SHELL SCRIPTS

QUÉ ES EL SHELL

- Intérprete de comandos del sistema.
- Interfaz de texto de altas prestaciones que sirve fundamentalmente para tres cosas:
  - **Administrar el sistema operativo**
  - **Lanzar aplicaciones e interactuar con ellas**
  - **Como entorno de programación**

QUÉ ES UN SCRIPT

- Fichero que contiene una secuencia de comandos
- Contiene estructuras de control: sentencias condicionales, bucles, funciones
- Permite automatizar tareas de administración repetitivas
- Para ser ejecutado → permiso de ejecución

CONSIDERACIONES

- Si el directorio en el que se encuentra el shell script está incluido en la variable PATH, el Shell script podrá ser ejecutado desde cualquier directorio simplemente tecleando su nombre

```
$ miscript
```

- Si el directorio en el que se encuentra el script no está en la variable PATH, para ejecutarlo es necesario:
  - Escribir un punto separado del nombre del script por un espacio
  - Indicar la ruta del script (absoluta o relativa)

```
$ . /home/alumno/Documentos/miscript
```

- Si la variable PATH contiene el símbolo "." →podremos ejecutar directamente (sin incluir el punto) los scripts situados en el directorio actual. Pero cuando vayamos a ejecutar scripts fuera del directorio actual actuaremos como en el caso anterior.
- Otra forma de ejecutar el script, sin necesidad de ponerle permisos de ejecución es llamar al intérprete de comandos y darle como parámetro el nombre del script

```
$ bash /home/alumno/Documentos/miscript
```

- Todos los scripts en bash deben comenzar con la línea:

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

ENTRADA Y SALIDA DE DATOS

- ENTRADA DE DATOS:
  - Entrada estándar: teclado
  - Redirección de entrada “<“
  - Comando read

- SALIDA DE DATOS:
  - Salida estándar: pantalla
  - Redirección de salida “ >”, ”>>”
  - Comando echo
  - Comando printf

alumnos.sh

```
#!/bin/bash
echo "Introduce el apellido"
read ape
echo "Introduce tu nombre"
read nom
echo "Te llamas $nom $ape"
```

```
[alumno@localhost ~]$ ./alumnos.sh
Introduce el apellido
Romero
Introduce tu nombre
Rosa
Te llamas Rosa Romero
[alumno@localhost ~]$
```

alumnos2.sh

```
#!/bin/bash
printf "Introduce el apellido"
read ape
printf "Introduce tu nombre"
read nom
printf "Te llamas $nom $ape"
```

```
[alumno@localhost ~]$ chmod u+x alumnos2.sh
[alumno@localhost ~]$ ./alumnos2.sh
Introduce el apellidoRomero
Introduce tu nombreRosa
Te llamas Rosa Romero[alumno@localhost ~]$
```

PARÁMETROS Y ARGUMENTOS

- 2 categorías:
  - parámetros posicionales
  - parámetros especiales

PARÁMETROS POSICIONALES

- Son simplemente los argumentos pasados al script cuando es invocado.
- Se almacenan en las variables reservadas 1,2,3,...9,10,11,... y pueden ser llamados con las expresiones $1,$2...${10},${11}...
- Nota: El shell Bourne está limitado a los parámetros del 0 al 9.

PARÁMETROS ESPECIALES

- $0→ nombre del script o programa
- $#→ nº de parámetros pasados al script
- $*→ muestra una lista con todos los parámetros en formato "$1 $2 …" menos $0
- $@ lista de todos los parámetros en forma de elementos distintos:"$1" "$2"… menos $0
- $$→ PID del proceso
- $!→ PID del último proceso ejecutado en 2º plano
- $?→ Salida del último proceso ejecutado

param.sh

```
#!/bin/bash
echo "Nombre: $0"
echo "Número de parámetros: $#"
echo "parámetros: 1=$1 2=$2 3=$3"
echo "Lista: $*"
echo "Elementos: $@"
```

```
[alumno@localhost ~]$ ./param.sh casa perro gato
Nombre: ./param.sh
Número de parámetros: 3
parámetros: 1=casa 2=perro 3=gato
Lista: casa perro gato
Elementos: casa perro gato
```

INICIALIZAR PARÁMETROS

Es posible pasar directamente parámetros al shell gracias al comando **set**. Un simple comando como:

```
$ set param1 param2 param3
```

Inicializará automáticamente los parámetros posiciónales "$1,$2,$3" con los valores "param1,param2,param3", borrando de este modo los antiguos valores si existieran.

