Tema 3

Creación de scripts Programación en Shell

Administración de Sistemas Operativos

Índice

[Esquema 3](#_Toc170128074)

[Material de estudio 4](#_Toc170128075)

[3.1. Introducción y objetivos. 4](#_Toc170128076)

[3.2. Parámetros y variables. 5](#_Toc170128077)

[3.3. Gestión de la entrada y la salida de datos 8](#_Toc170128078)

[3.4. Condicionales y otras estructuras de control. 11](#_Toc170128079)

[3.5. Funciones 18](#_Toc170128080)

[3.6. Referencias Bibliográficas. 19](#_Toc170128081)

[A fondo 19](#_Toc170128082)

[Entrenamientos 20](#_Toc170128083)

[Test 25](#_Toc170128084)

Esquema

Tabla

Descripción generada automáticamente

Material de estudio

3.1. Introducción y objetivos.

Los comandos de la Shell ofrecen una serie de funcionalidades que pueden verse potenciadas mediante el uso de algunos elementos adicionales como **variables, funciones o estructuras de control**.

La **programación en Shell,** también conocida como Shell scripting, consiste integrar los comandos junto a estos elementos adicionales dentro de scripts que nos ayudan entre otras cosas a automatizar tareas rutinarias

La finalidad de esta unidad es familiarizarte con los elementos principales del lenguaje de programación de **shell-scripts** para poder entender, modificar y reutilizar scripts de terceros, así como elaborar otros de cosecha propia.

Los objetivos que se pretenden conseguir al final de esta unidad son:

* **Comprender y utilizar variables y parámetros**: Los objetivos incluyen aprender a declarar y utilizar variables, así como a pasar argumentos a los scripts mediante parámetros.
* **Manejar el uso de la entrada y salida de datos**: Gestionar las diferentes maneras de proporcionar entradas a un script así como entender las acciones que podemos llevar a cabo con la salida resultante de los mismos.
* **Dominar las estructuras de control**: Comprender las estructuras de control de flujo, como las sentencias **if**, **for** y **while**, para tomar decisiones y repetir acciones en los scripts.
* **Familiarizarse con el uso de las funciones**: Interiorizar la organización y reutilización de código mediante el uso de las funciones.
* **Crear scripts sencillos**: Aplicar los conocimientos adquiridos para escribir scripts simples que realicen tareas como la manipulación de archivos o la automatización de tareas rutinarias.
* **Depurar y optimizar scripts**: Los estudiantes deben saber cómo identificar y depurar errores en los scripts, además de aprender a optimizar el rendimiento de los mismos.

3.2. Parámetros y variables.

Los parámetros y variables son datos asociados a un nombre que se guardan en una determinada posición de memoria. Una vez declarados, los parámetros y variables pueden ser utilizados, accediendo a su valor, desde cualquier parte del script con solo escribir su nombre precedido del símbolo ( $ ).

* **Parámetros**

Los parámetros guardan datos muy concretos, por lo que distinguimos entre distintos tipos:

* Parámetros posicionales: Guardan los argumentos pasados al script cuando este es ejecutado desde la línea de comandos. Estos parámetros se almacenan automáticamente en variables reservadas $1, $2, $3, … siendo $1 la variable que guarda el primer argumento aportado.
* Parámetros especiales: Se guardan en variables especiales y permiten realizar operaciones sobre ellos como la iteración. Algunos ejemplos son:

$#: Número total de parámetros pasados al script

$0: Nombre del script invocado

$$: PID del Shell que ejecuta el script

$@: Variable que recoge en un array todos los parámetros aportados.

* **Variables estandar**

Una variable puede almacenar varios tipos de dato como un número o una cadena. En Bash scripting, como en otros lenguajes de programación, existe una sintaxis propia para definir variables y asignarles un valor:

#Declaración de una variable (definición y asignación de valor).

nombre=”Pepe”

#acceso e impresión por pantalla del valor de la variable

echo “hola $nombre”

* **Arrays**

Por otro lado, tenemos las conocidas como estructuras de datos o arrays. Al contrario que las variables estándar, los arrays son objetos capaces de almacenar de forma estructurada varios datos al mismo tiempo en diferentes posiciones. Este conjunto de datos es iterable, es decir, podemos recorrer un array para consultar, acceder y operar con cualquiera de los datos que se encuentran en sus posiciones, veamos un ejemplo de cómo declararlos:

#Declaración de un array (definición y asignación de valores)

array = (1 2 3 4 5 6 7 8 9 10)

#Declaración de array con diferentes tipos de dato

Array\_mixta=(Uno 2 tres 4 cinco)

Array\_mixta contiene datos de tipo numérico (2 y 4) así como datos de tipo cadena de caracteres o string (Uno, tres y cinco).

Los arrays pueden asemejarse a listas donde cada dato es guardado en una posición.

