

Scripts en bash

¿Qué es un *script*?

Cuando hablamos de *script* sencillamente estamos tratando una secuencia de comandos que se ejecutan de forma ordenada y que cumplen una función en específico.

Conforman una manera sencilla de automatizar tareas y generar información simplificando la interacción del usuario.

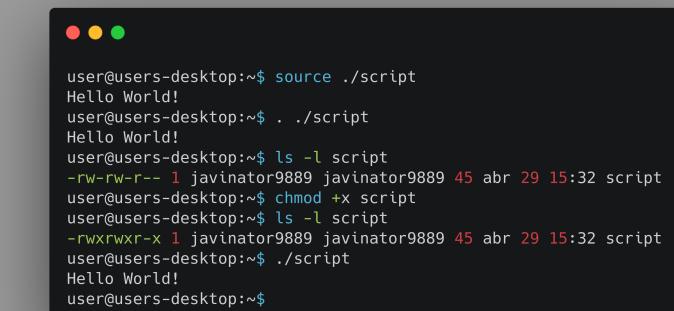
Al final, un *script* no es más que un texto en ASCII puro al cual le damos un significado según lo que pongamos en la cabecera y en su interior.

```
#!/bin/env bash
# Esto conforma el "preludio", donde se especifica
# qué tipo de fichero es.
#
# Se podría poner directamente /bin/bash,
# /bin/sh, etc. Pero se usa "env" para adaptar
# el script a la configuración del equipo (ya que
# la ubicación de los comandos puede cambiar)

# Añadimos diversas sentencias que conformarían la ejecución
# del programa
echo "Hello World!"

# Finalizamos indicando el código de fin. '0' es que todo
# ha ido bien, y cualquier otro valor no. Se puede usar
# para notificar a otras aplicaciones si han habido errores.
exit 0
```

Scripts en bash



A screenshot of a terminal window on a Mac OS X desktop. The window has red, yellow, and green title bar buttons. The terminal shows the following session:

```
user@users-desktop:~$ source ./script
Hello World!
user@users-desktop:~$ . ./script
Hello World!
user@users-desktop:~$ ls -l script
-rw-r--r-- 1 javinator9889 javinator9889 45 abr 29 15:32 script
user@users-desktop:~$ chmod +x script
user@users-desktop:~$ ls -l script
-rwxrwxr-x 1 javinator9889 javinator9889 45 abr 29 15:32 script
user@users-desktop:~$ ./script
Hello World!
user@users-desktop:~$
```

Sin embargo, definir únicamente el contenido en un fichero no sirve: eso solo es un fichero en texto plano.

Hace falta darle permisos de ejecución al *script* anterior, ya que en otro caso no será un programa. Sin embargo, dar permisos de ejecución puede ser “peligroso”. ¿Cómo se pueden depurar los *scripts* para evitar poner en riesgo el sistema?

Utilizando el comando **source** (o **.**) se puede ejecutar un programa que no tenga permisos de ejecución, siendo la opción a elegir para probar una aplicación antes de darle los permisos de ejecución definitivos.

Hay que tener en cuenta que, si ponemos un “**exit**”, al ejecutar la aplicación con “**source**” cerrará el terminal al finalizar.

En las pruebas que hemos realizado antes hemos puesto “Hello World!” entre comillas dentro del script, pero... ¿son necesarias?

Probad a ejecutar los siguientes comandos:

```
echo esto es un asterisco * sin comillas
```

```
echo esto es un dolar y tres letras $ABC sin comillas
```

```
echo "esto es un asterisco * entre comillas dobles"
```

```
echo 'esto es un asterisco * entre comillas simples'
```

```
echo "esto es un dolar y tres letras $ABC entre comillas dobles"
```

```
echo 'esto es un dolar y tres letras $ABC entre comillas simples'
```

¿Qué sucede?

En bash (y otras consolas basadas en Bourne Shell) existen caracteres especiales los cuales se conocen como *shell expansions*.

Ya hemos trabajado con ellos con relativa frecuencia (por ejemplo, las llaves '{' y '}'), pero vamos a listar algunos de ellos:

- { } → generador de cadenas de texto según lo escrito.
- ~ → directorio *home*.
- \${ } → expansión de parámetros.
- \$(cmd) → sustitución de comandos.
- \$[expr] → expansión aritmética.
- *? → expansión de ficheros.