EJEMPLO

param2.sh

```
#!/bin/bash
echo "Situación inicial"
echo "Nombre: $0"
echo "Número de parámetros: $#"
echo "parámetros: 1=$1 2=$2 3=$3"
echo "Lista: $*"
echo "Elementos: $@"
echo "Si redefinimos dos de las variables anteriores"
set coche moto
echo "Situación actual"
echo "Número de parámetros: $#"
echo "parámetros: 1=$1 2=$2 3=$3"
echo "Lista: $*"
echo "Elementos: $@"
```

```
[alumno@localhost ~]$ chmod u+x param2.sh
[alumno@localhost ~]$ ./param2.sh casa perro gato
Situación inicial
Nombre: ./param2.sh
Número de parámetros: 3
parámetros: 1=casa 2=perro 3=gato
Lista: casa perro gato
Elementos: casa perro gato
Si redefinimos dos de las variables anteriores
Situación actual
Número de parámetros: 2
parámetros: 1=coche 2=moto 3=
Lista: coche moto
Elementos: coche moto
```

EL COMANDO SHIFT

- El comando interno "**shift**" permite desplazar los parámetros.
- El valor del 1er parámetro ($1) es reemplazado por el valor del 2º parámetro ($2), el del 2º parámetro ($2) por el del 3º parámetro ($3), …
- Es posible indicar como argumento (**shift n**) el número de posiciones que hay que desplazar los parámetros.

EJEMPLO

param3.sh

```
#!/bin/bash
echo "Asignamos valores a los parámetros"
set rojo amarillo verde
echo "valores: rojo, amarillo y verde"
echo ""
echo "Número de parámetros: $#"
echo "parámetros: 1=$1 2=$2 3=$3"
echo "Lista: $*"
echo "Elementos: $@"
echo "Si añdimos un shift"
shift
echo "Situación actual"
echo "Número de parámetros: $#"
echo "parámetros: 1=$1 2=$2 3=$3"
echo "Lista: $*"
echo "Elementos: $@"
```

```
[alumno@localhost ~]$ ./param3.sh
Asignamos valores a los parámetros
valores: rojo, amarillo y verde

Número de parámetros: 3
parámetros: 1=rojo 2=amarillo 3=verde
Lista: rojo amarillo verde
Elementos: rojo amarillo verde
Si añdimos un shift
Situación actual
Número de parámetros: 2
parámetros: 1=amarillo 2=verde 3=
Lista: amarillo verde
Elementos: amarillo verde
[alumno@localhost ~]$
```

param3b.sh

```
#!/bin/bash
echo "Asignamos valores a los parámetros"
set rojo amarillo verde azul negro
echo "valores: rojo, amarillo, verde, azul y negro"
echo "--"
echo "Número de parámetros: $#"
echo "parámetros: 1=$1 2=$2 3=$3"
echo "Lista: $*"
echo "Elementos: $@"
echo "--"
echo "Si añdimos un shift de 2 posiciones"
shift 2
echo "Situación actual"
echo "Número de parámetros: $#"
echo "parámetros: 1=$1 2=$2 3=$3"
echo "Lista: $*"
echo "Elementos: $@"
```

```
[alumno@localhost ~]$ chmod u+x param3b.sh
[alumno@localhost ~]$ ./param3b.sh
Asignamos valores a los parámetros
valores: rojo, amarillo, verde, azul y negro
--
Número de parámetros: 5
parámetros: 1=rojo 2=amarillo 3=verde
Lista: rojo amarillo verde azul negro
Elementos: rojo amarillo verde azul negro
--
Si añdimos un shift de 2 posiciones
Situación actual
Número de parámetros: 3
parámetros: 1=verde 2=azul 3=negro
Lista: verde azul negro
Elementos: verde azul negro
```

VARIABLES

- Están inicializadas a nulo (vacío)
- No hay tipos
- Para referirse a su contenido se usa $
- Declaración: nombre=valor *(sin espacios antes ni después del signo igual)*
- Se distinguen mayúsculas de minúsculas
- Ámbito local.

