# Scripting avanzado

¡Felicitaciones! si llegaste hasta este punto ya tienes los conocimientos para hacer tu script básico… Pero, ¿es esto todo lo que puedo hacer? imprimir frases por pantalla? La respuesta es : Por supuesto que no, en lo que queda de documento nuestra tarea será enseñarte las herramientas de desarrollo para hacer un script mas poderoso.

## Un viejo conocido

Nombremos algunos de los temas que veremos: condicionales, iteradores, arreglos… esto se nos hace familiar, pero de donde? Exacto, suena a los típicos temas que veríamos si queremos aprender un lenguaje de programación.
¡Es por que son conceptos realmente similares! Pero ahora nuestro código puede tener mayor impacto en el ecosistema del ordenador por lo que debemos ser mas cautelosos con lo que estemos desarrollando.

## Operadores

### Operadores matemáticos

Si queremos representar una operación matemática dentro de un script de bash debemos usar la siguiente sintaxis:

```
$((A + B))
```

Los valores A y B pueden ser tanto como números “literales” (los que tecleamos) o variables que contengan valores numéricos:

```
$((unEntero - otroEntero))
```

Podemos usar el resultado de esta operación como si fuera un valor numérico cualquiera.

Además de los operadores matemáticos básicos podremos usar los modificativos

Ej: `$((unEntero += otroEntero))`

### Operadores relacionales

Esta misma sintaxis se utiliza para los operadores lógicos, solo basta con reemplazar el signo

```
$((unValor > otroValor))
```

El resultado que nos devolverá será del tipo booleano, pero si lo imprimimos por pantalla notaremos que nos devuelve 1 si la expresión es verdadera y 0 si la expresión es falsa, es por que en esencia estos son los verdaderos valores de las expresiones booleanas, pero se las representa con *true* y *false* por cuestiones de semántica.

## Condicionales

Bueno ya vimos que los booleanos también existen aca, pero como los usamos exactamente, pues con unos buenos condicionales, si hay booleanos hay un if que los usara, y es exactamente lo que veremos  a continuación:

La sintaxis es la siguiente:

```
if [ variable1 -eq variable2 ]; then
     echo "son iguales"
fi
```

(*recordar respetar los espaciados*)

Es muy parecido a un if que veríamos en otros lenguajes pero hay elementos atípicos:
*; then* indica el fin de la expresión a evaluar y el inicio del consecuente.
*fi* indica el fin de toda la expresión
Por último *-eq*, es solo una expresión equivalente a los operadores relacionales que ya vimos, a continuación una tabla de equivalencias:

- -eq : ==
- -ne : !=
- -gt : >
- -ge : >=
- -lt : <
- -le : <=

Al igual que un típico if podemos encadenar un else en caso de querer tener un consecuente en caso de que la expresión de negativa:

```
if [ variable1 -eq variable2 ]; then
     echo "son iguales"
else
     echo "no son iguales"
fi
```

Por último podremos usar operadores lógicos en esto, pero tendremos que modificar un poco la sintaxis:

```
a=10
b=20
```

```
if [[ $a -lt 100 && $b -gt 100 ]]; then
   echo "is true"
else
   echo "is false"
fi

if [[ $a -lt 100 || $b -gt 100 ]]; then
   echo "is true"
else
   echo "is false"
fi
```

## Condicionales con archivos

Los condicionales también nos sirven para control de archivos, mas específicamente nos permiten corroborar ciertas características de los archivos que evaluamos.

La sintaxis general es la siguiente:

```
if [ -@ nombreArchivo ]; then
      comandos…
fi
```

Es muy parecido a un if tradicional pero no estamos enfrentados a dos condiciones. Pero qué son los caracteres del principio de la sentencia? (-@). Bueno, el @ es solo ilustrativo, podemos usar ciertos caracteres dependiendo la situación que queramos evaluar:

- **-e** Esta es la mas básica y mas usada, nos permite saber si existe un archivo con el nombre que estamos pasando.

- **-f** Preguntamos si el archivo con el nombre que pasamos es un archivo normal (osea que no es un directorio).

- **-s** Preguntamos si el archivo con el nombre que pasamos tiene tamaño 0.

- **-d** Preguntamos si el archivo con el nombre que pasamos es un directorio.

- **-r, -w, -x** Preguntamos si el archivo con el nombre que pasamos tiene permisos de lectura, escritura o ejecución respectivamente. Todo visto desde el usuario que esté ejecutando el script.

# Mas sobre variables

Hasta el momento las únicas variables que conocemos son las que asignamos valor por nuestra cuenta, pero podemos definir otro tipo de variables?

## Capturando retornos de comandos

Como habremos notado muchos comandos nos retornan un valor, podemos guardar ese retorno en una variable. Como?

```
variable=$(unComando)
```

## Capturar inputs externos

En este punto puedes tener la idea: "*che! mis scripts son muy rígidos y cerrados, no hay forma de interactuar con el exterior?*". Pues hay una herramienta para capturar valores del usuario (no siempre será el usuario, en realidad es una solicitud de input a una entidad superior) y luego capturar ese valor en una variable.

Para ello debemos primero crear la variable en la que se guardara ese valor:

`variableUsuario=" "` (*el espaciado entre las comillas es solo ilustrativo, se debe dejar vacío*)

Ahora, con el siguiente código pausaremos el flujo del programa y solicitaremos un input externo:

```
read -p "Ingrese un valor" variableUsuario
```

# Como el *switch*

Ya conocemos los condicionales, ya sabemos cómo recibir inputs externos, que podemos hacer con estos conocimientos? Exacto un switch!! Solo que aca se llama case y tiene una sintaxis muy particular:

```
case $expresion in
  caso1)
    comandos
  ;;
  caso2)
    comandos
  ;;
#...
  *)
    comandos
  ;;
```

```
esac
```

El uso de “*” indica que es el caso default.