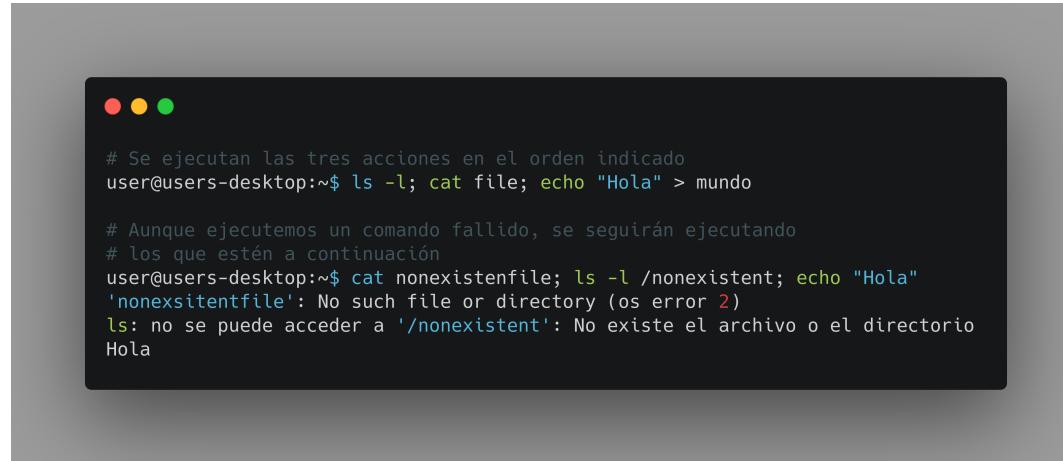
```
user@users-desktop:~$ echo a{b,c,d}e
abe ace ade
user@users-desktop:~$ echo ~
/home/user
user@users-desktop:~$ string=1234567abcdef
user@users-desktop:~$ echo ${string}
1234567abcdef
user@users-desktop:~$ echo ${string:7}
abcdef
user@users-desktop:~$ $(pwd)
user@users-desktop:~$ wami=$(pwd)
user@users-desktop:~$ echo $wami
/home/user
user@users-desktop:~$ echo $[ 5 + 3 ]
8
user@users-desktop:~$ echo $[ 17 / 3 ]
5
user@users-desktop:~$ echo *
Desktop Documents Downloads Music Videos Pictures
user@users-desktop:~$ echo D?*
Desktop Documents Downloads
```

https://www.gnu.org/software/bash/manual/html_node/Shell-Expansions.html

Por otra parte, existen formas de ejecutar comandos en una misma línea. El operador que se usa es ; el cual sirve de delimitador.

Es importante tener en cuenta que el orden de las operaciones se respeta siempre (estamos hablando de un operador secuencial). Sin embargo, los comandos se van a ejecutar siempre indiferentemente de si uno falla o no.

Si se quieren ejecutar comandos de forma condicional se utilizan otros operadores, como veremos a continuación.

