . mes_citations
```

Comme de nombreuses commandes Unix, `ls` peut prendre en argument plusieurs chemins à la fois.

On peut aussi copier un répertoire et tout son contenu en utilisant l' option `-r` (*recursive*), par exemple pour sauvegarder un répertoire :

```
$ cp -r mes_citations mes_citations_sauvegarde
```

4.5 Supprimer des fichiers et des répertoires

Supprimons maintenant le fichier `citation_copie.txt`, en utilisant la commande `rm` (*remove*).

```
$ rm citation_copie.txt
```

Le shell Unix n'a pas de corbeille à partir de laquelle nous pouvons récupérer les fichiers supprimés. Au lieu de cela, lorsque nous supprimons des fichiers, ils sont dissociés du système de fichiers afin que leur espace de stockage sur disque puisse être recyclé.

> Utiliser `rm` en toute sécurité
>
> Que se passe-t-il lorsque nous exécutons `rm -i mes_citations_sauvegarde/citation.txt` ? Pourquoi voudrions-nous cette protection lors de l'utilisation `rm` ?

Tentons maintenant de supprimer le répertoire `mes_citations` :

```
$ rm mes_citations
```

Nous obtenons un message d'erreur ! Cela se produit car `rm`, par défaut, ne fonctionne que sur les fichiers, pas sur les répertoires.

`rm` peut supprimer un répertoire et tout son contenu si nous utilisons l'option `-r` (*recursive*), et il le fera sans demander de confirmation :

```
$ rm -r mes_citations
```

Étant donné qu'il n'y a aucun moyen de récupérer des fichiers supprimés à l'aide du shell, `rm -r` il doit être utilisé avec beaucoup de prudence (vous pourriez envisager d'ajouter l'option interactive `rm -r -i`).

4.6 Utiliser les caractères génériques (*wildcards*) pour accéder à plusieurs fichiers à la fois

Créons maintement un nouveau répertoire contenant trois fichiers.

```
$ mkdir molecules
$ cd molecules
$ touch ethane.json methane.json propane.yml
```

`*` est un caractère générique qui correspond à **zéro ou plusieurs caractères**.

Grâce au caractère *wildcard* `*`, on peut notamment :

- lister les fichiers contenant la séquence de lettres "thane";

  ```
  $ ls *thane*
  ```

- lister les fichiers ayant l'extension `.json`.

```
$ ls *.json
```

`?` est également un caractère générique, mais il correspond à **un seul caractère**. Ainsi, `?ethane.json` correspond à `methane.json`, alors que `*ethane.json` correspond à la fois à `ethane.json` et à `methane.json`.

Les caractères génériques peuvent être utilisés en combinaison les uns avec les autres, par exemple `??thane.json` correspond à deux caractères suivis de `thane.json`.

> **Expansion des caractères génériques**
>
> Lorsque l'interpréteur de commandes rencontre un caractère générique, il l'étend pour créer la liste des noms de fichiers correspondants avant d'exécuter la commande demandée.
>
> Ainsi, les commandes comme `ls` voient les listes de noms de fichiers correspondant à ces expressions, mais pas les caractères génériques eux-mêmes. **C'est le shell, et non les autres programmes, qui s'occupe de l'expansion des caractères génériques**.

5 *Pipes* et filtres

À présent, déplaçons nous dans le répertoire `~/tp1` et créons un fichier nommé `personnes.txt` avec le contenu suivant en utilisant Nano :

```
Ada
Dennis
Kenneth
Grace
Linus
Steve
Leslie
Sergey
Alan
```

5.1 Capturer la sortie des commandes

On souhaite déterminer combien de noms (c'est-à-dire de lignes) sont présents dans le fichier.

Il est facile de répondre à cette question lorsque le fichier est de taille modeste comme c'est le cas actuellement, mais comment faire si celui-ci contenait plusieurs milliers de lignes ?

Notre premier pas vers une solution est d'exécuter la commande :