En este caso se ha declarado un array con diez posiciones siendo la primera posición [0] y la última posición [9]. Podemos acceder a estas posiciones de la siguiente forma:

#Acesso al valor de la primera posición e impresión en pantalla

${array[0]}

De forma similar a las variables, también utilizamos el símbolo ( $ ) para acceder a las posiciones de un array, la diferencia es que debemos introducir entre llaves el nombre del array y especificar entre corchetes la posición concreta a la que queremos acceder.

A continuación veremos algunos ejemplos para las operaciones básicas que tenemos disponibles a la hora de trabajar con arrays. El array que se utilizará como referencia es el siguiente:

# Declaración del array que usaremos para las explicaciones

array = (1 2 3 4 5 6 7 8 9 10)

* **Acceso a los elementos del array imprimiéndolos por pantalla**

|  |  |
| --- | --- |
| **Ejemplo** | **Descripción** |
| **echo ${array[1]}** | Acceso e impresión de la segunda posición del array |
| **echo ${array[@]}** | Acceso e impresión de todas las posiciones del array en una línea |
| **echo ${array[@]:1:3}** | Acceso e impresión de un rango de posiciones  Comienza desde la posición uno e imprime un total de tres posiciones |

Tabla 1. Acceso a los elementos de un array. Fuente: elaboración propia.

* **Conteo de los elementos o posiciones del array para saber su longitud total.**

|  |  |
| --- | --- |
| **Ejemplo** | **Descripción** |
| **echo ${#array[@]}** | Conteo de los elementos de un array (longitud del array) |
| **echo ${#array[1]}** | Conteo de los caracteres dentro de una posición (longitud del elemento) |

Tabla 2 Conteo de elementos de un array. Fuente: elaboración propia.

* **Modificación de los elementos de un array.**

|  |  |
| --- | --- |
| **Ejemplo** | **Descripción** |
| * **array[1]=”dos”** | Modificamos el segundo elemento cambiándolo por la palabra “dos”. (Recordemos que ocupa la posición 1 del array ya que el primer elemento corresponde a la posición 0). |

Tabla 3. Modificación de los elementos de un array. Fuente: elaboración propia.

* **Eliminación de un array completo o de sus elementos.**

|  |  |
| --- | --- |
| **Ejemplo** | **Descripción** |
| * **unset array[2]** | Eliminamos el tercer elemento del array (recordemos que ocupa la posición 2) |
| **unset array** | Eliminación del array completo |

Tabla 4. Eliminación de un array completo o de sus elementos. Fuente: elaboración propia.

3.3. Gestión de la entrada y la salida de datos

Existen varias formas de introducir datos al script, así como guardar otros desde el script en ficheros externos:

* **Comando “echo”**

Es el comando para producir la salida estándar de mensajes por pantalla que serán visibles en la línea de comandos durante la ejecución del script. El comando echo puede utilizarse también para imprimir el valor de variables y consta de varias opciones:

|  |  |
| --- | --- |
| **Opción** | **Comando** |
| **echo -n** | No genera el salto delinea después de imprimir el mensaje, es decir, lo próximo que se imprima irá justo a continuación. |
| **echo -e** | Habilita las opciones de barra invertida que se describen a continuación. Nota: Si utilizamos la opción -e deberemos incluir el texto siguiente entre comillas dobles. |
| **echo -e “hola \\ adiós”** | Inserta una barra inclinada en el texto. |
| **echo -e “hola \c adiós”** | No muestra el texto escrito después del símbolo (\c). |
| **echo -e “hola \r adiós”** | No muestra el texto escrito antes del símbolo (\r). |
| **echo -e “hola \n adiós”** | Inserta un salto de línea entre las palabras. |
| **echo -e “hola \t adiós”** | Inserta una tabulación entre las palabras. |
| **echo -e “hola \e adiós”** | Simula la pulsación del botón “escape” |
| **echo -e “hola \b adiós”** | Simula la pulsación del botón “retroceso” |

Tabla 5. Opciones del comando “echo”. Fuente: elaboración propia.

* **Comando “read”**

Desde dentro del script podemos solicitar información al usuario para después guardarla en una variable, esto lo conseguimos a través del comando “read”, veamos algún ejemplo:

#Enviamos un mensaje por pantalla solicitando el dato al usuario

echo “Por favor, introduzca su nombre”

#Usamos el comando read para guardar el valor introducido en la variable nombre

read nombre

El comando “read” admite la opción (-p) que nos permite indicar el mensaje y recoger el input del usuario en una sola línea prescindiendo del comando echo anterior, veamos la sintaxis con un ejemplo:

read -p “por favor, introduzca su nombre” nombre

* **Operadores de redirección**

Los operadores de redirección en Linux son herramientas esenciales que permiten controlar el flujo de datos entre comandos, archivos y dispositivos. A continuación, se presentan los operadores de redirección más comunes en Linux

Imaginemos que queremos realizar un script que nos liste el contenido de un directorio y guarde el resultado en un fichero llamado listado.txt. Podemos hacerlo de la misma manera que lo haríamos desde la línea de comandos, pero automatizado dentro de un script, el operador de redirección utilizado es ( > ), veamos un ejemplo:

ls -l ./alumnos/cursos > listado.txt

Dentro de un script también podemos acceder a los datos de un fichero externo leyendo sus líneas con una estructura de control de flujo denominada while en conjunto con el operador de redirección ( < ) para indicarle el fichero que deseamos recorrer (estudiaremos los bucles while más adelante). El código quedaría así:

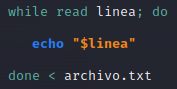


Figura 1. Ejemplo de lógica interna de script. Fuente: elaboración propia.

