# Curso desarrollado script 1ª Parte V.1.4

## Estructura de Bash

Comencemos creando un nuevo archivo con la extensión `.sh`. Como ejemplo, podríamos crear un archivo llamado `mi_script.sh`.

Para crear ese archivo, puedes usar el comando `touch`:

```
touch mi_script.sh
```

O puedes usar tu editor de texto en su lugar:

```
nano mi_script.sh
```

Para ejecutar un archivo de script bash con el intérprete de shell bash, la primera línea del archivo debe indicar la ruta absoluta al ejecutable de bash:

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

Esto también se llama un **Shebang**.

Todo lo que hace el shebang es instruir al sistema operativo para que ejecute el script con el ejecutable `/bin/bash`.

Sin embargo, bash no siempre está en el directorio `/bin/bash`, particularmente en sistemas que no son Linux. Por lo tanto, es posible que prefieras usar:

```
#!/usr/bin/env bash
```

Esto busca el ejecutable de `bash` en los directorios listados en la variable de entorno `PATH`.

---

## Hola Mundo en Bash

Una vez que tenemos nuestro archivo `mi_script.sh` creado y hemos especificado el shebang de bash en la primera línea, estamos listos para crear nuestro primer script "Hola Mundo".

Para hacer eso, abre el archivo `mi_script.sh` nuevamente y agrega lo siguiente después de la línea `#!/bin/bash`:

```
#!/bin/bash
echo "¡Hola Mundo!"
```

Guarda el archivo y sal.

Después de eso, haz que el script sea ejecutable ejecutando:

```
chmod +x mi_script.sh
```

Luego, ejecuta el archivo:

```
./mi_script.sh
```

Verás un mensaje "¡Hola Mundo!" en la pantalla.

Otra forma de ejecutar el script sería:

```
bash mi_script.sh
```

Como bash se puede usar interactivamente, podrías ejecutar el siguiente comando directamente en tu terminal y obtendrías el mismo resultado:

```
echo "¡Hola Mundo!"
```

Juntar un script es útil una vez que tienes que combinar múltiples comandos.

---

## Variables en Bash

Como en cualquier otro lenguaje de programación, también puedes usar variables en Bash Scripting. Sin embargo, no hay tipos de datos, y una variable en Bash puede contener números y caracteres.

Para asignar un valor a una variable, todo lo que necesitas hacer es usar el signo `=`:

```
nombre="MiApp"
```

**Aviso:** como nota importante, no puedes tener espacios antes ni después del signo `=`.

Después de eso, para acceder a la variable, tienes que usar el signo `$` y referenciarla como se muestra a continuación:

```
echo $nombre
```

Envolver el nombre de la variable entre llaves no es obligatorio, pero se considera una buena práctica, y te aconsejaría que las uses siempre que puedas:

```
echo ${nombre}
```

El código anterior mostraría: `MiApp`, ya que este es el valor de nuestra variable `nombre`.

A continuación, actualicemos nuestro script `mi_script.sh` e incluyamos una variable en él.

De nuevo, puedes abrir el archivo `mi_script.sh` con tu editor de texto favorito:

```
nano mi_script.sh
```

Añadiendo nuestra variable `nombre` aquí en el archivo, con un mensaje de bienvenida. Nuestro archivo ahora se ve así:

```
#!/bin/bash
nombre="MiApp"
echo "Hola $nombre"
```

Guárdalo y ejecuta el archivo usando el comando a continuación:

```
./mi_script.sh
```

Verías la siguiente salida en tu pantalla:

```
Hola MiApp
```

Aquí hay un resumen del script escrito en el archivo:

- `#!/bin/bash` - Primero, especificamos nuestro shebang.
- `nombre="MiApp"` - Luego, definimos una variable llamada `nombre` y le asignamos un valor.
- `echo "Hola $nombre"` - Finalmente, mostramos el contenido de la variable en la pantalla como un mensaje de bienvenida usando `echo`.

