# Script bash : état de sortie et les tests

- Objet : Script bash : enchaînement de commandes et redirection
- Niveau requis :
  débutant, avisé
- Commentaires : Fix Me!
- Débutant, à savoir : Utiliser GNU/Linux en ligne de commande, tout commence là !.
- Suivi :
  - Création par Hypathie le 18/03/2014
  - Testé par Hypathie le Juin 2014
- Commentaires sur le forum : Lien vers le forum concernant ce tuto [1)]

Contributeurs, les Fix Me! sont là pour vous aider, supprimez-les une fois le problème corrigé ou le champ rempli !

**Nota : Les autres wiki :**

- debuter-avec-les-scripts-shell-bash
- script-bash-variables-arguments-parametres
- modification de variable et de paramètre
- script-bash-enchainement-de-commandes-et-etat-de-sortie
- 
- script-bash-les-tableaux
- script-bash-les-fonctions

## État de sortie et code de retour

### Le code de retour

Il ne faut pas confondre le code de retour et le résultat d'une commande. Le résultat est ce qui s'inscrit sur la sortie standard.

1. L'exécution de tous programmes et de toutes fonctions renvoie une valeur numérique appelée code de retour.
2. Il est envoyé `0` si tout c'est bien passé lors de l'exécution, et un nombre entre `1 et 255` s'il y a eu une erreur.
3. Pour récupérer le code de retour on utilise le paramètre spécial `$?`.
4. Il y a des codes de retour particuliers, par exemple :

```
blabla
```

```
echo $?
```

retour de la commande

```
127
```

> On peut chercher dans les codes d'erreur de bash que 127 est le code de retour pour les commandes qui n'ont pas pu être trouvées.

L'état de sortie vrai ou faux ( 0 ou autre chose) est utilisé avec :

1. enchaînements conditionnels :
2. les tests ( commande `test`, `[[ ]]`, `if/else`, `case`)

## L'enchaînement conditionnel est fondé sur le code de retour

Parmi les opérateurs d'enchaînement de commandes ci-dessous :
(voir : les opérateurs de contrôle)

```
||    &&    ;    <retour-chariot>
```

ceux fonctionnant sur le code de retour, sont :

1. l'opérateur `et`, `cmd1 && cmd2` : avec cet opérateur, la commande 2 est exécutée si le code retour de la commande 1 est `0` (c'est-à-dire, si elle a fonctionné).
2. l'opérateur `ou`, `cmd1 || cmd2` : la commande 2 est exécutée si le code de retour de la première est différente de zéro, c'est-à-dire si elle n'a pas fonctionné.

- Exemple :

Soit le dossier “Mon-dossier” non-vide; le dossier “mon-dossier” vide; et le dossier “mondossier” inexistant.

script

```
#!/bin/bash
cd ~/Mon-dossier && ls # => titi  toto
cd ~/mon-dossier || pwd # pas de retour : la première commande renvoie
0
            # car "rester sur place" n'est pas une erreur
cd ~/mondossier 2>/dev/null ||\
 echo "le dossier mondossier n'existe pas" &&\
 read -p "voulez-vous le créer [oui/non] ? " reponse &&\
 ( [ $reponse == non ] && echo " le dossier ne sera pas créé") ||\
 ( [ $reponse == oui ] && echo " le dossier va être créé" ) #; && mkdir
~/mondossier
```

> Les antislash ne sont pas obligatoires, ils servent à rendre plus lisible les longues commandes qui se suivent sur une ligne.
> On inhibe là le `<retour-chariot>` !

Les opérateurs && et || s'utilisent comme les opérateurs binaires.

Avec eux eux, c'est soit `0` soit autre chose que `0`, c'est-à-dire `1`, ou encore soit vrai, soit faux.
Voir la liste des opérateurs binaires

## Inverser le code de retour de la sortie d'une commande

```
! commande
```

- Exemple :

```
whoami
```

```
echo $?
```

retour de la commande

```
 0
```

```
! whoami
```

```
echo $?
```

retour de la commande

```
1
```

## Utiliser la commande exit

- Syntaxe : `exit nombre`

> Avec un nombre de 1 à 3 chiffre(s).

### La commande "exit"

`exit` permet de remplacer le code de retour de la dernière commande d'un script.