Cada iteración del bucle guardará una línea del fichero “archivo.txt” en la variable $línea y después la imprimirá por pantalla con el comando echo.

Veamos un ejemplo a modo de repaso donde se observa entrada, manipulación y salida de datos:

Tenemos un fichero llamado Prueba\_paises.txt cuyo contenido es el siguiente:



Figura 2. Ejemplo de lógica interna de script. Fuente: elaboración propia.

Vamos a crear un script al que le vamos a pasar como parámetro este fichero y se encargará de separarlo, en dos ficheros distintos, uno llamado países.txt y otro que llamado capitales.txt con la información que le corresponde a cada uno:

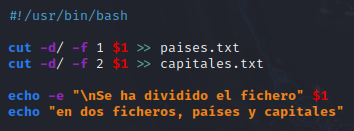


Figura 3. Ejemplo de lógica interna de script. Fuente: elaboración propia.

El resultado después de la ejecución es el siguiente:

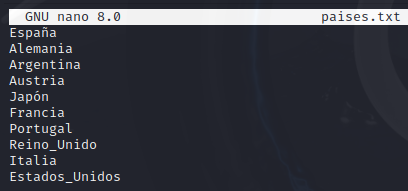
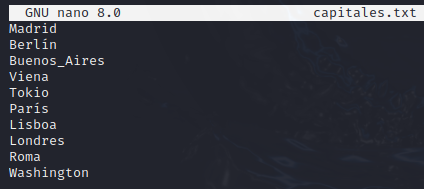


Figura 4. Contenido de los ficheros.txt. Fuente: elaboración propia.

3.4. Condicionales y otras estructuras de control.

Antes de entrar en las estructuras de control de flujo, es recomendable conocer los operadores lógicos y los operadores aritméticos que nos permiten gestionar las condiciones de funcionamiento para dichas estructuras de control:

* **Operadores lógicos**

|  |  |
| --- | --- |
| **Operador en bash** | **Significado** |
| **-lt** | Representa las palabras en inglés “less than” que significan “menor que…”, equivalente en otros lenguajes al signo ( **<** ). Ejemplo: 1 -lt 2 |
| **-gt** | Representa las palabras en inglés “greater than” que significan “mayor que…”, equivalente en otros lenguajes al signo ( **>** ). Ejemplo: 2 -gt 1 |
| **-le** | Representa las palabras en inglés “less equal” que significan “menor o igual que…”, equivalente en otros lenguajes al signo ( **<=** ). |
| **-ge** | Representa las palabras en inglés “greater equal” que significan “mayor o igual que…”, equivalente en otros lenguajes al signo ( **>=** ). |
| **-eq** | Representa la palabra en inglés “equal” que significan “exactamente igual que…”, equivalente en otros lenguajes al signo ( **==** ). |
| **!** | Representa la palabra en inglés “not” que implica negación |
| **-!eq** | Representa las palabras en inglés “not equal” que significan “distinto de…”, equivalente en otros lenguajes al signo ( **<>**  o **!=** ). |
| **&&** | Representa la palabra en inglés “and” que significa (y) o adicionalmente |
| **||** | Representa la palabra en inglés “or” que implica opcionalidad |

Tabla 6. Operadores lógicos. Fuente: elaboración propia.

* **Operadores aritméticos**

|  |  |
| --- | --- |
| **Operador en bash** | **Función** |
| **+** | Suma: Se utiliza para sumar dos o más números. |
| **-** | Resta: Se utiliza para restar dos o más números. |
| **\*** | Multiplicación: Se utiliza para multiplicar dos o más números. |
| **/** | División: Se utiliza para dividir dos o más números. |
| **%** | Módulo: Devuelve el resto resultante de una división |
| **\*\*** | Exponente: Se utiliza para elevar un número a otro |
| **num=$((num+1))** | Suma una unidad a la variable num, en ocasiones se admite (num++) |
| **num=$((num-1))** | Resta una unidad a la variable num, en ocasiones se admite (num--) |

Tabla 7. Operadores aritméticos. Fuente: elaboración propia.

Cabe destacar que el operador de multiplicación ( \* ) pude utilizarse precedido de la barra inclinada de esta manera ( /\* ), encontramos esta variante cuando no queremos que bash confunda este operador con el carácter comodín de un comando.

La ejecución de un script se produce normalmente de izquierda a derecha y de arriba abajo, línea por línea hasta terminar el código, a esto se le conoce como “flujo del script”, sin embargo, existen las denominadas “estructuras de control de flujo” que permiten alterar el funcionamiento de este “flujo normal”.

