1. ¿Qué es el Shell Scripting? ¿A qué tipos de tareas están orientados los script? ¿Los scripts deben compilarse? ¿Por qué?

El Shell Scripting es la práctica de escribir secuencias de comandos (scripts) utilizando un intérprete de comandos o shell en un sistema operativo Unix-like, como GNU/Linux.

¿Que puedo hacer con shell scripts?

- Automatización de tareas
- Aplicaciones interactivas
- Aplicaciones con interfaz grafica (con el comando zenity, por ejemplo)

los scripts en Shell Scripting no necesitan compilarse en el sentido tradicional que se aplica a los lenguajes de programación compilados, como C o C++. Los scripts en Shell Scripting son interpretados directamente por el shell en lugar de ser compilados en código de máquina.

2. Investigar la funcionalidad de los comandos echo y read

Echo: El comando echo se utiliza para mostrar texto o mensajes en la pantalla. Puedes utilizarlo para imprimir mensajes, variables u otra información en la salida estándar (generalmente la pantalla) del terminal. Sua sintaxis básica es la siguiente:

```
echo <mensaje>
```

Ejemplo:

```
echo "Hola, mundo"
```

Esto imprimirá "Hola, mundo" en la pantalla.

También puedes usar variables con echo para mostrar el contenido de una variable. Por ejemplo:

```
variable="Hola Mundo "
```

```
echo $variable
```

Esto imprimirá el contenido de *variable* en la pantalla.

Comando read: El comando read se utiliza para leer la entrada del usuario desde el teclado y asignarla a una variable en el script. Con read, puedes solicitar al usuario que ingrese datos, como números o texto, y luego utilizar esos datos en el script. Su sintaxis básica es la siguiente:

```
read <variable>
```

Por ejemplo, para solicitar al usuario que ingrese su nombre y asignarlo a la variable nombre, puedes hacer lo siguiente:

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

```
read nombre
```

El valor ingresado por el usuario se almacenará en la variable nombre y estará disponible para su uso posterior en el script.

¿Como se indican los comentarios dentro de un script?

Dentro de un script de Shell Scripting, los comentarios se indican utilizando el símbolo #. Cualquier texto que siga a # en una línea se considera un comentario y no se ejecutará como parte del script.

```
# Esto es un comentario en una línea
```

¿Cómo se declaran y se hace referencia a variables dentro de un script?

Para declarar una variable, simplemente asigna un valor a un nombre de variable. No es necesario declarar el tipo de variable, ya que las variables en Shell Scripting son de tipo dinámico.

```
mi_variable="Hola, mundo"
```

```
numero=42
```

Es importante que no haya espacios alrededor del signo igual (=) al asignar un valor a una variable.

Para hacer referencia a una variable, utiliza el nombre de la variable precedido por el signo del dólar ($)

```
echo $mi_variable
```

3. sustitución de comandos. ¿Qué significa esto?

Es una característica en Shell Scripting que permite tomar la salida de un comando y utilizarla como entrada o parte de una expresión en otro comando o contexto.

Sustitución de comandos con acento grave (` `):

```
fecha_actual=`date`
```

Sustitución de comandos con $():

```
fecha_actual=$(date)
```

Ambos ejemplos son equivalentes y almacenan la salida del comando *date* en la variable

4. Parametrización: ¿Cómo se acceden a los parámetros enviados al script al momento de su invocación? ¿Qué información contienen las variables $#, $*, $? Y $HOME dentro de un script?

Los scripts pueden recibir argumentos en su invocación. Supongamos que tienes un script llamado mi_script.sh y deseas pasarle dos parámetros:

```
./mi_script.sh parametro1 parametro2
```

Dentro del script mi_script.sh, puedes acceder a estos parámetros de la siguiente manera:

```
echo "El primer parámetro es: $1"
```

```
echo "El segundo parámetro es: $2"
```

Cuando ejecutas el script de la manera que se muestra arriba, verás la siguiente salida:

```
El primer parámetro es: parametro1
```

```
El segundo parámetro es: parametro2
```

Puedes acceder a más parámetros utilizando $3, $4, y así sucesivamente, para los siguientes parámetros.

$# (Número de Parámetros):

La variable $# contiene el número de parámetros que se pasaron al script. Por ejemplo, si ejecutas un script con tres argumentos, $# será igual a 3.

```
echo "Número de parámetros: $#"
```

***$ (Todos los Parámetros):**

La variable $* representa todos los parámetros que se pasaron al script como una sola cadena. Los parámetros se separan por espacios en blanco.

$? (Código de Salida del Último Comando):

La variable $? contiene el código de salida del último comando ejecutado en el script. Un valor de 0 generalmente indica que el comando se ejecutó con éxito

La variable $HOME contiene la ruta al directorio personal del usuario que ejecutó el script. Puedes usar $HOME para referenciar el directorio personal del usuario en tus scripts.