También puedes añadir múltiples variables en el archivo como se muestra a continuación:

```
#!/bin/bash
nombre="MiApp"
saludo="Hola"
echo "$saludo $nombre"
```

Guarda el archivo y ejecútalo de nuevo:

```
./mi_script.sh
```

Verías la siguiente salida en tu pantalla:

```
Hola MiApp
```

Ten en cuenta que no necesitas necesariamente añadir punto y coma (`;`) al final de cada línea.

También puedes pasar variables en la Línea de Comandos fuera del script de Bash y pueden ser leídas como parámetros:

```
./mi_script.sh Juan amigo
```

Este script toma dos parámetros, `Juan` y `amigo`, separados por espacio. En el archivo `mi_script.sh` tenemos lo siguiente:

```
#!/bin/bash
echo "Hola" $1
```

`$1` es el primer parámetro (`Juan`) en la Línea de Comandos.

Del mismo modo, podría haber más parámetros y todos se referencian por el signo `$` y su respectivo orden. Esto significa que `amigo` se referencia usando `$2`. Otro método útil para leer variables es `$@`, que lee todos los parámetros.

Así que ahora cambiemos el archivo `mi_script.sh` para entenderlo mejor:

```
#!/bin/bash
echo "Hola" $1
#$1: primer parámetro
echo "Hola" $2
#$2: segundo parámetro
echo "Hola a todos" $@
#$@: todos los parámetros
```

La salida para:

```
./mi_script.sh Juan amigo
```

Sería la siguiente:

```
Hola Juan
Hola amigo
Hola a todos Juan amigo
```

## Entrada de Usuario en Bash

Con el script anterior, definimos una variable y mostramos su valor en la pantalla con `echo $nombre`.

Ahora, sigamos adelante y pidamos al usuario que ingrese datos. Para hacerlo, abre el archivo (`mi_script.sh`) con tu editor de texto y actualiza el script de la siguiente manera:

```
#!/bin/bash
echo "¿Cuál es tu nombre?"
read nombre
echo "Hola $nombre"
echo "¡Bienvenido al sistema!"
```

Lo anterior le pedirá al usuario que ingrese datos y luego almacenará esa entrada como una cadena/texto en una variable. Luego podemos usar la variable e imprimir un mensaje de vuelta.

La salida del script anterior sería:

- Primero ejecuta el script:

```
./mi_script.sh
```

- Luego, se te pedirá que ingreses tu nombre:

```
¿Cuál es tu nombre?
Ana
```

- Una vez que hayas escrito tu nombre, simplemente presiona enter y obtendrás la siguiente salida:

```
Hola Ana
¡Bienvenido al sistema!
```

Para reducir el código, podríamos cambiar la primera declaración `echo` con `read -p`. El comando `read` usado con el flag `-p` imprimirá un mensaje *antes* de pedirle la entrada al usuario:

```
#!/bin/bash
read -p "¿Cuál es tu nombre? " nombre
echo "Hola $nombre"
echo "¡Bienvenido al sistema!"
```

¡Asegúrate de probar esto tú mismo también!

Comentarios en Bash

Como con cualquier otro lenguaje de programación, puedes agregar comentarios a tu script. Los comentarios se usan para dejarte notas a ti mismo a través de tu código. Para hacerlo en Bash, necesitas agregar el símbolo `#` al principio de la línea. Los comentarios nunca se mostrarán en la pantalla.

Aquí hay un ejemplo de un comentario:

```
#Esto es un comentario y no se mostrará en la pantalla
```

Vamos a agregar algunos comentarios a nuestro script:

```
#!/bin/bash
#Preguntar al usuario por su nombre
read -p "¿Cuál es tu nombre? " nombre
#Saludar al usuario
```

```
echo "Hola $nombre"
echo "¡Bienvenido al sistema!"
```

Los comentarios son una excelente manera de describir algunas de las funcionalidades más complejas directamente en tus scripts, para que otras personas puedan orientarse en tu código con facilidad.

