

Computación en Física

Harvey Rodriguez Gil

Universidad EIA

8 de Agosto de 2024

Ciclos - for

El intérprete nos permite realizar una iteración sobre valores. Por medio del comando `for` podemos hacer esto. La sintaxis es la siguiente:

```
for var in list
do
    comandos
done
```

Esto también se puede escribir en una sola línea de código:

```
for var in list; do comandos; done
```

Ciclos - for

Una primera alternativa es definir los valores que se quieren iterar posterior de la palabra `in` del comando `for`:

```
hrodriguezgi ~ — bash — bash — 80x24
bash-3.2$ cat test1
#!/bin/bash

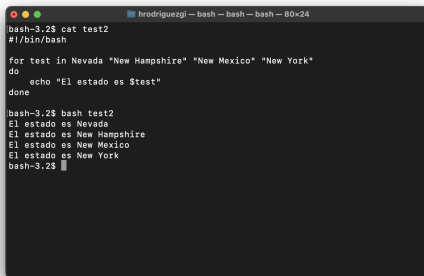
for test in Alabama alaska Arizona Arkansas California Colorado
do
    echo "El siguiente estado es $test"
done
echo "El ultimo estado fue $test"

test="Connecticut"
echo "Ahora el estado es $test"

bash-3.2$ bash test1
El siguiente estado es Alabama
El siguiente estado es alaska
El siguiente estado es Arizona
El siguiente estado es Arkansas
El siguiente estado es California
El siguiente estado es Colorado
El ultimo estado fue Colorado
Ahora el estado es Connecticut
bash-3.2$
```

Ciclos - for

Cuando se cuentan con elementos en la lista que tienen un espacio como lo es el caso de **New York** en el ejemplo, se deben colocar los elementos entre comillas (")



```
hrodriguezgi ~ — bash — bash — 80x24
bash-3.2$ cat test2
#!/bin/bash

for test in Nevada "New Hampshire" "New Mexico" "New York"
do
    echo "El estado es $test"
done

bash-3.2$ bash test2
El estado es Nevada
El estado es New Hampshire
El estado es New Mexico
El estado es New York
bash-3.2$
```

Ciclos - for

También es posible adicionar elementos a una variable, ya sea con un valor sencillo como en el ejemplo o por la concatenación de dos o más variables:

```
bash-3.2$ cat test3
#!/bin/bash

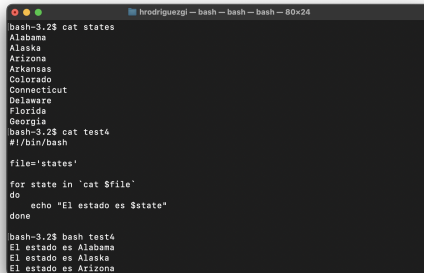
lista="Alabama Alaska Arizona Arkansas Colorado"
lista=$lista" Connecticut"

for state in $lista
do
    echo "El estado es $state"
done

bash-3.2$ bash test3
El estado es Alabama
El estado es Alaska
El estado es Arizona
El estado es Arkansas
El estado es Colorado
El estado es Connecticut
bash-3.2$
```

Ciclos - for

Con ayuda de los backticks incluso podemos extender la funcionalidad de `for` por medio de algún comando que se ejecute y nos devuelva un objeto que se pueda iterar:



```
bash-3.2$ cat states
Alabama
Alaska
Arizona
Arkansas
Colorado
Connecticut
Delaware
Florida
Georgia
bash-3.2$ cat test4
#!/bin/bash

file='states'

for state in `cat $file`
do
    echo "El estado es $state"
done

bash-3.2$ bash test4
El estado es Alabama
El estado es Alaska
El estado es Arizona
```

Ciclos - for

Los ciclos for pueden controlarse por medio de dos comandos `break` que le indica a bash salir del ciclo aun cuando no se haya terminado (cumplimiento de alguna condición)



```
class_10 ~ hrodriguezgi@Harveys-MacBook-Pro ~ .sica/class_10 -- zsh -- 80x24
+ class_10 cat test10
#!/bin/bash

for var1 in {1..10}
do
    if [ $var1 -eq 5 ]
    then
        break
    fi
    echo "Valor de la variable: $var1"
done
echo "El ciclo se ha completado"

+ class_10 bash test10
Valor de la variable: 1
Valor de la variable: 2
Valor de la variable: 3
Valor de la variable: 4
El ciclo se ha completado
+ class_10
```