- Exemples :

Dans ce script, puisque la correspondance est juste, la commande “exit 1” est exécutée, et on sort du programme.

mon-script

```
var=bonjour
```

```
if [ $var == bonjour ] ; then
    echo "$var est correspond à bonjour"
    exit 1
fi
```

```
bonjour est correspond à bonjour
```

```
./mon-script
```

```
echo $?
```

retour de la commande

```
1
```

**"exit" force à sortir du programme**

script

```
#!/bin/bash
var=bonjour
if [ $var == coucou ] ; then
    echo "$var est correspond à coucou"
    exit 0
else
    echo "$var ne correspond pas à coucou"
    echo $?
    exit 1
    echo $?
fi
echo "$var"
```

```
bonjour ne correspond pas à coucou
 0
```

**"exit" et les paramètres passés au script**

Lançons ce script avec aucun argument ou un autre que “a”.

script

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

```
if [[ $1 == a ]] ; then
    exit 1
fi

echo "On a pas utilisé le bon argument, le test n'a pas permis
d'exécuter exit"
exit 0
```

```
On a pas utilisé le bon argument, le test n'a pas permis d'exécuter exit
```

#### Convention et sortie de programme par défaut

> Par convention on finit un script par `exit 0`
>
> Si on ne finit pas par `exit 0`, il s'exécute un `exit $?` par défaut, ce qui est équivalent à un `exit` (tout court).

script

```
#!/bin/bash
var=bonjour
echo "$var"
echho " " 2>/dev/null # affichage d'erreur vers le puits
echo $?
exit 13
```

```
bonjour
127
```

- Ou encore :

```
echo $?
```

retour de la commande

```
13
```

## État de sortie et les tests

#### À savoir :

- les opérateurs lexicographiques et leur syntaxe
- les opérateurs de comparaison numérique et leur syntaxe

- La commande test ou les crochets :conclusion-sur-les-operateurs-lexicographiques-et-les-operateurs-de-comparaison-numerique

**Attention au signe =** Ne pas confondre le signe = de l'affectation d'une variable par une valeur (voir ci-dessus affectation directe)
avec l'opérateur de correspondance = (ou == ) utilisé dans les tests.
Dans les tests sur les entiers l'égalité est représentée par l'option `-eq` !

## Composition avec les tests; valeurs (vides ou nulles) déclarées dans le script

De même que la composition de commandes vu plus haut, on se sert de la composition avec les tests. Et oui les doubles crochets et la commande test sont des commandes ! 

- séquentielle : `cmd1 ; cmd2`
- parallèle : `cmd1 & cmd2`
- sur erreur (ou) : `cmd1 || cmd2`
- sur succès (et) : `cmd1 && cmd2`
- rappel des options des commandes de test :

1. `-z $chaine` : teste si la variable ne contient rien
1. `-n $chaine` : teste si la variable contient quelque-chose

script

```
#!/bin/bash
#var3 est nulle, non déclarée, (ou non initialisée) : sa valeur est nulle
#var2 est initialisée mais sans valeur : sa une valeur vide
var1=ma_variable
var2=
var2bis=" "

[ ${var1} ] && echo "$var1"    # => ma_variable
[ -n ${var1} ] && echo "$var1" # => ma_variable
[ -n $var1 ] && echo "$var1"   # => ma_variable
#ou encore :
test $var1 && echo "ok"        # => ok
test -n $var1 && echo "ok var1 contient quelque-chose" # => ok var1 contient quelque-chose
test -z $var1 || echo "NON: var1 ne contient pas rien" # => NON: var1 ne contient pas rien
echo " "

[ -z $var2 ] && echo "var2: ${#var2} a une valeur vide : ne contient
```

```
rien" # => var2: 0 a une valeur vide : ne contient rien
[ -n $var2 ] && echo "var2: une valeur vide contient 0 : du vide !" #
=> var2: une valeur vide contient 0 : du vide !
[ -z $var2bis ] && echo "var2bis: comme var2"  # => var2bis: comme var2
[ -n $var2bis ] && echo "var2bis: comme var2" # => var2bis: comme var2
echo " "
[ -n $var3 ] && echo "ET avec -n: une variable nulle contient aussi du
vide" # => ET avec -n: une variable nulle contient aussi du vide
[ -n $var3 ] || echo "OU avec -n"  # PAS DE RETOUR puisque la première
commande a renvoyé le code de retour 0.
[ -z $var3 ] && echo " OU avec -z une variable nulle contient aussi du
vide" # => OU avec -z une variable nulle contient aussi du vide
[ -z $var3 ] || echo " OU avec -z" # PAS DE RETOUR puisque la première
commande a renvoyé le code de retour 0.
```

## Tests sur paramètres passés au scripts

### Alternatives : case et paramètres passés aux scripts

Attention, `case` n'utilise pas d'expression régulière, il s'agit plutôt de “pattern matching”.
Voir : les symboles reconnus par cases sont ceux servant à la manipulation des fichiers.