## Argumentos en Bash

Puedes pasar argumentos a tu script de shell cuando lo ejecutas. Para pasar un argumento, solo necesitas escribirlo justo después del nombre de tu script. Por ejemplo:

Bash

```
./mi_script.sh tu_argumento
```

En el script, podemos usar `$1` para hacer referencia al primer argumento que especificamos. Si pasamos un segundo argumento, estaría disponible como `$2` y así sucesivamente.

Creemos un script corto llamado `argumentos.sh` como ejemplo:

Bash

```
#!/bin/bash
echo "El argumento uno es $1"
echo "El argumento dos es $2"
echo "El argumento tres es $3"
```

Guarda el archivo y hazlo ejecutable:

Bash

```
chmod +x argumentos.sh
```

Luego ejecuta el archivo y pasa 3 argumentos:

Bash

```
./argumentos.sh perro gato pajaro
```

La salida que obtendrías sería:

```
El argumento uno es perro
El argumento dos es gato
El argumento tres es pajaro
```

Para hacer referencia a todos los argumentos, puedes usar `$@`:

Bash

```
#!/bin/bash
echo "Todos los argumentos: $@"
```

Si ejecutas el script nuevamente:

```
./argumentos.sh perro gato pajaro
```

Obtendrás la siguiente salida:

```
Todos los argumentos: perro gato pajaro
```

Otra cosa que debes tener en cuenta es que `$0` se usa para hacer referencia al script mismo. Esta es una excelente manera de autodestruir el archivo si lo necesitas o simplemente obtener el nombre del script.

Por ejemplo, creemos un script que imprima el nombre del archivo y lo elimine después:

```
#!/bin/bash
echo "El nombre del archivo es: $0 y va a ser auto-borrado."
rm -f $0
```

Debes tener cuidado con la autoeliminación y asegurarte de tener una copia de seguridad de tu script antes de auto-borrarlo.

## Arrays en Bash

Si alguna vez has programado, probablemente ya estés familiarizado con los arrays. Pero en caso de que no seas un desarrollador, lo principal que necesitas saber es que, a diferencia de las variables, los arrays pueden contener varios valores bajo un solo nombre.

Puedes inicializar un array asignando valores divididos por espacios y encerrados entre `()`. Ejemplo:

```
mi_array=("valor 1" "valor 2" "valor 3" "valor 4")
```

Para acceder a los elementos en el array, necesitas hacer referencia a ellos por su índice numérico. **Aviso:** ten en cuenta que necesitas usar llaves `{}`.

- **Acceder a un solo elemento** (esto mostraría: `valor 2`, ya que el índice empieza en 0):

  ```
  echo ${mi_array[1]}
  ```

- Esto devolvería el **último elemento**: `valor 4`

  ```
  echo ${mi_array[-1]}
  ```

- Al igual que con los argumentos de la línea de comandos, usar `@` devolverá **todos los elementos** del array, de la siguiente manera: `valor 1 valor 2 valor 3 valor 4`

```
echo ${mi_array[@]}
```

- Anteponer al array un signo de almohadilla (`#`) mostraría el **número total de elementos** en el array, en nuestro caso es 4:

```
echo ${#mi_array[@]}
```

Asegúrate de probar esto y practicarlo por tu cuenta con diferentes valores.

---

## Rebanado de Arrays (Array Slicing)

Aunque Bash no soporta un verdadero "rebanado" de arrays (slicing) como otros lenguajes, puedes lograr resultados similares usando una combinación de indexación de arrays:

```
#!/bin/bash
array=("A" "B" "C" "D" "E")

# Imprimir todo el array
echo "${array[@]}" # Salida: A B C D E

# Acceder a un solo elemento
echo "${array[1]}" # Salida: B

# Imprimir un rango de elementos (requiere Bash 4.0+)
# Sintaxis: ${array[@]:inicio:cantidad}
echo "${array[@]:1:3}" # Salida: B C D

#Imprimir desde un índice hasta el final
echo "${array[@]:3}" # Salida: D E
```

Cuando trabajes con arrays, usa siempre `[@]` para referirte a todos los elementos, y encierra la expansión del parámetro entre comillas para preservar los espacios en los elementos del array.