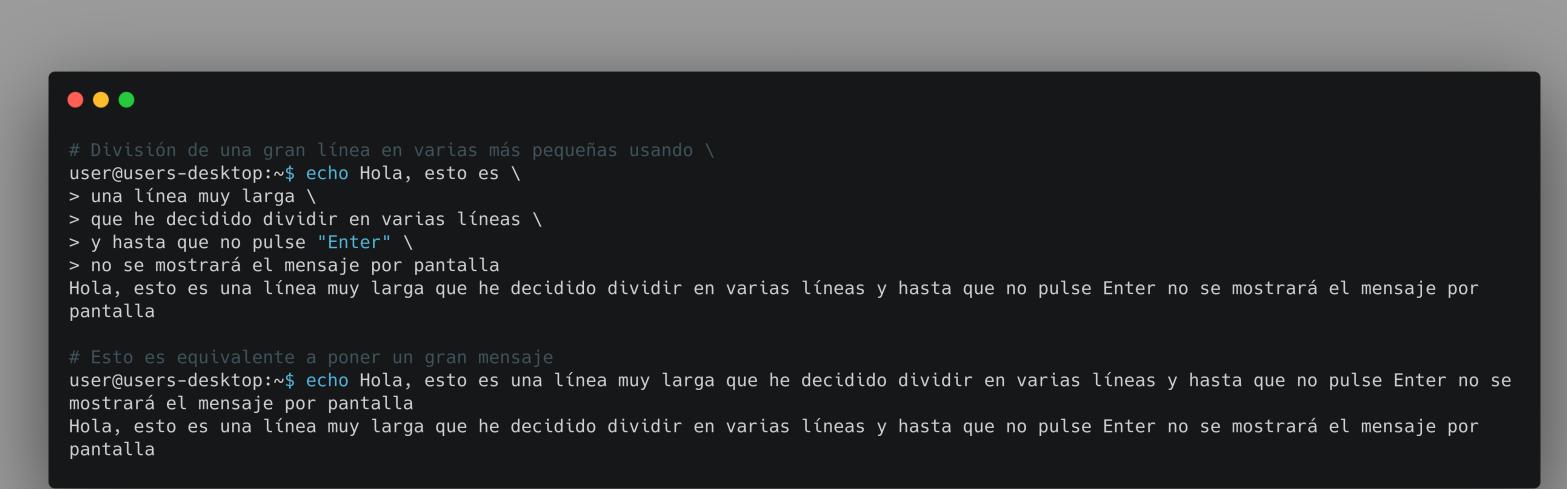


```
# Se ejecutan las tres acciones en el orden indicado
user@users-desktop:~$ ls -l; cat file; echo "Hola" > mundo

# Aunque ejecutemos un comando fallido, se seguirán ejecutando
# los que estén a continuación
user@users-desktop:~$ cat nonexistentfile; ls -l /nonexistent; echo "Hola"
'nonexistentfile': No such file or directory (os error 2)
ls: no se puede acceder a '/nonexistent': No existe el archivo o el directorio
Hola
```

Como “antítesis” de lo anterior, es posible dividir una línea bastante larga en varias líneas más pequeñas. Si el ; servía para separar instrucciones en una única línea, el operador \ permite dividir sentencias muy largas en múltiples líneas.

Es importante tener en cuenta que, a nivel de comando, es exactamente lo mismo que si hubiésemos escrito una única línea larga:



```
# División de una gran línea en varias más pequeñas usando \
user@users-desktop:~$ echo Hola, esto es \
> una línea muy larga \
> que he decidido dividir en varias líneas \
> y hasta que no pulse "Enter" \
> no se mostrará el mensaje por pantalla
Hola, esto es una línea muy larga que he decidido dividir en varias líneas y hasta que no pulse Enter no se mostrará el mensaje por pantalla

# Esto es equivalente a poner un gran mensaje
user@users-desktop:~$ echo Hola, esto es una línea muy larga que he decidido dividir en varias líneas y hasta que no pulse Enter no se
mostrará el mensaje por pantalla
Hola, esto es una línea muy larga que he decidido dividir en varias líneas y hasta que no pulse Enter no se mostrará el mensaje por
pantalla
```

Variables y expansiones

Variables y expansiones

Las variables en los *scripts* de bash son muy simples: es necesario definir únicamente el nombre de la variable y asignar un valor.

No es necesario en ningún momento declarar la variable antes de usarla y se considera que el valor asignado siempre es alfanumérico, esto es, una cadena de caracteres.

Esto produce que ciertas operaciones no “funcionen” como se espera en un inicio ya que serán operaciones sobre cadenas de texto.

La situación anterior se puede apreciar en el ejemplo:

```
#!/bin/bash

# Creamos una variable cuyo contenido es
# "Hola mundo"
DECIR="Hola mundo"

# Con $ accedemos directamente al contenido
echo $DECIR

# Creamos una nueva variable con el valor '4'
NUMERO=4
# y mostramos por pantalla la suma de dicha variable
# con el '3'. ¿O no?
echo $NUMERO+3
#####
user@users-desktop:~$ chmod +x script
user@users-desktop:~$ ./script
Hola mundo
43
```