5.¿Cual es la funcionalidad de comando exit? ¿Qué valores recibe como parámetro y cual es su significado?

Para terminar un script usualmente se utiliza la funcion exit:

- Causa la terminacion de un script
- Puede devolver cualquier valor entre 0 y 255:
  El valor 0 indica que el script se ejecuto de forma exitosa
  Un valor distinto indica un codigo de error
  Se puede consultar el exit status imprimiendo la variable $?

Puede recibir un valor numérico como parámetro, que se denomina "código de salida" o "código de retorno". El código de salida es una forma de indicar si el programa se ejecutó con éxito o si ocurrió algún error durante su ejecución.

Sintaxis básica:

```
exit [código de salida]
```

Comprobación del Código de Salida: Después de que un programa o script se ejecuta y sale, puedes comprobar su código de salida utilizando la variable especial $?. Por ejemplo:

```
./mi_script.sh
codigo_salida=$?
echo "El código de salida fue: $codigo_salida"
```

Utilidad: El código de salida es útil en scripts y programas para automatizar procesos, controlar flujos de trabajo y detectar errores. Puedes usarlo en combinación con condicionales (if) para tomar decisiones basadas en el resultado de la ejecución de un programa.

```
#!/bin/bash
# Realizar alguna tarea
if [ $? -eq 0 ]; then
  echo "La tarea se completó con éxito."
else
  echo "Ocurrió un error durante la tarea."
fi
# Finalizar el script con un código de salida
exit 1  # Código de salida no nulo para indicar un error
```

En este ejemplo, el script realiza una tarea y luego verifica el código de salida para determinar si se completó con éxito o si ocurrió un error. Luego, utiliza el comando exit para salir del script con un código de salida adecuado.

Nótese que el código de salida de un comando o script, generalmente representado por la variable $?, se puede verificar mientras el script está en ejecución.

6. El comando expr permite la evaluación de expresiones. Su sintaxis es: expr arg1 op arg2, donde arg1 y arg2 representan argumentos y op la operación de la expresión. Investigar que tipo de operaciones se pueden utilizar.

El comando expr se utiliza en la línea de comandos de Unix y en scripts de Shell para evaluar expresiones y realizar cálculos aritméticos o de cadenas. La sintaxis básica de expr es la siguiente:

```
expr EXPRESIÓN
```

Donde EXPRESIÓN es la expresión que deseas evaluar. Las expresiones pueden involucrar operadores aritméticos, operadores de cadenas y funciones incorporadas.

Ejemplo de expresión aritmética:

```
cadena1="Hola, "
cadena2="Mundo!"
resultado=$(expr "$cadena1" : '\(.*\)'"$cadena2")
echo "El resultado es: $resultado"
```

Ejemplo de expresión de cadenas:

```
cadena="¡Hola, Mundo!"
longitud=$(expr length "$cadena")
echo "La longitud de la cadena es: $longitud"
```

7. El comando “test expresión” permite evaluar expresiones y generar un valor de retorno, true o false. Este comando puede ser reemplazado por el uso de corchetes de la siguiente manera [expresión]. Investigar que tipo de expresiones pueden ser usadas con el comando test. Tenga en cuenta operaciones para: evaluación de archivos, evaluación de cadenas de caracteres y evaluaciones numéricas.

El comando test o su equivalente [ ] se utiliza en Shell Scripting para evaluar expresiones condicionales. La sintaxis básica es la siguiente:

```
test EXPRESIÓN
```

O usando la notación [ ]:

```
[ EXPRESIÓN ]
```

Donde EXPRESIÓN es la expresión condicional que deseas evaluar. Puedes utilizar operadores de comparación y otros operadores lógicos en la expresión.

Ejemplo de prueba de igualdad numérica:

```
numero1=5
numero2=5

if [ "$numero1" -eq "$numero2" ]; then
  echo "Los números son iguales."
else
  echo "Los números son diferentes."
fi
```

Ejemplo de prueba de igualdad de cadenas:

```
cadena1="hola"
cadena2="mundo"

if [ "$cadena1" = "$cadena2" ]; then
  echo "Las cadenas son iguales."
else
  echo "Las cadenas son diferentes."
fi
```

Ejemplo de prueba de existencia de archivo:

```
archivo="/ruta/al/archivo.txt"

if [ -e "$archivo" ]; then
  echo "El archivo existe."
else
  echo "El archivo no existe."
fi
```

8. Estructuras de control. Investigue la sintaxis de las siguientes estructuras de control incluidas en shell scripting: IF , CASE, WHILE, FOR, SELECT

Estructura de control if:

La estructura básica de un bloque if en un script de Bash o en la línea de comandos de Unix/Linux es la siguiente:

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

Ejemplo:

```
numero=15
if [ "$numero" -gt 10 ]; then
  echo "El número es mayor que 10."
else
  echo "El número no es mayor que 10."
fi
```

Estructura de control case:

La estructura básica de un bloque case en un script de Bash o en la línea de comandos de Unix/Linux es la siguiente:

```
case EXPRESIÓN in
  OPCIÓN1)
    # Código a ejecutar si la expresión coincide con OPCIÓN1
    ;;
  OPCIÓN2)
    # Código a ejecutar si la expresión coincide con OPCIÓN2
    ;;
  ...
  *)
    # Código a ejecutar si ninguna de las opciones coincide
    ;;
esac
```

Ejemplo:

```
opcion="B"
case "$opcion" in
  "A")
    echo "La opción es A."
    ;;
  "B")
    echo "La opción es B."
    ;;
  "C")
    echo "La opción es C."
    ;;
  *)
    echo "La opción no es válida."
    ;;
esac
```

Estructura de control while:

La estructura básica de un bucle while en un script de Bash o en la línea de comandos de Unix/Linux es la siguiente:

```
while [ CONDICIÓN ]; do
  # Código a ejecutar mientras la condición sea verdadera
done
```

Ejemplo:

```
contador=1
while [ $contador -le 5 ]; do
  echo "Número: $contador"
  contador=$((contador + 1))
done
```

Estructura de control for:

Existen dos tipos de for:

Bucle for clásico: Este bucle se utiliza para iterar sobre una secuencia de números o elementos definidos. La sintaxis del bucle for clásico es la siguiente:

```
for VARIABLE in SECUENCIA; do
  # Código a ejecutar para cada elemento de la secuencia
done
```

Ejemplo:

```
for i in {1..5}; do
  echo "Número: $i"
done
```

2. Bucle for mejorado o "foreach": Este bucle se utiliza para iterar sobre elementos de una lista, como elementos de un array. La sintaxis del bucle for mejorado es la siguiente:

```
for ELEMENTO in ELEMENTO1 ELEMENTO2 ELEMENTO3 ...; do
  # Código a ejecutar para cada elemento
done
```

Ejemplo:

```
nombres=("Juan" "María" "Pedro" "Luis")

for nombre in "${nombres[@]}"; do
  echo "Hola, $nombre"
done
```

Estructura de control select:

El comando select en Bash se utiliza para crear un menú interactivo que permite al usuario elegir una opción de una lista predefinida. La sintaxis básica del comando select es la siguiente:

```
select VAR in OPCIÓN1 OPCIÓN2 OPCIÓN3 ...; do
  # Código a ejecutar para manejar la opción seleccionada
done
```

VAR: Es una variable que almacenará la opción seleccionada por el usuario

Ejemplo:

```
select opcion in "Opción 1" "Opción 2" "Opción 3" "Salir"; do
  case "$opcion" in
    "Opción 1")
      echo "Has seleccionado la Opción 1."
      ;;
    "Opción 2")
      echo "Has seleccionado la Opción 2."
      ;;
    "Opción 3")
      echo "Has seleccionado la Opción 3."
      ;;
    "Salir")
      echo "Saliendo del menú."
      break
      ;;
    *)
      echo "Opción no válida. Introduce un número del 1 al 4."
      ;;
  esac
done
```

9. ¿Qué acciones realizan las sentencias break y continue dentro de un bucle? ¿Qué parámetros reciben?

Break: La sintaxis de la sentencia break en Bash scripting es muy simple. Se utiliza dentro de bucles (como for, while, o until) para salir de ellos prematuramente

Ejemplo:

```
contador=1

while [ $contador -le 5 ]; do
  if [ $contador -eq 3 ]; then
    break  # Salir del bucle cuando contador sea igual a 3
  fi
  echo "Número: $contador"
  contador=$((contador + 1))
done

echo "Bucle terminado."
```

En este ejemplo, el bucle while cuenta del 1 al 5, pero se utiliza break para salir del bucle cuando contador es igual a 3. Como resultado, el bucle se detiene prematuramente y el programa imprime "Bucle terminado."

La sintaxis del comando break en Bash scripting permite especificar un número opcional n que indica cuántos niveles de bucles anidados se deben romper. Esto es útil cuando tienes bucles anidados y deseas salir de uno específico sin afectar a los bucles más externos.

Ejemplo:

```
for i in {1..3}; do
  echo "Iteración externa: $i"
  for j in A B C; do
    echo "  Iteración interna: $j"
    if [ "$i" -eq 2 ] && [ "$j" = "B" ]; then
      break 2  # Salir de los dos niveles de bucles
    fi
  done
done
```

Continue:

La sintaxis del comando continue en Bash scripting se utiliza dentro de bucles (como for, while, o until) para saltar a la siguiente iteración del bucle actual, omitiendo el resto del código en ese ciclo.

