

LINUX

BASH

SCRIPTING

ÍNDICE

1. Intro	3
2. Hello world	4
3. Variables	5
4. Operadores	6
5. Maths	6
6. if	7
7. Case statements	7
8. While loops	8
9. For loops	9
10. Debugging	10
Exit codes	10
Set	10
11. Filesystem localización para Scripts	11
12. Data Streams	12
standard output (stdout)	12
standard error (stderr)	12
standard input	12
13. Functions	13
14. Scheduling p1	13
15. Scheduling p2	13
16. Argumentos	13
17. Otros recursos	14
18. Comandos útiles	14

1.Intro

Bash Scripting es la técnica de automatizar tareas en sistemas operativos basados en Unix utilizando el lenguaje de comandos Bash en el shell. Bash es un intérprete de comandos que permite escribir scripts por defecto de Ubuntu, los scripts son pequeños programas que ejecutan una serie de comandos en secuencia. Esto permite automatizar tareas, ahorrar tiempo y reducir la posibilidad de errores humanos. Los scripts en Bash pueden ser utilizados para una amplia gama de tareas, desde comprobar actualizaciones del sistema hasta la creación de complejas soluciones para problemas comunes.

¿Por qué usar Bash Scripting?

Existen muchas razones para usar Bash Scripting, entre ellas:

- Automatizar tareas repetitivas: Si realizas tareas tediosas y repetitivas con frecuencia, puedes escribir un script para automatizarlas y ahorrar tiempo y esfuerzo.
- Simplificar tareas complejas: Bash Scripting te permite combinar varios comandos en un solo script, lo que puede simplificar tareas complejas que de otro modo requerirían escribir muchos comandos individuales.
- Mejorar la productividad: Al automatizar tareas y simplificar procesos complejos, Bash Scripting te puede ayudar a mejorar tu productividad.
- Mayor control sobre tu sistema: Bash Scripting te da un mayor control sobre tu sistema operativo, ya que te permite personalizar y automatizar tareas a tu gusto.

Más información:

<https://www.hostinger.es/tutoriales/bash-script-linux>


2. Hello world

```
#!/bin/bash #shebang mas ruta del intérprete a usar por el script
```

¿Cómo saber el intérprete por defecto del shell?

```
ps
```

```
echo $SHELL #muestra su localización
```

 Los script por defecto se terminan en .sh pero NO es obligatorio. Lo que determina si se pueden ejecutar como tal son los permisos (que cambia según usuario). Cuando es ejecutable el color del archivo es **verde**

Como se vio anteriormente así se le otorgan permisos de ejecución a todos los usuarios

```
sudo chmod +x nombrescript.sh
```

Aunque quizás no es lo ideal y es necesario darle solo al owner o grupo. Para ello utilizar los permisos absolutos o simbólicos adecuados. Por normal general, para que se ejecute debe también permisos de lectura.

Ejecutar script

```
./nombrescript.sh #si no le damos los permisos no ejecuta.
```

 También funciona *bash nombrescript.sh*

```
#!/bin/bash

nombre="Manuel"
edad="30"

echo "Hola, my nombre es $nombre."
echo "Tengo $edad años."
```

 **OJO CON PONER ELEMENTOS EN LA MISMA LÍNEA EN EL SCRIPT**

3. Variables

Para crear y darle valor a una variable:

```
nombrevariable=valor  
nombrevariable="valorString" #tambien se pueden hacer string sin comillas
```

⚠ **NO PUEDE HABER ESPACIOS entre el nombre variable, = y valor de variable.**


```
$nombrevariable #para acceder (y no colisionar con otras)
```

Para acceder al valor debe preceder por el símbolo de dolar \$

```
echo $nombrevariable
```

En la terminal las variables no son persistentes (por ventana), se pierden al cerrar.

⚠ Double quotes "\$nombreVariable" imprime el valor pero con single quotes imprime el nombre de variable (con \$), es decir

⚠ Las variables son CASE SENSITIVE (diferencia mayúsculas y minúsculas) 

En mayúsculas suelen ser del sistema por que NO se recomienda crearlas así.