Ciclos - for

Por otro lado con el comando `continue` se le indica que continúe con el siguiente elemento de la iteración

```
class_10 — hrodriguezgi@Harveys-MacBook-Pro — .sica/class_10 — zsh — 80x24
➤ class_10 cat test12
#!/bin/bash

for (( var1 = 1; var1 < 15; var1++ ))
do
    if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
    then
        continue
    fi
    echo "Iteración número: $var1"
done

➤ class_10 bash test12
Iteración número: 1
Iteración número: 2
Iteración número: 3
Iteración número: 4
Iteración número: 5
Iteración número: 10
Iteración número: 11
Iteración número: 12
Iteración número: 13
Iteración número: 14
➤ class_10
```


Ciclos - while

A diferencia del `for`, el comando `while` no itera sobre una lista, si no que depende del cumplimiento de una condición para ejecutar un comando repetidamente. La sintaxis es la siguiente:

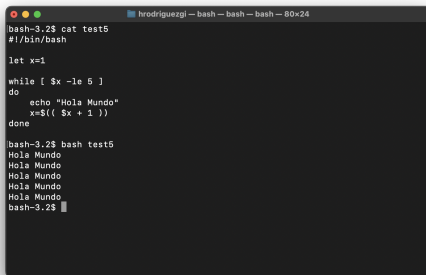
```
while condición  
do  
    comandos  
done
```

Esto también se puede escribir en una sola línea de código:

```
while condición; do comandos; done
```

Ciclos - while

Es importante tener en cuenta en los ciclos `while` de incrementar nuestra variable (si este es nuestro condicional), de otro modo se quedará infinitamente en el ciclo:



```
bash-3.2$ cat test5
#!/bin/bash

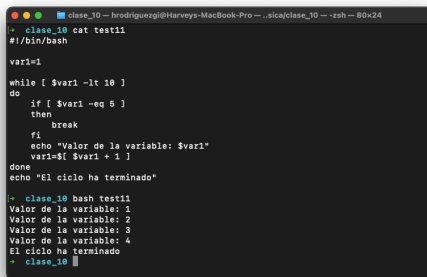
let x=1

while [ $x -le 5 ]
do
    echo "Hola Mundo"
    x=$(( $x + 1 ))
done

bash-3.2$ bash test5
Hola Mundo
Hola Mundo
Hola Mundo
Hola Mundo
Hola Mundo
bash-3.2$
```

Ciclos - while

Los ciclos `while` también pueden controlarse por medio de dos comandos `break` que le indica a `bash` salir del ciclo aun cuando no se haya terminado (cumplimiento de alguna condición)



```
class_10 ~ hrodriguezgi@Harveys-MacBook-Pro ~ .sica/class_10 -- zsh -- 80x24
➤ class_10 cat test11
#!/bin/bash

var1=1

while [ $var1 -lt 10 ]
do
    if [ $var1 -eq 5 ]
    then
        break
    fi
    echo "Valor de la variable: $var1"
    var1=$(( $var1 + 1 ))
done
echo "El ciclo ha terminado"

➤ class_10 bash test11
Valor de la variable: 1
Valor de la variable: 2
Valor de la variable: 3
Valor de la variable: 4
El ciclo ha terminado
➤ class_10
```

Ciclos - while

Por otro lado con el comando `continue` se le indica que continúe con el siguiente elemento de la iteración

```
class_10 — bash test13 | more — bash — more — 80x24
+ class_10 cat test13
#!/bin/bash

var1=0

while echo "Iteración: $var1"
[ $var1 -lt 15 ]
do
    if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
    then
        continue
    fi
    echo "    Iteracion interna: $var1"
    var1=$(( $var1 + 1 ))
done
+ class_10 bash test13 | more
Iteración: 0
    Iteracion interna: 0
Iteración: 1
    Iteracion interna: 1
Iteración: 2
    Iteracion interna: 2
Iteración: 3
    Iteracion interna: 3
```

Funciones

Como en los lenguajes de programación, las funciones nos permiten crear bloques de códigos que pueden ser reutilizados.

