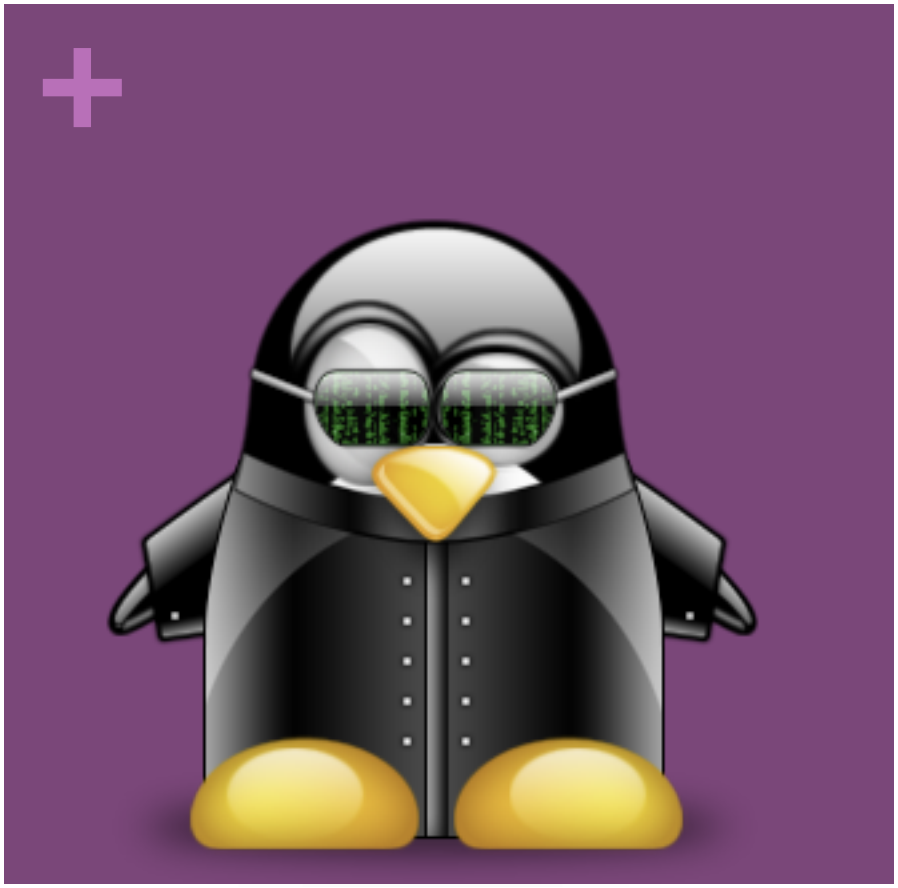
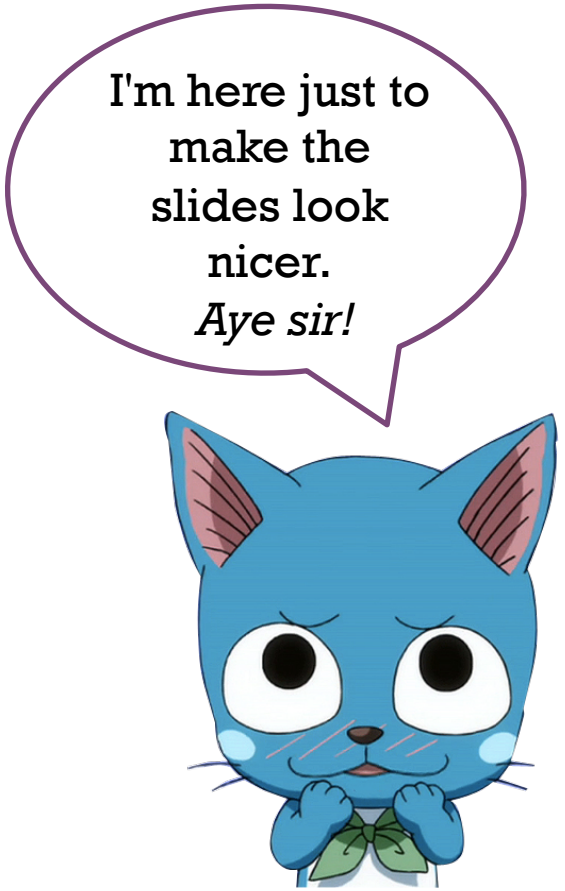


Introduction to Shell Scripts

David Beserra ☺

Agenda

- About the Shell
- Configuration Files
- About (Bash) Scripts
- Script Structure
- First Script
- Variables
- Operators
  - Number Comparison Operators
  - String Comparison Operators
  - Conditional Statements Operators
  - Operators for Testing Files Operators

