Dernière mise-à-jour : 2020/01/30 03:27

105.2 - Personnaliser ou écrire des scripts simples (4/60)

Les Scripts Shell

Le but de la suite de cette unité est de vous amener au point où vous êtes capable de comprendre et de déchiffrer les scripts, notamment les scripts de démarrage ainsi que les scripts de contrôle des services.

Écrire des scripts compliqués est en dehors de la portée de cette unité car il nécessite une approche programmation qui ne peut être adressée que lors d'une formation dédiée à l'écriture des scripts.

Exécution

Un script shell est un fichier dont le contenu est lu en entrée standard par le shell. Le contenu du fichier est lu et exécuté d'une manière séquentielle. Afin qu'un script soit exécuté, il suffit qu'il puisse être lu au quel cas le script est exécuté par un shell fils soit en l'appelant en argument à l'appel du shell :

/bin/bash myscript

soit en redirigeant son entrée standard :

/bin/bash < myscript

Dans le cas où le droit d'exécution est positionné sur le fichier script et à condition que celui-ci se trouve dans un répertoire spécifié dans le PATH de l'utilisateur qui le lance, le script peut être lancé en l'appelant simplement par son nom :

myscript

Dans le cas où le script doit être exécuté par le shell courant, dans les mêmes conditions que l'exemple précédent, et non par un shell fils, il convient de le lancer ainsi :

. myscript et **./myscript**

Dans un script il est fortement conseillé d'inclure des commentaires. Les commentaires permettent à d'autres personnes de comprendre le script. Toute ligne de commentaire commence avec le caractère **#**.

Il existe aussi un **pseudo commentaire** qui est placé au début du script. Ce pseudo commentaire permet de stipuler quel shell doit être utilisé pour l'exécution du script. L'exécution du script est ainsi rendu indépendant du shell de l'utilisateur qui le lance. Le pseudo commentaire commence avec les caractères **#!**. Chaque script commence donc par une ligne similaire à celle-ci :

```
#!/bin/sh
```

Puisque un script contient des lignes de commandes qui peuvent être saisies en shell intéractif, il est souvent issu d'une procédure manuelle. Afin de faciliter la création d'un script il existe une commande, **script**, qui permet d'enregistrer les textes sortis sur la sortie standard, y compris le prompt dans un fichier dénommé **typescript**. Afin d'illustrer l'utilisation de cette commande, saisissez la suite de commandes suivante :

```
[trainee@centos7 ~]$ script
Script started, file is typescript
[trainee@centos7 ~]$ pwd
/home/trainee
[trainee@centos7 ~]$ ls
aac  bca    Desktop    Downloads  fichier1  file   Music     Public     training    Videos  xyz
abc  codes  Documents  errorlog   fichier2  file1  Pictures  Templates  typescript  vitext
[trainee@centos7 ~]$ exit
exit
Script done, file is typescript
[trainee@centos7 ~]$ cat typescript
Script started on Tue 29 Nov 2016 03:58:33 CET
[trainee@centos7 ~]$ pwd
/home/trainee
[trainee@centos7 ~]$ ls
aac  bca    Desktop    Downloads  fichier1  file   Music     Public     training    Videos  xyz
abc  codes  Documents  errorlog   fichier2  file1  Pictures  Templates  typescript  vitext
[trainee@centos7 ~]$ exit
exit
```

```
Script done on Tue 29 Nov 2016 03:58:40 CET
```

Cette procédure peut être utilisée pour enregistrer une suite de commandes longues et compliquées afin d'écrire un script.