# Arreglos

Acá tenemos otro viejo conocido, los arreglos o arrays para los angloparlantes.
Se los define de la siguiente manera:

```
unArreglo=(1 2 3 4 5 6 7 8)
```

Como habrás notado aca es mas mas importante que nunca respetar el espacio, ya que definen dónde empieza un elemento y termina el anterior.

No tenemos porqué limitarnos a solo numeros, un arreglo puede contener varios tipos de datos:

```
otroArreglo=(1 2 4 “manzana” “otra fruta”)
```

## Ver tamaño de un arreglo

Si en algún momento necesitamos ver el tamaño de nuestro arreglo debemos usar el siguiente código:

```
${#unArreglo[*]}
```

Entonces si queremos guardar ese número en una variable solo debemos hacer lo siguiente:

```
tamanioArreglo=${#unArreglo[*]}
```

## Acceder a una posición específica

Si queremos acceder a un elemento del arreglo en la posición *x* (recordar que las posiciones de los arrays se empieza a contar desde 0) Entonces solo debemos escribir el siguiente código:

```
${unArreglo[x]}
```

Ejemplo:

```
otroArreglo=(1 2 4 “manzana” “otra fruta”)

echo ${otroArreglo[3]}
```

Por pantalla veríamos la palabra “manzana”.

## Manipulación de arreglos

Con un razonamiento como el anterior podemos arreglarnos para modificar los valores de arreglo:

```
unArreglo[x]=”unaCosa”
```

Ahora la posición x del arreglo guarda el string “unaCosa”

## Pseudo-Arreglos

Si quisiéramos hacer un arreglo con los números del 1 al 100 en un script de bash, se nos ocurren 2 opciones, o lo hacemos a mano (mala idea) o rellenamos un arreglo vacío con un for. Pero tenemos una tercera opción! Podemos generar un rango:

```
{1..100}
```

En si esto no es un arreglo, tiene mas semejanza a una colección, en palabras simples, estamos diciendo al ordenador que lo interprete como una colección de números cuyo valor inicial es el 1 y el final es el 100

Si usamos el comando:

```
echo {1..100}
```

obtendremos la siguiente salida:

1 2 3 4 5 6 7 8 9 10 11 … (llega hasta el 100 lo juro)

Podemos especificar el salto en cada digito:

```
echo {0..10..2}
0 2 4 6 8 10
```

y si invertimos el orden obtendremos una cuenta regresiva:

```
echo {10..0}
10 9 8 7 6 5 4 3 2 1 0
```

también funciona con letras:

```
echo {a..z}
a b c d e f … x y z
```

además de esto también nos sirve para controlar ciclos como el for:

```
for num in {1..10}
do
        echo "estoy imprimiendo el $num"
done
```

```
estoy imprimiendo el 1
estoy imprimiendo el 2
estoy imprimiendo el 3
estoy imprimiendo el 4
estoy imprimiendo el 5
estoy imprimiendo el 6
estoy imprimiendo el 7
estoy imprimiendo el 8
estoy imprimiendo el 9
estoy imprimiendo el 10
```

# Bucles

## El ciclo for

Vuelve un viejo conocido, el ciclo for, nos permite iterar sobre cada uno de los elementos de un arreglo.

Usaremos la siguiente sintaxis:

```
for var in ${unArreglo[*]}
do
     comandos…
done
```

Ej:

```
paises=(“Argelia” “Argentina” “Armenia” ….)

for unPais in ${paises[*]}
do
     echo “$unPais”
done
```

Este ciclo imprimirá por pantalla ,un país a la vez, del arreglo “países”

## El ciclo while

El ciclo while funciona de una forma parecida, pero necesitaremos la ayuda de un contador externo la mayoría de las veces:

Ej:

```
contador=0

while [ $contador -lt 4 ]
do
        echo "estoy en la iteracion numero $contador"
        let $((contador += 1))
done
```

El bucle continuará mientras la expresión a la derecha del while resulte verdadera.
Existe también el ciclo until, la sintaxis es la misma que para el while, pere este mantendrá el ciclo mientras la expresión resulte falsa:

```
until [ condición ]
do
     //Bloque de instrucciones
done
```

## Ciclos especiales

La sintaxis de los ciclos puede variar en función de la tarea que queramos realizar, por ejemplo:

Un ciclo for que itera sobre cada elemento del listado capturado por el comando *ls*:

```
archivos=$(ls)

for archivo in ${archivos[@]}
do
     comandos…
done
```

otro ejemplo:

Un ciclo while que lee un archivo de texto linea por linea:

```
while IFS= read line
do
     echo “el contenido de esta línea es $line”

done < $rutaDelArchivo
```

# Funciones

Como dijimos anteriormente, si esto del scripting es parecido a otros lenguajes de pŕogramacion, entonces hay funciones aca? Si, y la sintaxis es la siguiente:

```
unaFuncion () {
      comandos…
}
```

¿Interesante no? Recordamos otra vez tener cuidado con el espaciado.
Pero cómo llamamos a esta función, sencillo, solo usamos el nombre de la función para invocarla:

```
unaFuncion
```

¿Y las funciones con parámetros? Pues la sintaxis es especial:

```
funcionConParametros () {
      echo "el primer parámetro es $1 y el segundo es $2"
}
```

Como podemos observar no se declara en ningún momento que se esperan parámetros, si no que se asume su existencia y se los usa directamente en el código, etiquetando al primer parámetro con los caracteres $1, al segundo $2 y así sucesivamente.

¿Cómo le paso parametros a una función con parámetros? De la misma manera que a un script:

```
funcionConParametros manzana banana
```

El resultado sería:

*el primer parámetro es manzana y el segundo banana.*