TEMA 6

IMPORTANTE

- Si tienes dudas con cualquier comando puedes ejecutar antes: `comando --help` Ej: chown --help
- Intenta usar rutas absolutas, si no te dicen lo contrario, para que no pierdas el hilo de donde estas y a donde quieres ir. Ej: nano /home/usuario/fichero
- Rutas absolutas: tienes que escribir la ruta completa desde raíz (/)
- Rutas relativas: escribes la ruta desde el directorio donde estás
- ~: indica el home del usuario, es lo mismo que poner /home/nombreusuario
- ./: el directorio en el que estás.
- PWD: comando para comprobar en que directorio estoy

INTRODUCCIÓN

SHEBANG O HASHBANG (!#)

Secuencia de caracteres que se utiliza en el inicio de los scripts de Unix/Linux para indicar el intérprete que se utilizará para ejecutar el script. El shebang comienza con el carácter "#" (hash) seguido del carácter "!" (bang) y del path absoluto o relativo del intérprete. Por ejemplo, el shebang para utilizar Bash como intérprete sería: `#!/bin/bash` No solo sirve para el intérprete bash, también puedes utilizar otros interpretes como por ejemplo Pyhton: `#!/usr/bin/python3` **Es importante destacar que el shebang debe ser la primera línea del archivo y no debe contener espacios ni caracteres adicionales antes o después de la secuencia de caracteres "#!".**

COMENTARIOS (#)

Son una manera de incluir notas y explicaciones en el código, sin que sean interpretados como comandos por el intérprete de Bash. También se pueden utilizar para eivtar que se ejecute un trozo de código.

- **Comentarios en línea**: son comentarios que se incluyen en una sola línea del script. Se indican con el carácter "#" (hash) y todo lo que se escriba después de él hasta el final de la línea será ignorado por el intérprete. `echo "Este es un mensaje" # Este es un comentario`

- **Comentarios en bloque**: son comentarios que se incluyen en varias líneas del script. Se indican con el carácter "#" (hash) al principio de cada línea de comentario y se finalizan con un último carácter "#" (hash).

```
  # Este es un comentario de bloque que explica la sección de código
siguiente
  # Este código realiza una tarea importante
  # y debe ser ejecutado con precaución
```

IMPRIMIR (echo)

Se utiliza para imprimir texto en la salida estándar (por defecto, la pantalla del terminal).

Sintaxis: `echo [opciones] [cadena]` **RECUERDA**: puedes usar el parámetro `--help` para ver todas las opciones del comando.

Donde opciones son una serie de parámetros que pueden modifican el comportamiento del comando, y cadena es el texto que se desea imprimir.

- `-n`: se utiliza para evitar que se imprima una nueva línea después de la cadena. Esto puede ser útil si se quiere concatenar varias cadenas de texto.

```
echo -n "Hola, "

#___RESULTADO
echo "mundo!"
```

- `-e`: se utiliza para permitir el uso de caracteres especiales en la cadena, como por ejemplo \n para representar una nueva línea.

```
echo -e "Hola,\n¿cómo estás?"

#___RESULTADO
Hola,
¿cómo estás?
```

- `-E`: se utiliza para deshabilitar el uso de caracteres especiales en la cadena.

```
  echo -E "Este es un ejemplo de uso de la opción -E para deshabilitar la
interpretación de caracteres especiales como \n, \t y \\ en la cadena."

  #___RESULTADO
  Este es un ejemplo de uso de la opción -E para deshabilitar la
interpretación de caracteres especiales como \n, \t y \ en la cadena.
```

LEER (read)

Se puede utilizar el comando read para solicitar datos al usuario por medio del teclado. El comando read lee una línea de entrada desde el teclado y almacena el resultado en una variable.

Sintaxis:: `read [-options] variable` **RECUERDA**: puedes usar el parámetro `--help` para ver todas las opciones del comando.

