

Gestion d'identité en ligne

Chapitre 3 Scripter avec le shell



Pablo Rauzy <pr@up8.edu>
pablorausy.fr/teaching/idweb

Scripter avec le shell

- ▶ Un *shell* est un *interpréteur de commandes*.
 - Il présente une *invite de commande* (un “prompt”) à son utilisatrice (dans laquelle on voit généralement son login, le nom d’hôte de la machine, et le répertoire de travail courant).
 - Une fois une *ligne de commande* entrée par l’utilisatrice, le shell la lit et l’exécute avant de rendre la main.
- ▶ Il ne fait pas confondre le shell avec la *console* (qu’on appelle également le *terminal*) qui est le logiciel dans lequel le shell s’exécute dans un environnement graphique.

Historique

- ▶ Le premier shell date de 1971, il s'agit du Thompson shell (du nom de son auteur Ken Thompson, un des co-créateur du système UNIX avec Denis Ritchie).
- ▶ Il est plus tard remplacé par le Bourne shell (du nom de Stephen Bourne) en 1977.
- ▶ En 1988, Brian Fox écrit Bash (*Bourne again shell*), une version libre et améliorée du Bourne shell, pour le projet GNU.
 - C'est toujours aujourd'hui le shell par défaut sur Linux, même si des projets un peu plus récents comme zsh lui sont parfois préférés.

- ▶ Une *commande* peut être intégrée au shell ou être un programme externe qui sera exécuté par le shell.
- ▶ Les programmes exécutés par le shell héritent de son *environnement*, de son *répertoire de travail*, et par défaut la console dans laquelle le shell s'exécute est utilisé pour leur *entrée standard*, *sortie standard*, et *sortie d'erreur*.

Rappels de quelques commandes

► On a déjà vu plusieurs commandes de base lors du TP Console :

- **date** : afficher la date courante ;
- **whoami** : afficher le login de l'utilisateur ;
- **hostname** : afficher le nom d'hôte de la machine ;
- **echo** : afficher ses arguments ;
- **pwd** : afficher le répertoire de travail ;
- **cd** : changer le répertoire de travail ;
- **ls** : lister des fichiers ;
- **mv** : déplacer / renommer des fichiers ;
- **cp** : copier des fichiers ;
- **rm** : supprimer des fichiers ;
- **mkdir** : créer un répertoire ;
- **rmdir** : supprimer un répertoire ;
- **man** : consulter une page de manuel ;
- **exit** : quitter le shell.

► N'hésitez pas à (re)consulter leur page de manuel.

Quelques commandes supplémentaires

- ▶ Voici quelques commandes supplémentaires permettant de manipuler le texte qui leur arrive sur leur entrée standard (ou qui est lu dans les fichiers dont les chemins leurs sont donnés en argument) :
 - **cat** : copier ;
 - **head** : afficher les premières lignes ;
 - **tail** : afficher les dernières lignes ;
 - **tr** : transformer certains caractères en d'autres ;
 - **cut** : découper les lignes ;
 - **sort** : trier les lignes ;
 - **uniq** : dédupliquer les lignes consécutives identiques ;
 - **nl** : numérotter les lignes ;
 - **grep** : sélectionner des lignes suivant un *motif* ;
 - **sed** : éditer les lignes ;
 - **wc** : compter les lignes / mots / caractères.
- ▶ N'hésitez pas à consulter leur page de manuel.

Redirections

- ▶ Il est possible d'utiliser des *redirections* sur une ligne de commande pour changer les entrées/sorties standards d'un programme pour des fichiers :
 - `> f` redirige la sortie standard dans le fichier `f` (remis à zéro) ;
 - `>> f` redirige la sortie standard dans le fichier `f` (à la suite) ;
 - `2> f` redirige la sortie d'erreur dans le fichier `f` (remis à zéro) ;
 - `2>> f` redirige la sortie standard dans le fichier `f` (à la suite) ;
 - `< f` redirige l'entrée standard depuis le fichier `f`.
- ▶ Exemple :
 - ```
$ echo 1312 > f
$ cat f
1312
$ tr 123 ABC < f
ACAB
```

- ▶ Souvent on veut rediriger directement la sortie standard d'un programme vers l'entrée standard d'un autre, sans devoir passer par un fichier intermédiaire.
- ▶ On peut faire cela avec un *tuyau* ("*pipe*" en anglais), noté avec le caractère `|`.
  - `a | b` exécute le programme `a` et le programme `b` en redirigeant la sortie standard de `a` dans l'entrée standard de `b`.
- ▶ Exemple :
  - `$ echo 1312 | tr 123 ABC`  
ACAB

