Scripts

En informática, un **script**, archivo de órdenes, archivo de procesamiento por lotes o, cada vez más aceptado en círculos profesionales y académicos, **guion**, es un programa usualmente simple, que por lo regular se almacena en un archivo de texto plano.

Los guiones son casi siempre interpretados, pero no todo programa interpretado es considerado un guion. El uso habitual de los guiones es realizar diversas tareas como combinar componentes, interactuar con el sistema operativo o con el usuario. Por este uso es frecuente que los intérpretes de órdenes sean a la vez intérpretes de este tipo de programas.

En el sistema operativo

- En UNIX

Los archivos guion suelen ser identificados por el sistema a través de uno de los siguientes encabezamientos en el contenido de la música, utilizado por hackers o programadores: **shebang**[1]

```
#!/bin/bash ; #!/bin/ksh ; #!/bin/csh
```

Aunque en entornos UNIX la mayoría de los guiones son identificados por dicho encabezamiento, también pueden ser identificados a través de la extensión "**.sh**", siendo esta quizá menos importante que el encabezamiento, ya que casi todos los sistemas no necesitan dicha extensión para ejecutar el guion, por lo tanto, esta suele ser añadida por tradición, o más bien, es útil para que el usuario pueda identificar estos archivos a través de una interfaz de línea de comandos sin necesidad de abrirlo.

Difieren de los programas de aplicación, debido a que los últimos son más complejos; además, los guiones son más bien un programa que le da instrucciones a otros más avanzados.

- En Windows y DOS

En el sistema operativo DOS, a los guiones creados para ser interpretados por **cmd.exe** o el obsoleto COMMAND.COM se les conoce como archivos «**batch**» (procesamiento por lotes) y acaban en **.bat**. En el sistema operativo Windows, existen varios lenguajes interpretados como Visual Basic Script, JavaScript, WScript, Batch.

- En diseño web

Guiones del lado del cliente

Los guiones del lado del cliente se deben incluir con la etiqueta **<script>**, incluyendo el atributo **type** con el tipo **MIME**.

Generalmente se usa JavaScript, pero se puede usar VBScript (solo Internet Explorer o Google Chrome). Tiene como objetivo, por lo general, AJAX o manipulación del DOM[2].

Guiones del lado del servidor

No tienen los problemas de accesibilidad que pueden presentar los guiones del lado del cliente. También permiten modificar las cabeceras HTTP, u obtenerlas. Además, permiten acceso a bases de datos y otros archivos internos.

El **script** es un documento que contiene instrucciones, escritas en códigos de programación. El **script** es un lenguaje de programación que ejecuta diversas funciones en el interior de un programa de computador.

---

1 **Shebang** es, en la jerga de Unix, el nombre que recibe el par de caracteres **#!** que se encuentran al inicio de los programas ejecutables interpretados

2 Document Object Model o DOM ('Modelo de Objetos del Documento' o 'Modelo en Objetos para la Representación de Documentos') es esencialmente una interfaz de plataforma que proporciona un conjunto estándar de objetos para representar documentos HTML, XHTML y XML, un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos. A través del DOM, los programas pueden acceder y modificar el contenido, estructura y estilo de los documentos HTML y XML, que es para lo que se diseñó principalmente.

Los **scripts** se encargan de cumplir las siguientes funciones:

- Combinar componentes.
- Interaccionar con el sistema operativo o con el usuario.
- Controlar un determinado programa o aplicación.
- Configurar o instalar sistemas operacionales, especialmente en los juegos, se usa para controlar las acciones de los personajes.

Algunos lenguajes de programación, usada como script son: ActionScript, JavaScript, Lua, PHP, Python, ShellScript, Ruby, VBScript.

Scripts de bash

Un **script** no es más que un archivo que contiene un conjunto de órdenes para realizar una acción.

Vamos a crear nuestro primer **script**. Para ello en un editor de texto escribiremos lo siguiente y lo guardaremos con el nombre **hola.sh**

```
#!/bin/bash
# Este es nuestro primer programa
echo Hola Mundo
```

A continuación iremos a la terminal y lo ejecutaremos:

```
~$ ./hola.sh
```

La primera línea de nuestro **script** le indica al sistema que tiene que usar la **shell BASH**. La segunda línea es un comentario para consumo humano, todas las líneas que comiencen por **#** son ignoradas por la computadora y nos sirven para incluir comentarios destinados a programadores o usuarios. En la tercera línea tenemos el comando **echo** que sirve para imprimir texto en la pantalla.