- syntaxe de case :

```
case $variable in
     expression)
         instructions
         ;;
     ...
esac
```

- Explications :

1. case “teste” la valeur du paramètre passé au script avec chaque “expression” ;
2. et en fonction de la réussite ou de l'échec du test, il y a exécution ou non des commandes placées au niveau de “instructions” ;
3. Case sert à conditionner l'exécution des commandes en fonction d'argument choisi ;
4. On se sert de “l'étoile” pour permettre que soit exécuter quelque chose quand n'importe quel autre paramètre que ceux des expressions, est passé au script;

- Ne pas oublier :

1. le double point virgule qui permet de clôturer chaque test ;
2. esac pour finir.

- exemple :

script

```
#!/bin/bash
# passer le paramètre 'coco' à ce script ; puis 'cucu' ; titi puis
toto, puis celui que voulez.
case $1 in
    coco)
        echo "Vous avez passé le paramètre 'coco', ré-essayez avec
'cucu'"
    ;;
    cucu)
        echo "Vous avez passé le paramètre 'cucu'"
        echo "un peu d'humour !"
        echo "Ré-essayé avec titi, puis avec toto."
    ;;
    titi|toto)
        echo "vous avez passé le paramètre $1"
    ;;
    *)
        echo "Vous avez choisi $1"
esac
```

Vous avez tout en mains pour comprendre ceci : Fonctionnalités avancées du Shell: selecteur-case

#### if et les paramètres passés au script

Voir structure conditionnelle if pour ce qui suit.

Un script peut opérer un test sur les chaînes de caractères passées au script depuis le terminal.

Ci-dessous, les valeurs de la variable1 (var1) et de la variable2 (var2) peuvent être les arguments passés au script. *(Lancez “mon-script” successivement sans argument, puis un, deux trois, etc. arguments.)*

script

```
#!/bin/bash
var1=$1
var2=$2
echo $1
echo $2
if [ $# == 1 ] ; then
 echo "ERREUR: vous avez entré $@, mais il faut deux arguments !"
  elif [ $# == 2 ] ; then
```

```
   echo "Les deux arguments que vous avez entré sont $1 et $2"
  elif [ $# == 0 ] ; then
    echo "ERREUR: vous n'avez pas entré d'arguments, il en faut deux !"
fi
```

## Tests sur les valeurs déclarées dans le script

### Avec la structure conditionnelle if

Avec la structure conditionnelle if, on peut aussi faire des tests sur la (les) valeur(s) déclarée(s) dans le script.

script

```
#!/bin/bash
var1=23
var2=36

if [ $var1 -eq $var2 ] ; then
    echo "$var1 et $var2 sont égales"
else
    echo "$var1 et $var2 ne sont pas égales"
fi

if [ $var1 != $var2 ] ; then
    echo "$var1 et $var2 sont inégales"
fi

if [ ${#var1} = ${#var2} ] ; then
    echo "$var1 et $var2 sont des chaînes de même longueur"
fi
echo "les longueurs sont de : ${#var1} et de : ${#var2}"
```

### Conclusion

Les chaînes testées par un script peuvent être aussi le contenu d'un fichier.

- Voir ce script : http://www.quennec.fr/linux/programmation-shell-sous-linux/les-bases-de-la-programmation-shell/ex%C3%A9cution-de-tests/la-5

Avec if + $paramètre ; case ; et l'utilisation de la commande test, les valeurs des variables que l'on testent dans le script peuvent être passées depuis l'extérieur du script, ou par déclaration/affectation dans le script, avant le(s) test(s).

Pour modifier la valeur d'une variable d'un script (que le script appelle une valeur extérieure; chaîne de caractère donnée par le terminale, récupération du résultat d'une commande, contenu d'un fichier, etc. ou que cette variable soit affectée dans le script lui-même) on utilise les boucles while et for.

On peut aussi modifier la valeur d'une variable d'un script, en exportant depuis le terminal une nouvelle valeur (voir "export" )

# La suite c'est ici

script-bash-les-tableaux

1)

N'hésitez pas à y faire part de vos remarques, succès, améliorations ou échecs !