Pour illustrer l'écriture et l'exécution d'un script, éditez le fichier **myscript** avec **vi** :

```
$ vi myscript [Entrée]
```

Éditez votre fichier ainsi :

```
pwd
ls
```

Sauvegardez votre fichier. Lancez ensuite votre script en passant le nom du fichier en argument à /bin/bash :

```
[trainee@centos7 ~]$ vi myscript
[trainee@centos7 ~]$ /bin/bash myscript
/home/trainee
aac  codes  Downloads  fichier2  myscript    Public       typescript  xyz
abc  Desktop    errorlog   file      Music    Templates  Videos
bca  Documents  fichier1   file1     Pictures    training   vitext
```

Lancez ensuite le script en redirigeant son entrée standard :

```
[trainee@centos7 ~]$ /bin/bash < myscript
/home/trainee
aac  codes  Downloads  fichier2  myscript    Public       typescript  xyz
abc  Desktop    errorlog   file      Music    Templates  Videos
bca  Documents  fichier1   file1     Pictures    training   vitext
```

Pour lancer le script en l'appelant simplement par son nom, son chemin doit être inclus dans votre PATH:

```
[trainee@centos7 ~]$ echo $PATH
```

```
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/trainee/.local/bin:/home/trainee/bin
```

Dans le cas de RHEL/CentOS, même si PATH contient $HOME/bin, le répertoire n'existe pas :

```
[trainee@centos7 ~]$ ls
aac  codes      Downloads  fichier2  myscript  Public     typescript  xyz
abc  Desktop    errorlog   file      Music      Templates  Videos
bca  Documents  fichier1   file1     Pictures   training   vitext
```

Créez donc ce répertoire :

```
[trainee@centos7 ~]$ mkdir bin
```

Ensuite déplacez votre script dans ce répertoire et rendez-le exécutable pour votre utilisateur :

```
[trainee@centos7 ~]$ mv myscript ~/bin
[trainee@centos7 ~]$ chmod u+x ~/bin/myscript
```

Exécutez maintenant votre script en l'appelant par son nom à partir du répertoire **/tmp** :

```
[trainee@centos7 tmp]$ myscript
/tmp
hsperfdata_root  systemd-private-e526abcf335b40949dfc725f28456502-cups.service-u0xGiL
```

Placez-vous dans le répertoire contenant le script et saisissez les commandes suivantes :

- ./myscript
- . myscript

```
[trainee@centos7 tmp]$ cd ~/bin
[trainee@centos7 bin]$ ./myscript
/home/trainee/bin
myscript
[trainee@centos7 bin]$ . myscript
```

```
/home/trainee/bin
myscript
```

A faire : Notez bien la différence entre les sorties de cette dernière commande et la précédente. Expliquez pourquoi.

La commande read

La commande **read** lit son entrée standard et affecte les mots saisis dans la ou les variable(s) passée(s) en argument. La séparation entre le contenu des variables est l'espace. Par conséquent il est intéressant de noter les exemples suivants :

```
[trainee@centos7 bin]$ read var1 var2 var3 var4
fenestros edu is great!
[trainee@centos7 bin]$ echo $var1
fenestros
[trainee@centos7 bin]$ echo $var2
edu
[trainee@centos7 bin]$ echo $var3
is
[trainee@centos7 bin]$ echo $var4
great!
```

Important: Notez que chaque champs a été placé dans une variable différente. Notez aussi que par convention les variables déclarées par des utilisateurs sont en miniscules afin de les distinguer des variables système qui sont en majuscules.

```
[trainee@centos7 bin]$ read var1 var2
fenestros edu is great!
[trainee@centos7 bin]$ echo $var1
```

```
fenestros
[trainee@centos7 bin]$ echo $var2
edu is great!
```

Important : Notez que dans le deuxième cas, le reste de la ligne après le mot *fenestros* est mis dans **$var2**.