---

## Rebanado de Cadenas (String Slicing)

En Bash, puedes extraer porciones de una cadena (string) usando el rebanado. La sintaxis básica es:

`${cadena:inicio:longitud}`

Donde:

- `inicio` es el índice inicial (basado en 0)
- `longitud` es el número máximo de caracteres a extraer

Veamos algunos ejemplos:

```
#!/bin/bash
texto="ABCDE"

# Extraer desde el índice 0, un máximo de 2 caracteres
echo "${texto:0:2}" # Salida: AB

# Extraer desde el índice 3 hasta el final
echo "${texto:3}" # Salida: DE

# Extraer 3 caracteres comenzando desde el índice 1
echo "${texto:1:3}" # Salida: BCD

# Si la longitud excede los caracteres restantes, se detiene al final
echo "${texto:3:3}"
# Salida: DE (solo 2 caracteres disponibles)
```

Nota que el segundo número en la notación de rebanado representa la **longitud máxima** de la subcadena extraída, no el índice final. Esto es diferente de otros lenguajes de programación como Python.

En Bash, si especificas una longitud que se extendería más allá del final de la cadena, simplemente se detendrá al final de la cadena sin generar un error.

Por ejemplo:

```
texto="¡Hola, Mundo!"

# Extraer 5 caracteres comenzando desde el índice 7
echo "${texto:8:5}" # Salida: Mundo

# Intentar extraer 10 caracteres comenzando desde el índice 7
# (aunque solo quedan 6 caracteres)
echo "${texto:7:10}" # Salida: Mundo!
```

En el segundo ejemplo, aunque pedimos 10 caracteres, Bash solo devuelve los 6 caracteres disponibles desde el índice 7 hasta el final de la cadena. Este comportamiento puede ser particularmente útil cuando no estás seguro de la longitud exacta de la cadena con la que estás trabajando.

---

## Expresiones Condicionales de Bash

En informática, las sentencias condicionales, expresiones condicionales y construcciones condicionales son características de un lenguaje de programación que realizan diferentes cómputos o acciones

dependiendo de si una condición booleana especificada por el programador evalúa como verdadera o falsa.

En Bash, las expresiones condicionales son usadas por el comando compuesto `[[` y el comando interno `[` para probar atributos de archivos y realizar comparaciones aritméticas y de cadenas.

> Nota:
> En bash clásico existe el comando `test` que en bash es equivalente a `[` , pero los dobles corchetes son una mejora y es compatible con ptros operadores
> 1.- No tienes que utilizar las comillas con las variables, los dobles corchetes trabajan perfectamente con los espacios. Así `[ -f "$file" ]` es equivalente a `[[ -f $file ]]`.
> 2.- Con `[[` puedes utilizar los operadores `||` y `&&` , así como para las comparaciones de cadena.
> 3.- Puedes utilizar el operador `=~` para expresiones regulares, compo por ejemplo `[[ $respuesta =~ ^s(i)?$ ]]`
> 4.- También puedes utilizar comodines como por ejemplo en la expresión `[[ abc = a\* ]]`

---

Aquí hay una lista de las expresiones condicionales de Bash más populares. Algunas se usan mucho.