Variables y expansiones

```
# Demostraciones varias de cómo usar $()
user@users-desktop:~$ echo pwd
pwd
# Como no hemos usado la expansión $(cmd), "echo"
# repite lo que hemos puesto a continuación
user@users-desktop:~$ echo $(pwd)
/home/user
# Creamos una nueva variable cuyo contenido será:
# red + fecha actual en formato DDMYY + .conf
# Por ejemplo: red130521.conf
user@users-desktop:~$ IF_TODAY=$(date +%d%m%y).conf"
# Podemos usar directamente la variable en otro comando
user@users-desktop:~$ cp -v /etc/network/interfaces $IF_TODAY
'/etc/network/interfaces' -> 'red130521.conf'
# También se puede usar la expansión $(cmd) directamente
# en un comando, sin necesidad de crear una variable. Sin
# embargo, se considera una buena práctica ya que en caso
# de cambio solo es necesario editar la variable en sí.
# Además, es más sencillo de depurar.
user@users-desktop:~$ echo "Hay $(find ~ -iname \"*.sh\" | wc -l) scripts en $HOME"
Hay 1291 scripts en /home/user
```

Sin embargo, crear variables cuyo contenido es fijo (constantes) no tiene mucho sentido, ya que para ello se puede sustituir directamente (a no ser que se quieran usar para no repetir código).

Una idea muy interesante es el usar variables para guardar la salida de un comando y poder usarla más adelante (similar al mecanismo de una tubería).

El punto de usar variables es que se cuenta con una mayor flexibilidad y, sobre todo, la capacidad de reutilizar el contenido.

El operador que se usa para ello es `$(cmd)` el cual, internamente, genera una “*mini-shell*” para realizar la ejecución.

Variables y expansiones

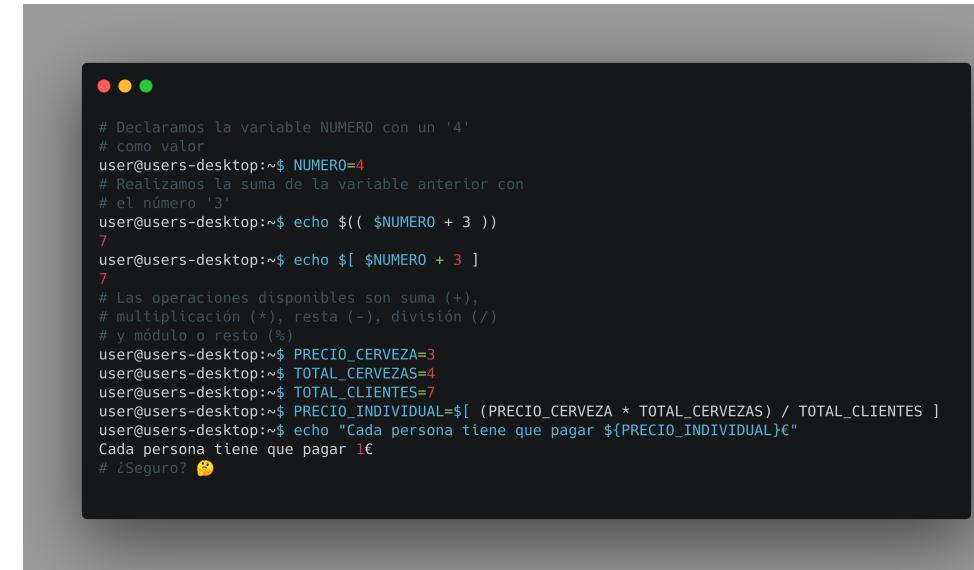
En uno de los ejemplos anteriores vimos que no era posible realizar una suma de dos números, sino que bash concatenaba uno con otro. Esto se debe a lo que se ha comentado anteriormente de que todo se trata como si fuesen cadenas de texto.

Para poder hacer operaciones aritméticas es necesario hacer uso del operador:

`$((expr)) o ${[expr]}`

Ambos son exactamente iguales, únicamente cambia la forma de escribirlos.