Para imprimir TODAS las variables del sistema:

```
env
```

Para imprimir una del sistema:

- echo \$HOME
- echo \$USER

Subshell

Un subshell es un nuevo shell sólo para ejecutar un programa. Un subshell puede acceder a las variables globales establecidas por el 'shell padre' pero no a las variables locales. Se usa también para acceder a comandos del sistema [Más información](#)

```
files=$(ls) #output del comando (incluido strings) se guarda en la  
variable
```

```
date #imprime hora del sistema  
variable=$(date)
```

4. Operadores

Comparadores

! : invierte la comparación
-eq : equal
-ne : not equal
-gt : greater than
-lt : lower than
= : para strings
== : para strings (específico de bash)

Operadores lógicos:

-a : AND
-o : OR

5. Math

expr: evalúa una expresión. Es necesario preceder la operación con **expr** para realizar una operación matemática.

⚠ Ojo con los espacios

```
expr $var1 + $var2
```

```
expr $var1 \* $var2 #realiza la multiplicación. \ es necesario para "escapar" el comodín
```

División

<https://www.linuxjournal.com/content/mastering-division-variables-bash>

```
x=20 y=5  
expr $x / $y  
echo $((x / y)) #usando bash arithmetic xpansion
```

⚠ Ojo a los espacio antes de la /

⚠ Por defecto la division trabaja SOLO con enteros

Para tener precisión decimal usar **printf** o el comando **bc**

```
x=10.5 y=-2  
echo "scale=4; $x / $y" | bc
```

6.if

```
if [ $variable -eq valoracomparar ]
then
    echo "la condicion es true"
else
    echo "la condicion es falsa"
fi
```

⚠ **Ojo a los espacios en la evaluación del if del principio y final**

Los corchetes (brackets) del if NO son SIEMPRE obligatorios. Al ponerlos se asume que se ejecuta **test**. Ejecuta **man test** para info

```
#!/bin/bash

mynum=300

if $mynum -gt 200 #OJO PARÉNTESIS PUEDE FUNCIONAR PERO CON ERROR
then
    echo "The condition is true."
else
    echo "The variable does not equal 200"
fi
```

```
#!/bin/bash

if [ -f ifscript.sh ] #aquí los [] SÍ son obligatorios!
then
    echo "el archivo existe"
else
    echo "el archivo no existe"
fi
```

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

```
command=/usr/bin/htop
```

```
if [ -f $command ]
```

```
then
```

```
    echo "$command está disponible. Ejecutar!HACKEANDO"
```

```
sleep 2 #confirmar que se ejecuta cuando esta instalado
```

```
else
```

```
    echo "$command no existe o no está instalado en el sistema"
```

```
    sudo apt update && sudo apt install -y htop #-y: asume yes
```

```
fi
```

```
$command #deberia confirmar la localizacion y ejecutarlo
```


7. Case statements

```
case expression in
    pattern1)
        # codigo a ejecutar si la expresion coincide con pattern1
        ;;
    pattern2)
        # codigo a ejecutar si la expresion coincide con pattern2
        ;;
    *)
        # code to execute if none of the above patterns match expression
        ;;
esac
```

8. While loops

```
#!/bin/bash

myvar=1

while [ $myvar -le 10 ]
do
    echo $myvar
    myvar=$(( $myvar +1 ))
    sleep 0.5
done
```

```
#!/bin/bash

while [ -f archivowhile ]
do
    echo "el archivo existe" #descomentar >> logfile.log
    sleep 2
done

echo "El archivo no existe. Exiting." #descomentar >> logfile.log
```

9. For loops

```
#!/bin/bash

for i in 1 2 3 4 5
#equivalente a
#for i in {1..5}
do
    echo $i
done
```

10. Argumentos

```
./nombrescript valor1 valor2 #pasa los valores dados a cada una de las variables según su posición
```

En el script la primera posición se asignará a la primera variable a **1**, mientras que el segundo a **2**, accediendo a ellos con el dólar **\$**

```
#!/bin/bash
echo "Has introducido como primer argumento $1"
echo "Has introducido como segundo argumento $2"
```

Script para calcular el numero de objetos de un directorio dado por como argumento

```
#!/bin/bash
lineas=$(ls -lh $1 | wc -l)

echo "Hay $((lineas-1)) objetos en el directorio $1."

#este script tiene un "bug", si no se le pasa argumento mostrará el
numero de archivos del directorio en que te encuentras (no debería),
pero no imprimirá el directorio. Ver abajo solución
```