## Expresiones de Archivo

- Verdadero si el archivo existe. (sustituido por -e )
  ~~`[[ -a ${archivo} ]]`~~
- Verdadero si el archivo existe y es un archivo especial de bloque.
  `[[ -b ${archivo} ]]`
- Verdadero si el archivo existe y es un archivo especial de caracteres.
  `[[ -c ${archivo} ]]`
- Verdadero si el archivo existe y es un directorio.
  `[[ -d ${archivo} ]]`
- Verdadero si el archivo existe. (Alternativa a -a)
  `[[ -e ${archivo} ]]`
- Verdadero si el archivo existe y es un archivo regular.
  `[[ -f ${archivo} ]]`
- Verdadero si el archivo existe y es un enlace simbólico.
  `[[ -h ${archivo} ]]`
- Verdadero si el archivo existe y es legible.
  `[[ -r ${archivo} ]]`
- Verdadero si el archivo existe y tiene un tamaño mayor que cero.
  `[[ -s ${archivo} ]]`
- Verdadero si el archivo existe y es escribible.
  `[[ -w ${archivo} ]]`

- Verdadero si el archivo existe y es ejecutable.
  `[[ -x ${archivo} ]]`
- Verdadero si el archivo existe y es un enlace simbólico. (Alternativa a -h)
  `[[ -L ${archivo} ]]`

---

## Expresiones de Cadena

- Verdadero si la variable de shell `nombre_var` está definida (se le ha asignado un valor).
  `[[ -v nombre_var ]]`
  (Aquí, `nombre_var` es el *nombre* de la variable. El operador `-v` espera un nombre de variable, no su valor, así que si pasas `${nombre_var}` en lugar de `nombre_var`, la expresión devolverá falso).
- Verdadero si la longitud de la cadena es cero.
  `[[ -z ${cadena} ]]`
- Verdadero si la longitud de la cadena no es cero.
  `[[ -n ${cadena} ]]`
- Verdadero si las cadenas son iguales. (`=` debe usarse con el comando `test` para conformidad POSIX).
  `[[ ${cadena1} == ${cadena2} ]]`
- Verdadero si las cadenas no son iguales.
  `[[ ${cadena1} != ${cadena2} ]]`
- Verdadero si `cadena1` se ordena antes que `cadena2` lexicográficamente.
  `[[ ${cadena1} < ${cadena2} ]]`
- Verdadero si `cadena1` se ordena después que `cadena2` lexicográficamente.
  `[[ ${cadena1} > ${cadena2} ]]`

---

## Operadores aritméticos

- Devuelve verdadero si los números son iguales
  `[[ $arg1 -eq $arg2 ]]`
- Devuelve verdadero si los números no son iguales
  `[[ $arg1 -ne $arg2 ]]`
- Devuelve verdadero si arg1 es menor que arg2
  `[[ $arg1 -lt $arg2 ]]`
- Devuelve verdadero si arg1 es menor o igual que arg2
  `[[ $arg1 -le $arg2 ]]`
- Devuelve verdadero si arg1 es mayor que arg2
  `[[ $arg1 -gt $arg2 ]]`
- Devuelve verdadero si arg1 es mayor o igual que arg2

`[[ $arg1 -ge $arg2 ]]`

Como nota al margen, arg1 y arg2 pueden ser enteros positivos o negativos.

Al igual que con otros lenguajes de programación, puedes usar condiciones Y (AND) y O (OR):

Bash

```
[[ condicion_1 ]] && [[ condicion_2 ]] # Y
[[ condicion_1 ]] || [[ condicion_2 ]] # O
```

---

## Operadores de estado de salida

- devuelve verdadero si el comando fue exitoso sin ningún error

  [[ $? -eq 0 ]]
- devuelve verdadero si el comando no fue exitoso o tuvo errores

  [[ $? -gt 0 ]]

---

## Condicionales de Bash

En la última sección, cubrimos algunas de las expresiones condicionales más populares. Ahora podemos usarlas con sentencias condicionales estándar como `if`, `if-else` y `switch case`.