Estas estructuras son más conocidas como condicionales y estructuras de repetición o bucles. A continuación, se describen algunas de las más comunes:

* **Estructura condicional [ IF ELSE ]**

Son estructuras a las que debemos especificarles una condición, por ejemplo

[ variable1 -lt variable2 ]. Esta estructura permite la ejecución de un bloque de código distinto en función de si la condición establecida se cumple o no, veamos la

sintaxis con un ejemplo:

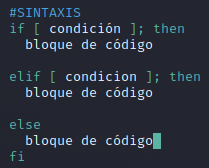
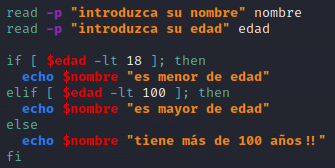


Figura 5. Ejemplo de lógica interna de script. Fuente: elaboración propia.

En el ejemplo podemos observar cómo primero se solicita por consola el valor para dos variables, nombre y edad.

Posteriormente se ejecuta el condicional ( if ) que comprueba la primera condición:

* **If [ $edad -lt 18 ].** Si el valor de la variable edad es menor que 18 se ejecutará el código asociado , en caso contrario la ejecución pasará a la sentencia “elif” que comprueba una segunda condición
* **elif [ $edad -lt 100 ].** Si el valor de la variable edad es menor que 100 y no ha cumplido la anterior condición, quiere decir que el valor de edad está entre 18 y 99 por lo tanto se ejecuta el bloque de código asociado, en caso contrario la ejecución saltaría a la última sentencia, “else”
* **else.** Esta sentencia engloba todas las posibilidades que no recoge la sentencia “if” y la sentencia “elif” ejecutando entonces su bloque de código asociado.

La estructura de control finaliza con la palabra reservada “fi” (if a la inversa).

* **Estructura condicional [ CASE ]**

Cuando se trata de comprobar condiciones simples, basadas en opciones, se suele utilizar la estructura condicional case, que simplemente recoge un número determinado de valores para una variable y los compara con el valor real para ejecutar el bloque de código que le corresponde a cada opción, veamos la sintaxis con un ejemplo :

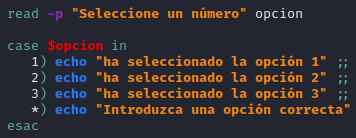
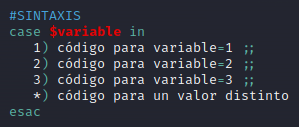


Figura 6. Ejemplo de lógica interna de script. Fuente: elaboración propia.

En este caso solicitamos por consola un numero de opción y con la sentencia “case” comprobamos si la opción introducida es 1), 2) o 3) ejecutando el bloque de código que le corresponde a cada una. Finalmente contemplamos la opción \*) que recoge las opciones distintas a las anteriormente especificadas. La estructura case termina con la palabra reservada esac, (case a la inversa).

* **Estructura de repetición o bucle [ WHILE ]**

La estructura de repetición o bucle while permite repetir un bloque de código basándose en una condición, es decir, si la condición se cumple, se ejecutará el código en el interior del bucle y después se volverá a comprobar si la condición sigue cumpliéndose, en caso afirmativo, volverá a ejecutarse el código hasta que la condición establecida deje de cumplirse. Veamos la sintaxis con un ejemplo:

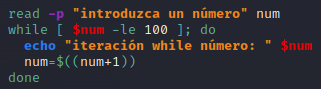
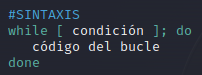


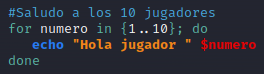
Figura 7. Ejemplo de lógica interna de script. Fuente: elaboración propia.

El ejemplo imprimirá el código del bucle 100 veces ya que la condición para continuar ejecutando es que el valor de la variable num sea menor que 100, dentro del bucle se suma una unidad a la variable por cada iteración y cuando el valor de la variable num sea 101 no cumplirá la condición y por tanto finalizará el bucle.

La palabra clave en la sintaxis del bucle while que cierra la estructura es “done”.

* **Estructura de repetición o bucle [ FOR ]**

Se utiliza cuando queremos repetir un bloque de código un número determinado de veces, veamos la sintaxis con un ejemplo:



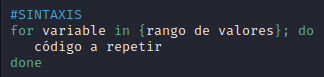


Figura 8. Ejemplo de lógica interna de script. Fuente: elaboración propia.

Con este tipo de bucle , podemos recorrer un array para recuperar sus datos o realizar cualquier operación con ellos, Veamos la sintaxis con un ejemplo:

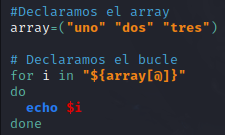


Figura 9. Ejemplo de lógica interna de script. Fuente: elaboración propia.

Existe una variante del bucle for equivalente al bucle while con la siguiente sintaxis:

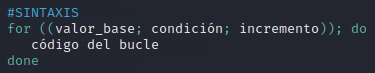
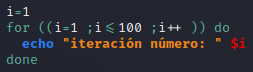


Figura 10. Ejemplo de lógica interna de script. Fuente: elaboración propia.