```
$ wc -l personnes.txt > compte.txt
```

La commande `wc` (*word count*) permet de déterminer **le nombre de lignes, mots ou octets présents** dans un ou plusieurs fichiers. Lisez la page du manuel dédié avec `man wc` pour en savoir plus.

Le symbole supérieur à, `>`, indique à l'interpréteur de commandes de rediriger la sortie de la commande vers un fichier au lieu de l'afficher à l'écran. Le shell créera le fichier s'il n'existe pas. Si le fichier existe, il sera écrasé silencieusement, ce qui peut entraîner une perte de données et nécessite donc une certaine prudence.

Utilisez `ls` pour confirmer que le fichier existe bien.

Nous pouvons maintenant afficher le contenu de `compte.txt` à l'écran en utilisant la commande suivante

```
$ cat compte.txt
```

La commande `cat` tire son nom de "concatenate" (concaténer), c'est-à-dire joindre ensemble, et elle affiche le contenu des fichiers l'un après l'autre. Dans ce cas, il n'y a qu'un seul fichier, donc `cat` nous montre seulement ce qu'il contient.

> ***less***
>
> `cat` présente l'inconvénient de toujours afficher le fichier entier sur votre écran.
> Plus utile en pratique est la commande `less`. Elle affiche un écran complet du fichier, puis s'arrête. Vous pouvez avancer d'un écran en appuyant sur la touche `Espace`, ou reculer d'un écran en appuyant sur `B`. Appuyez sur `Q` pour quitter.

Créons un second fichier nommé `personne-suite.txt` avec Nano, contenant quelques noms additionnels :

```
Bill
Alonzo
John
```

On peut concaténer le contenu des deux fichiers dans un nouveau fichier de la manière suivante.

```
$ cat personnes.txt personne-suite.txt > noms.txt
```

5.2 Filtrage de la sortie

Ensuite, nous allons utiliser la commande `sort` pour **trier** tous les noms.

```
$ sort noms.txt
```

La commande `sort` ne modifie pas le fichier en entrée mais affiche le résultat du tri à l'écran. Nous pouvons placer la liste triée des noms dans un autre fichier temporaire appelé `noms-tries.txt` en ajoutant `> noms-tries.txt` après la commande.

Une fois que nous avons fait cela, nous pouvons exécuter une autre commande appelée `head` pour obtenir les trois premières lignes de `noms-tries.txt` :

```
$ head -n 3 noms-tries.txt
```

tail

Alors que `head` permet d'afficher les premières lignes d'un fichier, la commande `tail` permet d'en afficher les dernières lignes.

Utilisez `tail` pour afficher les 4 dernières lignes du fichier `noms-tries.txt`.

Que signifie `>>` ?

Nous avons vu l'utilisation de `>`, mais il existe un opérateur similaire `>>` qui fonctionne légèrement différemment.

Découvrons les différences entre ces deux opérateurs en affichant quelques chaînes de caractères. On rappelle que la commande `echo` permet d'afficher des chaînes de caractères :

```
$ echo La commande echo affiche du texte
```

Testez maintenant les commandes ci-dessous pour révéler la différence entre les deux opérateurs :

```
$ echo bonjour > fichier-test-1.txt
```

et :

```
$ echo bonjour >> fichier-test-2.txt
```

Conseil : essayez d'exécuter chaque commande deux fois de suite, puis examinez les fichiers de sortie.

5.3 Passer la sortie d'une commande à une autre commande

Dans notre exemple précédent de tri des noms, nous utilisons deux fichiers intermédiaires `noms.txt` et `noms-tries.txt` pour stocker les sortie. Cette façon de travailler est déroutante car même une fois que vous avez compris ce que font `cat`, `sort` et `head`, ces fichiers intermédiaires rendent difficile la compréhension de ce qui se passe.

Nous pouvons rendre cela plus facile à comprendre en exécutant `sort` et `head` ensemble :

```
$ sort noms.txt | head -n 3
```

La barre verticale, |, entre les deux commandes s'appelle un pipe. Elle indique au shell que nous voulons utiliser la **sortie** de la commande de gauche comme **entrée** de la commande de droite.

Ainsi, le fichier `noms-tries.txt` n'est plus nécessaire.

5.4 Combiner plusieurs commandes

Rien ne nous empêche d'enchaîner les pipes consécutivement. Nous pouvons par exemple envoyer la sortie de `cat` directement à `sort`, puis la sortie résultante à `head`. Cela supprime tout besoin de fichiers intermédiaires.

Nous allons commencer par utiliser un pipe pour envoyer la sortie de `cat` à `sort` :

