

Administration Système sous Linux (Ubuntu Server)

Grégory Morel

2022-2023

CPE Lyon - 3IRC / 4ETI / 3ICS

Cours 2

Bash Partie 2 : langage de script

Variables d'environnement

Variables d'environnement

Les **variables d'environnement** (v.e.) sont des variables de configuration **globales**, utilisées par les programmes pour modifier certains comportements

Exemple : la variable d'environnement **LANG** détermine la langue que les logiciels utilisent pour communiquer avec l'utilisateur

Pour afficher le contenu d'une variable, on utilise **printenv variable**¹:

```
greg@cpe:~$ printenv LANG  
fr_FR.UTF-8
```

1. Si on ne spécifie aucune variable, elles sont toutes affichées

Variables d'environnement

Pour récupérer la valeur d'une v.e., on la fait précéder du symbole \$

```
greg@cpe:~$ echo USER
USER
greg@cpe:~$ echo $USER
batman
```

⚠ La variable est immédiatement remplacée par sa valeur :

```
greg@cpe:~$ $USER
La commande « batman » n'a pas été trouvée
```

Variables d'environnement

Pour **créer** une valeur d'environnement, il faut utiliser **export** ou **declare -x** :

```
greg@cpe:~$ export VAR="abcdef" ; printenv VAR
abcdef
```

💡 Sinon, on crée une *variable de Shell* **locale** (i.e. connue du Shell courant seulement, et donc uniquement pour la session en cours)

```
greg@cpe:~$ VAR="abcdef" ; printenv VAR
greg@cpe:~$
greg@cpe:~$ echo $VAR
abcdef
```

💡 La commande **set** liste **toutes** les variables (locales ou d'environnement) du shell courant

Variables d'environnement

Pour **modifier** la valeur d'une variable **existante** :

```
greg@cpe:~$ printenv LANG
fr_FR.UTF-8
greg@cpe:~$ LANG="en_US.UTF-8"
greg@cpe:~$ printenv LANG
en_US.UTF-8
```

- ⚠ Ne pas mettre d'espace entre la variable et le signe '='
- ⚠ La modification est **temporaire**, et n'est effective que pour la session courante

Variables d'environnement

Pour que la création ou la modification d'une variable soit **permanente**, il faut ajouter la commande au fichier `~/.bashrc`¹ qui est lu à chaque démarrage de bash

⚠ Ce fichier n'est lu qu'au *démarrage* de bash ; pour forcer bash à le relire immédiatement, il faut le **sourcer** :

```
source ~/.bashrc
```

💡 **.bashrc** ne concerne que l'utilisateur courant ; si on veut toucher *tous* les utilisateurs, il faut modifier le fichier *global* `/etc/bash_bashrc`

1. Les fichiers dont le nom se termine par **rc** sont très souvent des fichiers de configuration

Variables d'environnement

Exemple : la variable **PATH**

Elle indique à **bash** où trouver les commandes tapées par l'utilisateur.

```
greg@cpe:~$ printenv PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
:/bin:/usr/games:/usr/local/games:/snap/bin
```

Quand on tape une commande, **bash** regarde successivement dans chacun de ces dossiers jusqu'à la trouver.

Si on veut utiliser une commande se trouvant dans un dossier ne figurant pas dans `$PATH`, on a deux possibilités :

- indiquer à chaque fois le chemin complet vers commande
- modifier la variable `PATH` :

```
PATH=$PATH:~/mondossier
```

Pour que le changement soit permanent, on rajoute cette ligne à la fin du fichier `~/.bashrc`

Important!

 Ne jamais mettre le dossier courant (.) dans la variable **PATH**!

