Bash : les opérateurs de comparaison numérique

- Objet : Suite de la série de wiki visant à maîtriser bash par les caractère.
- Niveau requis :
  débutant
- Commentaires : Bash, ligne de commande et scripts
- Débutant, à savoir : Utiliser GNU/Linux en ligne de commande, tout commence là !.
- Suivi :
  à-tester
  - Création par Hypathie le 08/04/2014
  - Testé par Hypathie en Avril 2014
  - Modifié par agp91 le 21/02/2022
- Commentaires sur le forum : Lien vers le forum concernant ce tuto [1)]

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

Introduction

Dans la page du manuel de bash, **les opérateurs des commandes de test** sont nommés **primitives**.

Bash dispose de plusieurs commandes pour **réaliser des comparaisons numérique**.

- Les commandes de test :
  - Les commandes internes **[** et **test**.
  - Et la commande composée **[[**.
- Les commandes d'évaluation numériques :
  - La commande composée **((** et la commande interne **let**.

- Les commandes **((** et **let** sont équivalentes.
- Les commandes **[** et **test** sont équivalentes.

- Les commandes **[** et **test** sont disponibles dans leurs versions externe : **/usr/bin/[** et **/usr/bin/test**.
  - Elles ont toutes les deux la même page de manuel (**man [** ou **man test**).
- Les commandes internes disposent de primitive que n'ont pas les commandes externes.

Rappels :

- Une commande de test renvoie le code de retour **0** (considéré comme vrai) lorsque le test réussi et **1** (considéré comme faux) lorsqu'il échoue.
- Le code retour d'une commande est mémorisé dans le paramètre spécial **$?**.
- L'opérateur de contrôle **&&** exécute la commande suivante, si la commande précédente a renvoyé un code de retour égal à **0**.
- L'opérateur de contrôle **||** exécute la commande suivante, si la commande précédente a renvoyé un code de retour supérieur à **0**.

Comparaison numérique avec [et [[

Les commandes de test dispose de 6 primitives binaires pour effectuer des comparaisons numériques.

Syntaxe

- **test expr1 OP expr2**
- **[expr1 OP expr2]**
- **[[expr1 OP expr2]]**

- Avec :
  - **Expr1** et **expr2** sont sujets au développement des paramètres.
  - Avec la commande **[[**, **expr1** et **expr2** sont sujets à l'évaluation arithmétique. Les commandes **[** ou **test**, ne le permettent pas.
  - **OP**, l'une des primitives du tableau suivant.

Liste des primitives de comparaison numérique	
Primitives	**Retours**
`-eq`	est égal à
`-ne`	n'est pas égal à
`-gt`	est plus grand que
`-ge`	est plus grand ou égal à
`-lt`	est plus petit que
`-le`	est plus petit ou égal à

Exemples

```
[ 25 -eq 20 ]   # Teste si 25 est égale à 20
echo $?
```

```
1
```

```
var1=17
var2=18
echo "$var1 est-il plus grand que $var2 : $([ $var1 -gt $var2 ]; echo $?)"
echo "$var1 est-il plus petit que $var2 : $([[ $var1 -lt $var2 ]]; echo $?)"
echo "$var1 est-il différent de $var2 : $(test $var1 -ne $var2 ; echo $?)"

unset var1 var2
```

```
17 est-il plus grand que 18 : 1
17 est-il plus petit que 18 : 0
17 est-il différent de 18 : 0
```

Copions le code ci-dessous dans le fichier **mon_script**.
Puis exécutons le avant de le supprimer.

mon_script

```
#!/bin/bash
a=2 ; b=1
if [ "$a" -gt "$b" ] ; then
    echo "$a est plus grand que $b"
fi

if test 100 -gt 99
then
    echo "vrai"   #réponse : vrai
else
    echo "faux"
fi
```

```
bash mon_script

rm -f mon_script
```

```
2 est plus grand que 1
vrai
'mon_script' supprimé
```

Particularités de la commande <nowiki>[[</nowiki>

Seule la commande **[[** permet de tester une expression arithmétique.

```
test 21+21 -eq 42 ; echo -e $?\\n

[ 21*2 -eq 42 ] ; echo -e $?\\n

[[ 84-42 -eq 42 ]] ; echo $?
```

```
bash: test: 21+21 : nombre entier attendu comme expression
2

bash: [: 21*2 : nombre entier attendu comme expression
2

0
```

Lorsqu'une commande interne **le code de retour renvoie 2**, cela signifie un mauvais usage de cette commande.