- Reglas para nombrar variables:
  - Se puede usar letras minúsculas, mayúsculas, cifras y caracteres de subrayado.
  - El primer carácter no puede ser una cifra.
  - El tamaño de un nombre suele ser ilimitado, aunque conviene no excederse.
  - Se recomienda que las variables de usuario se escriban en minúsculas, para diferenciarlas de las variables de sistema, que suelen ir en mayúsculas.
  - No puede contener caracteres especiales, particularmente el espacio

```
[alumno@localhost ~]$ variable1=casa azul
bash: azul: comando no encontrado...
[alumno@localhost ~]$ variable1="casa azul"
[alumno@localhost ~]$ echo $variable1
casa azul
[alumno@localhost ~]$ variable2=dividendo/divisor
[alumno@localhost ~]$ echo$ variable2
bash: echo$: comando no encontrado...
Un comando similar es: 'echo'
[alumno@localhost ~]$ variable2="dividendo/divisor"
[alumno@localhost ~]$ echo $variable2
dividendo/divisor
[alumno@localhost ~]$ variable3=multiplicando*multiplicador
[alumno@localhost ~]$ echo $variable3
multiplicando*multiplicador
```

```
[alumno@localhost ~]$ variable4="La casa azul"
[alumno@localhost ~]$ echo variable4
variable4
[alumno@localhost ~]$ variable5='La casa azul'
[alumno@localhost ~]$ echo$ variable5
bash: echo$: comando no encontrado...
Un comando similar es: 'echo'
[alumno@localhost ~]$ variable6="La casa azul'
>
```

```
[alumno@localhost ~]$ echo "Vas a ver la variable5:$variable5"
Vas a ver la variable5:La casa azul
[alumno@localhost ~]$ echo 'Vas a ver la variable5:$variable5'
Vas a ver la variable5:$variable5
```

SUPRESIÓN Y PROTECCIÓN

- Se suprime una variable con el comando **unset**. Si pedimos que devuelva el contenido, será una cadena vacía

```
[alumno@localhost ~]$ variable6="La casa azul"
[alumno@localhost ~]$ echo $variable6
La casa azul
[alumno@localhost ~]$ unset variable6
[alumno@localhost ~]$ echo $variable6

[alumno@localhost ~]$
```

- Se puede proteger una variable en modo escritura y contra su eliminación con el comando **readonly**.
  - Una variable en modo de sólo lectura, incluso vacía, es exclusiva. No existe ningún medio de sustituirla en escritura y de eliminarla, salvo saliendo del Shell.

```
[alumno@localhost ~]$ variable7=protegida
[alumno@localhost ~]$ readonly variable7
```

```
[alumno@localhost ~]$ variable7=otrovalor
bash: variable7: variable de sólo lectura
[alumno@localhost ~]$ unset variable7
bash: unset: variable7: no se puede borrar: variable es de solo lectura
```

EXPORTACIÓN

- Por defecto una variable es accesible únicamente desde el shell donde ha sido definida.
- El comando **`export`** permite exportar una variable de manera que su contenido sea visible por los scripts y otros subshells.
- Se pueden modificar las variables exportadas en el script, pero estas modificaciones sólo se aplican al script o al subshell

EJEMPLOS DE VARIABLES

- DE SISTEMA:
  - $HOME→ directorio raíz del usuario
  - $LOGNAME→ login del usuario que ha iniciado sesión
  - $PATH→ Lista de directorios donde el shell va a buscar los comandos, scripts y otros binarios.
  - $SHELL→ intérprete de comandos asignado
  - $PS1→ prompt del shell
  - $MAIL→ directorio de buzón de correo
  - $PWD→ directorio actual