Supposons en effet qu'un attaquant réussisse à placer un script malveillant dans un dossier fréquemment utilisé :

- si . apparaît au début du **PATH**, et si le malware porte le même nom qu'une commande légitime (par exemple **ls**), c'est lui qui sera exécuté
- si . apparaît à la fin du **PATH**, le risque est diminué, mais existe encore si le malware porte le nom d'un programme légitime mais pas installé sur la machines

Pour **supprimer** une variable, on utilise la commande **unset** :

```
greg@cpe:~$ export VAR=toto ; printenv VAR  
toto  
greg@cpe:~$ unset VAR ; printenv VAR  
greg@cpe:~$
```

Introduction aux scripts Bash

Éditeurs de texte

- **nano** : rudimentaire, mais peut afficher la coloration syntaxique; utile pour éditer rapidement des fichiers textes
- **vim** : éditeur très puissant, extensible, peut être utilisé comme un véritable environnement de développement
- **emacs** : concurrent de vim; pas installé par défaut, très puissant, extensible et personnalisable

⚠ Ces éditeurs sont très différents du "bloc-note" de Windows, en particulier au niveau des raccourcis clavier

<https://doc.ubuntu-fr.org/{nano|vim|emacs}>

Hello World!

Un script Bash commence toujours par la ligne suivante¹ :

```
#!/bin/bash
```

La séquence `#!` est appelée **shebang** : elle indique à l'OS que le fichier est un script. Juste derrière, on indique le chemin vers un **interpréteur** capable d'exécuter le script

La suite du script est la liste des commandes à exécuter :

```
echo "Hello, World!"
```

Par convention, on donne une extension **.sh** aux scripts

1. Non obligatoire, mais permet de s'assurer que le script est exécuté par le bon interpréteur

Hello World!

On a ensuite besoin de rendre le script **exécutable** :

```
greg@cpe:~$ chmod u+x hello.sh
```

On peut enfin exécuter notre script :

```
greg@cpe:~$ ./hello.sh  
Hello, World!
```

? Que signifie **./** avant le nom de la commande?
Pour quelle raison Linux impose-t-il ce mécanisme?

Les variables

Par défaut, toutes les variables sont **globales** (même celles définies dans une fonction).

Pour définir une variable locale à une fonction, on utilise le mot-clé **local** :

```
#!/bin/bash
myvar='A'

my_function () {
    local myvar='B'
    echo "Dans my_function: myvar vaut $myvar"
}
my_function
echo "Après my_function, myvar vaut toujours $myvar"
-----
Dans my_function: myvar vaut B
Après my_function, myvar vaut toujours A
```

Les guillemets

Guillemets simples : chaîne littérale (le contenu n'est pas interprété)

```
var=toto  
echo 'contenu de var : $var'  
-----  
contenu de var : $var
```

Guillemets doubles : chaîne interprétée

```
var=toto  
echo "contenu de var : $var"  
-----  
contenu de var : toto
```

Exécuter une commande

Pour exécuter une commande dans un script, il suffit de taper son nom :

```
pwd
-----
/home/batman
```

On peut récupérer le résultat d'une commande avec la syntaxe `$(...)`¹ ou ``...``² :

```
res=$(pwd)      # ou res=`pwd`
echo "vous êtes dans le dossier : $res"
-----
vous êtes dans le dossier : /home/batman
```

-
1. Plus précisément, `(...)` exécute la commande dans un *sous-shell* et `$` récupère la sortie
 2. La syntaxe ``...`` n'est pas pratique quand on doit imbriquer des commandes

Demander de saisir une valeur

On peut demander à l'utilisateur de saisir des valeurs avec **read** :

```
read -p 'Saisissez deux valeurs a et b : ' a b
```

La commande **read** a de nombreuses options intéressantes :

- p** : affiche un message
- t** : limite de temps
- s** : n'affiche pas le texte saisi (pour des mots de passe par exemple)
- n** : limite le nombre de caractères

Paramètres

On peut aussi passer des paramètres directement sur la ligne de commande :

```
greg@cpe:~$ ./mon_script.sh param1 param2
```

Dans le script, les variables suivantes permettent de manipuler les paramètres :

\$# : nombre de paramètres

\$0 : nom du script

\$1 : premier paramètre

\$2 : second paramètre

etc.