Donde options son una serie de parámetros opcionales que pueden modificar el comportamiento del comando, y variable es el nombre de la variable en la que se desea almacenar el resultado.

```
echo "Por favor, ingresa tu nombre:"
read nombre
```

```
echo "Hola, ${nombre}!"
```

- `-p`: se utiliza para imprimir un mensaje de prompt antes de leer la entrada de texto. Por ejemplo: `read -p "Ingrese su nombre: " nombre`
- `-s`: se utiliza para ocultar la entrada de texto mientras se está escribiendo, como por ejemplo para leer contraseñas. Por ejemplo: `read -s -p "Ingrese su contraseña: " contraseña`
- `-n`: se utiliza para especificar el número máximo de caracteres que se leerán. Por ejemplo: `read -n 1 -p "¿Está seguro que desea continuar? (s/n): " respuesta` En este ejemplo, el comando read solo leerá un carácter de la entrada de texto.

VARIABLES

Una variable es un espacio de memoria reservado para almacenar un valor. Las variables en Bash pueden contener diferentes tipos de datos, como cadenas de texto, números enteros y valores booleanos.

Sintaxis: `nombre_variable=valor` **Es importante tener en cuenta que no debe haber espacios antes o después del signo igual, ya que esto provocaría un error** Ejemplos:

```
nombre="Juan"
edad=25
activo=true
```

Tipos:

- **Variables de entorno**: son variables que se utilizan para configurar el entorno de trabajo del usuario. Estas variables se definen en el archivo de configuración `.bashrc` o `.bash_profile`, y están disponibles para todos los procesos que se ejecuten en el entorno.
- **Variables de posición**: son variables que se utilizan para almacenar los argumentos que se pasan al script de Bash desde la línea de comandos. Por ejemplo, el primer argumento se almacena en la variable `$1`, el segundo argumento en `$2`, y así sucesivamente
- **Variables de shell**: son variables que se utilizan para configurar el comportamiento de la shell de Bash. Por ejemplo, la variable `$PS1` se utiliza para definir el prompt de la shell
- **Variables de usuario**: son variables que se utilizan para almacenar información específica del usuario. Estas variables pueden ser creadas y utilizadas por el usuario según sus necesidades.

VARIABLES DE POSICIÓN

Estas variables son muy útiles para hacer que los scripts de Bash sean más flexibles y reutilizables, ya que permiten que los argumentos del script cambien sin tener que modificar el propio script.

- `$0`: Esta variable almacena el nombre del script actual.
- `$1, $2, $3, etc`: Estas variables almacenan los argumentos que se pasan al script de Bash. `./mi_script.sh hola mundo` entonces `$1` será igual a "hola" y `$2` será igual a "mundo".
- `$#`: Esta variable almacena el número total de argumentos pasados al script. `./mi_script.sh hola mundo` entonces $# será igual a 2.

- `$*`: Esta variable almacena todos los argumentos como una única cadena de texto separada por espacios. `./mi_script.sh hola mundo` entonces $* será igual a "hola mundo"
- `$@`: Esta variable almacena todos los argumentos como una lista separada por espacios. `./mi_script.sh hola mundo` entonces $@ será igual a "hola" "mundo".
- `$?`: Esta variable almacena el estado de salida del último comando ejecutado. Si el comando se ejecuta correctamente, entonces `$?` será igual a 0. Si el comando falla, entonces `$?` será un valor distinto de 0.

```
ls
echo "El estado de salida es: $?"
```

- `$$`: Esta variable almacena el número de identificación del proceso actual `echo "El ID del proceso actual es: $$"`
- `$!`: Esta variable almacena el ID del proceso más reciente en segundo plano.

```
sleep 5 &
echo "El ID del proceso en segundo plano es: $!
```

- `$-`: Esta variable almacena las opciones de línea de comandos utilizadas para iniciar el shell actual `echo "Las opciones de línea de comandos son: $-"`

CONCATENAR VARIABLES

Se puede utilizar el operador de concatenación + o simplemente escribir las variables juntas sin ningún operador. Es recomendable concatenar variables sin utilizar el operador + para evitar confusiones y errores en el código

```
var1="Hola"
var2="mundo"
var3=$var1$var2
echo $var3
```

o usando el operador +

```
var1="Hola"
var2="mundo"
var3=$var1+$var2
echo $var3
```

OPERACIONES ARIGMÉTICAS

Es importante tener en cuenta que para realizar operaciones aritméticas en Bash, se debe encerrar la operación entre $((...)).