Elle est aussi la seule à accepter les chaînes vides qui sont alors évaluées à **0**.

```
[[ 0 -eq "" ]] ; echo $?
[[ "" -ne 0 ]] ; echo $?
[[ "" -eq "" ]] ; echo $?
```

```
0
1
0
```

Idem pour les chaînes de caractères sans espace (mots). Un mot est substitué par **0**.

```
[[ mot -eq 0 ]] ; echo $?
[[ 0 -ne mot ]] ; echo $?
[[ mot1 -eq mot2 ]] ; echo $?
```

```
0
1
0
```

Attention :

mon_script

```
#!/bin/bash
var1=8
var2=7
if test "$e" -gt "$f" ; then
```

```
  echo " $var1 est plus grand que $var2 "
fi
printf \\n
test 8 -gt 7 && echo "$var1 est plus grand que $var2"
```

```
bash mon_script

rm -v mon_script
```

retour de la commande

```
mon_script: ligne 4 : test:  : nombre entier attendu comme
expression

8 est plus grand que 7
'mon_script' supprimé
```

Mauvais usages

Le caractère **$** est obligatoire pour développer les paramètres. Sans, ils sont interprétés comme des mots.

Avec la commande **[** (ou **test**), les chaînes vides ou les mots sont à proscrire.

```
test 42 -eq "" ; echo $?
[ 42 -lt "" ] ; echo $?
echo =====
test "" -gt 42 ; echo $?
[ "" -ge 42 ] ; echo $?
```

```
bash: test:  : nombre entier attendu comme expression
2
bash: [:  : nombre entier attendu comme expression
2
=====
bash: test:  : nombre entier attendu comme expression
2
bash: [:  : nombre entier attendu comme expression
2
```

```
test mot -eq 0  ; echo $?
[ 0 -eq mot ]   ; echo $?
test 0 -le 0    ; echo $?
[ "mot" -ge 0 ]   ; echo $?
```

```
echo $?
```

```
bash: test: mot : nombre entier attendu comme expression
2
bash: [: mot : nombre entier attendu comme expression
2
bash: test: 0 : nombre entier attendu comme expression
2
bash: [: mot : nombre entier attendu comme expression
2
```

Avec **[** (ou **test**) et **[[**, les opérandes ne peuvent être des chaînes de caractères qui comportent des espaces.

```
test "Du texte" -eq 0 ; echo $?
[ 0 -ge "Du texte" ]  ; echo $?
[[ "Du texte" -ge 0 ]]    ; echo $?
```

```
bash: test: Du texte : nombre entier attendu comme expression
2
bash: [: Du texte : nombre entier attendu comme expression
2
bash: [[: Du texte : erreur de syntaxe dans l'expression (le symbole erroné
est « texte »)
1
```

Il est remarquable que le test `[[ "Du texte" -ge 0 ]]`, ne renvoie pas le **code de retour 2**.
Pourtant il s'agit bien d'un mauvais usage de la commande `[[`.

Les deux opérandes sont obligatoires.

```
test 42 -eq ; echo $?
[ 42 -lt ] ; echo $?
[[ 42 -ne ]]
echo $?
echo =====
test -gt 42 ; echo $?
[ -ge 42 ] ; echo $?
[[ -le 42 ]]
echo $?
```

```
bash: test: 42 : opérateur unaire attendu
2
bash: [: 42 : opérateur unaire attendu
2
bash: argument « ]] » inattendu pour l'opérateur binaire conditionnel
bash: erreur de syntaxe près de « ]] »
```

```
2
=====
bash: test: -gt : opérateur unaire attendu
2
bash: [: -ge : opérateur unaire attendu
2
bash: opérateur binaire conditionnel attendu
bash: erreur de syntaxe près de « 42 »
2
```

Ainsi que les espaces.
Lorsqu'il n'y pas d'espaces entre les opérandes et l'opérateur, l'ensemble est considéré comme une chaîne de caractères
Puisque la chaîne est non vide, le test n'échoue pas.

```
test 420-ne420  ; echo $?
[ 420-gt42 ]    ; echo $?
[[ 42-lt420 ]]  ; echo $?
```

```
0
0
0
```

Ne pas utiliser les opérateurs **<** et **>** avec **[** (ou **test**) et **[[** pour réaliser des comparaisons numériques.
Car avec ces commandes, ces opérateur sont des opérateurs de comparaison lexicographique.
(voir Bash : Les opérateurs sur chaînes).

