BASH Unidad 5

Scripts, variables, funciones y control de flujo

Tabla de contenidos

Configuración del entorno

Durante una sesión con la shell se preserva cierta información en memoria. Esta información es lo que se denomina el *entorno* y contiene:

- **Variables** que indican el nombre de usuario, las rutas de los directorios donde se buscan los ejecutables, el nombre del fichero donde se entregan los correos del usuario en el sistema y muchos más. Se puede ver la lista completa de variables usando el comando printenv.
- **Alias**, que no son sino nombres cortos que sirven como abreviaciones de comandos más largos —por ejemplo porque incluyen múltiples opciones—. Para comprobar si un comando es un alias se puede usar el comando type —p. ej. `type ls`—
- **Funciones** de la shell, que se pueden entender como scripts dentro de scripts:

```
yo@mihost:~$ whoami () {
> echo "$USER"
> }
```

Se puede ver la lista completa de elementos del entorno usando el comando **set**.

¿Cómo se configura el entorno?

Cuando el usuario se autentica en el sistema, el programa BASH se inicia y lee una serie de archivos de configuración llamados *archivos de inicio*. La secuencia exacta depende del tipo de sesión de la shell que se haya iniciado:

- **Una shell de login** es aquella que es iniciada después de que el usuario introduce su nombre de usuario y su contraseña en el programa login; por ejemplo, cuando se inicia sesión en la consola.
- **Una shell de no-login** es aquella que típicamente se inicia cuando lanzamos una sesión de terminal desde el escritorio de la interfaz gráfica.

Las shell de login leen uno o más de los siguientes archivos de inicio::

Archivo	Contenido
/etc/profile	Archivo de configuración global que se aplica a todos los usuarios
~/.bash_profile	Archivo de inicio personal del usuario. Puede ser usado para ampliar o sobrescribir la configuración del archivo global.
~/.bash_login	Si no se encuentra ~/.bash_profile, BASH intenta leer este script.
~/.profile	Si no se encuentra ni ~/.bash_profile ni ~/.bash_login, BASH intenta leer este archivo. Esto es así por defecto en distribuciones basadas en Debian, como es el caso de Ubuntu.

Las shell de no-login leen los siguientes archivos de inicio:

Archivo	Contenido
/etc/bash.bashrc	Archivo de configuración global que se aplica a todos los usuarios.
~/.bashrc	Archivo de inicio personal del usuario. Puede ser usado para ampliar o sobrescribir la configuración del archivo global.

Además de leer los archivos de configuración mencionados anteriormente, **las shells de no-login también heredan el entorno de su proceso padre, que normalmente es una shell de login.**

Ejemplos

Echa un vistazo en tu sistema y mira qué tipo de archivos de configuración tienes. Recuerda que el nombre comienza por "." por lo que debes usar **ls -a** para mostrar los archivos ocultos.

Si, por ejemplo, echamos un vistazo a un .bash_profile típico, este podría contener algo así:

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin
export PATH
```

Las líneas que comienzan con “#” son comentarios. Así que las primeras cosas interesantes comienzan en la línea 4 con:

```
if [ -f ~/.bashrc ]; then
      . ~/.bashrc
fi
```

Que traducido viene a decir que **"si el archivo "~/.bashrc" existe, entonces incluir el archivo "~/.bashrc"** —igual que como el #include de C++—. Así que aquí es donde nos aseguramos que BASH lee los contenidos de .bashrc incluso cuando la shell no es de login.

Lo siguiente que hace nuestro archivo de configuración es **actualizar la variable PATH para que incluya el directorio ~/bin**.

```
PATH=$PATH:$HOME/bin
```

La variable PATH contiene la lista de directorios en los que la shell debe ir a buscar los ejecutables. Cuando se ejecuta un comando sin indicar la ruta —p. ej. `ls` en lugar de `/bin/ls`— BASH busca el ejecutable correspondiente para ejecutarlo solamente en los directorios indicados en PATH

Por último, se ejecuta **export PATH**. El comando **export le dice a la shell que haga visibles los cambios de la variable PATH a todos los procesos hijos de la shell**.