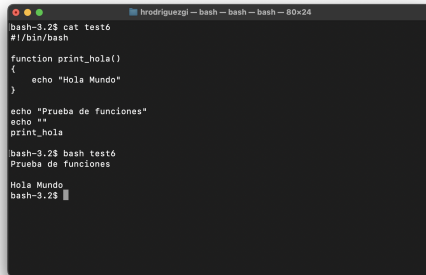
```
function_name()  
{  
    comandos  
}
```

O también se puede escribir de la siguiente forma:

```
function function_name  
{  
    comandos  
}
```

Funciones

Se debe tener presente que definir una función no implica que se utilice en el script, por lo tanto en el momento de querer hacer uso de ella, deberemos invocarla:

A terminal window with a dark background and light text. The window title is 'hrodriguezgi — bash — bash — 80x24'. The terminal shows the following commands and output:

```
bash-3.2$ cat test6
#!/bin/bash

function print_hola()
{
    echo "Hola Mundo"
}

echo "Prueba de funciones"
echo ""
print_hola

bash-3.2$ bash test6
Prueba de funciones

Hola Mundo
bash-3.2$
```

Funciones - Variables

Las variables en los scripts son por defecto globales en el script, sin embargo podemos definir las de forma local si lo requerimos:

```
function_name()  
{  
    local var1='C'  
}
```

De esta forma, la variable `var1` solo tendrá alcance dentro de la función.

Funciones - Variables

```
hrodriguezg1 ~ bash — bash — bash — 80x24
bash-3.2$ cat test7
#!/bin/bash

var1="A"
var2="B"

mi_funcion () {
    local var1="C"
    var2="D"
    echo "Dentro de la función: var1: $var1, var2: $var2"
}

echo "Antes de ejecutar la función: var1: $var1, var2: $var2"

mi_funcion

echo "Despues de la función: var1: $var1, var2: $var2"

bash-3.2$ bash test7
Antes de ejecutar la función: var1: A, var2: B
Dentro de la función: var1: C, var2: D
Despues de la función: var1: A, var2: D
bash-3.2$
```

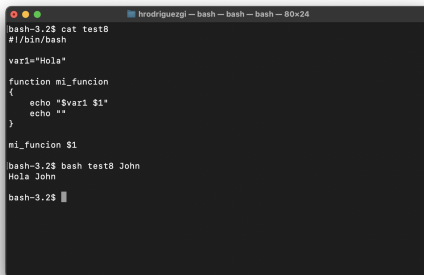

Funciones - Argumentos

Las funciones son capaces también de tomar los valores de los argumentos al momento de ejecutarse un script, bajo la misma lógica: \$1, \$2, ... , \$n

```
function_name()  
{  
    var1=$1  
    local var2=$2  
}
```

Funciones - Argumentos

Al igual que lo visto anteriormente, podemos utilizar la cantidad de argumentos que requiera nuestro script y ser llamados desde las funciones, incluso podemos hacer uso de `$@` que utilizaría todos los argumentos recibidos:



```
bash-3.2$ cat test8
#!/bin/bash

var1="Hola"

function mi_funcion
{
    echo "$var1 $1"
    echo ""
}

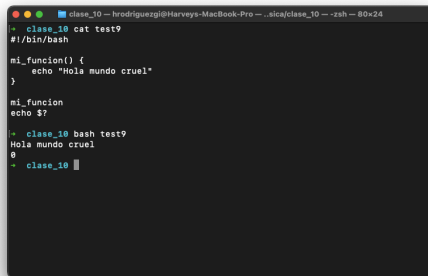
mi_funcion $1

bash-3.2$ bash test8 John
Hola John

bash-3.2$
```

Funciones - Salida

Las funciones a diferencia de los lenguajes no permiten devolver un valor cuando son invocadas. Al igual que lo visto anteriormente retornan el estado del último comando ejecutado en ella

A terminal window titled 'clase_10' with a prompt 'hrodriguezgi@Harveys-MacBook-Pro ~.sica/clase_10 -- zsh -- 80x24'. The user enters 'cat test9' and the terminal shows the contents of the file 'test9', which defines a function 'mi_funcion' that echoes 'Hola mundo cruel'. Then, the user enters 'mi_funcion' and the terminal outputs 'Hola mundo cruel'. Finally, the user enters 'echo \$?' and the terminal outputs '0', indicating the function executed successfully.

```
clase_10 — hrodriguezgi@Harveys-MacBook-Pro ~.sica/clase_10 — zsh — 80x24
➤ clase_10 cat test9
#!/bin/bash

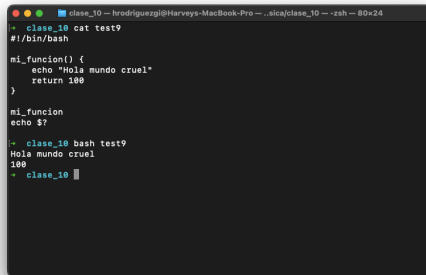
mi_funcion() {
    echo "Hola mundo cruel"
}

mi_funcion
echo $?

➤ clase_10 bash test9
Hola mundo cruel
0
➤ clase_10
```