---

## Sentencia If

El formato de una sentencia `if` en Bash es el siguiente:

Bash

```
if [[ alguna_prueba ]]
then
    <comandos>
fi
```

También está muy extendido la forma que se describe abajo. Es decir añadiendo el `then` después del `;` de la separación del `if` . Lo usaremos de forma idistinta.

```
if [[ alguna_prueba ]];then
    <comandos>
fi
```

Aquí hay un ejemplo rápido que te pediría que ingreses tu nombre en caso de que lo hayas dejado vacío:

```
#!/bin/bash
#Ejemplo de sentencia if de Bash

read -p "¿Cuál es tu nombre? " nombre

if [[ -z $nombre ]]
then
    echo "¡Por favor, ingresa tu nombre!"
fi
```

---

## Sentencia If Else

Con una sentencia `if-else`, puedes especificar una acción en caso de que la condición en la sentencia `if` no se cumpla. Podemos combinar esto con las expresiones condicionales de la sección anterior de la siguiente manera:

```
#!/bin/bash
# Ejemplo de sentencia if de Bash

read -p "¿Cuál es tu nombre? " nombre

if [[ -z $nombre ]]
then
    echo "¡Por favor, ingresa tu nombre!"
else
    echo "Hola $nombre"
fi
```

Puedes usar la sentencia `if` anterior con todas las expresiones condicionales de los capítulos anteriores:

```
#!/bin/bash
admin="usuario_admin"

read -p "Ingresa tu usuario: " usuario
#Comprueba si el usuario proporcionado es el admin
if [[ "$usuario" == "$admin" ]]
then
    echo "¡Eres el usuario administrador!"
else
    echo "Tú NO eres el usuario administrador."
fi
```

Aquí hay otro ejemplo de una sentencia `if` que comprobaría tu ID de Usuario actual y no te permitiría ejecutar el script como usuario root:

```
#!/bin/bash
if [[ $EUID == 0 ]]; then
    echo "Por favor no ejecutes como root"
    exit
fi
```

Si pones esto al principio de tu script, saldrá en caso de que el EUID sea 0 y no ejecutará el resto del script.

También puedes probar múltiples condiciones con una sentencia `if`. En este ejemplo queremos asegurarnos de que el usuario no sea ni el usuario admin ni el usuario root. Usaremos el operador Y (AND) en este ejemplo, indicado por `&&`.

```
#!/bin/bash
admin="usuario_admin"

read -p "Ingresa tu usuario: " usuario
# Comprueba si el usuario proporcionado es el admin
if [[ "$usuario" != "$admin" ]] && [[ $EUID != 0 ]];
then
    echo "No eres el admin o root, ¡pero ten cuidado!"
else
    echo "¡Eres el usuario admin! ¡Esto podría ser destructivo!"
fi
```

Si tienes múltiples condiciones y escenarios, puedes usar la sentencia `elif` con las sentencias `if` y `else`.

```
#!/bin/bash
read -p "Ingresa un número: " num

if [[ $num -gt 0 ]]; then
    echo "El número es positivo"
elif [[ $num -lt 0 ]]; then
    echo "El número es negativo"
else
    echo "El número es 0"
fi
```

---

## Sentencias Switch case

Como en otros lenguajes de programación, puedes usar una sentencia `case` para simplificar condicionales complejos cuando hay múltiples opciones diferentes. Así que, en lugar de usar varias sentencias `if` y `if-else`, podrías usar una única sentencia `case`.

La sintaxis de la sentencia `case` de Bash se ve así:

Bash

```
case $alguna_variable in
    patron_1)
        comandos
        ;;
    patron_2 | patron_3)
        comandos
        ;;
    *)
        comandos_por_defecto
        ;;
esac
```

Un resumen rápido de la estructura:

- Todas las sentencias `case` comienzan con la palabra clave `case`.
- En la misma línea que la palabra clave `case`, necesitas especificar una variable o una expresión seguida de la palabra clave `in`.
- Después de eso, tienes tus patrones de caso, donde necesitas usar `)` para identificar el final del patrón.
- Puedes especificar múltiples patrones divididos por una barra vertical: `|`.
- Después del patrón, especificas los comandos que te gustaría que se ejecutaran en caso de que el patrón coincida con la variable o la expresión que has especificado.
- Todas las cláusulas deben terminarse añadiendo `;;` al final.
- Puedes tener una sentencia por defecto añadiendo un `*` como patrón.
- Para cerrar la sentencia `case`, usa la palabra clave `esac` (`case` escrito al revés).

