# Bash scripting cheatsheet: https://devhints.io/bash

# Ejercicio 1: ¿qué es shell scripting?

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

El **Shell Scripting** es una forma de programación que se realiza utilizando comandos de shell en sistemas operativos Unix-like (como Linux) o en el símbolo del sistema en Windows.

Un **script de shell** es un archivo de texto que contiene una serie de comandos que se ejecutan en secuencia de manera automática, lo que permite automatizar tareas y procesos en el sistema operativo.

Los scripts de shell están orientados a una amplia variedad de tareas, como:

1. **Automatización de tareas repetitivas:** los scripts pueden utilizarse para automatizar tareas que se realizan con frecuencia, como la copia de archivos, la generación de informes, la administración de usuarios, la descarga de archivos, etc.
2. **Administración del sistema:** los administradores de sistemas utilizan scripts para configurar, mantener y monitorear sistemas, redes y servidores. Por ejemplo, se pueden escribir scripts para realizar copias de seguridad, configurar cortafuegos, administrar registros de registro y realizar actualizaciones del sistema.
3. **Procesamiento de datos:** los scripts pueden utilizarse para procesar y manipular datos en archivos. Esto es útil en aplicaciones como el análisis de registros, la conversión de formatos de archivos y la extracción de información relevante de conjuntos de datos.
4. **Automatización de tareas de desarrollo:** los desarrolladores pueden utilizar scripts para automatizar tareas relacionadas con el desarrollo de software, como la compilación de código, la ejecución de pruebas, la implementación y la gestión de repositorios de código fuente.
5. **Interacción con aplicaciones y servicios:** los scripts pueden interactuar con aplicaciones y servicios a través de la línea de comandos, lo que permite automatizar flujos de trabajo que involucran múltiples programas o servicios.

En cuanto a la compilación de scripts, en el contexto del Shell Scripting, los scripts no necesitan ser compilados como lo haría un lenguaje de programación compilado, como C++ o Java. En cambio, los scripts de shell son interpretados por el intérprete del shell, lo que significa que el código se ejecuta directamente línea por línea sin necesidad de un proceso de compilación previo.

La razón principal por la que los scripts de shell no se compilan es la flexibilidad y la simplicidad que ofrecen. Esto permite a los usuarios y administradores de sistemas crear,

> modificar y ejecutar rápidamente scripts sin la necesidad de herramientas de desarrollo adicionales. Además, dado que los scripts suelen consistir en una secuencia de comandos que ya están disponibles en el sistema operativo, no es necesario un proceso de compilación para generar un ejecutable separado. Esto hace que los scripts de shell sean accesibles y fáciles de usar para una amplia gama de usuarios.
>
> ChatGPT

# Ejercicio 2: echo y read

**Investigar la funcionalidad de los comandos echo y read**

**echo** imprime cada *STRING* en la salida estándar, con un espacio entre cada una y una nueva línea después de la última.

**read** se utiliza para leer una línea desde la entrada estandar y almacenarla en una variable.

```
echo "Por favor, ingresa tu nombre:"
read nombre
echo "Hola, $nombre. ¡Bienvenido!"
```

## Ejercicio 2.a

**¿Cómo se indican los comentarios dentro de un script?**

Los comentarios comienzan con **#** y llegan hasta el final de la línea.

```
# Este es un comentario de una sola línea
echo "Hola, mundo"  # Esto también es un comentario
```

## Ejercicio 2.b

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

En Bash solo existen strings y arreglos, y los nombres son case-sensitive.

Para crear una variable: **nombre=”valor”**.

Para acceder a una variable: **echo $nombre**. Se pueden usar llaves para evitar ambigüedades: **echo ${nombre}esto_no_es_parte**

# Ejercicio 3: sustitución de comandos

**Sustitución de comandos**

Permite utilizar la salida de un comando como si fuese una cadena de texto normal, lo que permite guardarlo en variables o utilizarlo directamente dentro de otros comandos.

La sustitución de comandos permite obtener un comando para reemplazar el comando mismo. La sustitución de comandos se produce cuando un comando está encerrado entre un signo de dólar al principio y paréntesis, **$(command)**, o con acento grave, **`command`**.

