Shell Scripting - Introducción

En esta lección aprenderemos exactamente qué son los scripts, los componentes que forman un script, cómo utilizar variables en nuestros scripts, cómo realizar pruebas y tomar decisiones, cómo aceptar argumentos de la línea de comandos y cómo aceptar inputs de un usuario.

Un script es un programa de línea de comandos que contiene una serie de comandos. Los comandos contenidos en el script son ejecutados por un intérprete. En el caso de los scripts Shell, el Shell actúa como intérprete y ejecuta los comandos enumerados en el script uno tras otro.

Cualquier cosa que podamos ejecutar en la línea de comandos podemos ponerlo en un script Shell. Los scripts Shell son excelentes para automatizar tareas. Si nos encontramos ejecutando una serie de comandos para llevar a cabo una tarea determinada y necesitamos realizar esa tarea en el futuro, podemos, y probablemente deberíamos, crear un script shell para esa tarea.

Veamos un sencillo script Shell:

script.sh

```
#!/bin/bash
echo "Scripting is fun!"
```

```
$ chmod 755 script.sh
$ ./script.sh
Scripting is fun!
$
```

Si el script existe en la ruta actual, podemos simplemente escribir el nombre del script y este será ejecutado. Si no está en la ruta, entonces necesitamos especificar una ruta relativa o completa. En este caso, vamos a pretender que estamos posicionados en el directorio home y este es un script que acabamos de crear y queremos ejecutarlo desde nuestro directorio home.

Notemos que la primera línea del script comienza con un signo numeral (#), luego un signo de exclamación (!), seguido por la ruta al programa bash Shell. Algunas personas se refieren al signo de número como “sharp” y al signo de exclamación como “bang”. Así que el signo “#!” se puede pronunciar como “sharp bang”. El término “shebang” es una contracción inexacta de “sharp bang”.

Cuando la primera línea de un script comienza con un shebang, lo que sigue se utiliza como intérprete para los comandos listados en el script. Veamos tres ejemplos, cada uno utilizando un programa Shell diferente como intérprete:

Shebang

```
#!/bin/csh
echo "This script uses csh as the interpreter."
```

```
#!/bin/ksh
echo "This script uses ksh as the interpreter."
```

```
#!/bin/zsh
echo "This script uses zsh as the interpreter."
```

Cuando ejecutamos un script que contiene un shebang, lo que ocurre en realidad es que se ejecuta el intérprete y la ruta especificada se pasa como argumento al intérprete. Podemos comprobarlo examinando la tabla de procesos:

Iniciemos este script llamado “sleepy.sh” en segundo plano y miremos la tabla de procesos:

sleepy.sh

```
#!/bin/bash
sleep 90
```

```
$ ./sleepy.sh &
[1] 16796
$ ps -fp 16796
UID      PID  PPID  C STIME TTY          TIME CMD
jason    16796 16725  0 22:50 pts/0  00:00:00
/bin/bash ./sleepy.sh
$
```

Podemos ver que lo que realmente se está ejecutando es “/bin/bash ./sleepy.sh”. Ahora usemos una ruta completa al script:

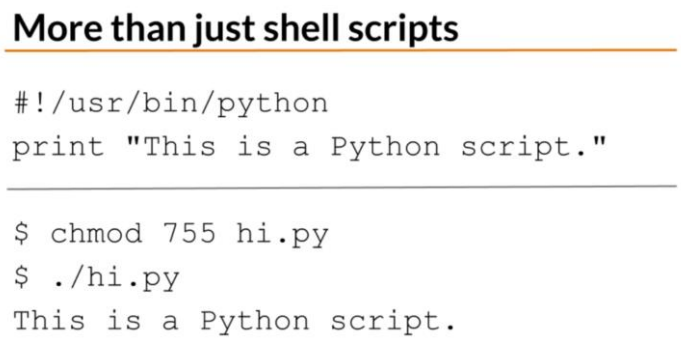
The interpreter executes the script

```
$ /tmp/sleepy.sh &
[1] 16804
$ ps -fp 16804
UID      PID  PPID  C STIME TTY      TIME CMD
jason    16804 16725  0 22:51 pts/0  00:00:00
/bin/bash /tmp/sleepy.sh
$
```

En este caso, “/bin/bash” se ha ejecutado con la ruta a “/tmp/sleepy.sh”.