EJEMPLOS DE VARIABLES

- ESPECIALES:
  - $? → Código de retorno del último comando ejecutado
  - $$ → PID del Shell activo
  - $! → PID del último proceso iniciado en segundo plano.
  - $ → Opciones del shell

COMANDOS TYPESET, EXPR Y LET

- **`typeset /declare`**→ Define variables y les da atributos. Opciones:
  - -i declara números enteros
  - -r para variables de solo lectura
  - –a para matrices o "arrays"
  - -f para funciones
  - -x para variables exportables
- **`expr`** →nos sirve para resolver expresiones matemáticas en la terminal. No usa decimales.
- **`let`** o **((...))** → nos permite trabajar con variables numéricas

```
[alumno@localhost ~]$ x=3
[alumno@localhost ~]$ let x=x+2
[alumno@localhost ~]$ echo $x
5
[alumno@localhost ~]$ let x=x*5
[alumno@localhost ~]$ echo $x
25
```

```
[alumno@localhost ~]$ echo $x
25
[alumno@localhost ~]$ y=40
[alumno@localhost ~]$ let y/x
[alumno@localhost ~]$ let z=y/x
[alumno@localhost ~]$ echo $z
1
```

```
[alumno@localhost ~]$ expr 1+2
1+2
[alumno@localhost ~]$ expr 1 + 2
3
[alumno@localhost ~]$ expr 4 - 1
3
[alumno@localhost ~]$ expr 4 * 4
expr: error de sintaxis
[alumno@localhost ~]$ expr 4 \* 4
16
[alumno@localhost ~]$ expr 12 / 2
6
```

ORDEN EXIT

- Si un programa se ejecuta correctamente (o un comando), se devuelve, por convenio un código de error 0 (cero)
- Si ha habido error → valor ≠0
- El código de error generado por un comando o programa se almacena en la variable $?. Su valor será el generado por la última instrucción del programa. Para variar este comportamiento, se usa el comando exit.
- El comando exit también se utiliza para interrumpir la ejecución de un script en cualquier momento
  - Sintaxis: exit [codigo de retorno, entre 0 y 255]
- Cuando usemos exit hay que invocar el script con bash + script, si no se sale de la sesión.

TABLAS, VECTORES, MATRICES

2 formas para declarar una tabla:

- la utilización de los corchetes [ ]

```
$Nombre[0]="Julio"
$Nombre[1]="Román"
$Nombre[2]="Francisco"
$echo ${Nombre[1]}
Román
```

- creación global

```
$Nombre=(Julio Román Francisco)
$echo ${nombre[2]}
Francisco
```

TABLAS, VECTORES, MATRICES

- El primer elemento es 0; el último, 1023. Para acceder al contenido de la tabla, hay que poner la variable Y el elemento entre llaves {}.
- Para listar todos los elementos

```
$echo ${Nombre[*]}
Julio Román Francisco
```

Para conocer el número de elementos

```
$echo ${#Nombre[*]}
3
```

Si el índice es una variable, no se pone $ delante:

```
$ idx=0
$ echo
${Nombre[idx]}
Julio
```

- `${nombre_array[x]}` →Para acceder al elemento x
- `${nombre_array[*]}` →Para consultar todos los elementos. Expande los elementos del array como si fueran una única palabra.
- `${nombre_array[@]}` → Para consultar todos los elementos. Se expande para formar cada elemento del array una palabra distinta.

COMANDO TEST

- Permite evaluar:
  - Archivos
  - Cadenas
  - Números

- Sintaxis:

```
test –opcion archivo/cadena/numero
o
test [expresión]
```

EVALUACIÓN DE FICHEROS

Archivos o directorios	
-f	Devuelve verdadero (0) si el archivo existe y es un archivo regular (no es un directorio no un archivo de dispositivo)
-s	Devuelve verdadero (0) si el archivo existe y si su tamaño es mayor que 0
-r	Devuelve verdadero (0) si el archivo existe y tiene permisos de lectura
-w	Devuelve verdadero (0) si el archivo existe y tiene permisos de escritura
-x	Devuelve verdadero (0) si el archivo existe y tiene permisos de ejecución
-d	Devuelve verdadero (0) si el archivo existe y es un directorio