Rappel : Avec **[** (ou **test**), les opérateurs **<** et **>** s'utilisent protégés (voir Bash, comparaison lexicographique avec [ou test).

```
test 425 \> 4242 ; echo $?
[ 426 > 4242 ]       ; echo $?
[[ 4242 < 427 ]] ; echo $?
```

retour des commandes

```
0
0
0
```

Lexicographiquement 425, 426 et 427 sont supérieurs (placés après dans l'ordre lexicographique), mais sont inférieurs (plus petits) numériquement à 4242.

Comparaison numérique avec ((

La commandes d'évaluation numérique **((** (commande composée) et la commande **let** (commande

interne), permettent de réaliser des comparaisons numériques.
Elles disposent de 6 opérateurs de comparaison.

Syntaxe

- **let expr1<OP>expr2**
- **let “expr1 <OP> expr2”**
- **((<expr1> <OP> <expr2>))**

- Avec :
  - **Expr1** et **expr2** sont sujets au développement des paramètres et à l’évaluation arithmétique.
  - **<OP>** l'un des opérateurs donnés dans le tableau suivant.

Opérateurs de comparaison numérique des commandes let et ((	
Opérateurs	**Retours**
==	Vrai si égale
!=	Vrai si différent
>	Vrai si plus grand que
>=	Vrai si plus grand ou égal que
<	Vrai si plus petit que
<=	Vrai plus petit ou égal que

La commande **let** ne supporte qu'un seul argument.
Si l'expression arithmétique à évaluer comporte des espaces, l'expression doit être protégée par des guillemets simples ou doubles.
Si l'expression n'est pas protégée, les opérateurs commençant par **<** ou **>** doivent être protégés
(Voir plus bas : Mauvais usages.)

Exemples

```
(( 42 == 42 ))          # Est-ce que 42 est égale à 42.
echo $?              # Affiche le code de retour.
let 24>=24           # Est-ce que 24 est supérieur ou égale à 24
echo $?              # Affiche le code de retour.
```

```
0
0
```

```
nombre1=12
nombre2=13
(( $nombre1 > $nombre2 ))    # Est-ce-que nombre1 (12) est strictement
supérieur au nombre2 (13).
echo $?              # Affiche le code de retour.
let "$nombre1 != $nombre2"    #Est-ce-que nombre1 (12) est différent du
```

```
nombre2 (13)
echo $?             # Affiche le code de retour.

unset nombre1 nombre2       # Suppression des paramètres nombre1 et nombre2
```

```
1
0
```

Les paramètres peuvent être transmis sans **$** (Sauf les paramètres positionnels et les paramètres spéciaux).

```
n1=42
n2=24
(( n1 > n2 )) && echo "$n1 > $n2" || echo "$n1 < $n2"
let n1==n2 && echo "$n1 est égale à $n2" || echo "$n1 est différent de $n2"

unset n1 n2
```

```
42 > 24
42 est différent de 24
```

Comme nous l'avons vu au dessus, les paramètres n'ont pas besoin du caractère **$** pour être développés.
Un simple mot sera alors interprété comme un paramètre.
Si ce paramètre n'existe pas, le mot sera substitué par **0**.

```
(("mot" == 0))
echo '(("mot" == 0)) revoie le code de retour' $?
let "mot <= 0"
echo 'let "mot <= 0" revoie le code de retour' $?

n="mot"
((n!=0))
echo -e "Avec n=\"$n\" ; ((n!=0)) renvoie le code de retour $?"
let n==0
echo -e "Avec n=\"${n}\" ; let n==0 renvoie le code de retour $?"

unset n
```

```
(("mot" == 0)) revoie le code de retour 0
let "mot <= 0" revoie le code de retour 0
Avec n="mot" ; ((n!=0)) renvoie le code de retour 1
Avec n="mot" ; let n==0 renvoie le code de retour 0
```

Si un paramètre existe mais que sa valeur est vide, son développement retournera 0.

```
n=
((n==0))    ; echo $?
let 'n != 0'    ; echo $?
```

```
unset n
```

```
0
1
```

Avec la commande **((**, les espaces ne sont pas obligatoires.

```
n1=4242
n2=2424
if ((n1>n2))
then
    echo "$n1 est supérieur à $n2"
else
    echo "$n1 est inférieur à $n2"
fi

unset n1 n2
```

```
4242 est supérieur à 2424
```