Cabe destacar que existen dos palabras clave más que suelen utilizarse dentro de los bucles, estas son continue y break. Normalmente se utilizan en un condicional interno del bucle para que causen su efecto en caso de cumplirse la condición:

* **Continue**: Se utiliza para indicar al bucle que debe saltar a la siguiente iteración obviando el código que pueda haber después de la sentencia continue, veamos la sintaxis con un ejemplo:

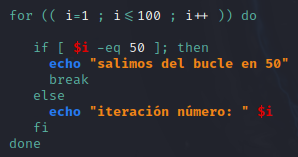


Figura 11. Ejemplo de lógica interna de script. Fuente: elaboración propia.

Primero declaramos un bucle for que comienza con la variable ( i ) en valor 1, en cada iteración sumará una unidad a dicha variable y el bucle será iterado mientras el valor de ( i ) sea menor o igual que 100.

En el interior del bucle declaramos un condicional que establece un límite, es decir, mientras el valor de ( i ) sea distinto de 50 imprimirá la frase “iteración número: ( i ) pero cuando se cumpla la condición imprimirá “salimos del bucle en 50” y acto seguido ejecuta un break que simplemente romperá el bucle terminando así la ejecución del for.

* **Break**: Se utiliza para romper el bucle, independientemente de las iteraciones que puedan quedar pendientes hasta que la condición del bucle se cumpliese, este será interrumpido de inmediato tras la sentencia break, veamos la sintaxis con un ejemplo:

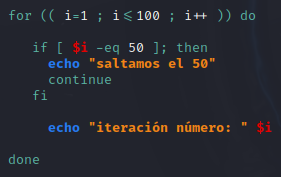


Figura 12. Ejemplo de lógica interna de script. Fuente: elaboración propia.

Primero declaramos un bucle for que comienza con la variable ( i ) en valor 1, en cada iteración sumará una unidad a dicha variable y el bucle será iterado mientras el valor de ( i ) sea menor o igual que 100.

En el interior del bucle declaramos un condicional que saltará la ejecución normal de la iteración número 50, es decir, mientras el valor de ( i ) sea distinto de 50 imprimirá la frase “iteración número: ( i ) pero cuando se cumpla la condición imprimirá “saltamos el 50” y acto seguido ejecuta un continue que simplemente saltará el resto de la ejecución de esta iteración para continuar con la siguiente de inmediato.

**Nota**: debemos prestar especial atención a los espacios en la sintaxis de todas las estructuras de control de flujo ya que en bash scripting son determinantes, por ejemplo, si declaramos un condicional sin respetar los espacios de los corchetes de la siguiente manera:

If [$i -eq 50]; then

El script no funcionará correctamente

3.5. Funciones

Las funciones en bash son estructuras capaces de encapsular un bloque de código para después ser reutilizado dentro del script y evitar así la repetición innecesaria de líneas de código. Veamos las dos maneras posibles de declarar una función:

* Utilizando la palabra reservada ( function ) o la sintaxis simplificada

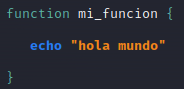
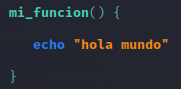


Figura 13. Ejemplo de lógica interna de script. Fuente: elaboración propia.

En cualquiera de los casos podemos invocar a la función para que se ejecute el código de su interior simplemente tecleando su nombre. Por supuesto, podemos utilizar las variables dentro de la función, por ejemplo:

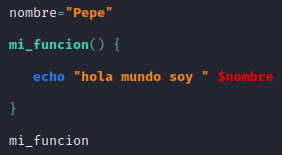


Figura 14. Ejemplo de lógica interna de script. Fuente: elaboración propia.

Podemos declarar variables internas que solo tienen efecto dentro de la función, para ello debemos incluir la palabra local delante de la declaración de la variable

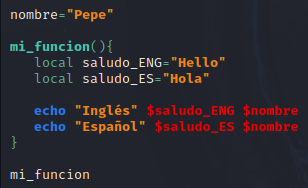


Figura 15. Ejemplo de lógica interna de script. Fuente: elaboración propia.

3.6. Referencias Bibliográficas.

Podemos encontrar más información en la documentación oficial denominada “manual de referencia de bash” dentro de la web gnu.org visitando el siguiente enlace:

<https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html>

Cabe destacar que podemos consultar una breve descripción de cada comando y de sus opciones ejecutando el comando de ayuda –help o man dentro de la línea de comandos, pongamos como ejemplo el comando read para visualizar como recibir está ayuda :

read - -help o man read

A fondo

Curso de Linux: Comandos básicos e introducción a la shell bash.

https://www.youtube.com/watch?v=qWrgoCP6q3M&list=PLN9u6FzF6DLTRhmLLT-ILqEtDQvVf-ChM

Esta lista de reproducción de Youtube encontramos un recorrido por los comandos más comunes de Linux así como los primeros pasos en bash scripting

Curso básico de Bash scripting desde cero

[Curso de BASH SCRIPTING Desde CERO 🐧✅ - YouTube](https://www.youtube.com/playlist?list=PLTDhSVtYdnF7IbtGoKz9UTra4Dae8_tBK)

Esta lista de reproducción cuenta con un curso de ocho videos cortos donde se pueden aprender los fundamentos de bash scripting.