Funciones - Salida

Sin embargo, al igual que en los bash script, podemos personalizar el código de salida con la palabra `return`



```
clase_10 — hrodriguezgi@Harveys-MacBook-Pro — .sica/clase_10 — zsh — 80x24
➤ clase_10 cat test9
#!/bin/bash

mi_funcion() {
    echo "Hola mundo cruel"
    return 100
}

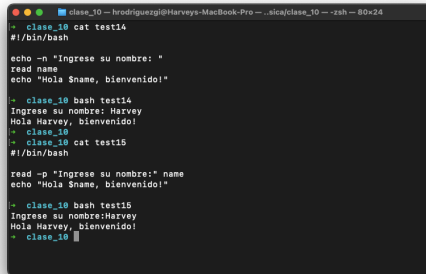
mi_funcion
echo $?
```

➤ clase_10 bash test9

```
Hola mundo cruel
100
➤ clase_10
```

Interactuando con el usuario

El comando `read` nos permite interactuar con el usuario en un script, esperando que ingrese un dato a través del teclado:



```
class_10 — hrodriguezgi@Harveys-MacBook-Pro — .sica/class_10 — zsh — 80x24
➤ class_10 cat test14
#!/bin/bash

echo -n "Ingrese su nombre: "
read name
echo "Hola $name, bienvenido!"

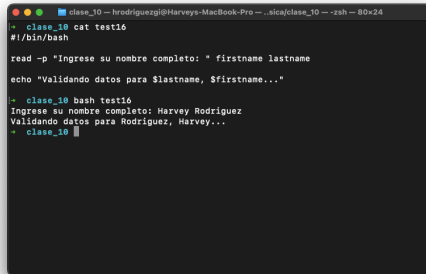
➤ class_10 bash test14
Ingrese su nombre: Harvey
Hola Harvey, bienvenido!
➤ class_10
➤ class_10 cat test15
#!/bin/bash

read -p "Ingrese su nombre:" name
echo "Hola $name, bienvenido!"

➤ class_10 bash test15
Ingrese su nombre:Harvey
Hola Harvey, bienvenido!
➤ class_10
```

Interactuando con el usuario

También se pueden leer múltiples argumentos al momento de ejecutar el programa:



```
clase_10 — hrodriguezgi@Harveys-MacBook-Pro — .sica/clase_10 — zsh — 80x24
+ clase_10 cat test16
#!/bin/bash

read -p "Ingrese su nombre completo: " firstname lastname

echo "Validando datos para $lastname, $firstname..."

+ clase_10 bash test16
Ingrese su nombre completo: Harvey Rodriguez
Validando datos para Rodriguez, Harvey...
+ clase_10
```

Interactuando con el usuario

También se pueden leer desde un archivo por medio del comando `read` en conjunto con `while`:

```
class_10 — hrodriguezi@Harveys-MacBook-Pro — .sica/class_10 — zsh — 80x24
+ class_10 cat course.txt
Introduction to the Unix Shell

The Unix command line has survived and thrived for almost fifty years
because it lets people to do complex things with just a few
keystrokes. Sometimes called "the duct tape of programming", it helps
users combine existing programs in new ways, automate repetitive
tasks, and run programs on clusters and clouds that may be halfway
around the world.

+ class_10 cat test17
#!/bin/bash

count=1
cat course.txt | while read line
do
    echo "Lines $count: $line"
    count=$((count + 1))
done
echo "Archivo procesado"

+ class_10
```

Interactuando con el usuario

Esta es la salida del programa anterior

```
class_10 — hrodriguezgi@Harveys-MacBook-Pro — .sica/class_10 — zsh — 80x24
+ class_10 bash test17
Linea 1: Introduction to the Unix Shell
Linea 2:
Linea 3: The Unix command line has survived and thrived for almost fifty years
Linea 4: because it lets people to do complex things with just a few
Linea 5: keystrokes. Sometimes called "the duct tape of programming", it helps
Linea 6: users combine existing programs in new ways, automate repetitive
Linea 7: tasks, and run programs on clusters and clouds that may be halfway
Linea 8: around the world.
Linea 9:
Archivo procesado
+ class_10
```


Vamos a practicar

- ▶ Crear un script que calcule el factorial de un número que ingrese el usuario
- ▶ Crear un script que reciba pregunte:
 - ▶ Número 1
 - ▶ Número 2
 - ▶ Operación a realizar

La salida deberá ser el resultado de la operación. Utilizar funciones para cada operación.