La forma con acento grave es más antigua y tiene dos desventajas: 1) el acento grave puede confundirse fácilmente a la vista con las comillas simples, y 2) el acento grave no puede anidarse dentro de los acentos graves. La forma **$(command)** puede anidar varias expansiones de comando dentro de cada una.

```
#!/bin/bash

echo -n "Introduzca su nombre y apellido: "
read nombre apellido
echo "Fecha y hora actual:" $(date)
echo "Su apellido y nombre es: $apellido $nombre"
echo "Su usuario es: `whoami`"
echo "Su directorio actual es: $(pwd)"
```

# Ejercicio 4: parámetros

**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?**

| Variable | Descripción |
|---|---|
| **$0** | Contiene el nombre del script |
| **$1, $2, $3, ...** | Contienen a cada uno de los parámetros |
| **$#** | Contiene la cantidad de argumentos pasados al script |
| **$*** | Contiene todos los argumentos como un *string* de caracteres |
| **$@** | Contiene todos los argumentos como una lista |
| **$?** | Contiene el código de salida del último comando ejecutado |

```
#!/bin/bash

echo '$0' "contiene la invocación al script"
echo '$0:' $0
echo

echo '$#' "contiene el número de argumentos"
echo '$#:' $#
echo

echo '$*' "contiene todos los argumentos como una cadena"
echo '$*:' $*
echo

echo '$@' "contiene todos los argumentos como una lista"
echo '$@' $@
echo

echo '$1, $2, $3, ..., $9, ${10}, ${11}, $n' "contienen cada uno de los
parámetros individuales".
echo $1, $2, $3, ...
echo
```

*parametros.sh*

```
iso@debian12-vm:~$ parametros.sh primer segundo tercero cuarto
$0 contiene la invocación al script
$0: ./parametros.sh

$# contiene el número de argumentos
$#: 4

$* contiene todos los argumentos como una cadena
$*: primer segundo tercero cuarto

$@ contiene todos los argumentos como una lista
$@ primer segundo tercero cuarto

$1, $2, $3, ..., $n contienen cada uno de los parámetros individuales.
primer, segundo, tercero, …
```

```
iso@debian12-vm:~$ /home/iso/shell-script/parametros.sh
$0 contiene la invocación al script
$0: /home/iso/shell-script/parametros.sh
...

iso@debian12-vm:~$ ~/shell-script/parametros.sh
$0 contiene la invocación al script
$0: /home/iso/shell-script/parametros.sh
...
```

# Ejercicio 5: exit

**¿Cuál es la funcionalidad de comando exit? ¿Qué valores recibe como parámetro y cuál es su significado?**

> **exit** se utiliza para terminar un script. Al ejecutarse el script finaliza y devuelve un valor entre 0 y 255. El valor 0 indica que el script se ejecutó de forma exitosa, el resto de los valores indican un error.

# Ejercicio 6: expr (evaluación de expresiones)

**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.**

> **`expr EXPRESSION`** imprime el valor de ***EXPRESSION*** en la salida estándar. Una línea en blanco debajo separa grupos de precedencia creciente.
>
> Tenga en cuenta que **muchos operadores deben escaparse** o “encomillarse” para las shells. Las comparaciones son aritméticas si ambos ARGs son números, de lo contrario son lexicográficas. Las coincidencias de patrones devuelven la cadena coincidente entre \( y \) o null; si no se utilizan \( y \), devuelven el número de caracteres coincidentes o 0.
>
> El estado de salida es 0 si EXPRESIÓN no es nula ni 0, 1 si EXPRESIÓN es nula o 0, 2 si EXPRESIÓN es sintácticamente inválida y 3 si se ha producido un error.

| Expresión | Descripción |
|---|---|
| **ARG1 \| ARG2** | **ARG1** si no es nulo ni 0, de lo contrario **ARG2**. |
| **ARG1 & ARG2** | **ARG1** si ninguno de los argumentos es nulo o 0, de lo contrario, 0. |
| **ARG1 < ARG2** | **ARG1** es menor que **ARG2**. |
| **ARG1 <= ARG2** | **ARG1** es menor o igual a **ARG2**. |
| **ARG1 = ARG2** | **ARG1** es igual a **ARG2**. |
| **ARG1 != ARG2** | **ARG1** es diferente de **ARG2**. |
| **ARG1 >= ARG2** | **ARG1** es mayor o igual a **ARG2**. |
| **ARG1 > ARG2** | **ARG1** es mayor que **ARG2**. |
| **ARG1 + ARG2** | Suma aritmética de **ARG1** y **ARG2**. |
| **ARG1 - ARG2** | Resta aritmética de **ARG1** y **ARG2**. |
| **ARG1 * ARG2** | Producto aritmético de **ARG1** y **ARG2**. |
| **ARG1 / ARG2** | Cociente aritmético de **ARG1** dividido por **ARG2**. |
| **ARG1 % ARG2** | Resto aritmético de **ARG1** dividido por **ARG2**. |
| **STRING : REGEXP** | Coincidencia de patrón anclada de **REGEXP** en **STRING**. |
| **match STRING REGEXP** | Igual que **STRING : REGEXP**. |
| **substr STRING POS LENGTH** | Subcadena de **STRING**, comenzando en la posición **POS** contada desde 1, con longitud **LENGTH**. |
| **index STRING CHARS** | Índice en **STRING** donde se encuentra cualquier carácter de **CHARS**, o 0 si no se encuentra. |
| **length STRING** | Longitud de **STRING**. |
| **+ TOKEN** | Interpretar **TOKEN** como una cadena, incluso si es una palabra clave como 'match' o un operador como '/'. |
| **( EXPRESSION )** | Valor de la **EXPRESSION**. |

man page expr(1)