Opción	Papel
`-f`	Fichero normal.
`-d`	Un directorio.
`-c`	Fichero en modo carácter.
`-b`	Fichero en modo bloque.
`-p`	Tubería nombrada (named pipe).
`-r`	Permiso de lectura.
`-w`	Permiso de escritura.
`-x`	Permiso de ejecución.
`-s`	Fichero no vacío (al menos un carácter).
`-e`	El fichero existe.
`-L`	El fichero es un vínculo simbólico.
`-u`	El fichero existe, SUID-Bit activado.
`-g`	El fichero existe, SGID-Bit activado.

EJEMPLO

```
ls -l
-rw-r--r--   1 seb   users    1392 Ago 14 15:55 dump.log
lrwxrwxrwx   1 seb   users       4 Ago 14 15:21 vínculo_fic1 -> fic1
lrwxrwxrwx   1 seb   users       4 Ago 14 15:21 vínculo_fic2 -> fic2
-rw-r--r--   1 seb   users     234 Ago 16 12:20 lista1
-rw-r--r--   1 seb   users     234 Ago 13 10:06 lista2
-rwxr--r--   1 seb   users     288 Ago 19 09:05 param.sh
-rwxr--r--   1 seb   users     430 Ago 19 09:09 param2.sh
-rwxr--r--   1 seb   users     292 Ago 19 10:57 param3.sh
drwxr-xr-x   2 seb   users    8192 Ago 19 12:09 dir1
-rw-r--r--   1 seb   users    1496 Ago 14 16:12 resultado.txt
-rw-r--r--   1 seb   users    1715 Ago 16 14:55 pepito.txt
-rwxr--r--   1 seb   users      12 Ago 16 12:07 ver_a.sh

test -f vínculo_fic1 ; echo $?
1

test -x dump.log ; echo $?
1

test -d dir1 ; echo $?
0
```

EJEMPLOS

```
[ -s fich1.txt ] o  test -s fich1.txt
```

Ambas expresiones son equivalentes y devuelven 0 (cierto) si existe fich1.txt (tanto si es fichero como si es directorio) y su longitud es mayor que 0.

```
[ -f /tmp/fich1.txt ]  o  test -f /tmp/fich1.txt
```

Ambas expresiones son equivalentes y devuelven 0 (cierto) si existe el fichero fich1.txt (debe ser un fichero)

```
[ -z $mi_nombre ] o  test -z $mi_nombre
```

Devuelven 0 si la variable mi_nombre contiene la cadena nula.

```
[ $mi_nombre = "Pepe" ]  o   test $mi_nombre = "Pepe"
```

Devuelven 0 si la variable mi_nombre contiene el valor Pepe

EVALUACIÓN DE CADENAS

- test -z "variable“ → cero, retorno OK si la variable está vacía (p. ej.: test -z "$a")
- test -n "variable“ →no cero, retorno OK si la variable no está vacía (texto cualquiera).
- test "variable"= cadena → OK si las dos cadenas son idénticas.
- test "variable"!= cadena → OK si las dos cadenas son diferentes.

EJEMPLOS

```
a=
test -z "$a" ; echo $?
0
test -n "$a" ; echo $?
1
a=Julio
test "$a" = Julio ; echo $?
0
```

Cuidado con colocar correctamente las variables que contienen texto entre comillas, porque si la variable está vacía se producirá un error:

```
a=
b=pepito
[ $a = $b ]
bash: [: =: unary operator expected
```

Mientras que esta otra forma no lo producirá:

```
[ "$a" = "$b" ]
```

EVALUACIÓN DE VALORES NUMÉRICOS

- Se deben convertir las cadenas en valores numéricos para poder evaluarlas. Bash sólo gestiona valores enteros.
- La sintaxis es **`test valor1 opción valor2`** y las opciones son las siguientes

Valores numéricos	
-lt	Menor que
-le	Menor o igual que
-qt	Mayor que
-qe	Menor o igual que
-eq	Igual a
-ne	No igual a

