PROGRAMMATION EN BASH

Formation Linux

Variables • Scripts • Conditions • Boucles • Fonctions

PROGRAMMATION EN BASH

Objectifs

- Comprendre le système des scripts de démarrage
- Écrire des scripts simples

Plan du cours

1. Les variables en bash
2. Écrire des scripts
3. Conditions
4. Boucles
5. Fonctions

PROGRAMMATION EN BASH

1. Les variables en bash

Variables scalaires

- Pas d'espaces autour du =
- Pas de $ pour la déclaration
- Utilisation du $ pour utiliser la variable
  - Devant le nom
  - Le nom de la variable peut être mis entre { }
  - ${comme_ca}

```
[user@linux ~]$ os="GNU/Linux"
[user@linux ~]$ echo "This OS is $os"
This OS is GNU/Linux
```

Les tableaux

- Variable scalaire
- Utilisation des [ ]
  - Pour écrire dedans
  - Pour accéder aux valeurs
- Les tableaux associatifs dans Bash 4
  - Indexés par des chaînes de caractères

```
[user@linux ~]$ array[0]="dummy"
[user@linux ~]$ array[1]="foo"
[user@linux ~]$ array[2]="bar"
[user@linux ~]$ echo ${array[1]}
foo
```

2. Exécuter des commandes

- Back quote `
- Parenthèses
- Les quotes : " ' `
- Commentaires : #

```
[user@linux ~]$ var=`uname -sr`
[user@linux ~]$ var2=$(uname -sr)
[user@linux ~]$ echo $var $var2
Linux 2.6.25-2-686 Linux 2.6.25-2-686
```

```
# this is a comment
var="ls"
echo "var = $var" # print : var = ls
echo 'var = $var' # print : var = $var
echo `"$var"` # print : ls result

- Les guillemets doubles "  " interprètent les variables et remplacent $var par sa valeur.
- Les guillemets simples '  ' n'interprètent pas les variables : le texte est affiché exactement tel qu'il est écrit.
- Les accents graves `  `, aussi appelés backticks, exécutent la commande placée entre eux et affichent son résultat.
```

Écrire sur l'écran

- echo -e CHAÎNE
  - -e : active les séquences d'échappement \
- read [-p PHRASE] [variable]

```
[user@linux ~]$ read -p "What is the OS Name ? " os
What is the OS Name ? OpenBSD
[user@linux ~]$ echo "This OS is $os"
This OS is OpenBSD
```

```
read       # store the input into REPLY
read VAR   # store the input into VAR
```

3. Écrire un script

Le shebang

- Rappelez-vous le #!shebang
  - Première ligne du fichier
  - Permet de choisir le bon interpréteur
  - Bash par défaut si pas de shebang

```
#!/bin/bash

#!/usr/bin/perl

#!/usr/bin/python
```

Rendre un script exécutable

```
[user@linux ~]$ chmod +x script.sh
```

Lancer le script

```
./script.sh
It starts the program specified in the shebang to execute the script. Only exported variables are available
from within the script.

source script.sh
. script.sh
The script is executed from within the running shell. The script has read/write access on all the variables
defined by the shell.
```

Les arguments en ligne de commande

- Accès aux arguments

```
#!/bin/bash
# Argv script example
#
echo $0 $1 $2
```

- Facilités
  - $* : tous les arguments en tant que chaîne
  - $@ : sous forme de tableau

```
[user@linux ~]$ ./script labo linux
./script labo linux
```

PROGRAMMATION EN BAS H

4. Les codes de sortie et les tests

Les codes de sortie

- Instruction exit
- Convention sur les Unix
  - 0 : tout va bien
  - != 0 : échec
- L'instruction trap permet de récupérer les signaux
- echo $?

```
trap "echo You've pressed Ctrl-C" SIGINT
```

Les tests

- Permet de tester des booléens
  - [ ] : pour une condition
  - [[ ]] : pour plusieurs
  - Retourne 0 si vrai
  - Manuel : man test

```
[user@linux ~]$ test 1 -gt 0; echo $?
0
[user@linux ~]$ [ 1 -lt 0 ]; echo $?
1
[user@linux ~]$ [[ 1 -lt 0 && 3 -gt 2 ]]; echo $?
1
```

Les expressions mathématiques

- expr
  - Écrit le résultat sur STDOUT
  - Division entière (euclidienne)
- let
  - Dans bash
  - Pas de STDOUT
  - Fonctionne avec des variables
- Expansion arithmétique
  - $((3+2))

```
[user@linux ~]$ expr 4 + 3
7
[user@linux ~]$ i=0; let i++; echo $i
1
[user@linux ~]$ echo "3+2=$((3+2))"
3+2=5
```

5. Les conditions

If, then

```
if [ 'a' = 'a' ]; then echo "true"; fi

if [ 'a' = 'a' ]; then
    echo "true"
fi
```

If, then, else

```
if [ 'a' = 'a' ]; then echo "true";
else echo "false"; fi

if [ 'a' = 'a' ]; then
    echo "true"
else
    echo "false"
fi
```

If, then, elif

```
if [ 'a' = 'a' ]; then echo "true";
elif [ 'b' = 'b' ]; then echo "false"; fi

if [ 'a' = 'a' ]; then
    echo "true"
elif [ 'b' = 'b' ]; then
    echo "false"
fi
```

Case

```
case "$OS" in
    Unix)
        echo 'Your operating system is Unix'
        ;;
    [Bb][Ss][Dd])
        echo 'your operating system is a BSD'
        ;;
    Lin*)
        echo 'your operating system is GNU/Linux'
        ;;
    *)
        echo 'your operating system is unknown'
        ;;
esac
```

6. Les boucles

Tant que : while

- while condition ; do lot of things ; done
  - Entre dans la boucle si la condition est vraie

```
i=0
while [ $i -lt 3 ]; do
    echo $i
    let i+=1
done
```

À moins que : until

- until condition ; do lot of things ; done
  - Contraire du while, entre si la condition est fausse

```
i=0
until [ $i -eq 3 ]; do
    echo $i
    let i+=1
done
```

For each

- Boucle sur une liste de valeurs
  - Séparées par la variable $IFS (saut de ligne par défaut)
  - Chaînes et nombres
- {m..n} : génère une liste de nombres de m à n séparés par des espaces

```
for i in {1..5}; do
    echo $i
done

for chain in INPUT OUTPUT FORWARD; do
    echo $chain
done
```

Contrôle du flot de la boucle

- break : interrompre la boucle
- continue : reprend au début de la boucle

7. Les fonctions

Portée des variables

- Par défaut : toutes les variables sont globales
  - Visible et modifiable depuis l'extérieur
- Le mot-clé local
  - Protège la variable

Arguments

- On peut en déclarer
- Ou utiliser $1, $2...

Code de retour

- Toujours numérique
- Retourner une chaîne : écrire sur STDOUT

Syntaxe générale

```
function my_function() {
    local var1
    local var2="value"
    command1
    command2
    return val;
}
```

Appel de fonction

```
my_function # returns int
a=$?
str=$(my_string_function) # writes to STDOUT (i.e.: echo)
```