Aquí hay un ejemplo de una sentencia `case` de Bash:

```
#!/bin/bash
read -p "Ingresa una tipo de fruta (Manzana, Banana, Naranja): " fruta

case $fruta in
    Manzana)
        echo -n "La ${fruta} es roja o verde."
        ;;
```

```
    Banana | Limon)
        echo -n "La ${fruta} es amarilla."
        ;;
    Naranja)
        echo -n "La ${fruta} es naranja."
        ;;
    *)
        echo -n "${fruta} es una fruta desconocida"
        ;;
esac
```

Con este script, estamos pidiendo al usuario que ingrese un nombre de una fruta. Luego, con una sentencia `case`, comprobamos el nombre de la fruta y si coincide con alguno de nuestros patrones, imprimimos un mensaje. Si el nombre de la fruta no coincide con ninguna de nuestras sentencias `case`, imprimimos un mensaje por defecto.

---

## Bucles de Bash

Como con cualquier otro lenguaje, los bucles son muy convenientes. Con Bash puedes usar bucles `for`, bucles `while` y bucles `until`.

---

### Bucles For

Aquí está la estructura de un bucle `for`:

Bash

```
for var in ${lista}
do
    tus_comandos
done
```

Ejemplo:

Bash

```
#!/bin/bash
nombres="Ana Robero Carlos"
for nombre in ${nombres}
do
    echo "${nombre}"
done
```

Un resumen rápido del ejemplo:

- Primero, especificamos una lista de nombres y almacenamos el valor en una variable llamada `$nombres`.
- Después de eso, comenzamos nuestro bucle `for` con la palabra clave `for`.
- Luego definimos una nueva variable que representaría cada ítem de la lista que damos. En nuestro caso, definimos una variable llamada `nombre`, que representaría cada nombre de la variable `$nombres`.
- Luego especificamos la palabra clave `in` seguida de nuestra lista sobre la que iteraremos.
- En la siguiente línea, usamos la palabra clave `do`, que indica lo que haremos en cada iteración del bucle.
- Luego especificamos los comandos que queremos ejecutar.
- Finalmente, cerramos el bucle con la palabra clave `done`.

También puedes usar `for` para procesar una serie de números. Por ejemplo, aquí hay una forma de iterar del 1 al 10:

Bash

```
#!/bin/bash
for num in {1..10}
do
    echo $num
done
```

---

### Bucles While

La estructura de un bucle `while` es bastante similar a la del bucle `for`:

Bash

```
while [[ tu_condicion ]]
do
    tus_comandos
done
```

Aquí hay un ejemplo de un bucle `while`:

```
#!/bin/bash

contador=1
while [[ $contador -le 10 ]]
do
    echo $contador
```

```
    ((contador++))
done
```

Primero, especificamos una variable `contador` y la establecimos en 1, luego dentro del bucle, incrementamos el `contador` usando esta sentencia: `((contador++))`. De esa manera, nos aseguramos de que el bucle se ejecute solo 10 veces y no se ejecute para siempre. El bucle se completará tan pronto como el contador sea mayor que 10.

Creemos un script que pida al usuario su nombre y no permita una entrada vacía:

Bash

```
#!/bin/bash
read -p "¿Cuál es tu nombre? " nombre
while [[ -z $nombre ]]
do
    echo "Tu nombre no puede estar vacío. ¡Ingresa un nombre válido!"
    read -p "¿Ingresa tu nombre otra vez? " nombre
done
echo "Hola $nombre"
```

Ahora, si ejecutas lo anterior y simplemente presionas enter sin proporcionar una entrada, el bucle se ejecutará de nuevo y te pedirá tu nombre una y otra vez hasta que realmente proporciones alguna entrada.

---

### Bucles Until

La diferencia entre los bucles `until` y `while` es que el bucle `until` ejecutará los comandos dentro del bucle hasta que la condición se vuelva verdadera.