# Ejercicio 7: test (evaluar expresiones)

**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.**

Es mejor usar **`[[ expr ]]`** para realizar comprobaciones.

| Expresión | Descripción |
|---|---|
| **Expresiones** | |
| `( EXPR )` | **EXPR** es verdadera |
| `! EXPR` | **EXPR** es falsa |
| **Strings** | |
| `-z STR` | La longitud de **STR** es cero |
| `-n STR` **STR** | La longitud de **STR** es distinta de cero |
| `STR1 =, !=, <, > STR2` | Comparar strings |
| **Enteros** | |
| `INT1 -eq INT2` | **INT1** es igual a **INT2** |
| `INT1 -ne INT2` | **INT1** no es igual a **INT2** |
| `INT1 -lt INT2` | **INT1** es menor estricto que **INT2** |
| `INT1 -le INT2` | **INT1** es menor o igual que **INT2** |
| `INT1 -gt INT2` | **INT1** es mayor estricto que **INT2** |
| `INT1 -ge INT2` | **INT1** es mayor o igual que **INT2** |
| **Archivos** | |
| `-e FILE` | **FILE** existe |
| `-d FILE` | **FILE** existe y es un directorio |
| `-f FILE` | **FILE** existe y es un archivo regular |
| `-s FILE` | **FILE** existe y tiene un tamaño mayor que 0. |

man page test(1)

# Ejercicio 8: estructuras de control

**Estructuras de control. Investigue la sintaxis de las siguientes estructuras de control incluidas en shell scripting: if, case, while, for y select**

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

```
case $variable in
    valor1)
        # Código a ejecutar si variable coincide con valor1
        ;;
    valor2)
        # Código a ejecutar si variable coincide con valor2
        ;;
    *)
        # Código a ejecutar si no se cumple ninguna de las anteriores
        ;;
esac
```

```
select opcion in opcion1 opcion2; do
    case $opcion in
        opcion1)
            # Código a ejecutar para la opción 1
            ;;
        opcion2)
            # Código a ejecutar para la opción 2
            ;;
        *)
            # Código a ejecutar si se selecciona una opción no válida
            ;;
    esac
done
```

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

```
until [[ condición ]]; do
    # Código a ejecutar mientras la condición NO sea verdadera
done
```

```
# 1. for con una lista de elementos
for variable in ${arreglo[@]}; do
    # Código a ejecutar para cada elemento en la lista
done

# 2. For con una secuencia numérica
for ((i=1; i<=5; i++)); do
    # Código a ejecutar para cada valor de i en el rango 1-5
done

# 3. for con una expansión de llaves (brace expansion):
for letra in {a..z}; do
    # Código a ejecutar para cada letra en el rango de a a z
done

# 4. for con la expansión de archivos (globbing):
for archivo in *.txt; do
    # Código a ejecutar para cada archivo que coincida con *.txt
done

# 5. for con el resultado de un comando:
for elemento in $(comando); do
    # Código a ejecutar para cada elemento generado por el comando
done

# 6. for con un array
for elemento in "${mi_array[@]}"; do
    # Código a ejecutar para cada elemento en el array
done

# 7. For en un rango numérico usando seq
for i in $(seq 1 5); do
    # Código a ejecutar para cada valor en el rango 1-5
done
```

# Ejercicio 9: break y continue

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

**break** termina el bucle actual y pasa el control del programa al comando que sigue al bucle terminado. Se utiliza para salir de un bucle *for*, *while*, *until* o *select*.

```
break n;
```

**n** es un argumento opcional y debe ser mayor o igual a 1. **break 1** es equivalente a **break**.

**continue** omite los comandos restantes dentro del cuerpo del bucle que lo encierra para la iteración actual y pasa el control del programa a la siguiente iteración del bucle.

```
continue n;
```

**n** es opcional y puede ser mayor o igual a 1. Cuando se da **n**, se reanuda el bucle n-ésimo. **continue 1** es equivalente a **continue**.

Bash break and continue - Linuxize

# Ejercicio 10: variables

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

Bash soporta *strings* y *arrays*.

Bash es un lenguaje de **tipado débil y dinámico.** No es necesario declarar explícitamente el tipo de una variable y las variables pueden cambiar de tipo en tiempo de ejecución.

Los nombres de las variables son case-sensitive y pueden contener mayúsculas, minúsculas, numeros y el símbolo “_”, pero no pueden empezar con un número.