Valores numéricos	
-o	OR
-a	AND
!	NOT

EJEMPLOS

```
[ $num -eq 5 ] o test $num -eq 5
```

Devuelven 0 si la variable num contiene el valor 5.

```
[ $num -lt 5 ] o  test $num -lt 5
```

Devuelven 0 si la variable num contiene un valor menor que 5

```
[ $num -eq 5 -o $num -gt 10]  o  test $num -eq 5 -o $num -gt 10
```

Devuelven 0 si la variable num contiene el valor 5, o un valor mayor que 10.

```
[ $num -gt 5 -a $num -lt 10]  o  test $num -gt 5 -a $num -lt 10
```

Devuelven 0 si num es mayor que 5 y menor que 10

los símbolos “(“ y “)” tienen significado especial se debe usar la barra de escape (\).

ESTRUCTURAS CONDICIONALES

ESTRUCTURAS CONDICIONALES: IF

```
if [expresión];
then
código si 'expresión' es verdadera.
fi
```

```
#!/bin/bash
if [ "pirulo" = "pirulo" ]; then
echo expresión evaluada como verdadera
fi
```

```
#!/bin/bash
T1="pirulo"
T2="pirulon"
if [ "$T1" = "$T2" ]; then
echo expresión evaluada como verdadera
else
echo expresión evaluada como falsa
fi
if [ "$T1" != "$T2" ]; then
echo expresión evaluada como verdadera
else
echo expresión evaluada como falsa
fi
```

Fuente: Programación de comandos combinados (shell scripts). Remo Suppi Boldrito. UOC

EJEMPLO CON IF

```
# ***************************
# Escribir el menu principal
# ***************************
#/bin/bash
clear
printf "PROGRAMA DE ADMINISTRACION \n"
printf "\n \n \n \n"
printf "     1. Usuarios conectados \n"
printf "     2. Procesos en el terminal \n"
printf "     3. Procesos en el sistema \n"
printf "     0. Salir \n"
printf "\n \n \n \n \n \n"
printf "          Introduzca la opcion: "
read opcion
# ***********************
# Ejecucion de la opcion
# ***********************
if [ $opcion -eq 1 ]; then
clear
who
elif [ $opcion -eq 2 ]; then
clear
ps
elif [ $opcion -eq 3 ]; then
clear
ps -ef
elif [ $opcion -eq 0 ]; then
clear
else
printf "        Opcion incorrecta.\n "
fi
```

ESTRUCTURA CONDICIONAL: CASE

```
case $variable in

valor-1)
    comandos
;;
valor-2)
    comandos
;;
...
*)
    comandos
esac
```

```
nombre=gabriel

case $nombre in
txomin)     echo "No soy Gabriel"
            echo "Soy Txomin"
;;
gabriel)    echo "Soy Gabriel"
;;
*)          echo "No soy Txomin"
            echo "Ni tampoco soy Gabriel"
esac
```

Fuente: https://www.ehu.eus/ehusfera/hpc/2013/10/28/comando-case-en-linux/

ESTRUCTURA ITERATIVA: WHILE

```
while condicion
do
sentencias
done
```

Ejemplo: Trozo de código que nos solicita una respuesta hasta que sea “s” o “n”.

```
while [ $resp != “s” -a $resp != “n” ]; do
read resp
done
```

```
#/bin/bash
limite=5
i=0;
while (test $limite -gt $i)
do
        echo "Valor $i"
let i=$i+1
done
```

ESTRUCTURA ITERATIVA: FOR

Sintaxis 1:

```
for variable in lista_valores
do
sentencias
done
```

Sintaxis 2:

```
for variable
do
sentencias
done
```

Ejemplo:

```
for i in 1 2 3 4 5; do
echo " $i hola"
done
Resultado:    1 hola
              2 hola
              3 hola
              4 hola
              5 hola
```

Ejemplo:

```
$cat ejemplo_for2
for var; do
echo $var
done
$ ejemplo_for2 a b c d e f
a
b
c
d
e
f
```

ESTRUCTURA UNTIL

Sintaxis:

```
until condicion
do
sentencias
done
```

Ejemplo: Trozo de código que nos solicita una respuesta hasta que sea “s” o “n”.

```
until [ $resp = "s" -o $resp = "n" ]; do
read resp
done
```

RUPTURA DE LA EJECUCIÓN DE BUCLES

- Algunas veces necesitamos romper la ejecución normal de los bucles, porque se cumple alguna condición extra o nos interese saltarnos lo que queda de iteración, o bien puede que necesitemos salir del bucle completo.
- Las dos órdenes utilizadas en la programación en shell para hacer esto son: break y continue.