Estructura:

Bash

```
until [[ tu_condicion ]]
do
    tus_comandos
done
```

Ejemplo:

Bash

```
#!/bin/bash
contador=1
until [[ $contador -gt 10 ]]
```

```
do
    echo $contador
    ((contador++))
done
```

---

### Continue y Break

Como con otros lenguajes, también puedes usar `continue` y `break` con tus scripts de bash:

- continue le dice a tu script de bash que detenga la iteración actual del bucle y comience la siguiente iteración.

  La sintaxis de la sentencia continue es la siguiente:

  continue [n]

  El argumento [n] es opcional y puede ser mayor o igual a 1. Cuando se da [n], se reanuda el n-ésimo bucle contenedor. continue 1 es equivalente a continue.

  Bash

  ```
  #!/bin/bash
  for i in 1 2 3 4 5
  do
      if [[ $i -eq 2 ]]
      then
          echo "saltando el número 2"
          continue
      fi
      echo "i es igual a $i"
  done
  ```

- break le dice a tu script de bash que termine el bucle inmediatamente.

  La sintaxis de la sentencia break toma la siguiente forma:

  break [n]

  [n] es un argumento opcional y debe ser mayor o igual a 1. Cuando se proporciona [n], se sale del n-ésimo bucle contenedor. break 1 es equivalente a break.

  Ejemplo:

  Bash

  ```
  #!/bin/bash
  num=1
  while [[ $num -lt 10 ]]
  do
      if [[ $num -eq 5 ]]
      then
          break
      fi
  ```

```
    ((num++))
done
echo "Bucle completado"
```

También podemos usar el comando `break` con múltiples bucles. Si queremos salir del bucle de trabajo actual, ya sea interno o externo, simplemente usamos `break`, pero si estamos en un bucle interno y queremos salir del bucle externo, usamos `break 2`.[1]

Ejemplo:[2]

Bash

```
#!/bin/bash
for (( a=1; a<10; a++ ))
do
    echo "bucle externo: $a"
    for (( b=1; b<100; b++ ))
    do
        if [[ $b -gt 5 ]]
        then
            break 2
        fi
        echo "Bucle interno: $b"
    done
done
```

El script de bash comenzará con `a=1` y se moverá al bucle interno y cuando llegue a `b=6` (ya que `b > 5`), romperá el bucle externo. Podemos usar solo `break` en lugar de `break 2`, para romper el bucle interno y ver cómo afecta la salida.

---

### Funciones de Bash

Las funciones son una gran manera de reutilizar código. La estructura de una función en bash es bastante similar a la mayoría de los lenguajes:

Bash

```
function nombre_funcion() {
    tus_comandos
}
```

También puedes omitir la palabra clave `function` al principio, lo que también funcionaría:

Bash

```
nombre_funcion() {
    tus_comandos
}
```

Prefiero ponerla allí para una mejor legibilidad. Pero es una cuestión de preferencia personal.

Ejemplo de una función "¡Hola Mundo!":

Bash

```
#!/bin/bash
function saludar() {
    echo "¡Hola Mundo desde la Función!"
}
saludar
```

Aviso: Una cosa a tener en cuenta es que no debes añadir los paréntesis cuando llamas a la función.

Pasar argumentos a una función funciona de la misma manera que pasar argumentos a un script:

Bash

```
#!/bin/bash
function saludar() {
    echo "¡Hola $1!"
}
saludar "Mundo"
```

Las funciones deberían tener comentarios mencionando descripción, variables globales, argumentos, salidas y valores devueltos, si aplica:

```
#####
# Descripción: Función de saludo
# Globales:
#   Ninguna
# Argumentos:
#   Un solo argumento de entrada
# Salidas:
#   Valor del argumento de entrada
# Devuelve:
#   0 si es exitoso, distinto de cero si hay error.
#######
function saludar() {
    echo "¡Hola $1!"
}
```