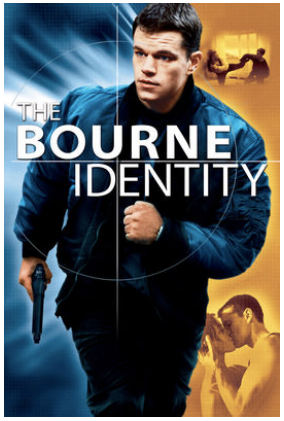


Agenda

- Decision Structures
  - if/then/else case
- Looping Constructs
  - for
  - while
- Parameters
- Functions

About Shell

- It is an interface between the user and the operating system.
  - It includes several built-in commands that allow users to utilize the services of the operating system.
- It implements a simple programming language that allows the development of small programs: shell scripts
- Bash is one among many available shells
  - Developped by Bourne: *Bourne Again SHell*



Configuration files

- The configuration of the user's working environment is done in three files stored in the user's *home* directory.
  - ~/.bash_profile
  - ~/.bashrc
  - ~/.bash_lougout
- These files allow the execution of commands at different times.

Configuration files

- ~/.bash_profile
  - This file is processed every time you log in.
  - The information added to this file will not be valid until it is read again. Therefore, you need to log out and then log in again.
- ~/.bashrc
  - This file is processed every time a sub-shell is generated.
    - *Anyway, how to create a sub-shell?*

Configuration files

- **~/.bash_logout**
  - This file is processed **just before** you log out.
  - So, for example, commands to delete temporary files can be executed before closing the session.

It can be convenient to place all shell customization information in the ~/.bashrc file and call the ~/.bashrc file (source ~/.bashrc) in the ~/.bash_profile file.

This way, when we launch a new shell, the environment will already be configured ;)

About shell scripts

- Scripts are simple files that contain sequences of Linux commands.
  - Nevertheless, they have control structures:
    - Decision Structures
    - Looping Constructs
- **Shell scripts are a simple solution for automating tasks in a Linux system.**
  - They interact with Linux commands.
  - They allow the manipulation of command outputs.
  - **Interpreted** language

Script Structure

- Start
  - #!/bin/bash
  - Cette ligne indique quel sera le shell utilisé pour interpréter les scripts
- Middle
  - Commands (any Linux command can be used).
  - Control Structures
  - Decision Structures
  - Looping Constructs
  - Comments
    - After the character #
- End
  - exit 0 (but it's not mandatory xD)

First script ☺

- You need to create a new file in a text editor (any).
  - Ex: nano my-script.sh
- Enter the commands for the script.
  - #!/bin/bash
  - # This script prints a simple text
  - echo Hello World
- The "echo" command prints the sentence passed as a parameter to the screen, moving the cursor to the next line.

First script ☺

- Running the script
  - Requires execution permission.
  - chmod +x my-script.sh
- Three ways to run
  - ./my-script.sh
  - source my-script.sh
  - bash my-script.sh

In fact, the form ./ simply indicates that we will execute a command local to the directory.

When you use the form "bash script," the script runs in a copy of the shell (sub-shell), with control being passed back to the parent shell after its completion.

When you use "./ script" or "source script," the script runs in the current shell.

Variables

- Variables have no defined type.
  - They are strings of characters.
  - If they consist only of digits (numbers), bash allows operations on integers (addition, subtraction).
- Operations
  - Assign a value to the variable
  - Access the variable
  - Print to the screen
  - Read a variable from the keyboard
  - Execute commands

Variables

- To assign a value
- You MUST NOT put spaces before and after the '='. Ex:
  - x=world
  - a="Hello World"
- To access the value:
  - y=$x #assign "world" to y
  - y=x #assign the character "x" to y
  - y=${x} #assign "world" to y
  - y=${x}nn #assign "worldnn" to y (concatenation)
  - y=$xnn #error, because there is no xnn variable. Then, the value "" is assigned to y
  - y=`ls` #interprets the ls command and assigns its output to y
  - y= "hello" #assigns the character string "hello" to y
  - On utilise "$" pour faire référence (à la valeur) de la variable
- To print on screen
  - y=''world''
  - echo 'Hello $y' #print Hello $y
  - echo "Hello $y" #print Hello world
  - echo Hello $y #print Hello world

Single quotes: content literal value

Double quotes: interprets the value of the variable

- Be **careful** about the difference between **single quotes** and **double quotes**

Variables

- Read a variable from the keyboard
  - read -p "Tell me, my friend, what's your name?: " prenom
  - echo $prenom
- A variable can contain commands
  - LS="ls"
  - LS_FLAGS="-l"
  - $LS $LS_FLAGS /home/david/ #lists files in /home/david