Ejercicios resueltos de Linux Shell Script en Bash

<https://www.youtube.com/watch?v=Y950V89-a-s&list=PLN9u6FzF6DLSdJLrA1U_ss6sPAm3DXDC4>

Esta lista de reproducción nos ofrece un listado de ejercicios resueltos para practicar la lógica de programación y el uso de las estructuras básicas de bash scripting.

Entrenamientos

Entrenamiento 1

* Enunciado

Crea un script llamado borrar\_vacios.sh al que le pases como parámetro la ruta de un directorio para que revise si en su interior existen directorios vacíos y en caso afirmativo proceda a eliminarlos. La ejecución del script solo debe mostrar por consola los echo, es decir no debe aparecer ningún mensaje de error estándar.

* Solución

Primero crearemos la estructura de directorios que nos servirá para comprobar el correcto funcionamiento del script:

* **mkdir -p prueba/dir1 prueba/dir2 prueba/dir3**

A continuación creamos el fichero del script e implementamos la siguiente lógica:

* **nano borrar\_vacios.sh**

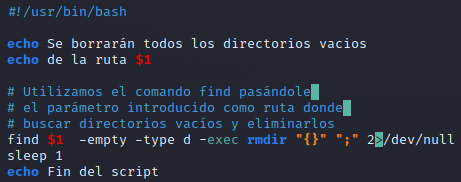


Figura 16. Ejemplo de lógica interna de script. Fuente: elaboración propia.

Por último guardamos el fichero y lanzamos el script pasándole como parámetro la ruta de directorios vacíos que hemos creado anteriormente.

* **bash borrar\_vacios.sh ./prueba**

Observamos que en la lógica del script, la variable $1 hace referencia a la ruta del parámetro sobre la cual ejecutamos el comando find y que estamos anulando la salida de errores estándar con 2>/dev/null

Entrenamiento 2

* Enunciado

Crea un script llamado compara\_numeros.sh que solicite al usuario introducir dos números por consola (num1 y num2). El script debe comparar estos dos números y devolver una de las siguientes situaciones según corresponda.

* El número num1 es mayor que el número num2
* El número num1 es menor que el número num2
* Num1 y num2 son el mismo número
* Solución

Creamos el fichero del script e implementamos la siguiente lógica:

* **nano compara\_numeros.sh**

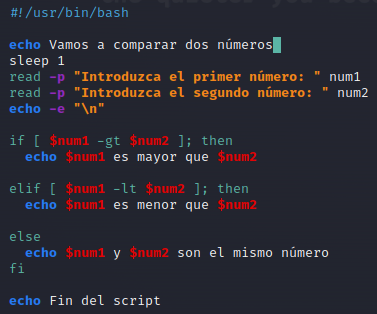


Figura 17. Ejemplo de lógica interna de script. Fuente: elaboración propia.

Observamos el funcionamiento de la estructura de control de flujo condicional que cubre las tres opciones posibles utilizando los operadores lógicos “menor que” (-lt) y “mayor que” (-gt).

Por otro lado observamos la opción -p para el comando read que nos permite introducir el mensaje que mostraremos por consola en el momento de solicitar al usuario el valor de la variable, evitando tener que ejecutar dos líneas de código, primero el comando echo con el mensaje y después el comando read para recoger la respuesta

Entrenamiento 3

* Enunciado

Crea un script llamado consulta\_dias\_mes.sh que muestre por pantalla los meses del año y pida al usuario que seleccione el número que corresponde con el mes a consultar. El script debe devolver el número de días que tiene dicho mes. (utiliza la estructura de control “case” para resolver el ejercicio y ten en cuenta que el proceso debe repetirse hasta que el usuario seleccione la opción “salir”)

* Solución

Creamos el fichero del script e implementamos la siguiente lógica:

* **nano consulta\_dias\_mes.sh**

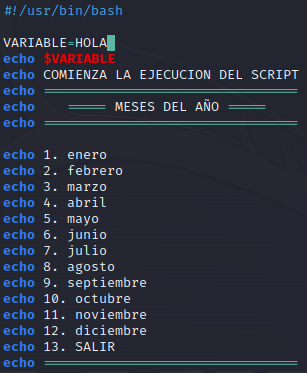
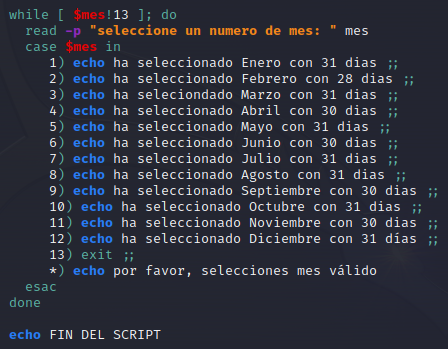


Figura 18. Ejemplo de lógica interna de script. Fuente: elaboración propia.