```
export PATH
```

Puesto que **de lo contrario los cambios en las variables son privados a la shell donde se hacen**. Como norma general, **las variables de entorno exportadas son copiadas de padres a hijos por el sistema operativo**, cada vez que se crea un nuevo proceso.

Alias

Un alias es una forma rápida de crear un nuevo comando que sea una abreviatura de otro con un nombre más largo. Tiene la siguiente sintaxis:

```
alias nombre=valor
```

Donde *nombre* es el nombre del nuevo comando y valor es el nombre del comando que va a ser ejecutado cuando se escriba *nombre* en la línea de comandos.

Ejemplos

Podemos editar nuestro .bashrc e incluir lo siguiente:

```
alias l='ls -l --color=auto'
```

Con esta línea hemos creado un nuevo comando llamado “l” que ejecutará un **ls -l** cada vez que sea invocado. **Para probar el nuevo comando, es necesario cerrar la consola y abrir una nueva para que se refresquen los cambios en el archivo ".bashrc"**.

Otro ejemplo que se puede probar es:

```
alias today='date +"%A, %B %-d, %Y"'
```

Este comando muestra la fecha de hoy con un formato adecuado.

Los alias se pueden crear también desde la línea de comandos, sin embargo, solo durarán mientras permanezca abierta la sesión actual de la shell.

Funciones de la Shell

Los alias están bien para comandos sencillos. Pero si se quiere crear algo más complejo, se debe trabajar con *funciones de la shell.* Este tipo de funciones se pueden pensar como scripts dentro de scripts —o pequeños sub-scripts—.

Ejemplos

Veamos uno. Abre otra vez .bashrc y reemplaza el alias "today" que escribimos antes con:

```
today() {
    echo -n "Today's date is: "
    date +"%A, %B %-d, %Y"
}
```

Las funciones de shell también se pueden definir directamente en la línea de comandos, sin embargo, como pasa con los alias, solo tendrán vigencia mientras dure la sesión actual.

Mi primer script

Para crear un script solo tenemos que abrir un editor y escribir algo así:

```
#!/bin/bash

# Mi primer script

tar cvzf $HOME/backup.tar.gz $HOME/Desktop $HOME/Documentos
```

Luego hay que guardarlo, con un nombre como `mi_script`, y darle permisos de ejecución para poder ejecutarlo directamente así:

```
yo@mihost:~$ ./mi_script
```

La primera línea del script es la línea *shebang*. Es opcional, pero si queremos utilizarla, siempre debemos ponerla en la línea 1 del script. En Linux y otros sistemas POSIX, si un script es lanzado como un ejecutable y la contiene, el sistema operativo interpreta lo que hay detrás de "#!", como la ruta al ejecutable que debe interpretar el script. Así que para ejecutar el script, el sistema ejecuta el intérprete indicado, pasándole como primer parámetro la ruta al archivo de script.

Si no ponemos la línea *shebang* o no damos permisos de ejecución al script, tenemos que ser nosotros los que indiquemos el intérprete y el archivo del script en el comando:

```
yo@mihost:~$ bash mi_script
```

BASH es una shell compatible con una anterior llamada Bourne Shell (1979). Si no vamos a usar en nuestro script características propias de BASH, podemos usar la línea *shebang*:

```
#!/bin/sh
```

Así el script será compatible con cualquier shell compatible Bourne Shell. Pero si estamos seguros de que queremos usar BASH, mejor utilizar:

```
#!/bin/bash
```

O mejor:

```
#!/usr/bin/env bash
```

Porque en algunas distribuciones de Linux y otros sistemas operativos —como macOS, algunos BSD, entornos chroot y sistemas embebidos— bash no está siempre instalado en `/bin/bash`, sino en otro lugar. Usar el comando env es más portable entre sistemas porque lo que hace es mirar la variable de entorno PATH para localizar el programa bash —o el que se indique como primer argumento— y ejecutarlo.

Volviendo a nuestro primer script, la segunda línea es un comentario. **BASH ignora cualquier texto detrás de un "#"**.

Finalmente, la tercera línea ejecuta el comando tar para hacer una copia de seguridad de nuestros archivos personales en el escritorio y en el directorio `Documentos`. Esta copia se guarda en un archivo de nombre `backup.tar.gz` dentro de nuestro directorio personal.

Añadir más funcionalidades al script es muy sencillo. Solo tenemos que añadir más comandos. En el fondo, **un script no es sino una lista de comandos**, de tal manera, en lugar de escribirlos a mano, los guardamos en un archivo para lanzarlos de forma más cómoda cuando nos hacen falta.

Heredoc

En esta sección construiremos una aplicación útil usando el lenguaje de script de BASH. Esta aplicación va a mostrar en la terminal información acerca del estado del sistema. A medida que construyamos nuestro script, descubriremos paso a paso las herramientas necesarias para resolver los problemas que se nos vayan presentando.

Escribiendo texto con un script de la shell

La información mostrada por nuestro script podría ser algo así:

```
Información del sistema
=======================

Información sobre los procesos:
```

```
  Datos del proceso 1
  Datos del proceso 2

Información de red:

  Datos de la interfaz de red 1
  Datos de la interfaz de red 2
```

Con lo que sabemos hasta ahora, podemos escribir un script que produzca el contenido deseado a través de la salida estándar:

```
#!/usr/bin/env bash

# sysinfo - Un script que informa del estado del sistema

echo "=== Información del sistema ==="
echo
echo "Información sobre los procesos:"
echo
echo "  Datos del proceso 1"
echo "  Datos del proceso 2"
echo
echo "  Información de red:"
echo
echo "  Datos de la interfaz de red 1"
echo "  Datos de la interfaz de red 2"
echo
```

La salida de este script puede almacenarse en un archivo usando una redirección:

```
yo@mihost:~$ ./sysinfo > sysinfo.txt
```

Como repetir **echo** no resulta muy cómodo, lo primero que podemos hacer es sustituir los múltiples usos del comando **echo** por una única llamada, incluyendo las distintas líneas entre comillas dobles:

```
#!/bin/bash

# sysinfo - Un script que informa del estado del sistema

echo "=== Información del sistema ===

Información sobre los procesos:
```

```
  Datos del proceso 1
  Datos del proceso 2

Información de red:

  Datos de la interfaz de red 1
  Datos de la interfaz de red 2"
```

Pero aunque esto es una mejora, tiene una limitación muy importante. Si quisiéramos imprimir una línea con comillas dobles como esta:

```
El nombre del equipo es "mihost".
```

tendríamos que escapar todas las comillas dobles incluidas en el texto usando el carácter de la barra invertida `\`.

```
echo "El nombre del equipo es \"mihost\"."
```

Una mejor forma de producir nuestra salida es mediante un *heredoc* —conjunción de *here-document—:*

```
#!/usr/bin/env bash

# sysinfo - Un script que informa del estado del sistema

cat << _EOF_
=== Información del sistema ===

Información sobre los procesos:

  Datos del proceso 1
  Datos del proceso 2

Información de red:

  Datos de la interfaz de red 1
  Datos de la interfaz de red 2
_EOF_
```

Un *heredoc* es una forma adicional de redirección que permite inyectar a través de la entrada estándar de un comando —en este caso a “cat”— **varias líneas del archivo del script como si fueran un archivo de texto.**

El comando **cat** por defecto muestra el contenido del archivo indicado como argumento en la línea de comandos:

```
yo@mihost:~$ cat /etc/passwd
```

Pero si no se le indica ninguno, intenta leer texto desde la entrada estándar para mostrarlo por la salida estándar. En el ejemplo anterior, las líneas que lee y muestra son las que están entre las palabras _EOF_ gracias a la expresión *heredoc*.

Variables

Empecemos construyendo el contenido real que mostrará el programa:

```
#!/usr/bin/env bash

# sysinfo - Un script que informa del estado del sistema

cat << _EOF_
=== Información del sistema ===

_EOF_
```

La cadena "Información del sistema" es el título o encabezado del informe sobre el sistema que va a generar el script. Por eso —para indicarlo de alguna manera— le hemos puesto los caracteres “=”.

Como ocurre con cualquier otro lenguaje, los scripts de BASH raramente se dan por terminados. Normalmente, son actualizados y mejorados por sus creadores y por otros que vienen detrás. Si en el futuro se desea personalizar el título "Información del sistema" por la frase "Información de mi sistema Linux", habría que buscar en el código todos los lugares donde se use dicha frase para reemplazarla.

Sin embargo, podemos mejorar el script gracias al uso de variables:

```
#!/usr/bin/env bash

# sysinfo - Un script que informa del estado del sistema

title="Información de mi sistema"

cat << _EOF_
=== $title ===

_EOF_
```

Ahora, gracias al uso de variables, solo tendríamos que cambiar la línea que asigna el contenido de la variable —que está en un lugar bien conocido al principio de script—. Como nuestro script es muy sencillo, esto parece un cambio trivial, sin embargo, a medida que los

---

scripts crecen en complejidad, usar las variables adecuadamente es cada vez más importante.

Las variables son zonas de memoria que se pueden usar para almacenar información y a la que se puede hacer referencia mediante un nombre:

- En el caso de nuestro script, **hemos creado una variable llamada "title" y puesto la frase "Información del sistema"** en la zona de memoria correspondiente.
- Dentro del *heredoc* **usamos $title para indicarle a la shell que realice una expansión de parámetros** y reemplace el nombre de la variable por el contenido de la misma antes de ejecutar la sentencia de **cat** y el *heredoc*.

Como ya vimos al estudiar la expansión de parámetros, cuando la shell ve una palabra que empieza por "$", la considera una variable e intenta sustituirla en el script por su contenido.

Más detalles sobre el uso de variables

- **¿Cómo se crea una variable?**
    - Añade una línea en tu script que contenga el nombre de la variable seguido inmediatamente por "=".
    - No se admiten espacios.
    - Después del signo igual, asigna la información que deseas almacenar.

- **¿De dónde vienen los nombres de variables?** El usuario decide los nombres de sus variables, sin embargo, hay algunas reglas que se deben respetar:
    - Los nombres deben comenzar con una letra.
    - No se permiten espacios dentro de los nombres. La convención es usar guión bajo en su lugar.
    - No se pueden usar signos de puntuación.

- **¿Cómo se crea una constante?** La mayoría de lenguajes de programación proveen facilidades para soportar el uso de valores que no están sujetos a cambio:
    - BASH también provee este tipo de funcionalidades, pero casi nunca se usan.
    - En lugar de ello, por convenio, **se utilizan nombres en mayúsculas para indicar que esa variable es constante**.

- **¿Cómo se usan las variables dentro de las expresiones aritméticas?** En una expresión aritmética se hace referencia al valor de las variables sin usar el carácter "$" antes del nombre —p. ej. `echo "horas: $((seconds / 3600))"`—

Variables de entorno

Como hemos visto, cuando se inicia sesión en la shell, algunas variables son inicializadas por los archivos de configuración que vimos anteriormente. A estas variables se las

---

denomina **variables de entorno**. Para ver las variables de entorno simplemente hay que ejecutar el comando printenv.

Por nuestra parte vamos a escoger una: `HOSTNAME`. Esta contiene el nombre del equipo y la vamos a en nuestro script de la siguiente manera:

```
#!/usr/bin/env bash

# sysinfo - Un script que informa del estado del sistema

title="Información de mi sistema"

cat << _EOF_
=== $title $HOSTNAME ===

_EOF_
```

Así nuestro script mostrará el nombre de la máquina donde se ejecute. **Por convenio, los nombres de las variables de entorno se escriben siempre en mayúsculas**, al igual que ocurre con las constantes, ya que es raro que sufran cambios.

Sustitución de comandos

Vamos a añadir una marca de tiempo —o *timestamp*— a nuestro informe para saber cuándo fue generado por última vez. También mostraremos qué usuario del sistema lo generó.

```
#!/usr/bin/env bash

# sysinfo - Un script que informa del estado del sistema

title="Información de mi sistema"

cat << _EOF_
=== $title $HOSTNAME ===

Actualizada el $(date +"%x %r%Z") por $USER
_EOF_
```

Sin embargo, parece mucho más interesante usar variables para esto, ya que así podemos cambiar fácilmente en el futuro cómo se muestra esta información. Puesto que el valor de estas variables no va a cambiar a lo largo de la ejecución del script, las crearemos usando mayúsculas para indicar que son constantes.

```
#!/usr/bin/env bash
```

```
# sysinfo - Un script que informa del estado del sistema

TITLE="Información del sistema para $HOSTNAME"
RIGHT_NOW=$(date +"%x %R %Z")
TIME_STAMP="Actualizada el $RIGHT_NOW por $USER"

cat << _EOF_
=== $TITLE ===

$TIME_STAMP
_EOF_
```

Encabezados con estilo

La terminal tiene algunas capacidades para dar formato al texto —aunque mucho más limitadas que la interfaz gráfica— que pueden ser interesantes para destacar algunas partes de la salida de nuestro programa.

Como mínimo, **la mayor parte de las terminales son capaces de interpretar las secuencias de escape ANSI** —más detalles en man console_codes—. Estas **son unas secuencias de caracteres especiales que cuando se envían a la terminal sirven para cambiar el color y la intensidad del texto o para mover el cursor**, entre otras muchas opciones.

Todas las secuencias empiezan por los caracteres ESC (carácter ASCII 27 en decimal y 1B en hexadecimal) y "[", seguidos por el resto de la secuencia que determina la acción o configuración deseada. Por ejemplo, la secuencia "\x1b[32m" cambia el color de la fuente al verde, mientras que "\x1b[1m" activa el uso de la negrita o aumenta la intensidad de la fuente[1][2].

Para utilizarlas solo tenemos que imprimirlas en la terminal. Por ejemplo, así podemos poner una palabra en negrita:

```
yo@mihost:~$ echo "Esto está en "$'\x1b[1m'NEGRITA$'\x1b[0m'.
```

donde:

- Usamos la sintaxis de entrecomillado `$'cadena'` para poder incluir el carácter ESC (1B en hexadecimal)

---

[1] ANSI Escape Codes es una web interesante para buscar y probar secuencias de escape ANSI antes de usarlas en nuestros scripts

[2] Al hacer scripts, la forma más portable de usar secuencias de escape ANSI es invocando el comando tput, indicando lo que queremos cambiar de la terminal. Este comando consulta una base de datos —denominada terminfo— donde se almacenan las capacidades de cada tipo de terminal y la secuencia de escape correspondiente —que pueden ser diferentes según la terminal—, prepara las secuencia de caracteres y la imprime en la terminal para que tenga efecto.

- `$'\x1b[1m'` es la secuencia de caracteres que pone la terminal en modo negrita.
- `$'\x1b[0m'` es la secuencia de caracteres que resetea la terminal al estilo por defecto.

Ahora podemos utilizar estas secuencias para darle un poco de estilo a la salida de nuestro script:

```
#!/usr/bin/env bash

# sysinfo - Un script que informa del estado del sistema

TITLE="Información del sistema para $HOSTNAME"
RIGHT_NOW=$(date +"%x %r%Z")
TIME_STAMP="Actualizada el $RIGHT_NOW por $USER"

TEXT_BOLD=$'\x1b[1m'
TEXT_GREEN=$'\x1b[32m'
TEXT_RESET=$'\x1b[0m'

cat << _EOF_
$TEXT_BOLD$TITLE$TEXT_RESET

$TEXT_GREEN$TIME_STAMP$TEXT_RESET
_EOF_
```

Lo que hemos hecho es guardar las secuencias en variables para poder utilizarlas fácilmente por todo el script.

💡Las secuencias de escape ANSI no son una característica de BASH, sino de la terminal y están disponibles en diferentes sistemas operativos. Por tanto, **podemos usarlas también en C, C++, Python, Java o cualquier otro lenguaje para darle estilo a la salida de texto de nuestro programa**.

Funciones de la Shell

A medida que los programas se hacen más largos y complejos, se vuelven más difíciles de diseñar, codificar y mantener. Así que normalmente es útil dividir una tarea grande y compleja en subtareas más pequeñas y simples.

Vamos a aprender a dividir un script monolítico en una serie de funciones separadas.

Tareas de nuestro script

A medida que nuestro script crece en complejidad, aplicaremos un diseño top-down para planificar la codificación de este.

Veamos las tareas que ya implementa nuestro script:

---

1. Configurar las variables.
2. Iniciar la impresión de la salida del programa.
3. Escribir el título en negrita en la terminal.
    a. Activar el estilo negrita de la terminal.
    b. Escribir el título en la terminal.
    c. Resetear el estilo de la terminal.
4. Escribir *timestamp* en verde en la terminal.
    a. Activar el color verde para la fuente en la terminal.
    b. Escribir el timestamp.
    c. Resetear el estilo de la terminal.
5. Terminar la impresión de la salida del programa.

Todas esas tareas están implementadas, pero queremos añadir otras. Vamos a añadir tareas adicionales después de la tarea 7:

4. Escribir *timestamp* en verde en la terminal.
5. **Escribir información sobre la versión del sistema.**
6. **Escribir el tiempo que lleva encendido el sistema.**
7. **Escribir el espacio en disco disponible.**
8. **Escribir el espacio ocupado por el directorio home.**
9. Terminar la impresión de la salida del programa.

Estaría bien que hubiera comandos que realizaran esas tareas. Si los hubiera, podríamos colocarlos en nuestro script así (**en negrita en el ejemplo**):

```
#!/usr/bin/env bash

# sysinfo - Un script que informa del estado del sistema

##### Constantes

TITLE="Información del sistema para $HOSTNAME"
RIGHT_NOW=$(date +"%x %r%Z")
TIME_STAMP="Actualizada el $RIGHT_NOW por $USER"

##### Estilos

TEXT_BOLD=$'\x1b[1m'
TEXT_GREEN=$'\x1b[32m'
TEXT_RESET=$'\x1b[0m'

##### Programa principal

cat << _EOF_
```

```
$TEXT_BOLD$TITLE$TEXT_RESET

$TEXT_GREEN$TIME_STAMP$TEXT_RESET
_EOF_

system_info
show_uptime
drive_space
home_space
```

Pero estos comandos no existen, así que tenemos que crearlos nosotros mediante funciones de la shell.