Ejemplo:

```
for i in {1..5}; do
  if [ $i -eq 3 ]; then
    continue  # Saltar a la siguiente iteración cuando i sea igual a 3
  fi
  echo "Número: $i"
done
```

En este ejemplo, el bucle for cuenta del 1 al 5, pero utiliza continue para saltar la iteración cuando i es igual a 3. Como resultado, el número 3 no se imprimirá, y el bucle continuará con las iteraciones restantes.

La sintaxis del comando continue en Bash scripting permite especificar un número opcional n que indica cuántos niveles de bucles anidados se deben saltar. Esto es útil cuando tienes bucles anidados y deseas saltar de uno específico sin afectar a los bucles más externos.

Ejemplo:

```
for i in {1..3}; do
  echo "Iteración externa: $i"
  for j in A B C; do
    echo "  Iteración interna: $j"
    if [ "$i" -eq 2 ] && [ "$j" = "B" ]; then
      continue 2  # Saltar al siguiente ciclo de los dos niveles de bucles
    fi
  done
done
```

10. ¿Qué tipo de variables existen? ¿Es shell script fuertemente tipado? ¿Se pueden definir arreglos? ¿Cómo?

Hay dos tipos de variables:

Variables de cadena (String Variables): Estas variables almacenan cadenas de texto o caracteres. Se definen sin ningún tipo de declaración de tipo y pueden contener texto, números, o cualquier combinación de caracteres.

Variables numéricas (Numeric Variables): Estas variables almacenan valores numéricos, como números enteros o números de punto flotante. Al igual que las variables de cadena, no es necesario declarar explícitamente el tipo de variable en Bash, ya que Bash es un lenguaje de programación de tipado dinámico

Para la catedra de ISO la distinción entre estos dos tipos de variables no tiene importancia excepto a la hora de usar operadores.

Operadores para condition:

Operador	Con strings	Con números
Igualdad	`"$nombre" = "Maria"`	`$edad -eq 20`
Desigualdad	`"$nombre" != "Maria"`	`$edad -ne 20`
Mayor	`A > Z`	`5 -gt 20`
Mayor o igual	`A >= Z`	`5 -ge 20`
Menor	`A < Z`	`5 -lt 20`
Menor o igual	`A <= Z`	`5 -le 20`

Bash es un lenguaje de programación débilmente tipado. Esto significa que no es necesario declarar explícitamente el tipo de una variable, y una variable puede cambiar de tipo durante la ejecución del programa sin restricciones.

en Bash es posible definir arreglos. Los arreglos son variables que pueden almacenar múltiples valores bajo un solo nombre. Sintaxis:

```
nombre_del_arreglo=(valor1 valor2 valor3 ...)
```

Ejemplo:

```
frutas=("manzana" "plátano" "naranja" "uva")

# Acceder a un elemento del arreglo
echo "La primera fruta es: ${frutas[0]}"

# Iterar a través de todos los elementos del arreglo
for fruta in "${frutas[@]}"; do
  echo "Fruta: $fruta"
done
```

Nótese que el primer índice de los arreglos de bash es 0

11. ¿Pueden definirse funciones dentro de un script? ¿Cómo? ¿Cómo se maneja el pasaje de parámetros de una función a la otra?

Las funciones en Bash son bloques de código reutilizables que pueden realizar una tarea específica. Sintaxis:

```
nombre_de_la_funcion() {
  # Código de la función
  # ...
}
```

Ejemplo:

```
# Definir una función llamada saludar
saludar() {
  echo "Hola, ¿cómo estás?"
}

# Llamar a la función
saludar
```

Pasar parámetros a una función en Bash:

```
nombre_de_la_funcion() {
  parametro1=$1
  parametro2=$2
  # ...
}
```

Dentro de la función, puedes acceder a los parámetros utilizando las variables $1, $2, $3, etc., donde $1 se refiere al primer parámetro, $2 al segundo, y así sucesivamente.

Ejemplo:

```
# Definir una función que toma dos parámetros
concatenar() {
  resultado="$1$2"
  echo "La concatenación de '$1' y '$2' es: $resultado"
}

# Llamar a la función y pasarle dos parámetros
concatenar "Hola, " "mundo"
```

En este ejemplo, la función "concatenar" toma dos parámetros, los concatena y muestra el resultado.

También puedes acceder a todos los parámetros pasados a la función utilizando "$@", que representa todos los parámetros como una lista.

Ejemplo:

```
# Definir una función que muestra todos los parámetros
mostrar_parametros() {
  echo "Los parámetros pasados a la función son: $@"
}

# Llamar a la función con varios parámetros
mostrar_parametros "uno" "dos" "tres"
```