Variables

Como cualquier otro lenguaje de programación, necesitamos variables que nos servirán para guardar datos en la memoria del computador hasta el momento que los necesitemos. Podemos pensar en una variable como una caja en la que podemos guardar un elemento (e.g, un número, una cadena de texto, la dirección de un archivo…) y, siguiendo con el símil, la memoria del ordenador no sería más que el conjunto de esas cajas.

Para asignar el valor a una variable simplemente simplemente debemos usar el signo igual (=). Las variables no tienen tipo de dato. Una variables puede contener un valor entero y a continuación un valor cadena:

```
nombre_variable=valor_variable
```

Es importante no dejar espacios ni antes ni después del signo igual.

Para recuperar el valor de dicha variable sólo hay que anteponer el símbolo de dolar **$** antes del nombre de la variable:

```
$nombre_variable
```

A lo largo de un **script** podemos asignarle diferentes valores a una misma variable:

```
#!/bin/bash
to_print='Hola mundo'
echo $to_print
to_print=5.5
echo $to_print
```

Nombre de las variables

Las variables pueden tomar prácticamente cualquier nombre, sin embargo, existen algunas restricciones:

- Sólo puede contener caracteres alfanuméricos y guiones bajos

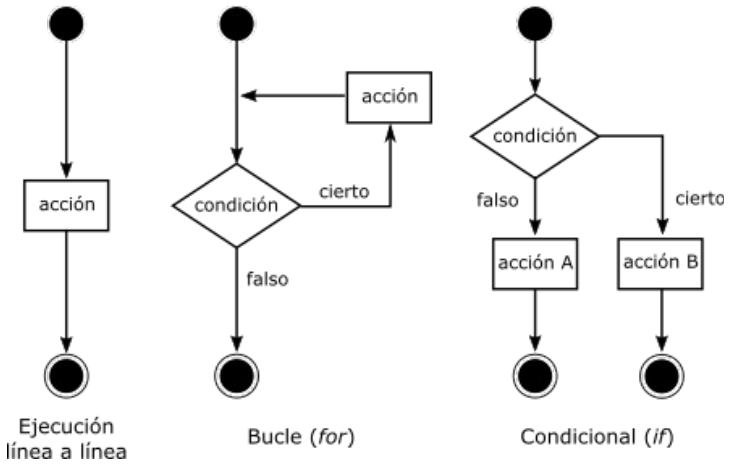
- El primer carácter debe ser una letra del alfabeto o guión bajo "_" (este último caso se suele reservar para casos especiales).
- No pueden contener espacios.
- Las mayúsculas y las minúsculas importan, "a" es distinto de "A".
- Algunos nombres son usado como variables de entorno y no los debemos utilizar para evitar sobrescribirlas (e.g. PATH).

De manera general, y para evitar problemas con las variables de entorno que siempre están escritas en mayúscula, deberemos escribir el nombre de las variables en minúscula.

Además, aunque esto no es una regla que deba obedecerse obligatoriamente, es conveniente que demos a las variables nombres que más tarde podamos recordar. Si abrimos un **`script`** tres meses después de haberlo escrito y nos encontramos con la expresión "**`m=3.5`**" nos será difícil entender que hace el programa. Habría sido mucho más claro nombrar la variable como "**`media=3.5`**".

Control de flujo

Como hemos visto los **`scripts`** se ejecutan línea a línea hasta llegar al final, sin embargo, muchas veces nos interesará modificar ese comportamiento de manera que el programa pueda responder de un modo u otro dependiendo de las circunstancias o pueda repetir trozos de código.