Sin embargo, es importante tener en cuenta que bash no respeta los decimales sino que las operaciones son siempre enteras.



```
# Declaramos la variable NUMERO con un '4'  
# como valor  
user@users-desktop:~$ NUMERO=4  
# Realizamos la suma de la variable anterior con  
# el número '3'  
user@users-desktop:~$ echo $(( $NUMERO + 3 ))  
7  
user@users-desktop:~$ echo ${[ $NUMERO + 3 ]}  
7  
# Las operaciones disponibles son suma (+),  
# multiplicación (*), resta (-), división (/)  
# y módulo o resto (%)  
user@users-desktop:~$ PRECIO_CERVEZA=3  
user@users-desktop:~$ TOTAL_CERVEZAS=4  
user@users-desktop:~$ TOTAL_CLIENTES=7  
user@users-desktop:~$ PRECIO_INDIVIDUAL=${(PRECIO_CERVEZA * TOTAL_CERVEZAS) / TOTAL_CLIENTES}  
user@users-desktop:~$ echo "Cada persona tiene que pagar ${PRECIO_INDIVIDUAL}€"  
Cada persona tiene que pagar 1€  
# ¿Seguro? 🤔
```

Variables y expansiones

```
● ● ●

# Declaramos la variable NUMERO con un '4'
# como valor
user@users-desktop:~$ NUMERO=4
# Realizamos la suma de la variable anterior con
# el número '3'. Esta vez, sin embargo, guardamos
# directamente su contenido en una variable
user@users-desktop:~$ let SUMA=NUMERO+3
# Mostramos por pantalla el resultado
user@users-desktop:~$ echo $SUMA
7

# Script sencillo para calcular la media
user@users-desktop:~$ cat notas.sh
#!/bin/bash
read -p '¿Qué nota has sacado en ISO?: ' NOTAISO
read -p '¿Qué nota has sacado en PAR?: ' NOTAPAR
read -p '¿Qué nota has sacado en BD?: ' NOTABD
let SUMA=NOTAISO+NOTAPAR+NOTABD
let MEDIA=SUMA/3
echo "La media es $MEDIA"
# En esta ejecución, sale que la media entera es '8'
user@users-desktop:~$ ./notas.sh
¿Qué nota has sacado en ISO?: 10
¿Qué nota has sacado en PAR?: 7
¿Qué nota has sacado en BD?: 9
La media es 8
# Nueva versión del script que usa "bc" para hacer
# el cálculo con decimales
user@users-desktop:~$ tail -n3 notas-ok.sh
let SUMA=NOTAISO+NOTAPAR+NOTABD
MEDIA=$(echo "scale=4; $SUMA/3" | bc -l)
echo "La media es $MEDIA"
# En esta nueva ejecución, sale que la media es
# 8.6666
user@users-desktop:~$ ./notas-ok.sh
¿Qué nota has sacado en ISO?: 10
¿Qué nota has sacado en PAR?: 7
¿Qué nota has sacado en BD?: 9
La media es 8.6666
```

Si el resultado de una operación aritmética se va a asignar a una variable se puede utilizar directamente la sentencia “`let`” delante del nombre de la variable:

`let SUMA=NUMERO+3`

Con la expresión anterior directamente se guardaría en `SUMA` el resultado de hacer `NUMERO + 3`.

Si quisiéramos que bash respetase los decimales es necesario utilizar un comando adicional. Dicho comando es `bc` (*basic calculator*) y permite trabajar con una precisión arbitraria.

En el [manual](#) hay más información sobre su uso.

Funciones

Funciones

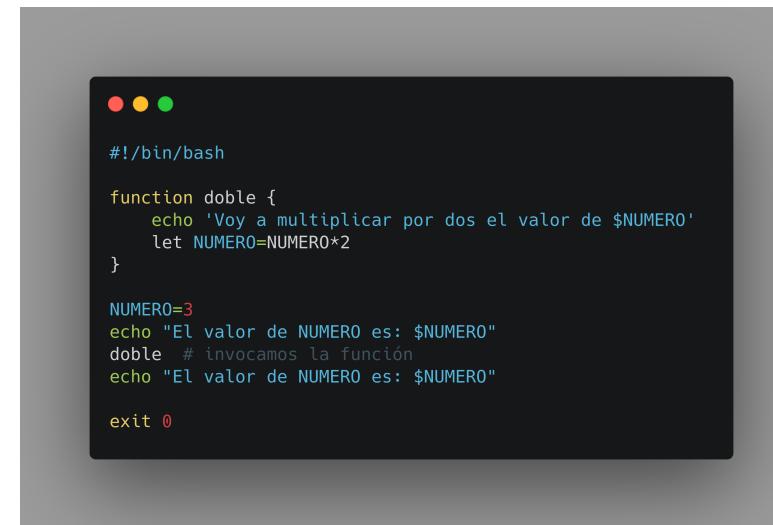
Las funciones son un mecanismo muy útil que existe a la hora de crear código y evitar repeticiones. En general, es un concepto que existe en todos los lenguajes de programación y que resulta útil a la hora de agrupar código.