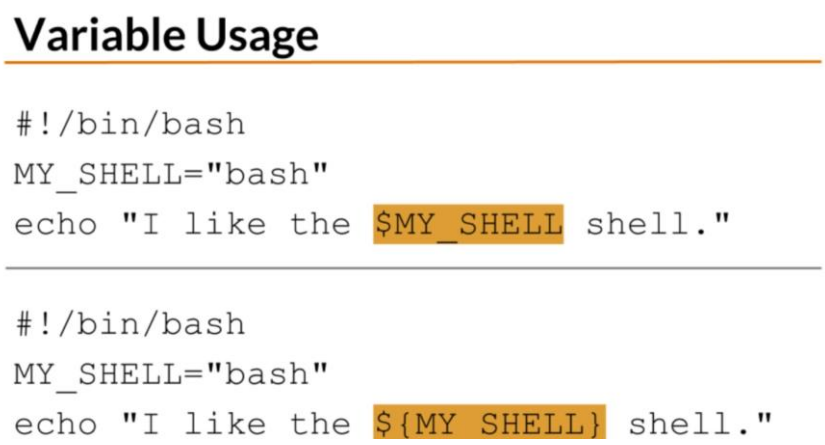
Si no proporcionamos un shebang y un intérprete en la primera línea del script, los comandos del script se ejecutarán utilizando nuestro shell actual. Aunque esto puede funcionar bien en muchas circunstancias, es mejor ser explícito y especificar el intérprete exacto que se utilizará con el script. Por ejemplo, hay características y sintaxis que funcionan bien con el Shell bash pero que no funcionan con el Shell csh.

Además, no tenemos que usar Shell como intérprete para todos nuestros scripts. Aquí hay un ejemplo de un script Python llamado “hi.py”:



Podemos utilizar variables en nuestro script Shell:

- Las variables son simplemente ubicaciones de almacenamiento que tienen nombre. Podemos pensar en las variables como pares nombre-valor.
- Para asignar un valor a una variable, utilice la sintaxis VARIABLE_NAME=”valor”.
- Debemos asegurarnos de no utilizar espacios antes o después del signo igual (=).
- Además, las variables distinguen entre mayúsculas y minúsculas y, por convención, los nombres de las variables se escriben en mayúsculas.
- Para utilizar una variable, preceda l nombre de la variable con un signo de pesos ($):



- También podemos encerrar el nombre de la variable entre corchetes y preceder la llave de apertura con un signo de pesos.
- La sintaxis de corchetes es opcional, a menos que necesite preceder o seguir inmediatamente a la variable con datos adicionales.

Veamos un ejemplo. Supongamos que queremos añadir las letras “ing” al valor almacenado en la variable “MY_SHELL”. Para ello, utilizamos “${MY_SHELL}” y a continuación “ing”:

```
#!/bin/bash
MY_SHELL="bash"
echo "I am ${MY_SHELL}ing on my keyboard."
```

Output:
I am bashing on my keyboard.

Si no encerramos el nombre de la variable entre corchetes, el Shell tratará el texto adicional como parte del nombre de la variable. Como no existe una variable con ese nombre, en este ejemplo no se pone nada en su lugar:

```
#!/bin/bash
MY_SHELL="bash"
echo "I am $MY_SHELLing on my keyboard."
```

Output:
I am on my keyboard.

También podemos asignar la salida de un comando a una variable. Para ello, encerramos el comando entre paréntesis y lo precedemos del signo de pesos. En este ejemplo, la salida del comando "hostname" se almacena en la variable "SERVER_NAME":

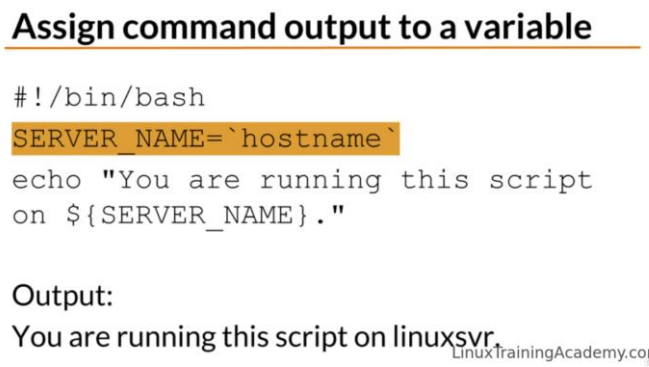
Assign command output to a variable

```
#!/bin/bash
SERVER_NAME=$(hostname)
echo "You are running this script
on ${SERVER_NAME}."
```

Output:
You are running this script on linuxsvr.