En este curso nos vamos a centrar en los controles de flujo más importantes:

- bucles (**`for loops`**)
- condicionales (**`if`**)

Bucles (`for`)

La sintaxis general de los bucles es la siguiente:

```
for VARIABLE in LISTA_VALORES;
do
      COMANDO 1
      COMANDO 2
      ...
      COMANDO N
done
```

Donde la lista de valores puede ser un rango numérico:

```
for VARIABLE in 1 2 3 4 5 6 7 8 9 10;
```

```
for VARIABLE in {1..10};
```

o una serie de valores:

```
for VARIABLE in file1 file2 file3;
```

o el resultado de la ejecución de un comando:

```
for VARIABLE in $(ls /bin | grep -E 'c.[aeiou]');
```

Hay que tener en cuenta que si pasamos un listado de valores pero lo ponemos entrecomillado, el ordenador lo enterá como un única línea:

```
for VARIABLE in "file1 file2 file3";
```

Un ejemplo simple de `for` sería:

```
#!/bin/bash
for numero in {1..20..2};
do
      echo Este es el número: $numero
done
```

Condicionales (`if`)

La sintaxis básica de un condicional es la siguiente

```
if [[ CONDICIÓN ]];
then
      COMANDO 1 si se cumple la condición
fi
```

También se puede especificar qué hacer si la condición no se cumple:

```
if [[ CONDICIÓN ]];
then
      COMANDO 1 si se cumple la condición
else
      COMANDO 2 si no se cumple la condición
fi
```

Incluso se pueden añadir más condiciones concatenando más `if`:

```
if [[ CONDICIÓN 1 ]];
then
      COMANDO 1 si se cumple la condición 1
elif [[ CONDICIÓN 2 ]];
then
      COMANDO 2 si se cumple la condición 2
else
      COMANDO 3 si no se cumple la condición 2
fi
```

Condicionales con números

Al comparar números podemos realizar las siguientes operaciones:

operador	significado
`-lt`	menor que (<)
`-gt`	mayor que (>)
`-le`	menor o igual que (<=)
`-ge`	mayor o igual que (>=)
`-eq`	igual (==)
`-ne`	no igual (!=)

```
#!/bin/bash
num1=$1  # la variable toma el primer valor que le pasamos al script
num2=$2  # la variable toma el segundo valor que le pasamos al script
if [[ $num1 -gt $num2 ]];
then
```

```
        echo $num1 es mayor que $num2
else
        echo $num2 es mayor que $num1
fi
```

Condicionales con cadenas de texto

A la hora de comparar cadenas de texto:

operador	significado
=	igual, las dos cadenas de texto son exactamente idénticas
!=	no igual, las cadenas de texto no son exactamente idénticas
<	es menor que (en orden alfabético ASCII)
>	es mayor que (en orden alfabético ASCII)
-n	la cadena no está vacía
-z	la cadena está vacía

```
#!/bin/bash
string1='reo'
string2='teo'
if [[ $string1 > $string2 ]];
then
        echo Eso es verdad
else
        echo Eso es mentira
fi
```

También podemos hacer comparaciones haciendo uso de **`wildcards`**:

```
#!/bin/bash
string1='reo'
if [[ $string1 = *e* ]];
then
        echo Eso es verdad
else
        echo Eso es mentira
fi
```

Condicionales con archivos

operador	Devuelve `true` si
-e name	**`name`** existe
-f name	**`name`** es un archivo normal (no es un directorio)
-s name	**`name`** NO tiene tamaño cero
-d name	**`name`** es un directorio
-r name	**`name`** tiene permiso de lectura para el **`user`** que corre el **`script`**
-w name	**`name`** tiene permiso de escritura para el **`user`** que corre el **`script`**
-x name	**`name`** tiene permiso de ejecución para el **`user`** que corre el **`script`**

Por ejemplo, podemos hacer un **`script`** que nos informe sobre el contenido de un directorio:

```
#!/bin/bash
for file in $(ls);
do
        if [[ -d $file ]];
        then
```

```
            echo directorio: $file
      else
            if [[ -x $file ]];
            then
                  echo archivo ejecutable: $file
            else
                  echo archivo no ejecutable: $file
            fi
      fi
done
```

Manipulación de cadenas de texto

Extraer subcadena

Mediante **`${cadena:posicion:longitud}`** podemos extraer una subcadena de otra cadena. Si omitimos **`:longitud`**, entonces extraerá todos los caracteres hasta el final de cadena.