Por lo general, la estructura de una función en bash es la siguiente:

```
function <nombre> {  
    código  
}
```

O bien:

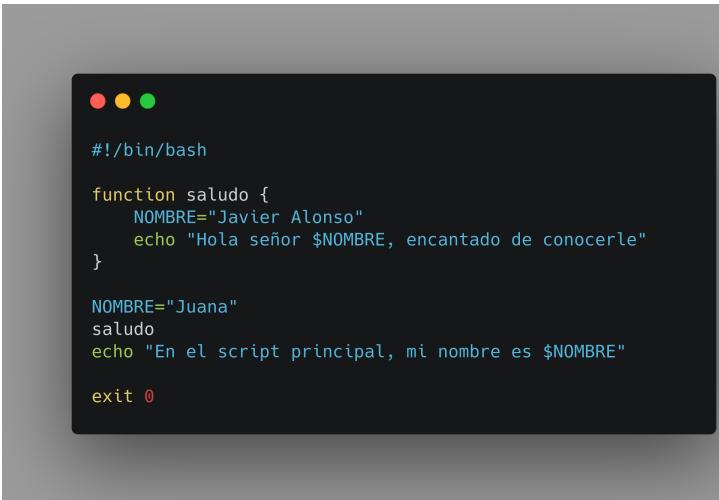
```
<nombre>() {  
    Código  
}
```



A screenshot of a terminal window on a Mac OS X system (indicated by the red, yellow, and green window control buttons). The terminal displays the following Bash script code:

```
#!/bin/bash  
  
function doble {  
    echo 'Voy a multiplicar por dos el valor de $NUMERO'  
    let NUMERO=NUMERO*2  
}  
  
NUMERO=3  
echo "El valor de NUMERO es: $NUMERO"  
doble # invocamos la función  
echo "El valor de NUMERO es: $NUMERO"  
  
exit 0
```

Funciones



A screenshot of a terminal window on a Mac OS X desktop. The window has the classic red, yellow, and green title bar buttons. The terminal itself is black with white text. It contains the following Bash script:

```
#!/bin/bash

function saludo {
    NOMBRE="Javier Alonso"
    echo "Hola señor $NOMBRE, encantado de conocerle"
}

NOMBRE="Juana"
saludo
echo "En el script principal, mi nombre es $NOMBRE"

exit 0
```

The script defines a function `saludo` that prints a greeting using the variable `NOMBRE`. It then sets `NOMBRE` to "Juana", calls the `saludo` function, and prints the value of `NOMBRE` again. Finally, it exits with status 0.

¿Qué ha sucedido en el ejemplo anterior?
¿Por qué el valor de `$NUMERO` ha cambiado si no estaba declarado?

Hay que tener en cuenta que, pese a que el *script* tiene una función la cual accede a una variable que todavía no ha sido declarada, el contenido del mismo solo será procesado en el momento en que llamemos a la función, es decir, cuando pongamos doble.