Observamos el funcionamiento de la estructura de control de flujo (case) que cubre las dice opciones posibles así como la introducción de un dato incorrecto

Por otro lado observamos que el case se encuentra dentro de un bucle while cuya condición de ejecución es que el usuario no seleccione la opción número 13 que corresponde con la orden “salir”.

Entrenamiento 4

* Enunciado

Crea un script llamado crear\_carpetas.sh que reciba un parámetro numérico para crear una estructura de directorios vacíos. El script debe ser capaz de crear un directorio con el nombre “año\_parámetro” y dentro de este deberán crearse doce carpetas, cada una de ellas con el nombre de un mes seguido del año recibido como parámetro. Finalmente se debe listar el contenido del directorio padre para mostrar la estructura de directorios creada. (Utiliza un bucle “for” para resolver el ejercicio).

* Solución

Creamos el fichero del script e implementamos la siguiente lógica:

* **nano crear\_carpetas.sh**

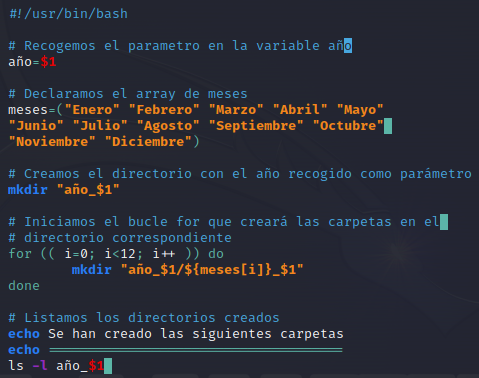


Figura 19. Ejemplo de lógica interna de script. Fuente: elaboración propia.

Observamos la declaración de un array de Strings con los meses del año, a continuación se declara la estructura de control de flujo (for) indicando la variable (i=0) que funcionará como contador incrementándose en 1 por cada iteración.

El bucle se ejecutará mientras el valor de la variable (i) sea menor que 12 y en cada iteración se accede a una posición del array para extraer su valor y utilizarlo como parte del nombre en la creación de un directorio junto al parámetro del año.

Entrenamiento 5

* Enunciado

Crea un script llamado cuenta\_atras.sh con una función que recoja el dato enviado como parámetro y haga las operaciones necesarias para simular una cuenta atrás mostrándola por pantalla, debes controlar que el parámetro introducido no sea ni cero ni un número negativo.

* Solución

Creamos el fichero del script e implementamos la siguiente lógica:

* **nano crear\_carpetas.sh**

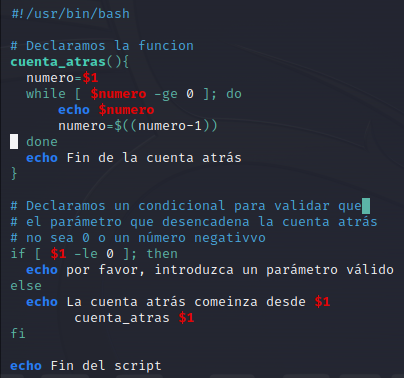


Figura 20. Ejemplo de lógica interna de script. Fuente: elaboración propia.

Observamos la declaración de la función “cuenta\_atras()” que recoge el número introducido como parámetro y ejecuta un bucle while restándole uno hasta llegar a cero. Posteriormente se observa un condicional que valida el parámetro introducido lanzando un mensaje de error en caso de ser cero o negativo y ejecutando la función “cuenta\_atras()” en caso contrario.

Test

**1.** ¿Qué hace el comando echo "Hola, mundo" en un script de Bash?

\_A. Imprime “Hola, mundo” en la pantalla.

B. Crea un archivo llamado “Hola, mundo”.

C. Ejecuta un programa llamado “Hola, mundo”.

D. Muestra un error, no existe el comando “Hola, mundo”1.

El comando echo seguido de un texto, imprime por pantalla dicho texto.

**2.** ¿Cuál es la sintaxis correcta para comentar una línea en un script de Bash?

A. // Este es un comentario.

\_B. # Este es un comentario.

C. /\* Este es un comentario \*/.

D. <!-- Este es un comentario --> 1.

Utilizamos el símbolo ( # ) para comentar el código escrito a continuación.

**3.** ¿Cómo se asigna un valor a una variable en Bash?

\_A. variable = 10

B. variable 🡪 10

C. variable : 10

D. variable > 10

La sintaxis para la declaración de variables en Bash es simple, nombre = valor.

**4.** ¿Cuál es el operador lógico equivalente a >= mayor o igual en Bash?

A. <>

B. !eq

\_C. -ge

D. -lt

El operador -ge hace referencia a las palabras en inglés “greater equal” que traducido al español significa mayor o igual que…

**5.** ¿Cuál es el significado de $1 en un script de Bash?

A. Suma un dollar a la variable.

B. Variable con valor 1.

\_C. Variable que contiene el primer parámetro pasado al script.