Crear una variable:

```
nombre="Pep"  # Sin espacios alrededor del =
```

Para acceder a una variable se usa **$nombre** o **${nombre}** (para evitar ambigüedades):

```
echo $nombre
echo ${nombre}esto_no_es_parte_de_la_variable
```

### Uso de comillas

No hacen falta, a menos que:

- El *string* tenga espacios.
- Sea una variable cuyo contenido puede tener espacios.
- Son importantes en las condiciones de las estructuras de control (if, while, etc.).

#### Comillas simples

```
unaVariable="un valor"
echo "Permiten usar $unaVariable"
echo "Y resultados de comandos: $(whoami)"
```

*Imprime:*

```
Permiten usar un valor
Y resultados de comandos: iso
```

#### Comillas simples

```
unaVariable="un valor"
echo 'No permiten usar $unaVariable'
echo 'Y resultados de comandos: $(whoami)'
```

*Imprime:*

```
Permiten usar $unaVariable
Y resultados de comandos: $(whoami)
```

### Alcance y visibilidad

Las variables no inicializadas son reemplazadas por un valor nulo o 0, según el contexto de evaluación.

Por defecto, las variables son **globales**:

```
#!/bin/bash

unaFuncion() {
    echo "2. antes: $unaVariable"     # 2.
    unaVariable=10
    echo "2. después: $unaVariable"   # 2.
}

echo "1. antes: $unaVariable"         # 1.
unaFuncion
echo "1. después: $unaVariable"       # 1.

unaVariable=100

echo "3. antes: $unaVariable"         # 3.
unaFuncion
echo "3. después: $unaVariable"       # 3.
```

*Imprime:*

```
1. antes:
2. antes:
2. después: 10
1. después: 10
3. antes: 100
2. antes: 100
2. después: 10
3. después: 10
```

Para definir una variable **local** a una función se utiliza ***local***:

```
#!/bin/bash

unaFuncion() {
    echo "2. antes: $unaVariable"     # 2.
    local unaVariable=10
    echo "2. después: $unaVariable"   # 2.
}

echo "1. antes: $unaVariable"         # 1.
unaFuncion
echo "1. después: $unaVariable"       # 1.

unaVariable=100

echo "3. antes: $unaVariable"         # 3.
unaFuncion
echo "3. después: $unaVariable"       # 3.
```

*Imprime:*

```
1. antes:
2. antes:
2. después: 10
1. después:
3. antes: 100
2. antes: 100
2. después: 10
3. después: 100
```

Las variables de entorno son heredades por los procesos hijos. Para **exponer** una variable global a los procesos hijos se usa el comando ***export:***

```
export variableGlobal="Mi variable global"
comando # comando verá entre sus variables de entorno a variableGlobal
```

## Arreglos

### Definición

```
arregloVacio=()
arregloInicializado=(1 2 3 4 5)
```

```
fruits[0]="Apple"
fruits[1]="Banana"
fruits[2]="Orange"
```

### Trabajando con arreglos

```
echo "${Fruits[0]}"           # Element #0
echo "${Fruits[-1]}"          # Last element
echo "${Fruits[@]}"           # All elements, space-separated
echo "${#Fruits[@]}"          # Number of elements
echo "${#Fruits}"             # String length of the 1st element
echo "${#Fruits[3]}"          # String length of the Nth element
echo "${Fruits[@]:3:2}"       # Range (from position 3, length 2)
echo "${!Fruits[@]}"          # Keys of all elements, space-separated
```

### Operaciones

```
Fruits=("${Fruits[@]}" "Watermelon")    # Push
Fruits+=('Watermelon')                  # Also Push
Fruits=( "${Fruits[@]/Ap*/}" )          # Remove by regex match
unset Fruits[2]                         # Remove one item
Fruits=("${Fruits[@]}")                 # Duplicate
Fruits=("${Fruits[@]}" "${Veggies[@]}") # Concatenate
lines=(`cat "logfile"`)                 # Read from file
```

# Ejercicio 11: funciones

### Definiendo funciones

```
unaFuncion() {
    echo "Hola, $1!"
}

# Sintaxis alternativa 1
function alternativa1() {
    echo "Hola, $1!"
}

#  Sintaxis alternativa 2
function alternativa2 {
    echo "Hola, $1!"
}
```

```
unaFuncion "Mate"
alternativa1 "Mati"
alternativa2 "Chino"
```

### return - levantando errores

**return** finaliza la ejecución de una función y puede retornar un valor entre 0 y 225 (por defecto devuelve 0). El valor 0 se considera como un valor de retorno exitoso.

```
myfunc() {
  return 1
}
```

```
if myfunc; then
  echo "success"
else
  echo "failure"
fi
```

### Devolviendo valores

```
myfunc() {
    local myresult='some value'
    echo "$myresult"
}
```

```
result=$(myfunc)
```