Shift

Imaginons maintenant le cas suivant :

```
greg@cpe:~$ ./mon_script.sh param1 param2 ... param15
```

Pour éviter d'avoir à gérer 15 variables \$1...\$15, on peut utiliser **shift** :

```
while ((" $#")); do  
    echo $1  
    shift  
done
```

```
-----  
greg@cpe:~$ ./mon_script.sh param1 param2 ... param15  
param1  
param2  
...  
param15
```

Paramètres spéciaux

`$*` permet de récupérer l'ensemble des paramètres sous la forme d'un seul argument :

```
for param in $*; do
    echo $param
done
-----
greg@cpe:~$ mon_script abc def ghi
abc
def
ghi
```

Paramètres spéciaux

⚠ Problème : si un paramètre contient des espaces, il est scindé en plusieurs arguments :

```
for param in $*; do
    echo $param
done

echo $1
-----
greg@cpe:~$ mon_script "abc def ghi"
abc
def
ghi
abc def ghi
```


Paramètres spéciaux

Et si on encadrait `$*` par des guillemets ?

```
for param in "$*"; do
    echo $param
done
-----
greg@cpe:~$ mon_script "abc def ghi"
abc def ghi
```

Cette fois ça marche ! Enfin, presque... :

```
for param in "$*"; do
    echo $param
done
-----
greg@cpe:~$ mon_script abc def ghi
abc def ghi
```

En effet, on ne peut plus distinguer les différents paramètres :

```
#!/bin/bash
ls -l "$*"
-----
greg@cpe:~$ touch toto titi
greg@cpe:~$ ls toto titi
toto titi      <-- ça fonctionne
greg@cpe:~$ mon_script toto titi
ls: erreur 'toto titi': fichier ou dossier inexistant
```

Paramètres spéciaux

Il nous faudrait un paramètre spécial qui :

- fournisse autant d'arguments qu'à l'origine lorsqu'il est entre guillemets
- protège chacun des arguments par des guillemets (pour préserver les espaces)

Ce paramètre existe : c'est `$@`

```
#!/bin/bash
ls -l "$@"
-----
greg@cpe:~$ mon_script toto titi
-rw-r--r-- 1 batman batman 0 févr.  1 11:39 titi
-rw-r--r-- 1 batman batman 0 févr.  1 11:39 toto
```

Comment faire pour afficher le *i*-ème paramètre, quand *i* est elle-même une variable ?

```
for i in $(seq 1 3); do
    echo $i
done
-----
greg@cpe:~$ mon_script a b c
1
2
3
```

Solution : on **déréférence** la variable, avec la syntaxe **`${!i}`** :

```
for i in $(seq 1 3); do
    echo ${!i}
done
-----
greg@cpe:~$ mon_script a b c
a
b
c
```

Tableaux

Création :

```
tab=(elem1 elem2 elem3 ...)
```

Initialiation / modification :

```
tab[5]=42
```

💡 l'indice peut être $>$ taille du tableau

Accès :

```
${tab[0]}
```

⚠ Accolades obligatoires (pourquoi?); par ailleurs les indices commencent à 0

Afficher tout le contenu du tableau :

```
${tab[*]}          (ou ${tab[@]})
```

Taille du tableau :

```
${#tab[*]}          (ou ${#tab[@]})
```

Conditions

Réaliser un test :

```
if [ $nom = "Astérix" ]; then
    echo "Idéfix"
elif [ $nom = "Tintin" ]; then
    echo "Milou"
else
    echo "Autre"
fi
```

 Les espaces entre les crochets et le test sont obligatoires !

On peut combiner plusieurs tests à l'aide des opérateurs logiques :

-a : ET

-o : OU

! : NON

Conditions

Tests possibles sur des chaînes de caractères :

Condition	Signification
<code>"\$chaine1" =¹ "\$chaine2"</code>	Teste si les deux chaînes sont identiques (sensible à la casse)
<code>"\$chaine1" != "\$chaine2"</code>	Teste si les deux chaînes sont différentes
<code>-z "\$chaine"</code>	Teste si la chaîne est vide
<code>-n "\$chaine"</code>	Teste si la chaîne est non vide

1. Pour les habitués des langages de programmation, il est possible d'utiliser `==`

Conditions

Tests possibles sur des nombres :