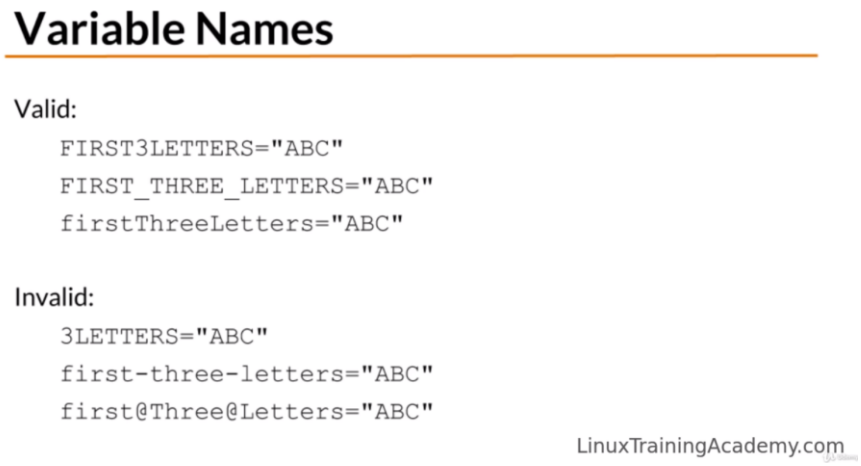
En este ejemplo, la salida del comando "hostname" es "linuxsvr".

También podemos encerrar el comando en comillas invertidas. Esto se trata de una sintaxix antigua que está siendo sustituida por la sintaxis "#()", sin embargo, podemos verlo en scripts antiguos:



Los nombres de las variables pueden contener letras, números y guion bajo. Pueden comenzar con letras o guion bajo, pero no pueden empezar con un dígito.

Lo siguientes son ejemplos de variables con nombres válidos y otros con nombres inválidos:



- Los nombres de variables válidos tienen una mezcla de mayúsculas, minúsculas, guion bajo y números.
- En los casos inválidos, vemos una variable que comienza con un número. Los otros dos ejemplos tienen guion y el signo @, que también son caracteres inválidos para los nombres de las variables.

Los scripts están diseñados para sustituir la necesidad de que una persona se siente físicamente ante un teclado a escribir muchos comandos. ¿Qué pasa si tenemos una tarea que queremos automatizar, pero que requiere diferentes acciones basadas en diferentes circunstancias? Si estamos tecleando los comandos, podemos tomarnos el tiempo de ver un estado y luego tomar una decisión basada en ese estado. En los scripts Shell, podemos hacer lo mismo con los test.

Para crear un test, coloque una expresión condicional entre paréntesis. Podemos probar varios tipos de situaciones. Por ejemplo, podemos comparar si dos string son iguales, si un número es mayor que otro o si un archivo existe:

Tests

Syntax:

```
[ condition-to-test-for ]
```

Example:

```
[ -e /etc/passwd ]
```

Este test comprueba si la ruta "/etc/password" existe. Si existe, devuelve true. Dicho de otro modo, el comando retorna con el estatus 0. Si el fichero no existe, devuelve false, es decir, el comando retorna el estatus 1.

Si estamos utilizando el intérprete de comandos bash, podemos ejecutar el comando "help test" para ver los distintos tipos de test que podemos realizar. A continuación, algunos ejemplos de los test disponibles:

File operators (tests)

-d FILE	True if file is a directory.
-e FILE	True if file exists.
-f FILE	True if file exists and is a regular file.
-r FILE	True if file is readable by you.
-s FILE	True if file exists and is not empty.
-w FILE	True if the file is writable by you.
-x FILE	True if the file is executable by you.

Arithmetic operators (tests)

arg1 –eq arg2 True if arg1 is equal to arg2.
arg1 –ne arg2 True if arg1 is not equal to arg2.

arg1 –lt arg2 True if arg1 is less than arg2.
arg1 –le arg2 True if arg1 is less than or equal to arg2.

arg1 –gt arg2 True if arg1 is greater than arg2.
arg1 –ge arg2 True if arg1 is greater than or equal to arg2.

Arithmetic operators (tests)

arg1 –eq arg2 True if arg1 is equal to arg2.
arg1 –ne arg2 True if arg1 is not equal to arg2.

arg1 –lt arg2 True if arg1 is less than arg2.
arg1 –le arg2 True if arg1 is less than or equal to arg2.

arg1 –gt arg2 True if arg1 is greater than arg2.
arg1 –ge arg2 True if arg1 is greater than or equal to arg2.