```
#!/usr/bin/env bash

# sysinfo - Un script que informa del estado del sistema

##### Constantes

TITLE="Información del sistema para $HOSTNAME"
RIGHT_NOW=$(date +"%x %r%Z")
TIME_STAMP="Actualizada el $RIGHT_NOW por $USER"

##### Estilos

TEXT_BOLD=$'\x1b[1m'
TEXT_GREEN=$'\x1b[32m'
TEXT_RESET=$'\x1b[0m'

##### Funciones

system_info()
{

}


show_uptime()
{

}


drive_space()
{
```

```
}


home_space()
{

}

##### Programa principal

cat << _EOF_
$TEXT_BOLD$TITLE$TEXT_RESET

$TEXT_GREEN$TIME_STAMP$TEXT_RESET
_EOF_

system_info
show_uptime
drive_space
home_space
```

Algunas cosas a tener en cuenta sobre las funciones:

- Deben estar definidas antes de que se intenten usar en el código.
- El cuerpo de la función —entre "{" y "}"— debe contener al menos un comando válido.
- Es opcional añadir los paréntesis () tras el nombre —como en C y C++— o añadir la palabra `function` antes del nombre.
- Por defecto, **todas las variables son globales**, tanto si se definen fuera como dentro de las funciones.

Actualmente, el script no es válido porque los cuerpos de las funciones están vacíos. Una forma simple de solucionarlo es escribir `return` dentro de cada función.

Implementando las funcionalidades reales

Mantén tus scripts en funcionamiento

Cuando estás escribiendo un programa, **es una buena costumbre el añadir pequeñas cantidades de código y ejecutar el script, luego añadir otra pequeña cantidad de código y volver a ejecutar**, y así sucesivamente. De esta forma, si se introduce un error en el código, será más fácil de localizar y corregir.

A medida que se añaden funciones al script, se puede utilizar una técnica llamada *stubbing*. Esta técnica funciona de la siguiente manera: imaginemos que queremos crear una función

---

llamada **system_info**, pero todavía no hemos resuelto todos los detalles de la misma. En lugar de parar el desarrollo del código hasta que la función esté terminada, se puede añadir un comando **echo** dentro de la función:

```
system_info()
{
    # Función de stub temporal
    echo "función system_info"
}
```

De esta forma, nuestro script se podrá seguir ejecutando, aunque no hayamos terminado la función **system_info**. Más tarde se reemplazará este código por una versión que implemente la funcionalidad deseada.

La razón de usar un **echo** es la de obtener alguna indicación de que la función está siendo ejecutada.

Veamos como sería crear todas las funciones pendientes de nuestro script como *stubs*:

```
#!/usr/bin/env bash

# sysinfo - Un script que informa del estado del sistema

##### Constantes

TITLE="Información del sistema para $HOSTNAME"
RIGHT_NOW=$(date +"%x %r%Z")
TIME_STAMP="Actualizada el $RIGHT_NOW por $USER"

##### Estilos

TEXT_BOLD=$'\x1b[1m'
TEXT_GREEN=$'\x1b[32m'
TEXT_RESET=$'\x1b[0m'

##### Funciones

system_info()
{
    # Función de stub temporal
    echo "función system_info"
}


show_uptime()
```

```
{
    # Función de stub temporal
    echo "función show_uptime"
}


drive_space()
{
    # Función de stub temporal
    echo "función drive_space"
}


home_space()
{
    # Función de stub temporal
    echo "función home_space"
}

##### Programa principal

cat << _EOF_
$TEXT_BOLD$TITLE$TEXT_RESET

$TEXT_GREEN$TIME_STAMP$TEXT_RESET
_EOF_

system_info
show_uptime
drive_space
home_space
```

Implementar la función show_uptime

La función **show_uptime** mostrará la salida del comando uptime. Este comando muestra algunos detalles muy interesantes acerca del sistema, incluyendo el tiempo que este ha estado encendido desde el último reinicio, el número de usuarios autenticados y la carga del sistema.