- Suma (+): `echo $((5 + 3))`
- Resta (-): `echo $((5 - 3))`
- Multiplicación (*): `echo $((5* 3))`
- División (/): `echo $((6 / 2))`
- Resto o Módulo (%): `echo $((7 % 2))`
- PreIncremento (++num): aumenta el valor de la variable en una unidad antes de utilizar su valor `echo $((++x))`
- PostIncremento (num++): utiliza el valor de la variable y luego la incrementa en una unidad `echo $((x++))`
- PreDecremento (--num): disminuye el valor de la variable en una unidad antes de utilizar su valor `echo $((--x))`
- PostDecremento (num--): utiliza el valor de la variable y luego la decrementa en una unidad `echo $((x--))`
- Potencia (**): `echo $((2** 3))`
- Operadores de comparación: Devuelven un valor booleano (true o false) según si la comparación es verdadera o falsa
  - Igualdad (==): `if [ 5 == 5 ]; then echo "true"; fi` resultado: `true`
  - Diferencia (!=): `if [ 5 != 6 ]; then echo "true"; fi` resultado: `true`
  - Mayor que (>): `if [ 5 > 3 ]; then echo "true"; fi` resultado: `true`
  - Menor que (<): `if [ 5 < 3 ]; then echo "true"; fi` resultado: `false`

ARRAYS

Un array es una colección ordenada de elementos que se pueden acceder y manipular mediante un índice numérico. Los arrays en Bash son unidimensionales, lo que significa que solo se pueden almacenar elementos en una sola dimensión (una lista). Cada elemento del array se separa con un espacio. **Sintaxis**: `nombre_array=(elemento1 elemento2 elemento3 ...)`

CREAR ARRAYS VACIOS

Se pueden crear arrays vacíos sin elementos de la siguiente manera: `nombre_array=()`

ACCEDER A ELEMENTO ESPECÍFICO

Para acceder a un elemento específico del array, se utiliza el índice numérico entre corchetes.

```
nombres=("Juan" "Pedro" "María")
echo ${nombres[0]}  # Imprime "Juan"
```

ACCEDER A TODOS LOS ELEMENTOS

Se puede acceder a todos los elementos del array con la notación de expansión de parámetros `${nombre_array[@]}`:

```
nombres=("Juan" "Pedro" "María")
echo ${nombres[@]}  # Imprime "Juan Pedro María"
```

TOTAL DE ELEMENTOS

Obtener el número total de elementos de un array

```
nombres=("Juan" "Pedro" "María")
echo ${#nombres[@]}  # Imprime 3
```

AGREGAR ELEMENTO AL PRINCIPIO

Para agregar un elemento al inicio de un array en Bash

```
nombres=("Juan" "Pedro" "María")
nombres=("Lucía" "${nombres[@]}")
echo ${nombres[@]}  # Imprime "Lucía Juan Pedro María"
```

AGREGAR ELEMENTO AL FINAL

Para agregar un elemento al final del array, se utiliza el operador +=.

```
nombres=("Juan" "Pedro" "María")
nombres+=("Lucía")
echo ${nombres[@]}  # Imprime "Juan Pedro María Lucía"
```

CAMBIAR ELEMENTO

Para cambiar un elemento específico de un array en Bash, se puede acceder al elemento utilizando su índice numérico y asignarle un nuevo valor.

```
nombres=("Juan" "Pedro" "María")
nombres[1]="Luis"
echo ${nombres[@]}  # Imprime "Juan Luis María"
```

También se puede utilizar la notación de expansión de parámetros

```
nombres=("Juan" "Pedro" "María")
nombres=("${nombres[@]:0:1}" "Luis" "${nombres[@]:2}")
echo ${nombres[@]}  # Imprime "Juan Luis María"
```

ELIMINAR ELEMENTO

También se pueden eliminar elementos del array con el comando unset.

```
nombres=("Juan" "Pedro" "María")
unset nombres[1]
echo ${nombres[@]}  # Imprime "Juan María"
```