Exercice

- Create a Bash script that receives the user's name and displays a welcome message: "Hello Patricia!"
- Create a Bash script that receives two user parameters (last name and first name) and displays both words separated and concatenated.
  - Example:
  - Parameters provided by the user: patricia and endo
  - Print on the screen: patricia, endo, and patriciaendo

Operators

- Comparison Operators for Numbers:
  - -eq: equal to
  - -ge: equal to or greater than
  - -gt: greater than
  - -lt: less than
  - -le: equal to or less than
  - -ne: not equal to

Operators

- Comparison Operators for Numbers:

```
X=3
Y=4
if [ $X -lt $Y ] #Is $X < $Y?
then
        echo "${X} is less than ${Y}"
else
        echo "${X} is not less than à ${Y}"
fi
```

Operators

- String Comparison Operators
  - =: equals to
  - !=: different to
- Associations between conditions
  - &&: logical and
  - ||: logical or

Example:

```
if [ $X = s ] || [ $X = S ]
if [ $X -lt $Y ] && [ $Z -gt $Y ]
```

Operateurs

- Opérateurs de test de fichiers
  - -e: l'entrée existe
  - -r: l'entrée peut être lue
  - -w: l'entrée peut être écrite
  - -x: l'entrée peut être exécutée
  - -s: a une taille supérieure à zéro
  - -f: est un fichier
  - -d: est un répertoire

Operateurs

- Opérateurs pour tester des fichiers
  - -O: L'utilisateur est propriétaire de l'entrée
  - -G: le groupe d'entrée est le même que le propriétaire
  - -nt: Vérifie si un fichier est plus récent qu'un autre
  - -ot: Vérifie si un fichier est plus ancien qu'un autre
  - -ef: Vérifie s'il s'agit du même fichier

+ Operateurs

- Operateurs de teste de fichier
- Example:

```
fichier='/etc/password'

if [ -e $fichier ]
then
        if [ -f $fichier ]
        then
                if [ -r $fichier ]
                        source $fichier
                else
                        echo "Je ne peux pas lire le fichier $fichier"
                fi
        else
                echo "$fichier n'est pas un fichier normal"
        fi
else
        echo "$fichier n'existe pas"
fi
```

Operators

- Arithmetic operators
  - +, -, *, /: basic arithmetic
  - **: potentiation
  - %: module (remainder)
  - +=, -=, *=, /=, %=,: arithmetic and assignment
  - “and”: bit shift
  - “= and ”=: offset and assignment
  - & and | : binary AND and OR
  - &= and |=: binary AND and OR with assignment
  - ! : NOT (binary)

Operators

- Mathematical operations:
  - a=$((b+c))
  - let a=b+c
  - let a+=1

Decision Structures

- if/then/else
  - The conditions tested are:
    - the exit status of the command execution
    - The output value of Boolean expressions
  - If the status is **zero**, the condition is considered **true**

Decision Structures

- if/then/else
  - Example: check if two files are equal with cmp
    - The cmp command is used to compare two files byte by byte. If a difference is found, it indicates the byte and line number where the first difference is found. If no difference is found, by default cmp returns no output.

```
if  cmp $file1 $file2 > /dev/null       #teste le statut de la commande cmp
then
            echo ''les fichiers sont égaux''
else
            echo ''les fichiers sont differents''
fi
```

Decision Structures

- if/then/else
  - There is a condition operator named test, which can also be represented as [condition], which returns 0 when the condition is true

```
if [ $n1 –lt $n2 ]              # $n1 est inferieur à $n2?
then
        echo ‘’$n1 est inferieur à $n2”
fi

if test  $n1 –lt $n2            # $n1 est inferieur à $n2?
then
        echo ‘’$n1 est inferieur à $n2”
fi
```

Decision Structures

- if/then/else
  - Main form
    - if-then

```
if command
then
        …
fi

if  cmp $file1 $file2 > /dev/null      #tests the status of the cmp command
then
        echo '' files are equal''
fi
```

Decision Structures

- if/then/else
  - Main forms
    - if-then-else

```
if command
then
        …

else
        …
fi

if  cmp $file1 $file2 > /dev/null        #Test the status of the "cmp" command.
then
        echo ‘’ The files are equal.”

else
        echo ‘’ the files are different”
fi
```

Decision Structures

- if/then/else
  - Main forms
    - if-then-elif-else

```
if command
then
        …
elif command2
then
        …
else
        …
fi
```

Decision Structures

- if/then/else
  - Main forms
    - if-then-else-elif

```
if  [ $n1 -lt $n2 ]       #compare variables $n1 and $n2
then
            echo ‘’$n1 < $n2”
elif [ $n1 -gt $n2 ]
then
            echo ‘’$n1 > $n2”
else
            echo ‘’$n1 = $n2”
fi
```

Decision Structures

- if/then/else

Basic

```
if [ condition ]
then
        command1
        command2
        ...
else
        command3
fi
```

Nested

```
if [ condition1] then
        command1
        command2
        ...
elif [ condition2 ]
then
        command3
        command4
        ...
elif [ condition3 ]
then
        command5
        command6
        ...
fi
```

Decision Structures - case

- Allows multiple decision options.