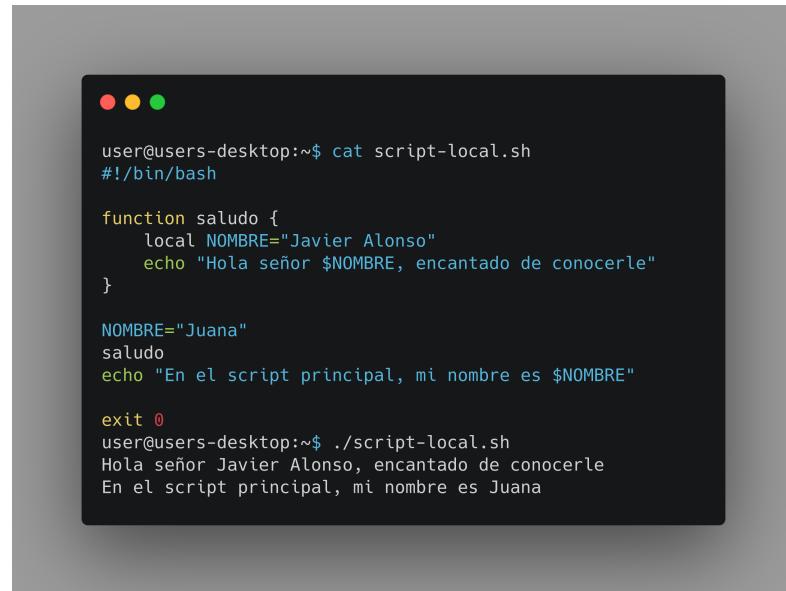
Sin embargo, ¿qué ocurre en el ejemplo de la izquierda? ¿Cuál será la salida producida? ¿Y el valor de la variable `NOMBRE` al final del *script*?

Funciones

En efecto, las variables en un *script* siempre son globales, esto es, son accesibles desde cualquier punto del fichero.

De esta forma, en los ejemplos anteriores pese a que las variables pudieran no estar declaradas como durante la ejecución del programa las hemos creado, automáticamente son accesibles desde el resto del *script*.

Pese a que esto pueda parecer una ventaja, tiene el problema de que “se ensucia” el *script* por la cantidad de variables globales que existen. Sin embargo, dentro de una función se puede usar la keyword *local* que permite que esa variable sea únicamente accesible desde dicha función.



The screenshot shows a terminal window with a dark background and light-colored text. It displays the contents of a file named 'script-local.sh'. The script defines a function 'saludo' that prints a greeting using a local variable 'NOMBRE'. It then sets 'NOMBRE' to 'Juana' and calls the 'saludo' function. Finally, it exits with status 0. When run, the script prints the expected greeting and the value of 'NOMBRE' from the environment.

```
user@users-desktop:~$ cat script-local.sh
#!/bin/bash

function saludo {
    local NOMBRE="Javier Alonso"
    echo "Hola señor $NOMBRE, encantado de conocerle"
}

NOMBRE="Juana"
saludo
echo "En el script principal, mi nombre es $NOMBRE"

exit 0
user@users-desktop:~$ ./script-local.sh
Hola señor Javier Alonso, encantado de conocerle
En el script principal, mi nombre es Juana
```

Funciones

```
./script.sh ARG1 ARG2 ARG3 ARG4 ARG5 ARG6 ARG7 ARG8 ARG9 ARG10
↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓
$0 $1 $2 $3 $4 $5 $6 $7 $8 $9 ${10}
Tecadmin.net
```

La utilidad de las funciones es evidente: agrupar código y re-usarlo cuando es necesario. Sin embargo, no es del todo útil que las variables tengan valores fijos o dependan del resto del problema, sino que lo interesante sería realizar una misma acción con distintos parámetros.

Es lo que permite la función conocida como “**paso de parámetros**”. Esto sirve tanto para funciones como para *scripts* en sí y es una característica muy útil si queremos variar la ejecución según el parámetro que se establezca.

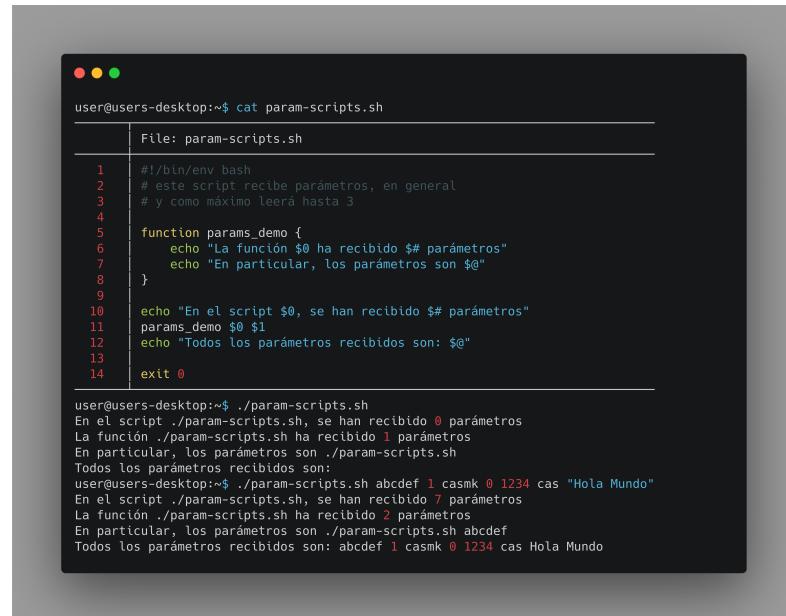
El acceso a los parámetros se realiza mediante el ‘\$’, donde \$0 se refiere al script/función en sí; \$@ son todos los argumentos y \$# es la cantidad de argumentos