Code de retour

La commande **read** renvoie un code de retour de **0** dans le cas où elle ne reçoit pas l'information **fin de fichier** matérialisée par les touches Ctrl+D.
Le contenu de la variable **var** peut être vide et la valeur du code de retour **0** grâce à l'utilisation de la touche Entrée :

```
[trainee@centos7 bin]$ read var
```

↵ Entrée

```
[trainee@centos7 bin]$ echo $?
0
[trainee@centos7 bin]$ echo $var

[trainee@centos7 bin]$
```

Le contenu de la variable **var** peut être vide et la valeur du code de retour **autre que 0** grâce à l'utilisation des touches Ctrl+D :

```
[trainee@centos7 bin]$ read var
```

Ctrl+D

```
[trainee@centos7 bin]$ echo $?
1
```

```
[trainee@centos7 bin]$ echo $var

[trainee@centos7 bin]$
```

La variable IFS

La variable IFS contient par défaut les caractères Espace, Tab et Entrée :

```
[trainee@centos7 bin]$ echo "$IFS" | od -c
0000000      \t  \n  \n
0000004
```

> **Important** : La commande **od** (*Octal Dump*) renvoie le contenu d'un fichier ou de l'entrée standard au format octal. Ceci est utile afin de visualiser les caractères non-imprimables. L'option **-c** permet de sélectionner des caractères ASCII ou des backslash dans le fichier ou dans le contenu fourni à l'entrée standard.

La valeur de cette variable définit donc le séparateur de mots lors de la saisie des contenus des variables avec la commande **read**. La valeur de la variable **IFS** peut être modifiée :

```
[trainee@centos7 bin]$ OLDIFS="$IFS"
[trainee@centos7 bin]$ IFS=":"
[trainee@centos7 bin]$ echo "$IFS" | od -c
0000000   :  \n
0000002
```

De cette façon l'espace redevient un caractère normal :

```
[trainee@centos7 bin]$ read var1 var2 var3
fenestros:edu is:great!
[trainee@centos7 bin]$ echo $var1
```

```
fenestros
[trainee@centos7 bin]$ echo $var2
edu is
[trainee@centos7 bin]$ echo $var3
great!
```

Restaurez l'ancienne valeur de IFS avec la commande IFS="$OLDIFS"

```
[trainee@centos7 bin]$ IFS="$OLDIFS"
[trainee@centos7 bin]$ echo "$IFS" | od -c
0000000      \t  \n  \n
0000004
```

La commande test

La commande **test** peut être utilisée avec deux syntaxes :

test *expression*

ou

[Espace*expression*Espace**]**

Tests de Fichiers

Test	Description
-f fichier	Retourne vrai si fichier est d'un type standard
-d fichier	Retourne vrai si fichier est d'un type répertoire
-r fichier	Retourne vrai si l'utilisateur peut lire fichier
-w fichier	Retourne vrai si l'utilisateur peut modifier fichier
-x fichier	Retourne vrai si l'utilisateur peut exécuter fichier
-e fichier	Retourne vrai si fichier existe

Test	Description
-s fichier	Retourne vrai si fichier n'est pas vide
fichier1 -nt fichier2	Retourne vrai si fichier1 est plus récent que fichier2
fichier1 -ot fichier2	Retourne vrai si fichier1 est plus ancien que fichier2
fichier1 -ef fichier2	Retourne vrai si fichier1 est identique à fichier2

LAB #1

Testez si le fichier **a100** est un fichier ordinaire :

```
[trainee@centos7 bin]$ cd ../training/
[trainee@centos7 training]$ test -f a100
[trainee@centos7 training]$ echo $?
0
[trainee@centos7 training]$ [ -f a100 ]
[trainee@centos7 training]$ echo $?
0
```

Testez si le fichier a101 existe :

```
[trainee@centos7 training]$ [ -f a101 ]
[trainee@centos7 training]$ echo $?
1
```

Testez si /home/trainee/training est un répertoire :

```
[trainee@centos7 training]$ [ -d /home/trainee/training ]
[trainee@centos7 training]$ echo $?
0
```

Tests de chaînes de caractère

Test	Description
-n chaîne	Retourne vrai si chaîne n'est pas de longueur 0
-z chaîne	Retourne vrai si chaîne est de longueur 0
string1 = string2	Retourne vrai si string1 est égale à string2
string1 != string2	Retourne vrai si string1 est différente de string2
string1	Retourne vrai si string1 n'est pas vide