Copions le code ci-dessous dans un fichier nommé **mon_script**.
Puis exécutons avant de le supprimer.

mon_script

```
#!/bin/bash
a=8 ; b=2
if (( "$a" < "$b" )) ; then
  echo "$a < $b"
else
  echo "$a n'est pas inférieur à $b"
fi
```

```
bash mon_script
echo $?

rm -v mon_script
```

```
8 n'est pas inférieur à 2
0
'mon_script' supprimé
```

L'exécution du script renverra toujours **0** (vrai), car le code de retour renvoyé est celui de la dernière commande exécutée, qui est **echo**.
(voir : Utilisation de la commande exit).

Maintenant, nous allons créer une fonction (**test_si_négatif**) qui teste si une expression numérique ou arithmétique est négative.

La commande **return** est identique à la commande **exit** (**return** s'utilise dans une fonction, **exit** dans un script).

```
test_si_négatif() {
    # Test_si_négatif <expression>
    # Retourne le code de retour 0 si <experssion> est négative
    # Retourne le code de retour 1 si <experssion> est positive
    # Retourne le code de reour 2 si la fonction est mal utilisée
    local rc=2                # Initialise le paramètre rc avec la valeur 2
    if (( $# == 0 ))            # Test si le nombre d'argument est 0
    then                        # Si oui,
        echo "Argument manquant" >&2        # Retourne un message sur le
canal d'erreur
    elif (( $# > 1 ))            # Si non, test si le nombre d'argument est
> à 1
    then                        # Si oui
        echo "Trop d'arguments" >&2        # Retourne un message sur le
canal d'erreur.
    elif (($1 >= 0))            # Si non, test si l'argument est positif ou
égale à 0
    then                        # Si oui,
        rc=1                    # Affecte 1 au paramètre rc
    elif (($1 < 0))            # Si non, test si l'argument est négatif
    then                        # Si oui,
        rc=0                    # Affecte 0 au paramètre rc
    fi
    return $rc                # Affecte $rc au code retour
}

test_si_négatif     ; echo -e $?\\n
test_si_négatif 42  ; echo $?
test_si_négatif -42 ; echo $?
test_si_négatif 42-84   ; echo $?

unset test_si_négatif
```

```
Argument manquant
2

1
0
0
```

Le 4eme usage de notre fonction montre que la commande **((** évalue une expression arithmétique avant de la tester.

Mauvais usages

Avec la commande **let**, si l'expression à évaluer n'est pas protégée,
Les espaces entre les opérandes et l'opérateur ne sont pas supportés.

```
let -42 < 0  ; echo $?
let 0 == 0  ; echo $?
```

```
bash: let: < : erreur de syntaxe : opérande attendu (le symbole erroné est
« < »)
1
bash: let: == : erreur de syntaxe : opérande attendu (le symbole erroné est
« == »)
1
```

Avec la commande **let**, si l'expression à évaluer est protégé par des guillemets simples,
Les paramètres sont développés uniquement s'ils ne disposent pas du caractère **$**.
S'il est fourni, une erreur est renvoyée.

```
n=42
let 'n==42' ; echo $?
let '$n > 42'    ; echo $?
```

```
0
bash: let: $n == 42 : erreur de syntaxe : opérande attendu (le symbole
erroné est « $n == 42 »)
1
```

Avec la commande **let**, si l'expression à évaluer n'est pas protégée, les opérateurs **<**, **<=**, **>** et **>=** doivent être protégés.
Les trois types de protection (****, entre guillemets simples **''** et entre guillemets doubles **“”** fonctionnent.
S'ils ne sont pas protégés, les opérateurs **<** et **>** sont des opérateurs de redirection.
Démonstration :

La commande **printf “\n”**, renvoie un saut de ligne.
La commande **echo -n**, n'ajoute pas de saut ligne à la fin de son retour.

```
p=$PWD
mkdir /tmp/test_let
cd /tmp/test_let

let 240>420  ; echo $?
let 241>=421 ; echo $?
let 243<423  ; echo $?
let 244<=424 ; echo $?

printf "\n"
echo -n "ls :"
ls
```

```
printf "\n"

cd $p
rm -rfv /tmp/test_let
```

```
bash: let: une expression est attendue
1
bash: let: une expression est attendue
1
bash: 423: Aucun fichier ou dossier de ce type
1
bash: =424: Aucun fichier ou dossier de ce type
1

ls : 420  '=421'

'/tmp/test_let/=421' supprimé
'/tmp/test_let/420' supprimé
répertoire '/tmp/test_let' supprimé
```

Les opérateurs **>** et **>=** ont créé respectivement les fichiers **420** et **=421**.
Les opérateurs **<** et **<=** ont recherché les fichiers **423** et **=424**, sans les trouver.