D. Símbolo para iniciar una función.

Para obtener el valor los parámetros proporcionados utilizamos $1, $2, $3, etc.

**6.** ¿Cuál de las siguientes opciones es correcta?

\_A. El shebang establece el intérprete de comandos para el script.

B. El shebang es la palabra reservada para declarar un array.

C. El shebang aparece siempre al final del script.

D. La extensión utilizada para los scripts es .shebang.

El shebang es la primera línea del script y proporciona la ruta del intérprete de comandos que utilizará dicho script en caso de no especificarlo cuando se lanza.

**7.** ¿Cómo se obtiene el valor del primer elemento de un array en Bash?

A. $array-0

\_B. ${array[0]}

C. ${array(0)}

D. $array{0}

Usamos las llaves { } para referenciar al array y los corchetes [ ] para acceder a una posición determinada.

**8.** ¿Cuál es la extensión utilizada para los ficheros de script?

A. [ .txt ]

B. [ .bs ]

C. [ .sc ]

\_D. [ .sh ]

La extensión .sh es la abreviatura de Shell utilizada para identificar los scripts de bash.

**9.** ¿Cuál es el operador lógico para la negación en Bash?

A. &&

B. ||

\_C. !

D. -eq

El operador ( ! ) puede asemejarse a la expresión “distinto de…”.

**10.** ¿Cómo se ejecuta un script de Bash que está en nuestro directorio actual?

\_A. bash ./script.sh o ./script.sh (si se tiene permiso de ejecución)

B. execute ./script.sh

C. sudo execute ./script.sh -b

D. nano ./script.sh

Con bash ./script.sh indicamos que, “bash”, será la Shell utilizada en el script, si lanzamos el script sin especificar la Shell deberemos tener permisos de ejecución.

**11.** ¿Cómo evitamos mostrar la salida estándar de un error de comandos en Bash?

A. >/dev/null

B. -nonerr

C. >/null

\_D. 2>/dev/null

2> hace referencia a la salida de error estándar que será redirigida a /dev/null siendo esta la ruta a un dispositivo virtual que absorbe y anula cualquier input que recibe.

**12.** ¿Cómo se obtiene el valor de una variable llamada “contador” en Bash?

\_A) $contador

B) ${contador}

C) $$contador

D) $[contador]

Utilizamos el símbolo $ seguido del nombre de la variable para recuperar su valor.

**13.** ¿Cuál es el significado de la primera línea de un script 🡪 #!/usr/bin/bash?

A. Se trata de la línea de saludo típica de los scripts en Bash.

\_B. Es el “shebang” proporciona la ruta para la Shell utilizada en el script.

C. Imprime por pantalla el nombre del script.

D. Recoge en una variable el parámetro pasado al ejecutar el script.

Si lanzamos el script sin especificar una shell y falta la línea de shebang, el script fallara ya que desconoce cuál es el intérprete que debe usar para ejecutarlo.

**14.** ¿Cómo solicitamos por pantalla al usuario un valor y lo recogemos en la variable “num” utilizando solo una línea de código?

A. echo inserte un número num

B. read “inserte un número” echo num

C. read inserte un número > num

\_D. read -p “inserte un número” num

La opción -p del comando read ahorra líneas de comando ya que posibilita lanzar el mensaje por pantalla y recoger el valor de la variable desde el input del usuario.

**15.** ¿Cuál de las siguientes sentencias se utiliza para cerrar un condicional en Bash?

A. done

B. exit

\_C. fi

D. end

Por norma general en bash scripting se utiliza la misma keyword de apertura (if) pero invertida para el caso del cierre, en este caso (fi). La excepción es el bucle while.

**16.** ¿Cómo se recogen todos los parámetros pasados en una sola variable?

\_A. $@

B. $inf

C. $$

D. $<>

La variable $@ recoge todos los argumentos que se han pasado al script en la ejecución desde la línea de comandos. Podemos iterar sobre ella igual que con otras estructuras de datos como los arrays.

**17.** ¿Cuál de las siguientes opciones se utiliza para cerrar un bucle while en Bash?

A. break

B. end

\_C. done

D. }

El bucle While se cierra con la palabra “Done”, es una de las excepciones a la norma.

**18.** ¿ Cuál de las siguientes sentencias se utiliza para cerrar un case en Bash?

\_A. esac

B. end

C. exit

D. done

Por norma general en bash scripting se utiliza la misma keyword de apertura (case) pero invertida para el caso del cierre, en este caso (esac). La excepción es el bucle while.

**19.** ¿Cómo se declara una función en Bash scripting?

A. function nombre\_funcion [ código ]

B. def nombre\_funcion { código }

\_C. function nombre\_funcion { código } o nombre\_funcion () { código }

D. f nombre\_funcion { código }

La sintaxis de la llamada a la función es simplemente tecleando el nombre de la misma.

**20.** ¿Cómo retardamos dos segundos la ejecución de la siguiente línea en un script?

A. after 2

B. sleep = “2”

C. $sleep 2

\_D. sleep 2

El comando sleep “duerme” la terminal por el número de segundos indicado a continuación del comando.