  - **Break**: sirve para salir del bucle y seguir la ejecución del programa.

  - **Continue**: sirve para saltarnos lo que queda de iteración y empezar la siguiente. Antes de empezar con la siguiente iteración, se volverá a chequear la condición.

Ejemplo: Programa que muestra por pantalla el contenido de todos los ficheros de texto del directorio actual, preguntando previamente al usuario si quiere visualizarlo. En cada iteración, también se le preguntará al usuario si quiere seguir visualizando ficheros o quiere parar.

```
for fichero in *.txt; do
printf "Desea visualizar el contenido del fichero $fichero?"
read resp
if [ $resp = "N" -o $resp = "n" ]; then
continue
fi
cat $fichero
printf "Quiere seguir examinando los ficheros de texto?"
read resp2
if [ $resp2 = "N" -o $resp2 = "n" ]; then
break
fi
done
```

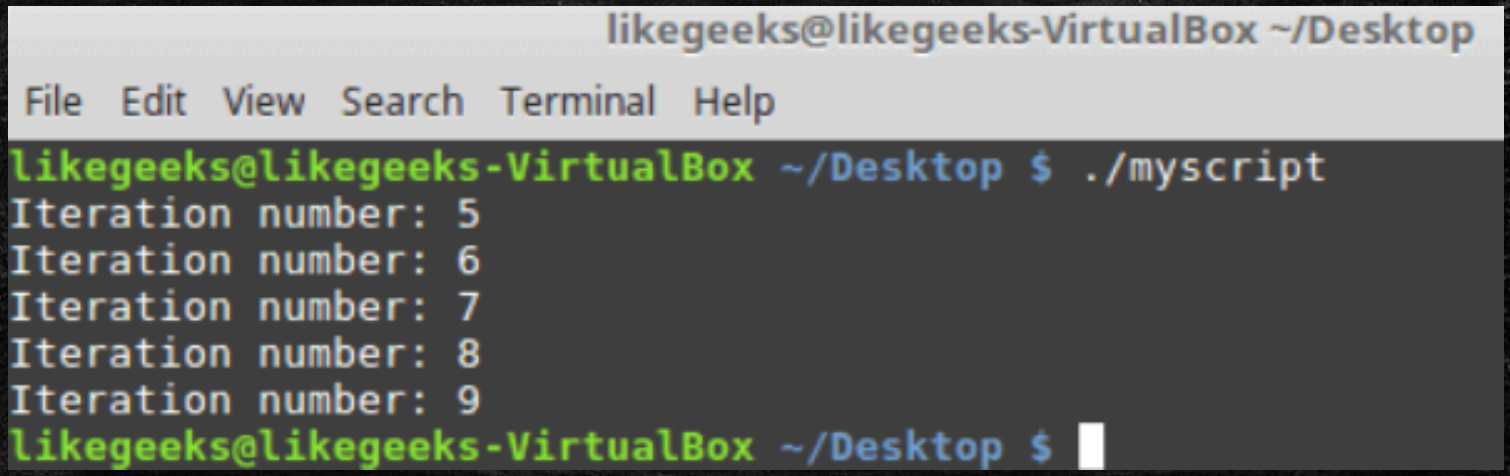
```bash
#!/bin/bash
for number in 10 11 12 13 14 15; do
    if [ $number -eq 14 ]; then
        break
    fi
    echo "Number: $number"
done
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File  Edit  View  Search  Terminal  Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscrip
Number: 10
Number: 11
Number: 12
Number: 13
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

