

0.1 Initiation à Bash

Sous linux vous entendrez souvent parler de terminal, en mode graphique sous kde par exemple, cela peut être assimilé à la petite tv noire à gauche de la barre des tâches. En réalité à partir du terminal, l'utilisateur peut dialoguer avec le système via un interpréteur de commandes shell. Il existe plusieurs variétés de shell, celui par défaut sous linux est Bash (bourne again shell), il est très pratique et permet de réaliser des scripts qui vous faciliteront la vie. Nous nous intéressons ici à l'aspect langage de programmation de Bash, l'aspect "lignes de commandes tapées au prompt" est traité dans Commandes utiles¹

0.1.1 1. Le premier script :

Un script Bash est un fichier texte, qui commence toujours par `#!/bin/bash` ou `#!/bin/sh`, et qui doit être rendu exécutable pour pouvoir s'exécuter ;-). Dans ce fichier on définit les commandes selon l'ordre dans lequel on souhaite qu'elles s'exécutent. **Exemple :** Ouvrez votre éditeur de texte préféré et copiez-y les lignes ci-dessous. que vous enregistrerez dans le fichier `mon_script.sh` :

```
#!/bin/bash
echo "Bonjour"
```

Rendons-le exécutable par :

```
chmod u+x mon_script.sh
```

Et pour l'exécuter ce sera :

```
/le_chemin_vers/mon_script.sh
```

Remarque : pour inclure des commentaires dans votre script - ce qui est indispensable dans tout programme/script qui se respecte - vous devez faire précéder la ligne du symbole #.

0.1.2 2. Variables et paramètres :

On peut définir des variables locales dans le script par `MA_VARIABLE='moi'`, et pour les manipuler il faut les préfixer du symbole \$, exemple :

```
#!/bin/bash
message=' Bonjour '
echo $message
```

Voici la syntaxe pour initialiser et pour utiliser un tableau en Bash :

```
mon_tableau[index]=variable # instantiation
${mon_tableau[index]} # appel d'un élément
${mon_tableau[*]} # appel de l'ensemble du tableau
```

L'index du tableau doit être un nombre !! Si vous mettez une chaîne de caractère, vous pourriez avoir l'impression que cela a marché, mais ce n'est pas vrai. En effet, l'index est automatiquement interprété comme une expression arithmétique de valeur positive ou nulle, et mettre autre chose qu'un nombre en tant qu'index revient à utiliser la première case du tableau (d'index '0'). **Exemple :**

¹<http://www.trustonme.net/didactels/130.html>

```
#!/bin/bash
nom[0]='Bonjour'
nom[1]='Monsieur'
#affiche 'Bonjour'
echo ${nom[0]}
#affiche 'Bonjour Monsieur'
echo ${nom[*]}
```

Les scripts Bash sont dotés de certains paramètres spéciaux. On peut récupérer leurs valeurs dans le script par :

- **\$0** correspond au nom du script lancé, **\$1** correspond au premier argument, **\$2** au deuxième argument ...
- **\$#** a pour valeur le nombre total de paramètres (**\$0** compris) passés au script
- **\$?** a pour valeur le code de retour de la dernière commande exécutée dans le shell
- **\$@** pour récupérer la concaténation de tous les paramètres, en les séparant par un espace

0.1.3 3. Les chaînes de caractères :

- **' '** délimitent une chaîne de caractères. A l'intérieur, tous les métacaractères perdent leur signification.
- **" "** délimitent une chaîne de caractères. A l'intérieur, tous les métacaractères perdent leur signification, à l'exception des métacaractères **'**, **\$** et ****
- **** protège le caractère qui suit, que ce soit un caractère normal ou un métacaractère du shell (sauf à l'intérieur d'une chaîne délimitée par des **'**).

Exemple :

```
#!/bin/bash
nom='Dupont'
echo 'Mr Dupont'
echo "Mr $nom"
echo "Mr \"$nom\""
```

Pour connaître la taille(longueur) d'une chaîne de caractères, utilisez la syntaxe suivante :

```
${#ma_variable}
```

Exemple :

```
nom='Dupont'
#affiche 6
echo ${#nom}
```

0.1.4 4. Les Conditions :

Cela peut paraître surprenant, mais il n'existe pas de type Booléen(**True/False** ou encore **0/1**) en Bash, ni d'autres types d'ailleurs, car Bash est un langage non-typé. Ceci étant, les conditions, sous leur forme booléenne, n'existent donc pas à proprement parler. Dans ce document, nous avons tout de même utilisé le mot 'condition' pour ne pas perturber le lecteur, mais sachez que ce sont en fait des 'commandes' qui jouent ce rôle. Je m'explique,