LAB #2

Testez si les deux chaînes sont égales :

```
[trainee@centos7 training]$ string1="root"
[trainee@centos7 training]$ string2="fenestros"
[trainee@centos7 training]$ [ $string1 = $string2 ]
[trainee@centos7 training]$ echo $?
1
```

Testez si la string1 n'a pas de longueur 0 :

```
[trainee@centos7 training]$ [ -n $string1 ]
[trainee@centos7 training]$ echo $?
0
```

Testez si la string1 a une longueur de 0 :

```
[trainee@centos7 training]$ [ -z $string1 ]
[trainee@centos7 training]$ echo $?
1
```

Tests sur des nombres

Test	Description
value1 -eq value2	Retourne vrai si value1 est égale à value2
value1 -ne value2	Retourne vrai si value1 n'est pas égale à value2
value1 -lt value2	Retourne vrai si value1 est inférieure à value2
value1 -le value2	Retourne vrai si value1 est inférieur ou égale à value2
value1 -gt value2	Retourne vrai si value1 est supérieure à value2
value1 -ge value2	Retourne vrai si value1 est supérieure ou égale à value2

LAB #3

Comparez les deux nombres **value1** et **value2** :

```
[trainee@centos7 training]$ read value1
35
[trainee@centos7 training]$ read value2
23
[trainee@centos7 training]$ [ $value1 -lt $value2 ]
[trainee@centos7 training]$ echo $?
1
[trainee@centos7 training]$ [ $value2 -lt $value1 ]
[trainee@centos7 training]$ echo $?
0
[trainee@centos7 training]$ [ $value2 -eq $value1 ]
[trainee@centos7 training]$ echo $?
1
```

Les opérateurs

Test	Description
!expression	Retourne vrai si expression est fausse

Test	Description
expression1 -a expression2	Représente un et logique entre expression1 et expression2
expression1 -o expression2	Représente un ou logique entre expression1 et expression2
\(expression\)	Les parenthèses permettent de regrouper des expressions

LAB #4

Testez si $file n'est pas un répertoire :

```
[trainee@centos7 training]$ file=a100
[trainee@centos7 training]$ [ ! -d $file ]
[trainee@centos7 training]$ echo $?
0
```

Testez si $directory est un répertoire **et** si l'utilisateur à le droit de le traverser :

```
[trainee@centos7 training]$ directory=/usr
[trainee@centos7 training]$ [ -d $directory -a -x $directory ]
[trainee@centos7 training]$ echo $?
0
```

Testez si l'utilisateur peut écrire dans le fichier a100 **et** /usr est un répertoire **ou** /tmp est un répertoire :

```
[trainee@centos7 training]$ [ -w a100 -a \( -d /usr -o -d /tmp \) ]
[trainee@centos7 training]$ echo $?
0
```

Tests d'environnement utilisateur

Test	Description
-o option	Retourne vrai si l'option du shell “option” est activée

LAB #5

```
[trainee@centos7 training]$ [ -o allexport ]
[trainee@centos7 training]$ echo $?
1
```

La commande [[expression]]

La commande **[[Espace expression Espace]]** est une amélioration de la commande **test**. Les opérateurs de la commande test sont compatibles avec la commande [[expression]] sauf **-a** et **-o** qui sont remplacés par **&&** et **||** respectivement :

Test	Description
!expression	Retourne vrai si expression est fausse
expression1 && expression2	Représente un **et** logique entre expression1 et expression2
expression1 || expression2	Représente un **ou** logique entre expression1 et expression2
(expression)	Les parenthèses permettent de regrouper des expressions

D'autres opérateurs ont été ajoutés :

Test	Description
string = modele	Retourne vrai si chaîne correspond au modèle
string != modele	Retourne vrai si chaîne ne correspond pas au modèle
string1 < string2	Retourne vrai si string1 est lexicographiquement avant string2
string1 > string2	Retourne vrai si string1 est lexicographiquement après string2