- ▶ Avant l'exécution d'une ligne de commande, le shell procède à son *expansion*.
- ▶ Durant cette étape, certaines syntaxe sont remplacées par des valeurs contextuelles :
  - `~` est remplacé par le chemin absolu du répertoire maison ;
  - `*` est remplacé par la liste des fichiers et dossiers du répertoire de travail ;
  - `*.txt` est remplacé par la liste des fichiers dont le nom termine par `.txt` ;
  - `abc-*` est remplacé par la liste des fichiers dont le nom commence par `abc-` ;
  - `abc-*.txt` est remplacé par la liste des fichiers qui dont le nom commence par `abc-` et termine par `.txt` ;
  - `?-*.txt` est remplacé par la liste des fichiers qui dont le nom commence par exactement un caractère suivi d'un tiret et termine par `.txt` ;
  - `*.{txt,md}` est remplacé par la liste des fichiers qui dont le nom termine par `.txt` ou `.md` ;
- ▶ À part ces expansions liées principalement aux fichiers, il y a aussi les *variables*.

- ▶ On peut utiliser des *variables* dans le shell pour réutiliser des valeurs d'une commande à une autre :
  - `x="abracadabra"` affecte la valeur `abracadabra` à la variable `x` ;
  - `$x` dans une ligne de commande sera remplacé par le contenu de la variable `x` lors de l'expansion.
- ▶ On peut délimiter le nom d'une variable avec des accolades :
  - `$ echo $xntesque`  
  
`$ echo ${x}ntesque`  
`abracadabrantesque.`
- ▶ Attention à l'absence d'espace autour du `=` pour désambiguer avec l'appel d'une commande.

# Variables numériques

► Par défaut, les variables ne contiennent que du texte ; on peut utiliser la syntaxe `$((...))` pour les opérations arithmétiques :

- `$ x=1+2`  
`$ echo $x`  
1+2
- `$ x=$((1+2))`  
`$ echo $x`  
3

## L'expansion des variables

► Le shell permet d'effectuer certaines opérations pratiques lors de l'expansion :

- `$ echo ${x%ra}`  
abracadab
- `$ echo ${x%r*}`  
abracadab
- `$ echo ${x%%r*}`  
ab
- `$ echo ${x#ab}`  
racadabra
- `$ echo ${x#*b}`  
racadabra
- `$ echo ${x##*b}`  
ra
- `$ echo ${x:4}`  
cadabra
- `$ echo ${x:4:3}`  
cad
- `$ echo ${x/a/Z}`  
Zbracadabra
- `$ echo ${x//a/Z}`  
ZbrZcZdZbrZ
- `$ echo ${x^}`  
Abracadabra
- `$ echo ${x^^}`  
ABRACADABRA
- `$ echo ${maj,,}` # supposons `maj=${x^^}`  
aBRACADABRA
- `$ echo ${maj,,,}`  
abracadabra
- `$ echo ${x:-valeur par défaut}`  
abracadabra
- `$ echo ${y:-valeur par défaut}`  
valeur par défaut
- `$ echo ${x:?message d'erreur}`  
abracadabra
- `$ echo ${y:?message d'erreur}`  
bash: y: message d'erreur

- On peut utiliser la sortie standard d'une ligne de commande comme valeur d'une pseudo-variable avec la syntaxe `$(...)` :
- `x=$(echo 1312 | tr 123 ABC)` affecte la valeur **ACAB** à la variable `x`.

## Conditions

- ▶ La variable spéciale `$?` contient le code de retour de la dernière commande.
  - Le code `0` signifie OK, toute autre valeur est un code d'erreur.
  - La commande `true` retourne `0`, la commande `false` retourne `1`.
- ▶ La syntaxe `if ...; then ...; else ...; fi` permet de faire un branchement conditionnel.
- ▶ Le code de retour de la commande après le `if` est utilisé pour déterminer le branchement.
- ▶ On utilise souvent la commande `test` pour cela, ou son raccourci `[]`.
  - Exemple :

---

```
1 if test -f article.txt; then
2 head -1 article.txt
3 else
4 echo "Le fichier article.txt n'existe pas." >&2
5 fi
```

---

# Boucles

- De la même manière que pour un branchement, on peut itérer avec la syntaxe **while ...; do ...; done**.