CONTROL DE FLUJO

COMPARACIONES

- **CADENAS ALFANUMÉRICAS**
  - Igual (=): `cadena1 = cadena2`
  - Distinto (!=): `cadena1 != cadena2`
  - Menor que (<): `cadena1 < cadena2`
  - Mayor que (>): `cadena1 > cadena2`
  - No es nulo (-n): `-n cadena1` longitud de cadena mayor que 0
  - Es nulo (-z): `-z cadena1` longitud de cadena igual a 0
- **VALORES NUMÉRICOS**
  - Igual (-eq): `5 -eq 2` resultado: `false`
  - Distinto (-ne): `5 -ne 2` resultado: `true`
  - Menor que (-lt): `5 -lt 2` resultado: `false`
  - Menor o igual que (-le): `5 -le 2` resultado: `false`
  - Mayor que (-gt): `5 -gt 2` resultado: `true`
  - Mayor o igual que (-ge): `5 -ge 2` resultado: `true`

IF-THEN / IF-ELSE

Es una de las estructuras de control de flujo más comunes en los scripts de Bash. Se utiliza para realizar una acción condicionalmente, dependiendo de si se cumple una determinada condición ☐ **y debe haber un espacio entre el corchete izquierdo y la condición, así como otro espacio entre la condición y el corchete derecho. Además, es importante que después de la condición se escriba un punto y coma ; y la palabra then en la misma línea. La línea que comienza con then debe terminar con un salto de línea.**

IF - THEN

Sintaxis:

```
if [ condición ]; then
  # acciones a realizar si la condición es verdadera
fi
```

Ejemplo:

```
read -p "Introduce un número: " num

if [ $num -gt 0 ]; then
  echo "El número introducido es positivo."
fi
```

IF - THEN - ELSE

La estructura else se utiliza para especificar qué hacer en caso de que la condición en el if no se cumpla. Es decir, si la condición evaluada es falsa, se ejecutará el bloque de código dentro de la estructura else.

Es importante destacar que, en la estructura if-then-else, siempre se ejecutará una de las dos ramas (la del if o la del else), nunca ambas. Sintaxis:

```
if [ condición ]; then
  # código a ejecutar si la condición es verdadera
else
  # código a ejecutar si la condición es falsa
fi
```

Ejemplo:

```
read -p "Introduce un número: " num

if [ $num -gt 0 ]; then
  echo "El número introducido es positivo."
else
  echo "El número introducido es negativo o cero."
fi
```

IF - THEN - ELIF - THEN - ELSE

La estructura de control if-elseif-else en Bash se utiliza cuando se desea evaluar múltiples condiciones **Es importante tener en cuenta que tanto el then como el elif y el else deben estar en la misma línea del if, y que se debe incluir el fi al final del bloque. Además, el bloque de código dentro de cada if, elif o else puede contener uno o varios comandos de Bash.**

Sintaxis:

```
if [ condición1 ]; then
  # código a ejecutar si la condición1 es verdadera
elif [ condición2 ]; then
```

```
  # código a ejecutar si la condición2 es verdadera
elif [ condición3 ]; then
  # código a ejecutar si la condición3 es verdadera
else
  # código a ejecutar si ninguna de las condiciones es verdadera
fi
```

Ejemplo:

```
edad=18

if [ $edad -lt 18 ]; then
  echo "Eres menor de edad."
elif [ $edad -ge 18 ] && [ $edad -le 65 ]; then
  echo "Eres mayor de edad y estás en edad de trabajar."
else
  echo "Eres mayor de edad y estás en edad de jubilarte."
fi
```

WHILE

El bucle while en Bash se utiliza para repetir un conjunto de comandos mientras se cumpla una condición. **Sintaxis:**

```
while [condición]
do
  comandos a repetir
done
```

Ejemplo:

```
i=1
while [ $i -le 10 ]
do
  echo $i
  i=$((i+1))
done
```

FOR

El bucle for en Bash se utiliza para repetir un conjunto de comandos un número fijo de veces o para iterar sobre una lista de elementos. **Sintaxis:**

```

```

```
for variable in lista
do
  comandos a repetir
done
```

Ejemplo:

```
for i in 1 2 3 4 5
do
  echo $i
done
```

Ejemplo 2: imprimir los nombres de los archivos de un directorio

```
for file in /home/usuario/documentos/*
do
  echo $file
done
```

CASE

El comando case en Bash se utiliza para seleccionar una serie de comandos a ejecutar según el valor de una variable. **Sintaxis:**

```
case variable in
patrón1)
  comandos si variable coincide con patrón1
  ;;
patrón2)
  comandos si variable coincide con patrón2
  ;;
*)
  comandos si variable no coincide con ninguno de los patrones anteriores
  ;;
esac
```