LAB #6

Testez si l'utilisateur peut écrire dans le fichier a100 **et** /usr est un répertoire **ou** /tmp est un répertoire :

```
[trainee@centos7 training]$ [[ -w a100 && ( -d /usr || -d /tmp ) ]]
[trainee@centos7 training]$ echo $?
```

```
0
```

Opérateurs du shell

Opérateur	Description
Commande1 && Commande2	Commande 2 est exécutée si la première commande renvoie un code vrai
Commande1 || Commande2	Commande 2 est exécutée si la première commande renvoie un code faux

LAB #7

```
[trainee@centos7 training]$ [[ -d /root ]] && echo "The root directory exists"
The root directory exists
[trainee@centos7 training]$ [[ -d /root ]] || echo "The root directory exists"
[trainee@centos7 training]$
```

L'arithmétique

La commande expr

La commande **expr** prend la forme :

expr Espace value1 Espace *opérateur* Espace value2 Entrée

ou

expr Tab value1 Tab *opérateur* Tab value2 Entrée

ou

expr Espace chaîne Espace : Espace *expression_régulière* Entrée

ou

expr Tab chaîne Tab : Tab *expression_régulière* Entrée

Opérateurs Arithmétiques

Opérateur	Description
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo
\(\)	Parenthèses

Opérateurs de Comparaison

Opérateur	Description
\<	Inférieur
\<=	Inférieur ou égal
\>	Supérieur
\>=	Supérieur ou égal
=	égal
!=	inégal

Opérateurs Logiques

Opérateur	Description
\\|	ou logique
\&	et logique

LAB #8

Ajoutez 2 à la valeur de $x :

```
[trainee@centos7 training]$ x=2
[trainee@centos7 training]$ expr $x + 2
4
```

Si les espaces sont retirés, le résultat est tout autre :

```
[trainee@centos7 training]$ expr $x+2
2+2
```

Les opérateurs doivent être protégés :

```
[trainee@centos7 training]$ expr $x * 2
expr: syntax error
[trainee@centos7 training]$ expr $x \* 2
4
```

Mettez le résultat d'un calcul dans une variable :

```
[trainee@centos7 training]$ resultat=`expr $x + 10`
[trainee@centos7 training]$ echo $resultat
12
```

La commande let

La commande let est l'équivalent de la commande ((expression)). La commande ((expression)) est une amélioration de la commande **expr** :

- plus grand nombre d'opérateurs
- pas besoin d'espaces ou de tabulations entre les arguments
- pas besoin de préfixer les variables d'un $
- les caractères spéciaux du shell n'ont pas besoin d'être protégés
- les affectations se font dans la commande
- exécution plus rapide

Opérateurs Arithmétiques

Opérateur	Description
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo
^	Puissance

Opérateurs de comparaison

Opérateur	Description
<	Inférieur
<=	Inférieur ou égal
>	Supérieur
>=	Supérieur ou égal
==	égal
!=	inégal

Opérateurs Logiques

Opérateur	Description
&&	et logique
||	ou logique
!	négation logique

Opérateurs travaillant sur les bits

Opérateur	Description
~	négation binaire
>>	décalage binaire à droite
<<	décalage binaire à gauche

Opérateur	Description
&	et binaire
|	ou binaire
^	ou exclusif binaire

LAB #9

```
[trainee@centos7 training]$ x=2
[trainee@centos7 training]$ ((x=$x+10))
[trainee@centos7 training]$ echo $x
12
[trainee@centos7 training]$ ((x=$x+20))
[trainee@centos7 training]$ echo $x
32
```

Structures de contrôle

If

La syntaxe de la commande If est la suivante :

```
if condition
then
    commande(s)
else
    commande(s)
fi
```

ou :

```
if condition
```

```
then
    commande(s)
    commande(s)
fi
```

ou encore :

```
if condition
then
     commande(s)
elif condition
then
     commande(s)
elif condition
then
     commande(s)
else
    commande(s)

fi
```