Condition	Signification
<code>\$num1 -eq \$num2</code>	Teste si les deux nombres sont égaux
<code>\$num1 -ne \$num2</code>	Teste si les deux nombres sont différents
<code>\$num1 -lt \$num2</code>	Teste si $\text{num1} < \text{num2}$
<code>\$num1 -le \$num2</code>	Teste si $\text{num1} \leq \text{num2}$
<code>\$num1 -gt \$num2</code>	Teste si $\text{num1} > \text{num2}$
<code>\$num1 -ge \$num2</code>	Teste si $\text{num1} \geq \text{num2}$

Conditions

Les écritures `[[...]]` et `((...))` sont des extensions (**non standard!**) de `[...]`

Permettent d'écrire des tests plus simplement :

- on peut utiliser les opérateurs mathématiques `=`, `<`, `>`, `<=`, `>=`
- on peut utiliser les opérateurs logiques `&&` et `||`
- on peut utiliser des expressions rationnelles

Différence :

- `[[...]]` utilise l'ordre **lexicographique** (où `5 > 23`, par exemple)
- `((...))` utilise l'ordre **numérique** (où `5 < 23`, par exemple)

Exemple :

`[$a -le $b -a $c -ge 1]` vs. `(($a <= $b && $c >= 1))`

Boucles

Boucle **while** :

```
while [ test ]  
do  
    echo 'Action en boucle'  
done
```

Boucle **for** sur une liste :

```
for fichier in $(ls)  
do  
    cp $fichier $fichier.bak  
done
```


Boucle **for** sur une suite de nombres :

```
for i in $(seq 1 10)  
do  
    echo $i  
done
```

Calcul numérique

Les opérations arithmétiques sont réalisées par l'opérateur `((...))` :

```
echo $((2 + 3 * (5 ** 2) ))  
-----  
77
```

 Cet opérateur ne travaille qu'avec des entiers. Pour des opérations plus complexes, on utilise `bc` (*basic calculator*), qui nécessite une syntaxe particulière :

```
LC_NUMERIC=C          # pour gérer le point décimal  
...  
half=$(echo "1 / 2" | bc -l)  
printf '%.3f\n' $half  # affichage avec 3 décimales  
-----  
0.500
```

Fonctions : syntaxe

Il existe deux syntaxes pour déclarer une fonction en bash :

```
ma_fonction () {  
    <instructions>  
}
```

ou :


```
function ma_fonction {  
    <instructions>  
}
```

? Comment passer des paramètres avec la deuxième syntaxe ?

Fonctions : paramètres

En fait, en bash, les paramètres des fonctions se passent et se récupèrent comme sur la ligne de commande :

```
function ma_fonction {  
    echo $1 $2  
}  
...  
ma_fonction arg1 arg2
```

 On doit toujours définir une fonction **avant** de l'utiliser

Fonctions : valeur de retour

L'instruction **return** s'utilise pour (et seulement pour) renvoyer le *statut* de la dernière instruction exécutée :

- **0** en cas de succès (valeur par défaut)
- **un entier entre 1 et 255** sinon

Ce code peut être récupéré par l'appelant grâce à la variable **\$?** :

```
ma_fonction () {  
    echo "un résultat"  
    return 3  
}  
ma_fonction  
echo $?  
-----  
un résultat  
3
```

Autre exemple :

```
greg@cpe:~$ ls
toto
greg@cpe:~$ ls toto ; echo $?
toto
0
greg@cpe:~$ ls tutu ; echo $?
ls: 'tutu': aucun fichier ou dossier
2      #Code d'erreur de ls pour "fichier introuvable"
```


Fonctions : valeur de retour

Pour exploiter le résultat d'une fonction, l'option la plus simple est d'utiliser une variable globale :

```
ma_fonction() {  
    func_result="un résultat"  
}  
  
ma_fonction  
echo $func_result  
-----  
un résultat
```

Fonctions : valeur de retour

Cependant il est préférable d'envoyer la valeur sur la **sortie standard**, en utilisant la commande **echo**, puis d'utiliser la **substitution de commande** vue précédemment :

```
ma_fonction() {  
    echo $((1 + 2))  
}  
  
echo $(( $(ma_fonction) * 3 ))  
-----  
9
```