```
show_uptime()
{
    echo "${TEXT_ULINE}Tiempo de encendido del sistema$TEXT_RESET"
    echo
    uptime
}
```

Usamos `${TEXT_ULINE}` en lugar de `$TEXT_ULINE` porque si no BASH se confundiría:

```
    echo "$TEXT_ULINETiempo de encendido del sistema$TEXT_RESET"
```

pensando que nos interesa la variable `TEXT_ULINETiempo`, que obviamente no existe. También sería correcto y, probablemente más elegante, utilizar las llaves en ambos casos:

```
    echo "${TEXT_ULINE}Tiempo de encendido del
sistema${TEXT_RESET}"
```

`TEXT_ULINE` se define al comienzo del script asignándole los caracteres de escape necesarios para activar el subrayado del texto:

```
TEXT_ULINE=$'\x1b[4m'
```

Implementar la función system_info

De forma similar podemos sustituir el *stub* de la función **system_info** por su implementación, para que muestre información sobre la versión del sistema operativo:

```
system_info()
{
    echo "${TEXT_ULINE}Versión del sistema${TEXT_RESET}"
    echo
    uname -a
}
```

El comando **uname** se utiliza para mostrar información sobre el sistema. Con la opción “-a” simplemente la muestra toda.

Actividad 1

Es hora de parar un momento y trabajar sobre un ejercicio propuesto.

Implementar la función drive_space

De forma similar a como hemos hecho con **show_uptime** y **system_info**, implementa la función **drive_space**. Esta función debe mostrar el espacio ocupado en las particiones / discos duros del sistema.

Para eso debe utilizar el comando df, que está especializado en proporcionar exactamente dicha información.

Control de flujo (I)

La shell proporciona diversos comandos que podemos usar para controlar el flujo de ejecución de nuestro programa.

Comando If

El comando **if** toma una decisión basándose en el *estado de salida* del comando indicado en la condición:

```
if comandos; then
comandos
[elif comandos; then
comandos...]
[else
comandos...]
fi
```

donde *comandos* es general una lista de comandos.

Estado de salida de un comando

Los procesos y los comandos —incluyendo los scripts y las funciones de la shell— cuando terminan, lanzan un valor al sistema que se denomina *estado de salida:*

- El valor es un entero entre 0 a 255.
- **Indica el éxito o el fallo de la ejecución del comando.**
- **Por convención, un valor de cero indica éxito, mientras que cualquier otro fallo**
- **La shell proporciona el parámetro o variable "$?", para que podamos examinar el estado de salida:**

```
yo@mihost:~$ ls -d /usr/bin
/usr/bin
yo@mihost:~$ echo $?
0
yo@mihost:~$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
yo@mihost:~$ echo $?
2
```

En el ejemplo anterior, como la primera ejecución de “ls” tiene éxito, el valor almacenado en "$?" es cero. Sin embargo, la segunda vez se produce un error, por lo que el valor de "$?" es 2:

- Cada comando puede proporcionar unos valores de estado de salida diferentes para ayudar a diagnosticar el error.

- Las páginas del manual de cada comando suelen incluir una sección denominada "Exit status" donde se describen los códigos devueltos por el comando en cuestión.
- Sin embargo, **un cero siempre indica éxito.**

El sistema proporciona dos comandos muy simples:

- **true**, que siempre se ejecuta con éxito, por lo que su estado de salida siempre es 0.
- **false**, que siempre se ejecuta sin éxito. En este caso su estado de salida suele ser 1.

```
yo@mihost:~$ if true; then
> echo "Es verdad";
> fi
It's true.

yo@mihost:~$ if false; then
> echo "Es verdad";
>fi
yo@mihost:~$
```

Comando test

El comando **test** permite comprobar ciertas condiciones:

- Si la condición es verdadera, **test** termina con estado de salida 0.
- En otro caso, su estado de salida es 1.

Por lo tanto, se trata de un comando muy útil, que por lo general se utiliza con el comando **if** para alterar el flujo del programa:

```
if test -f .bash_profile; then
    echo "Tienes un archivo .bash_profile. Todo correcto."
else
    echo "¡Ay! ¡No tienes un archivo .bash_profile!"
fi
```

En el ejemplo anterior usamos la expresión "-f .bash_profile" que le indica al comando **test** que compruebe si ".bash_profile" es un archivo. Si así fuera, el comando devolverá un 0 como estado de salida, en caso contrario el valor será 1.

Aquí tenemos una lista parcial de las condiciones que el comando puede evaluar, aunque en la ayuda se pueden encontrar muchas más:

Expresión	**Descripción**
-d archivo	Verdadero si *archivo* es un directorio.

-e archivo	Verdadero si *archivo* existe.
-f archivo	Verdadero si *archivo* existe y es un archivo regular..
-L archivo	Verdadero si *archivo* es un enlace simbólico.
-r archivo	Verdadero si *archivo* es legible por el usuario.
-w archivo	Verdadero si *archivo* es escribible por el usuario.
-x archivo	Verdadero si *archivo* es ejecutable.
archivo1 -nt archivo2	Verdadero si *archivo1* no es más antiguo que *archivo2*
archivo1 -ot archivo2	Verdadero si *archivo1* es más antiguo que *archivo2*
-z cadena	Verdadero si *cadena* tiene longitud 0.
-n cadena	Verdadero si *cadena* no tiene longitud 0.
cadena1 = cadena2	Verdadero si *cadena1* es igual a *cadena2*.
cadena1 != cadena2	Verdadero si *cadena1* no es igual a *cadena2*.

Comando []

El uso del comando test es tan común que tiene una segunda forma que es usada con mucha mayor frecuencia y que es completamente equivalente:

```
# Primera forma de test
test expression

# Segunda forma de test
[ expression ]
```

Lo que nos permite escribir el ejemplo anterior de la siguiente manera:

```
if [ -f .bash_profile ]; then
    echo "Tienes un archivo .bash_profile. Todo correcto."
else
    echo "¡Ay! ¡No tienes un archivo .bash_profile!"
fi
```

Otro ejemplo con comparaciones sería el siguiente:

```
if [ "$USER" != root ]; then
    echo "No tienes permisos de superusuario"
else
```

Nótense los espacios entre los corchetes —[y]— y la expresión que hay dentro, puesto que son obligatorios.

Comando exit y return

Los scripts son comandos y por eso siempre terminan con un estado de salida. Por defecto, el estado de salida de un script es el estado de salida del último comando que ejecuta. Sin embargo, los scripts bien escritos usan el comando **exit** para terminar controlando el valor devuelto. Como cuando programamos en C, C++ y otros lenguajes, es conveniente terminar con un valor diferente según la causa de la terminación, para que el programa que invocó el script, si lo necesita, pueda conocer con precisión el motivo de su finalización.

El comando **exit** provoca que el script finalice inmediatamente, estableciendo el estado de salida al valor indicado en el argumento. Por ejemplo:

```
exit 0
```

finaliza el script y devuelve 0 —éxito— mientras que:

```
exit 1
```

finaliza el script devolviendo 1 —fallo—.

Las funciones también son comandos, por lo que también tienen un estado de salida, que por defecto es el del último comando ejecutado. Pero si queremos controlar el estado de salida de una función debemos usar el comando **return**.

```
return 3
```

Actividad 2

Es hora de parar un momento y trabajar sobre un ejercicio propuesto.

Implementa la función home_space

Implementa la función **home_space**, de tal forma que:

1. Si el usuario que ejecuta el script es root, **home_space** debe mostrar es espacio ocupado por cada uno de los subdirectorios en `/home` (ojo, no el total de `/home` sino el de cada subdirectorio) en orden decreciente (primero el directorio que ocupa más espacio). Por ejemplo así:

   ```
   USADO     DIRECTORIO
   ```

```
1092345  vmunoz
923450   jttoledo
282334   jmtorres
...
```

2. Si el usuario que ejecuta el script NO es root, solo debe mostrar el espacio ocupado por el directorio personal del usuario:

```
USADO    DIRECTORIO
1092345  yo
```

Para resolver el ejercicio:

1. Recuerden que la variable de entorno $USER nos indica el nombre del usuario actual.
2. Debes usar el comando du, que recorre los directorios indicados como argumento, mostrando el espacio ocupado por cada archivo y cada subdirectorio de su interior.

   Por ejemplo:

   ```
   yo@mihost:~$ du /home/jmtorres /home/jttoledo
   ```

   o

   ```
   yo@mihost:~$ du /home/b*
   ```

   Como solo interesa el total del espacio ocupado por cada directorio, deberías buscar la opción que solo muestra la suma total del directorio, y no cada archivo y subdirectorio individualmente.