Funciones

Lo anterior no aplica únicamente a *scripts* en general, que ya da mucho juego, sino también a funciones en particular.

El uso es exactamente igual que con un *script* pero escribiendo sencillamente el nombre de una función.

En principio, se pueden pasar hasta 9 argumentos a una función (accediendo a ellos con la sintaxis `$1 .. $9`). Sin embargo, si se quieren usar más parámetros todavía será necesario usar la sintaxis `${10} .. ${99999}` para ello. Hay que tener en cuenta que una mayor cantidad de parámetros implica que el programa sea más pesado, pudiendo llegar a “fallar” en un momento dado.



```
user@users-desktop:~$ cat param-scripts.sh
File: param-scripts.sh
1 #!/bin/env bash
2 # este script recibe parámetros, en general
3 # y como máximo leerá hasta 3
4
5 function params_demo {
6     echo "La función $0 ha recibido $# parámetros"
7     echo "En particular, los parámetros son $@"
8 }
9
10 echo "En el script $0, se han recibido $# parámetros"
11 params_demo $0 $1
12 echo "Todos los parámetros recibidos son: $@"
13
14 exit 0

user@users-desktop:~$ ./param-scripts.sh
En el script ./param-scripts.sh, se han recibido 0 parámetros
La función ./param-scripts.sh ha recibido 1 parámetros
En particular, los parámetros son ./param-scripts.sh
Todos los parámetros recibidos son:
user@users-desktop:~$ ./param-scripts.sh abcdef 1 casmk 0 1234 cas "Hola Mundo"
En el script ./param-scripts.sh, se han recibido 7 parámetros
La función ./param-scripts.sh ha recibido 2 parámetros
En particular, los parámetros son ./param-scripts.sh abcdef
Todos los parámetros recibidos son: abcdef 1 casmk 0 1234 cas Hola Mundo
```

Funciones

```
user@users-desktop:~$ cat return-values.sh
File: return-values.sh
1 #!/bin/env bash
2
3 function try_access {
4     echo "Intentando acceder a un directorio que no existe..."
5     cd /tmp/nonexistent
6     echo "El valor de retorno es: $?"
7     cd ~
8     return $?
9 }
10
11 echo "Se llama a la función 'try_access' para ver su valor de retorno"
12 try_access
13 echo "La última llamada de la función ha devuelto un $?"
14 exit 0

user@users-desktop:~$ ./return-values.sh
Se llama a la función 'try_access' para ver su valor de retorno
Intentando acceder a un directorio que no existe...
./return-values.sh: línea 5: cd: /tmp/nonexistent: No existe el archivo o el directorio
El valor de retorno es: 1
La última llamada de la función ha devuelto un 0
user@users-desktop:~$ echo $?
0
```

Finalmente, en el flujo de ejecución de un *script* se tienen en cuenta siempre lo que se conocen como “*return values*” o valor de retorno.

Por defecto, se consideran los siguientes estados:

- 0, cuando la ejecución de una orden o *script* ha sido un éxito.
- ≥ 1 ó $\neq 0$, cuando la ejecución de una orden o *script* ha fallado.

Es fundamental que los *scripts* o funciones devuelvan siempre un valor. En una función, esto se indica con “*return*”; en un *script*, se usa “*exit*”.

Se puede acceder al valor de retorno con \$?.

Funciones