```
$ cat personnes-*.txt | sort
```

Nous pouvons ensuite envoyer cette sortie à travers un autre pipe, vers `head`, de sorte que le pipeline complet devienne :

```
$ cat personnes-*.txt | sort | head -n 3
```

> **i** Des outils conçus pour fonctionner ensemble
>
> Cette idée de lier des programmes ensemble est central dans la philosophie d'Unix.
>
> Au lieu de créer d'énormes programmes qui essaient de faire beaucoup de choses différentes, les programmeurs d'Unix se concentrent sur la création d'un grand nombre d'outils simples qui fonctionnent bien les uns avec les autres.
>
> Ce modèle de programmation est appelé "*pipes and filters*". Nous avons déjà vu les pipes ; un filtre est un programme comme wc ou sort qui transforme un flux d'entrée en un flux de sortie. Presque tous les outils Unix standards peuvent fonctionner de cette manière : à moins qu'on ne leur dise de faire autrement, ils lisent l'entrée standard, font quelque chose avec ce qu'ils ont lu, et écrivent sur la sortie standard.
>
> La clé est que tout programme qui lit des lignes de texte à partir de l'entrée standard et écrit des lignes de texte sur la sortie standard peut être combiné avec tout autre programme qui se comporte de la même manière. Vous pouvez et devez écrire vos programmes de cette façon afin que vous et d'autres personnes puissiez combiner ces programmes avec des *pipes* pour multiplier leur puissance.

6 Historique

La commande `history` vous permet de **lister toutes les commandes** que vous avez entrées depuis le début de la session. Essayons :

```
history
```

Par défaut, tout l'historique est affiché sur le terminal. Pour naviguer plus facilement nous pouvons envoyer la sortie de l'historique dans `less` :

```
history | less
```

Un des intérêts de l'historique est qu'il est possible de retrouver n'importe quelle précédente commande :

- `$ !3` donne la troisième commande de la session;
- `$ !-1` donne la dernière commande;
- `$ !echo` permet de chercher une commande commençant par `echo`.

On peut également chercher dans l'historique grâce à la combinaison de touches `CTRL` + `R` suivi de la recherche à effectuer puis d'`Entrée`.

En appuyant successivement sur `CTRL` + `R`, on peut remonter dans l'historique des commandes.

Cherchez la commande `head` précédemment executée.

Enfin, supprimons l'historique :

```
history -c
```

7 Variables

Le shell est à la fois un *REPL (read–eval–print loop)* et un langage de programmation. Ainsi, il est possible de déclarer et d'initialiser des variables avec le shell.

```
$ nom_domaine="https://fr.wikipedia.org"
$ chemin="/wiki/Alan_Turing"
```

On peut ensuite utiliser ces variables dans d'autres déclaration ou directement comme entrées pour des commandes.

```
$ url="$nom_domaine$chemin"
$ echo $url
```

8 Boucles

Les boucles sont une construction de programmation qui nous permet de **répéter une commande** ou un ensemble de commandes pour chaque élément d'une liste. En tant que telles, elles sont essentielles pour améliorer la productivité grâce à l'automatisation.

La syntaxe d'une boucle est la suivante :

```
for thing in list_of_things
do
    operation_using $thing # L'indentation dans la boucle n'est pas
     ↪  nécessaire, mais aide à la lisibilité
done
```

Supposons que nous souhaitions lire les lignes de notre fichier `personnes.txt` et afficher une phrase pour chacun des noms. Utilisons une boucle `for` pour cela.