- Exemple :

---

```
1 x=3
2 while test ! "$x" = "0"; do
3 echo $x
4 x=$((x-1))
5 done
```

---

- La syntaxe **for var in ...; do ...; done** permet d'itérer sur une liste (typiquement de fichier mais pas forcément).

- Exemple :

---

```
1 for f in *.txt; do
2 head -1 $f
3 done
```

---

# Fonctions

- ▶ On peut définir des fonctions avec le mot-clef **function** suivi de son nom suivi de son corp entre accolades.
- ▶ Le corp d'une fonction est une suite de commandes qui seront exécutées lors de l'utilisation de la fonction comme une commande.
  - Exemple :

---

```
1 function headline_all_caps {
2 first_line=$(head -1 article.txt)
3 echo ${first_line^^}
4 }
```

---

# Arguments

- ▶ Comme n'importe quelle commande, une fonction peut prendre des arguments et les utiliser.
- ▶ Les arguments n'ont pas de nom, ils sont numérotés :
  - \$1 est le premier argument, \$2 est le deuxième argument, ...
  - Exemple :

---

```
1 function headline_all_caps {
2 article=${1:?missing argument}
3 first_line=$(head -1 $article)
4 echo ${first_line^^}
5 }
```

---

- ▶ On peut connaître le nombre d'argument avec \$# et récupérer la liste avec \$@.
  - Exemple :

---

```
1 function list_args {
2 echo "$# arguments reçus :"
3 for arg in "$@"; do
4 echo $arg
5 done | nl
6 }
```

---

## Astuce d'utilisation : l'historique

- ▶ Il y a parfois des commandes un peu complexes qui demandent plusieurs essais pour être mises au point.
  - On peut remonter dans son historique de commande avec la touche flèche haut pour éditer les commandes précédentes plutôt que de repartir de zéro.
- ▶ L'historique du shell est préservé entre les sessions.
  - On peut le consulter avec la commande **history**.
  - On peut relancer la dernière commande qui commence par **xxx** avec **!xxx**.
  - On peut faire une recherche interactive dans l'historique en tapant **ctrl+r** (r comme "*reverse search*").

## Astuce d'utilisation : l'historique

- ▶ Il y a parfois des commandes un peu complexes qui demandent plusieurs essais pour être mises au point.
  - On peut remonter dans son historique de commande avec la touche flèche haut pour éditer les commandes précédentes plutôt que de repartir de zéro.
- ▶ L'historique du shell est préservé entre les sessions.
  - On peut le consulter avec la commande **history**.
  - On peut relancer la dernière commande qui commence par **xxx** avec **!xxx**.
  - On peut faire une recherche interactive dans l'historique en tapant **ctrl+r** (r comme "*reverse search*").
- ▶ Cela dit, quand on a des commandes complexes qu'on appelle souvent... c'est mieux d'en faire un *script*.

- ▶ Un *script shell* est un fichier texte qui contient une suite de *commandes* qui seront exécutées une par une par le shell (typiquement Bash) lors de l'exécution du script.
  - Par convention, on utilise souvent l'extension `.sh` pour ces fichiers.
- ▶ Pour exécuter un script, deux possibilités :
  - passer le fichier en argument à son interpréteur (`bash monscript.sh`);
  - rendre le fichier exécutable et l'appeler directement (`./monscript.sh`)  
dans ce cas, le fichier doit contenir un "shebang" : sa toute première ligne doit être `#!` suivi du chemin vers son interpréteur.

# Arguments

- ▶ Exactement comme dans une fonction.
- ▶ En plus, la variable `$0` contient le nom avec lequel le script a été appelé.

## Le PATH

- ▶ Quand vous taper un nom de commande, si ce n'est pas une commande intégrée du shell, celui-ci va chercher un programme du nom donné à différents endroits définis par la *variable d'environnement* **PATH**.
- ▶ Cette variable contient une liste de chemins vers des répertoires séparés par le caractère **:**.
  - `$ echo $PATH`  
`/usr/local/bin:/usr/bin:/bin`
- ▶ Dans votre fichier `~/.bashrc`, vous pouvez configurer votre **PATH** pour y ajouter un répertoire contenant vos propres scripts :
  - `export PATH="$HOME/local/bin:$PATH"`