```bash
#!/bin/bash
val=1
while [ $val -lt 5 ]; do
    if [ $val -eq 4 ]; then # Check number value
        break # The Code Breaks here <==
    fi
    echo "Iteration: $val" # The Printed Message
    val=$(($val + 1))
done
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File  Edit  View  Search  Terminal  Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Iteration: 1
Iteration: 2
Iteration: 3
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

```
#!/bin/bash
for ((number = 1; number < 10; number++)); do # El ciclo comienza aquí
    if [ $number -gt 0 ] && [ $number -lt 5 ]; then # Marque si el número e
        continue
    fi
    echo "Iteration number: $number" # El mensaje impreso
done
```



FUNCIONES Y PROCEDIMIENTOS

- Una función es un conjunto de instrucciones agrupadas bajo un nombre.
- Una función puede devolver uno o más valores. Si devuelve un valor entero se llama **función** y, en cualquier otro caso, se llama **procedimiento**.

Declaración de una función:

```
function nombre_funcion ()  {
        Instrucción 1
        Instrucción 2
           ........
        Instrucción N
}
```

Llamada a una función:

```
nombre_funcion /* no se ponen los paréntesis */
```

Ejemplo: Programa que recibe como argumentos dos números. Dentro del programa, se pide al usuario que introduzca la operación que quiere realizar con los dos números (suma o resta) y se llama a una función 'suma' o 'resta'.

```
#!/bin/bash
printf "Introduce la operación (s:suma, r:resta): "
read ope
function suma() {
        res=`expr $1 + $2`
        printf "El resultado es: $res \n \n"
}
function resta() {
        res=`expr $1 - $2`
        printf "El resultado es: $res \n \n"
}
if [ $ope = "s" ]; then
        suma $1 $2
elif [ $ope = "r" ]; then
        resta $1 $2
else
        printf "Operando incorrecto \n \n"
fi
```

DEPURACIÓN DE SCRIPTS

- Bash ofrece dos formas de depurar los shell scripts
  - -v : muestra cada línea completa del script antes de ser ejecutada
  - -x : muestra cada línea abreviada del script antes de ser ejecutada
- Uso:
  - #!/bin/bash –v
  - #!/bin/bash –x

programa1

```
#!/bin/bash -x
printf "Introduce la operación (s:suma, r:resta): "
read ope
function suma() {
        res=`expr $1 + $2`
        printf "El resultado es: $res \n \n"
}
function resta() {
        res=`expr $1 - $2`
        printf "El resultado es: $res \n \n"
}
if [ $ope = "s" ]; then
        suma $1 $2
elif [ $ope = "r" ]; then
        resta $1 $2
else
        printf "Operando incorrecto \n \n"
fi
```

```
[alumno@localhost ~]$ ./programa1 7 5
+ printf 'Introduce la operación (s:suma, r:resta): '
Introduce la operación (s:suma, r:resta): + read ope
s
+ '[' s = s ']'
+ suma 7 5
++ expr 7 + 5
+ res=12
+ printf 'El resultado es: 12 \n \n'
El resultado es: 12
```

programa1

```
#!/bin/bash -v
printf "Introduce la operación (s:suma, r:resta): "
read ope
function suma() {
        res=`expr $1 + $2`
        printf "El resultado es: $res \n \n"
}
function resta() {
        res=`expr $1 - $2`
        printf "El resultado es: $res \n \n"
}
if [ $ope = "s" ]; then
        suma $1 $2
elif [ $ope = "r" ]; then
        resta $1 $2
else
        printf "Operando incorrecto \n \n"
fi
```

```
[alumno@localhost ~]$ ./programa1 7 5
#!/bin/bash -v
printf "Introduce la operación (s:suma, r:resta): "
Introduce la operación (s:suma, r:resta): read ope
r
function suma() {
        res=`expr $1 + $2`
        printf "El resultado es: $res \n \n"
}
function resta() {
        res=`expr $1 - $2`
        printf "El resultado es: $res \n \n"
}
if [ $ope = "s" ]; then
        suma $1 $2
elif [ $ope = "r" ]; then
        resta $1 $2
else
        printf "Operando incorrecto \n \n"
fi
expr $1 - $2
El resultado es: 2
```