Ejemplo:

```
echo "Ingresa un número del 1 al 3: "
read numero

case $numero in
1)
  echo "El número es uno"
```

```
  ;;
2)
  echo "El número es dos"
  ;;
3)
  echo "El número es tres"
  ;;
*)
  echo "Número inválido"
  ;;
esac
```

Explicación: En este ejemplo, se solicita al usuario que ingrese un número del 1 al 3. Luego, el comando case evalúa el valor de la variable numero y ejecuta los comandos correspondientes al patrón que coincida con el valor de la variable. Si el valor de numero es 1, se ejecutan los comandos dentro del primer patrón, que en este caso es imprimir "El número es uno". Si el valor de numero es 2, se ejecutan los comandos dentro del segundo patrón, que en este caso es imprimir "El número es dos". Si el valor de numero es 3, se ejecutan los comandos dentro del tercer patrón, que en este caso es imprimir "El número es tres". Si el valor de numero no coincide con ninguno de los patrones anteriores, se ejecutan los comandos dentro del patrón "*)", que en este caso es imprimir "Número inválido".

UNTIL

El bucle until en Bash es similar al bucle while, con la única diferencia de que se ejecuta mientras la condición es falsa. Es decir, el bucle continuará ejecutándose hasta que la condición sea verdadera. **Sintaxis:**

```
until [condición]
do
  comandos a repetir
done
```

Ejemplo:

```
i=1
until [ $i -gt 10 ]
do
  echo $i
  i=$((i+1))
done
```

Explicación: En este ejemplo, el bucle until se utiliza para imprimir los números del 1 al 10 en la pantalla. El bucle comienza con la variable "i" inicializada en 1, y la condición es que "i" debe ser mayor que 10. Mientras la condición sea falsa, se ejecutan los comandos dentro del bucle, que en este caso es imprimir el valor actual de "i" en la pantalla con el comando "echo", y luego incrementar el valor de "i" en 1 mediante la expresión "i=$((i+1))". Cuando "i" llega a 11, la condición se cumple y el bucle termina.

FUNCIONES

¿Qué son las funciones?

Son bloques de código reutilizables que se pueden llamar desde cualquier parte de un script.

¿Cómo se declaran?

se definen utilizando la palabra clave "function" o mediante la sintaxis abreviada "()". **Sintaxis**:

```
function nombre_de_la_funcion {
  comandos_a_ejecutar
}
```

En esta sintaxis, "nombre_de_la_funcion" es el nombre de la función y "comandos_a_ejecutar" son los comandos que se ejecutarán cuando se llame a la función.

Ejemplo:

```
function imprimir_mensaje {
  echo "Hola, esta es una función en Bash"
}

imprimir_mensaje
```

En este ejemplo, la función "imprimir_mensaje" se define utilizando la palabra clave "function", y el comando "echo" se utiliza para imprimir un mensaje en la pantalla. Luego, se llama a la función utilizando el nombre de la función, "imprimir_mensaje", seguido de los paréntesis vacíos "()".

FUNCIONES CON PARÁMETROS

Los parámetros se pasan a la función dentro de los paréntesis de la función, separados por espacios. Los parámetros se pueden utilizar dentro de la función como variables.

Ejemplo:

```
function imprimir_mensaje {
  nombre=$1
  echo "Hola, $nombre. Esta es una función en Bash"
}

imprimir_mensaje "Juan"
```

En este ejemplo, la función "imprimir_mensaje" acepta un parámetro "nombre", que se utiliza dentro de la función para personalizar el mensaje que se imprime en la pantalla. Cuando se llama a la función, se pasa el parámetro "Juan" dentro de los paréntesis.

FUNCIONES QUE DEVUELVEN VALORES

Las funciones en Bash también pueden devolver valores a la sección de código que las llama. Para lograr esto, se utiliza el comando "return" dentro de la función.

Sintaxis:

```
function nombre_de_la_funcion {
  comandos_a_ejecutar
  return valor
}
```

Ejemplo:

```
function multiplicar {
  num1=$1
  num2=$2
  resultado=$((num1 * num2))
  return $resultado
}

echo "El resultado de multiplicar 5 y 7 es $(multiplicar 5 7)"
```