LAB #10

Créez le script **user_check** suivant :

```
#!/bin/bash
if [ $# -ne 1 ] ; then
  echo "Mauvais nombre d'arguments"
  echo "Usage : $0 nom_utilisateur"
  exit 1
fi
if grep "^$1:" /etc/passwd > /dev/null
```

```
then
  echo "Utilisateur $1 est défini sur ce système"
else
  echo "Utilisateur $1 n'est pas défini sur ce système"
fi
exit 0
```

Testez-le :

```
[trainee@centos7 training]$ chmod 770 user_check
[trainee@centos7 training]$ ./user_check
Mauvais nombre d'arguments
Usage : ./user_check nom_utilisateur
[trainee@centos7 training]$ ./user_check root
Utilisateur root est défini sur ce système
[trainee@centos7 training]$ ./user_check mickey mouse
Mauvais nombre d'arguments
Usage : ./user_check nom_utilisateur
[trainee@centos7 training]$ ./user_check "mickey mouse"
Utilisateur mickey mouse n'est pas défini sur ce système
```

case

La syntaxe de la commande case est la suivante :

```
case $variable in
modele1) commande
   ...
   ;;
modele2) commande
   ...
   ;;
```

```
modele3 | modele4 | modele5 ) commande
   ...
   ;;
esac
```

Exemple

```
  case "$1" in
      start)
          start
          ;;
      stop)
          stop
          ;;
      restart|reload)
          stop
          start
          ;;
      status)
          status
          ;;
      *)
          echo $"Usage: $0 {start|stop|restart|status}"
          exit 1
esac
```

Important : L'exemple indique que dans le cas où le premier argument qui suit le nom du script contenant la clause **case** est **start**, la fonction *start* sera exécutée. La fonction *start* n'a pas besoin d'être définie dans **case** et est donc en règle générale définie en début de script. La même logique est appliquée dans le cas où le premier argument est **stop**, **restart** ou **reload** et **status**. Dans tous les autres cas, représentés par une étoile, **case** affichera la ligne **Usage: $0 {start|stop|restart|status}** où $0 est remplacé par le nom du script.

Boucles

for

La syntaxe de la commande for est la suivante :

```
for variable in liste_variables
do
    commande(s)
done
```

while

La syntaxe de la commande while est la suivante :

```
while condition
do
    commande(s)
done
```

Exemple

```
U=1
while [ $U -lt $MAX_ACCOUNTS ]
do
useradd fenestros"$U" -c fenestros"$U" -d /home/fenestros"$U" -g staff -G audio,fuse -s /bin/bash 2>/dev/null
useradd fenestros"$U"$ -g machines -s /dev/false -d /dev/null 2>/dev/null
echo "Compte fenestros$U créé"
let U=U+1
```

```
done
```

Scripts de Démarrage

Quand Bash est appelé en tant que shell de connexion, il exécute des scripts de démarrage dans l'ordre suivant :

- **/etc/profile**,
- **~/.bash_profile** ou **~/.bash_login** ou **~/.profile** selon la distribution,

Dans le cas de RHEL/CentOS, le système exécute le fichier **~/.bash_profile**.

Quand un shell de login se termine, Bash exécute le fichier **~/.bash_logout** si celui-ci existe.

Quand bash est appelé en tant que shell interactif qui n'est pas un shell de connexion, il exécute le script **~/.bashrc**.

LAB #11

A faire : En utilisant vos connaissances acquises dans ce module, expliquez les scripts suivants ligne par ligne.

~/.bash_profile

```
[trainee@centos7 training]$ cat ~/.bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

```
# User specific environment and startup programs

PATH=$PATH:$HOME/.local/bin:$HOME/bin

export PATH
```

~/.bashrc

```
[trainee@centos7 training]$ cat ~/.bashrc
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# Uncomment the following line if you don't like systemctl's auto-paging feature:
# export SYSTEMD_PAGER=

# User specific aliases and functions
```

<html>

Copyright © 2004-2017 Hugh Norris.

Ce(tte) oeuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 France.

</html>