Les opérateurs de comparaison sont des opérateurs binaires, ils attendent donc 2 arguments (ou opérandes).
Les commande **((** et **let** retournent une erreur s'il manque un opérande.

```
(( == 0))   ; echo $?
let 0>=      ; echo $?
echo =====
(( 42 < ""))   ; echo $?
let ""!=24    ; echo $?
```

retour des commandes

```
bash: ((: == 0 : erreur de syntaxe : opérande attendu (le symbole
erroné est « == 0 »)
1
bash: let: une expression est attendue
1
=====
bash: ((: 42 <  : erreur de syntaxe : opérande attendu (le symbole
erroné est « <  »)
1
bash: let: !=24 : erreur de syntaxe : opérande attendu (le symbole
erroné est « !=24 »)
1
```

Les opérandes ne peuvent être des chaînes de caractères contenant des espaces.

L'option **-e** de la commande **echo** permet de développer les caractères protégés, (ici **\n** qui se développe en saut de ligne).
Le développement des caractères protégés se réalisent entre guillemets doubles.
Sans guillemets, il est nécessaire de protéger le caractère de protection (**\\n**).

```
(("Du texte" >= 0))
echo -e '* (("Du texte" >= 0)) renvoie le code de retour' $?\\n
let 424\>"Du texte"
echo -e '* let 424\>"Du texte" renvoie le code de retour' $?\\n

n="Du texte"
((0<=n))
echo -e "* Avec n=\"$n\" ; ((0<=n)) renvoie le code retour $?\n"
let n==0
echo "* Avec n=\"${n}\" ; let n==0 renvoie le code retour $?"

unset n
```

```
bash: ((: Du texte >= 0 : erreur de syntaxe dans l'expression (le symbole
erroné est « texte >= 0 »)
* (("Du texte" >= 0)) renvoie le code de retour 1

bash: let: 424>Du texte : erreur de syntaxe dans l'expression (le symbole
erroné est « texte »)
* let 424\>"Du texte" renvoie le code de retour 1

bash: ((: Du texte : erreur de syntaxe dans l'expression (le symbole erroné
est « texte »)
* Avec n="Du texte" ; ((0<=n)) renvoie le code retour 1

bash: let: Du texte : erreur de syntaxe dans l'expression (le symbole erroné
est « texte »)
* Avec n="Du texte" ; let n==0 renvoie le code retour 1
```

Remarquons que les commandes **let** et **((** renvoient **le code de retour 1** quand elles sont en erreur. Ce n'est pas la norme pour une commande interne de bash. Le code de retour devrait être **2**.

Pour aller plus loin : les opérateurs logiques.

Ils s'utilisent avec les tests.

Opérateur	Signification
!	Négation
-a	et

Opérateur	Signification
-o	ou

Et dans un ordre précis :

1. ! (négation)
2. -a (et)
3. -o (ou)

- Il doit toujours y avoir un espace autour des opérateurs: `!`, `-a`, `-o`.
- Ne pas confondre `-a` (opérateur logique) avec un opérateur de test sur les fichiers.
- Ne pas confondre `-o` (opérateur logique) avec `-ot` (test pour savoir si un fichier1 est plus ancien qu'un fichier2.
- Il est possible de modifier la priorité d'exécution des opérateurs en utilisant des parenthèses.
- Les parenthèses doivent être protégées par des anti-slash afin de ne pas être interprétées par le shell comme étant un regroupement de commandes: `\\\(…\)`

Exemples :

- Le fichier “toto” n'est pas un répertoire :

```
[ ! -d /etc/group ]
```

```
echo $?
```

retour de la commande

```
0
```

> Il est vrai (retour 0) que ce “n'est pas” 😉

- Le fichier mon-script existe et est exécutable :

```
[ -f mon-script -a -x mon-script ]
```

```
echo $?
```

retour de la commande

```
0
```

Tuto précédent

Bash : Les opérateurs de test sur fichiers

La suite c'est ici :

Bash : les symboles dans les calculs

1)

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