Structure

```
case $var in
   "$cond1" )
   commands...
   ;;
   "$cond2" )
   comands...
   ;;
esac
```

Example

```
case $opt in
            "-c")
                        complete= 1 ;;
            "-c")
                        short=1;
                        name="nothing";;
             *)
                        echo "inexistent option";
                        exit 1;;
esac
```

Exercice

- Write a bash script that takes two numbers as parameters and makes comparisons of "greater than", "less than" or "equal to"
  - Example: 2 and 10
  - 2 is less than 10

Repeat Loops - for

- There are several ways to do a *for* loop in bash.

```
for  variable in liste-de-valeurs
do
     commandes
done
```

```
for ((variable; condition; incrément ))
do
     commandes
done
```

```
for  planeta in Mercurio Venus Terra Marte
    echo $planeta
done

for  i in 1 2 3 4 5 6
    echo $i
done

for  i  in seq 1 100
    echo $i
done
```

```
for ((i=0; i<5; i++))
do
    echo $i
done
```

Boucles de Répétition - while

- Attention! Il faut toujours mettre un espace après le [et avant le]

```
while [ condition ]
do
    commandes
done
```

#Beserra-sensei, j'ai réussi à faire un code! Regarde!

```
while [ "$var" != "oui" ]
do
   read -p  "Est-ce que ce code mérite un 20/20?: oui ou non" var
   echo "reponse= $var"
done
```

Bien joué élève-chan !
Mais ce n'est pas le code que je t'ai demandé de faire! hihi!

Input parameters

- Parameters
  - $0: Script name
  - $n: nth parameter (where: 0 < n < 10)
  - $#: Number of parameters
  - $*: All parameters
  - $?: Exit status of the last executed command ("exit 1" returns 1)
  - $$: Process ID (PID) of the shell executing the script.

Input parameters

- Parameter Passing
- Script content.sh
  - #!/bin/bash
  - echo "$is a lecturer of the course$2"
- When you call the script, you pass the parameters you want :
  - ./content.sh david unixSystemAdministration
  - david a lecturer of the course unixSystemAdministration .

Input parameters

- Example:
  - Script that lists all the parameters:
  - listparameters.sh

```
#!/bin/bash

# Example of using input parameters

echo "Script name: $0"
echo "First parameter: $1"
echo "All parameters: $*"
echo "Number of parameters: $#"
echo "PID of this process: $$"
exit 0
```

Input parameters

- Running the script listparameters.sh

```
./listparameters.sh david fabrice angela luisa ali sabrine

Script name: listparameters
First parameter: david
All parameters: david fabrice angela luisa ali sabrine
Number of parameters: 6
PID of this process: 1989
```

Outputs

- Execution of Commands
  - Example:
    - $(ls)
- Recording the standard output in a variable.
  - Useful for commands that return only one line.
  - Does not preserve the line break.
    - x = `ls –l`
    - echo $ x

Functions

- These are code blocks that can be used anywhere in the script, as many times as needed.
- To execute the function, simply use the function name as if it were a bash command.
- Syntax for defining a function:

```
name_of_the_function ()
{
        #code
}
```

Functions

- Example

```
sqrt() {
        echo “Enter a number “
        read input
        echo “$the input’ssquare is: $((entree**2))”
}

echo “Running program” sqrt
```

Functions

- **Important Note**
  - Variables defined within functions have global scope and are not deleted once the function is finished.
  - To define local variables, use the **"local"** keyword *followed by the variable name at the beginning of the function*.

```
sqrt() {
        local input
        echo "Entrez un nombre: "
        read input
        echo "$the square of the input is: $((input**2))"
}

echo "Running the program" sqrt
```

Functions

- Parameter Passing and Return of Values

```
sum() {
        result= `expr $1 + $2`
        return $result
}

echo –n “5 + 7 = ”
Sum 5 7
echo $?

echo `sum 5 7`
```

$1 and $2 receive the values of the function's parameters.

"return" indicates the return value of the function.

$? receives the function's return value. Only **integer values** can be returned by functions.