```
for nom in $(cat personnes.txt)
do
  echo "$nom est un(e) informaticien(ne) célèbre."
done
```

`$(cat personnes.txt)` substitue le résultat de la commande `cat personnes.txt` dans la commande principale contenant la boucle for. `$nom` est une variable qui prend pour valeurs successives le contenu des lignes du fichier.

9 Scripts shell

Nous sommes enfin prêts à voir ce qui fait du shell un environnement de programmation si puissant. Nous allons prendre les commandes que nous répétons fréquemment et les enregistrer dans un fichier afin de pouvoir réexécuter toutes ces opérations plus tard en tapant une seule commande. Un ensemble de commandes enregistrées dans un fichier est généralement appelé un **script shell**.

Non seulement l'écriture de scripts shell rendra votre travail plus rapide (vous n'aurez pas à retaper les mêmes commandes encore et encore) mais elle le rendra également plus précis (moins de risques de fautes de frappe) et plus reproductible.

Commençons par ouvrir un nouveau fichier `script.sh` qui deviendra notre script shell.

```
$ nano script.sh
```

Une fois dans l'éditeur Nano, écrivons un script qui lit les lignes de notre fichier `personnes.txt`, tri les lignes par ordre alphabétique et crée un fichier portant le nom des trois premières lignes.

```
selection=$(sort personnes.txt | head -n 3)

for nom in $selection
do
  touch "$nom.txt"
done
```

Vérifiez que le répertoire de travail contient maintenant un fichier appelé `script.sh`.

Une fois que nous avons enregistré le fichier, nous pouvons demander au shell d'exécuter les commandes qu'il contient. Notre shell s'appelle `bash`, nous exécutons donc la commande suivante :

```
$ bash script.sh
```

Et si nous voulons réaliser cette opération sur un fichier arbitraire ? Nous pourrions éditer `script.sh` à chaque fois pour changer le nom du fichier, mais cela serait fastidieux. Au lieu de cela, éditons `script.sh` et rendons-le plus polyvalent :

```
$ nano script.sh
```

Maintenant, dans Nano, remplaçons le texte `personnes.txt` par la variable spéciale appelée `$1` :

```
selection=$(sort "$1" | head -n 3)

for nom in $selection
do
  touch "$nom.txt"
done
```

Dans un script shell, `$1` signifie "le premier nom de fichier (ou autre argument) sur la ligne de commande". Nous pouvons maintenant exécuter notre script comme ceci :

```
$ bash script.sh personnes.txt
```

ou sur un fichier différent comme ceci :

```
$ bash script.sh <nom_du_fichier>
```

Actuellement, nous devons modifier `script.sh` chaque fois que nous voulons ajuster le nombre de fichiers qui est créé. Corrigeons cela en configurant notre script pour qu'il utilise plutôt deux arguments de ligne de commande.

Après le premier argument de ligne de commande (`$1`), chaque argument supplémentaire que nous fournissons sera accessible via les variables spéciales `$2`, `$3`, etc., qui se réfèrent aux deuxième et troisième arguments de ligne de commande, respectivement.

```
$ nano script.sh
```

```
selection=$(sort "$1" | head -n $2)

for nom in $selection
do
  touch "$nom.txt"
```

```
done
```

Nous devons également mettre `$1` entre guillemets pour gérer le cas où le nom de fichier contiendrait un espace.

Nous pouvons maintenant exécuter :

```
$ bash script.sh personnes.txt 4
```

Cela fonctionne, mais il faudra peut-être un moment à la prochaine personne qui lira `script.sh` pour comprendre ce qu'il fait. Nous pouvons améliorer notre script en ajoutant quelques **commentaires** en haut :

```
$ nano script.sh
```

```
# Tri les lignes du fichier par ordre alphabétique
# puis sélectionne les premières lignes.
selection=$(sort "$1" | head -n $2)

# Crée un fichier vide pour chaque nom dans la sélection.
for nom in $selection
do
  touch "$nom.txt"
done
```

Un commentaire commence par le caractère `#` et se termine à la fin de la ligne.

Que faire si nous voulons traiter de nombreux fichiers dans un seul pipeline ? Par exemple, si nous voulons prendre en compte tous les fichiers correspondant au motif `personnes-*.txt`.

Nous souhaitons pouvoir lancer la commande suivante pour créer trois fichiers à partir des noms présents dans les fichiers correspondants à `personnes-*.txt`.