Por ejemplo en la cadena **`string=abcABC123ABCabc`**

```
echo ${string:0}          :     abcABC123ABCabc
echo ${string:0:1}        :     a (primer carácter)
echo ${string:7}          :     23ABCabc
echo ${string:7:3}        :     23A (3 caracteres desde posición 7)
echo ${string:7:-3}       :     23ABCabc (desde posición 7 hasta el final)
echo ${string: -4}        :     Cabc (atención al espacio antes del menos)
echo ${string: -4:2}      :     Ca (atención al espacio antes del menos)
```

Borrar subcadena

Hay diferentes formas de borrar subcadenas de una cadena:

- **`${cadena#subcadena}`** : borra la coincidencia más corta de subcadena desde el principio de cadena
- **`${cadena##subcadena}`** : borra la coincidencia más larga de subcadena desde el principio de cadena

Por ejemplo, en la cadena **`string=abcABC123ABCabc`**

```
echo ${string#a*C}        :     123ABCabc
echo ${string##a*C}       :     abc
```

Reemplazar subcadena

También existen diferentes formas de reemplazar subcadenas de una cadena:

- **`${cadena/buscar/reemplazar}`** : Sustituye la primera coincidencia de buscar con reemplazar
- **`${cadena//buscar/reemplazar}`** : Sustituye todas las coincidencias de buscar con reemplazar

Por ejemplo, en la cadena **`string=abcABC123ABCabc`**

```
echo ${string/abc/xyz}    :     xyzABC123ABCabc.
echo ${string//abc/xyz}   :     xyzABC123ABCxyz.
```

Operaciones aritméticas

Por último, **`Bash`** también permite la operaciones aritméticas con número enteros:

- **`+ -`** : suma, resta

```
~$ $num=10
~$ echo $((num + 2))
```

- `**` : potencia

```
~$ echo $((num ** 2))
```

- `* / %` : multiplicación, división, resto (módulo)

```
~$ echo $((num * 2))
~$ echo $((num / 2))
~$ echo $((num % 2))
```

- `VAR++ VAR–` : post-incrementa, post-decrementa

```
~$ echo $((num++))
~$ echo $num
```

- `++VAR –VAR` : pre-incrementa, pre-decrementa

```
~$ echo $((++num))
~$ echo $num
```

Parallel

parallel es un programa que permite la ejecución en paralelo de diferentes trabajos siempre que dispongamos de un ordenador o ordenadores con más de un procesador.

Aunque este programa no tiene nada que ver con los **scripts** de **Bash**, sí que hacer uso de éstos no resultará muy útil para preparar el archivo de entrada para **parallel**. Este programa requiere pasar un archivo donde cada línea es trabajo a realizar y puede consistir tanto en un comando como en un pequeño **script** a ejecutar. Y es ahí donde saber algo de **scripting** en **Bash** nos puede facilitar la tarea. Veamos un ejemplo de **input** para **parallel**:

```
to_run.txt:
zcat file1.fasta.gz | grep '>' | sed 's/>//' | bgzip > sample_names1.txt.gz
zcat file2.fasta.gz | grep '>' | sed 's/>//' | bgzip > sample_names2.txt.gz
zcat file3.fasta.gz | grep '>' | sed 's/>//' | bgzip > sample_names3.txt.gz
zcat file4.fasta.gz | grep '>' | sed 's/>//' | bgzip > sample_names4.txt.gz
zcat file5.fasta.gz | grep '>' | sed 's/>//' | bgzip > sample_names5.txt.gz
zcat file6.fasta.gz | grep '>' | sed 's/>//' | bgzip > sample_names6.txt.gz
```

parallel no viene instalado por defecto en algunas distribuciones Linux, por tanto, lo primero que debemos hacer es instalarlo:

```
~$ sudo apt-get install parallel
```

Con la opción **-j** podemos especificarle el número de trabajos simultáneos. Así pues, si tenemos 4 procesadores disponibles y queremos correr un trabajo por procesador correríamos:

```
~$ parallel -j 4 to_run.txt
```

Ejercicios

1. Haz un **script** que cree 10 archivos **.txt** en un directorio (use **touch** para crearlos)
2. Haz un **script** que comprima con **gzip** sólo los archivos 5 y 9.
3. Escribe un **script** que cambie la extensión del archivo que contenga un 3 en su nombre de **.txt** a **.md**
4. Crea un **script** que copie todos los archivos (no directorios) de **/etc** a un directorio en su "**home directory**".
5. Prepara un **script** que cuenta el número de directorios y archivos que hay en **/etc**
6. Haz un **script** que devuelva el número de archivos que ha guardado