Bash considère qu'une commande s'est bien déroulée lorsqu'elle reçoit comme valeur de sortie '0' (zéro), toute autre valeur correspondant à une exécution non réussie (entièrement ou partiellement). Donc, si, dans la structure de commande, la commande qui joue le rôle de condition renvoie '0', alors cela correspond à un True dans le type Booléen, et vice-versa. Toute condition est analysée par la commande **test**(interne à Bash), sans pour autant qu'elle soit obligatoirement écrite. Voici trois exemples pour expliciter ce qui vient d'être dit : **Exemples :**

```
if test -f mon_fichier; then
echo "le fichier existe"
fi

if [ -f mon_fichier ]; then
echo "le fichier existe"
fi

if [ ./ma_commande ]; then
echo "la commande s'est terminée avec succès"
fi
```

Quelques comparaisons utiles :

- f teste l'existence d'un fichier
- d teste l'existence d'un répertoire
- x teste si le fichier existe et est exécutable
- r teste si le fichier existe et est ouvert en lecture
- w teste si le fichier existe et ouvert en écriture
- s teste si le fichier existe et a une taille supérieur à 0 octet

Pour les comparaisons arithmétiques (voir la deuxième partie de ce tutoriel pour plus d'informations), il peut être plus agréable d'utiliser une autre syntaxe :

```
if (( $nombre_1 == $nombre_2 )); then
echo "il y a égalité"
fi
```

Remarque : il faut TOUJOURS laisser un espace entre les crochets "[]" et entre chaque argument ; sinon vous obtiendrez des messages d'erreurs tels que "syntax error in expression" ou "syntax error near unexpected token".

0.1.5 5. Les structures de contrôles :

Comme dans tout langage de programmation, avec Bash il est possible de casser la séquence d'exécution des commandes, en utilisant les structures de contrôles habituelles. Voici la syntaxe à utiliser.

0.1.6 5.1 IF :

```
if condition_1 ; then
commandes1
elif condition_2 ; then
```

```
commandes2
else
commandes3
fi
```

0.1.7 5.2 FOR :

```
for variable in liste_de_cas do
commandes
done

for (( expr1 ; expr2 ; expr3 )) do
commandes
done
```

0.1.8 5.3 WHILE :

```
while condition do
commandes
done
```

0.1.9 5.4 UNTIL :

```
until condition do
commandes
done
```

0.1.10 5.5 CASE :

```
case variable in
cas1)
commande1
;;
cas2)
commande2
;;
*)
commande_par_defaut
;;
esac

( cas1 et cas2 sont des chaînes de caractères. )
```

0.1.11 6. Les fonctions :

On peut définir des fonctions en Bash, cela peut être très utile pour structurer ses programmes. La syntaxe est la suivante :

```
ma_fonction() {  
  corps  
}
```

Pour appeler la fonction dans le script ce sera :

```
ma_fonction param1 param2 param3 ...
```

A l'intérieur du corps de la fonction, les paramètres sont disponibles dans les variables \$0, \$1, \$2 ... Le nombre de paramètres étant toujours \$#. Une variable utilisée à l'intérieur d'une fonction est globale ! Pour éviter ce phénomène, il faut rajouter **local** à la déclaration de la variable :

```
local MA_VARIABLE
```

Exemple :

```
#!/bin/bash  
valeur=0  
calcul() {  
  somme=$(( $1 + $2 ))  
  local valeur=$somme  
}  
calcul 5 9  
#retourne "14"  
echo $somme  
#retourne "0"  
echo $valeur
```

Une fonction se termine soit après l'exécution de la dernière commande située avant l'accolade fermante, auquel cas le code de retour est celui de cette dernière commande, soit après exécution d'une commande **return n**, avec "n" un nombre compris entre 1 et 255.

Remarque : la déclaration de la fonction dans le script doit ABSOLUMENT se faire avant son appel, autrement, vous obtiendrez un "command not found". Pour utiliser vos fonctions depuis d'autres scripts : Exemple fichier mesfonctions.inc.sh

```
ma_fonction() {  
  mavariable1=$1  
  mavariable2=$2  
}
```

Exemple d'inclusion dans un autre script sh : **Remarque** : il faut que le sh inclus soit dans le même répertoire que le sh appelant.

```
# On inclut le fichier de fonctions  
. ./mesfonctions.inc.sh  
# On definit les parametres à passer à la fonction  
mon_parametre1="mon premier parametre"
```

```
mon_parametre2="mon deuxieme parametre"  
# On appelle la fonction avec ces parametres  
ma_fonction mon_parametre1 mon_parametre2  
#suite du code...
```