```
$ bash script.sh personnes-*.txt
```

Nous ne pouvons pas utiliser `$1`, `$2`, etc. car nous ne savons pas combien de fichiers existent. Utilisons plutôt la variable spéciale `$@`, qui signifie "tous les arguments de la ligne de commande du script shell" :

```
selection=$(sort "$@" | head -n 3)

for nom in $selection
do
  touch "$nom.txt"
done
```

Vérifiez maintenant que le script fonctionne comme prévu.

Finalement, ajoutons une ligne au début de notre script.

```
#!/bin/bash

selection=$(sort "$@" | head -n 3)

for nom in $selection
do
  touch "$nom.txt"
done
```

L'en-tête qui commence par `#!` s'appelle le **shebang**, il indique l'emplacement du programme interpréteur qui doit être utilisé pour exécuter le script. En l'occurrence, il s'agit du shell Bash.

Ensuite, rendons le script exécutable avec la commande suivante (on verra la commande `chmod` plus en détails dans un des TP suivants):

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

Nous pouvons maintenant exécuter le script sans avoir à appeler `bash` explicitement.

```
./script.sh
```

Exercice

Écrivez un script nommé `supprimer.sh` qui :
- prend un nombre arbitraire de noms de fichiers en argument;
- enregistre chacun de ces noms de fichiers dans un fichier temporaire;
- supprime chacun de ces fichiers;
- affiche en ordre alphabétique le nom de chacun des fichiers supprimés, précédé de la mention "Fichiers supprimés".

Par exemple,

```
$ supprimer.sh B.txt C.txt A.txt
Fichiers supprimés :
- A.txt
- B.txt
- C.txt
```

10 Pour aller plus loin

- RTFM. Les man pages de Linux contiennent énormément d'information utiles pour maîtriser le shell et votre système d'exploitation. Prenez l'habitude de les consulter !

- Suivez un tutoriel en ligne tel que learnshell.org ou linuxcommand.org.
- Utilisez ShellCheck pour vérifier que vos scripts shell sont corrects et ne comportent pas de vulnérabilités.

À retenir

- `echo` affiche des chaînes de caractères à l'écran.
- `pwd` affiche le répertoire de travail actuel de l'utilisateur.
- `ls [path]` affiche la liste d'un fichier ou d'un répertoire spécifique ; `ls` seul affiche la liste du répertoire de travail actuel.
- `cd [path]` change le répertoire de travail actuel.
- `.` seul signifie "le répertoire actuel" ; `..` signifie "le répertoire au-dessus du répertoire actuel".

- `cp [old] [new]` copie un fichier.
- `mkdir [chemin]` crée un nouveau répertoire.
- `mv [old] [new]` déplace (renomme) un fichier ou un répertoire.
- `rm [chemin]` supprime (efface) un fichier.
- `*` correspond à zéro ou plusieurs caractères dans un nom de fichier, ainsi `*.txt` correspond à tous les fichiers se terminant par `.txt`.
- `?` correspond à un caractère unique dans un nom de fichier, ainsi `?.txt` correspond à `a.txt` mais pas à `any.txt`.

- `wc` compte les lignes, les mots et les caractères dans ses entrées.
- `cat` affiche le contenu de ses entrées.
- `sort` trie ses entrées.
- `head` affiche les 10 premières lignes de son entrée.
- `tail` affiche les 10 dernières lignes de son entrée.
- `command > [file]` redirige la sortie d'une commande vers un fichier (en écrasant tout contenu existant).
- `command >> [file]` ajoute la sortie d'une commande à un fichier.
- `[first] | [second]` est un pipeline : la sortie de la première commande est utilisée comme entrée de la seconde.

- `history` permet de lister toutes les commandes depuis le début de la session.
- Une boucle `for` répète les commandes une fois pour chaque chose dans une liste.
- Utilisez `$name` pour développer une variable (c'est-à-dire obtenir sa valeur). `${name}` peut également être utilisé.

- `bash [nom du fichier]` exécute les commandes enregistrées dans un fichier.
- `$@` fait référence à tous les arguments de ligne de commande d'un script shell.
- `$1`, `$2`, etc., font référence au premier argument de ligne de commande, au deuxième argument de ligne de commande, etc.
- `#!/bin/bash` s'appelle le shebang et indique l'emplacement du shell qui doit être utilisé pour exécuter le script.