Por ello debemos evaluar en el código que se han introducido el **número de argumentos** requeridos con **\$#**

```
#!/bin/bash
lineas=$(ls -lh $1 | wc -l)

if [ $# -ne 1 ] #numero de argumentos
then
    echo "El script requiere pasarle la ruta de directorio por argumento"
    echo "Prueba otra vez"
    exit 1
fi

echo "Hay $((lineas-1)) objetos en el directorio $1"
```

11. Debugging

Exit codes

```
echo $?
```

Al ejecutarlo tras un comando satisfactorio devuelve **0 (cero)**

Al ejecutarlo tras un comando satisfactorio devuelve **un valor que no sea 0** (cuyo número representa el tipo de error)

 **Ojo con usarlo tras un if-else porque siempre devolverá un cero (satisfactorio).**

exit (manual)

<https://www.geeksforgeeks.org/how-to-use-exit-code-to-read-from-terminal-from-script-and-with-logical-operators/>

Set

set -x

Esta opción activa el modo de **depuración**, que hace que Bash imprima en el terminal cada comando que ejecuta, precedido por un **signo +**.

set -e

Hará que Bash muestre un error si falla cualquier comando del script, lo que facilita la identificación y corrección de errores en el script.

set -u

El script falla si existe una variable no inicializada

Set permite realizar muchas más acciones. Más información:

<https://www.reiser.cl/2016/03/04/bash-la-orden-set-script/>

<https://www.fpgenred.es/GNU-Linux/set.html>

12. Data Streams

standard output (stdout)

Se denota con: 1

```
find /etc -type f 1> find_resultok.txt
```

Si no se escribe el 1 es implícito (ie. lo pone por defecto)

standard error (stderr)

Se denota con: 2

```
find /etc -type f 2> find_errors.txt
```

Los dos de arriba se podrían escribir en una sola línea

```
find /etc -type f 1> find_resultok.txt 2>find_errors.txt
```

Para ambos se usa: &

```
find /etc -type f &> file.txt #notar que permite dividir entre ambos
```

standard input

```
read variable
```

- Para añadir texto en la misma línea y omitir *echo* añadir opción **-p**
- Para ocultar caracteres (útil para contraseñas) añadir opción: **-s**

```
#!/bin/bash
read -p "Usuario: " usuario
read -sp "Bienvenido, $usuario, introduce tu contraseña: " password
echo -e "\nEl usuario es $usuario y la contraseña es $password"
```

13. Filesystem localización para Scripts

```
echo $PATH
```

Salida

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:  
/usr/local/games:/snap/bin
```

Localización recomendable para los scripts

```
/usr/local/bin/nombrescript
```

Se recomienda cambiarle el owner y grupo a root

Al realizar esto, si se interroga por el script en el shell con **which nombrescript**, devolverá su localización.

Localización recomendable de Logs

```
/var/log
```

14. Funciones

```
#!/bin/bash

# Function
saludar() {
    local name="$1" # Capturar el primer argumento pasado a la funcion
    echo "Hola, $nombre!"
}

# Leer el nombre introducido por el usuario
read -p "Introduce tu nombre: " nombre

# Llamar a la función con la variable como argumento
saludar "$nombre" #notar que la llamada NO lleva paréntesis
```

15. Scheduling

at

```
sudo apt install at #instalacion
```

Script a ejecutar

```
#!/bin/bash
logfile=log_results.log

echo "The script se ejecuto en la siguiente fecha: $date" > $logfile
#la linea de arriba solo crea el archivo con el mensaje de echo y la fecha
```

Programar el scheduling en la consola

```
at 11:15 -f ./nombrescript.sh #confirmar si el script corrio (ie., se creó el archivo)
```

```
at 10:00 123024 -f ./nombrescript.sh #para dar una fecha. Ojo al formato de fecha que es americano
```

Para ver trabajos en cola: **atq**

Para borrar un trabajo en cola: **atrm numeroEnlaCola** #valor de la izquierda

16. Otros recursos

<https://www.freecodecamp.org/news/bash-scripting-tutorial-linux-shell-script-and-command-line-for-beginners/>

Cheat Sheet

<https://devhints.io/bash>

17. Comandos útiles

`sleep tiempo`

`cron`

`date`

`-f`: existencia de un archivo/file

`-d`: existencia de un directorio

`-y`: opción para asumir yes (que acepte los prompts y el script continúe automáticamente)

`set`