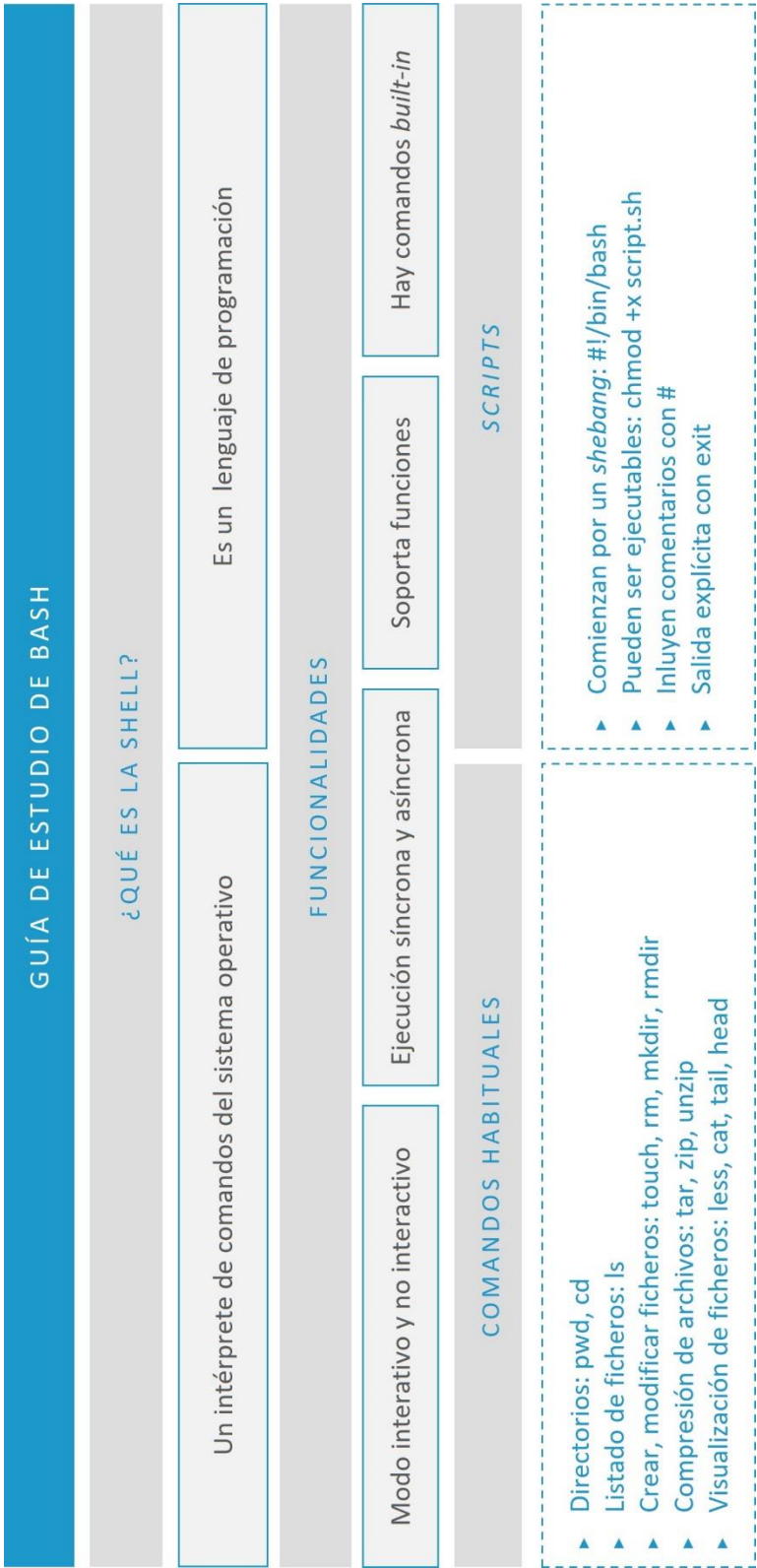


Administración de Sistemas en la Cloud

Guía de estudio de Bash

Índice

Esquema	3
Ideas clave	4
3.1. Introducción y objetivos	4
3.2. Qué es la <i>shell</i>	4
3.3. Comandos habituales	9
3.4. <i>Scripts</i> de Bash	24
3.5. Referencias bibliográficas	26
A fondo	27
Test	28



Esquema

3.1. Introducción y objetivos

Teniendo en cuenta el concepto de *script* y las ventajas que ofrece para la automatización de tareas de administración, este tema servirá de introducción para una de las herramientas de *scripting* más usadas en servidores Linux: la *shell* Bash.

Los **objetivos** que se pretenden conseguir son:

- ▶ Entender el concepto de *shell* y de *scripts* de *shell*.
- ▶ Aprender los comandos básicos de Bash.
- ▶ Familiarizarse con la ejecución de *scripts* de automatización.

3.2. Qué es la *shell*

Dicho de un modo muy sencillo, la *shell* es un **macroprocesador que ejecuta comandos**. Esta definición indica que hay una funcionalidad donde textos y símbolos se combinan para crear expresiones más grandes.

La *shell* es, a la vez, una intérprete de comandos y un lenguaje de programación. En su rol de intérprete de comandos, la *shell* ofrece al usuario una rica interfaz de utilidades o herramientas de GNU. Las características del lenguaje de programación, por su parte, permiten que estas herramientas se combinen.

Bash

Hay múltiples *shells* disponibles en entornos Linux y Unix. Aquí se usará Bash, ya que está disponible prácticamente la totalidad de las distribuciones. En caso de no estarlo, es relativamente sencillo traducir *scripts* de Bash a *scripts* de *shell* Bourne.

Historia de Bash

La *shell* Bourne original data de 1979 (Robbins, 2010), cuando se empezó a distribuir en entornos UNIX. Todavía se encuentra en `/bin/sh` en muchas distribuciones y, de hecho, ha evolucionado solo en contadas ocasiones. Otra *shell* de la época, la *shell* C (*csh*) ofrecía funcionalidades útiles para uso interactivo, como control de procesos e historial de comandos. Era habitual entre los administradores el uso de *sh* para programación de *scripts* y de *csh* para sesiones interactivas. Fue en 1983 cuando se presentó la *shell* Korn, o *ksh*, que se mantuvo compatible con *sh*, pero con funcionalidades interactivas de *csh*. Finalmente, Bash, de *bourne again shell*, apareció en 1989 como un clon de *sh* escrito desde cero, incorporando muchas funcionalidades de *ksh* desde entonces.

Características de la *shell*

La *shell* puede funcionar en modo interactivo y en modo no interactivo. En **modo interactivo**, la *shell* acepta entrada desde el teclado, ya sea el teclado local o un teclado remoto en una sesión SSH. El sistema operativo arranca una *shell* en el momento en el que un usuario inicia una nueva sesión (Van Vugt, 2015, pp. 1-26), ya sea una sesión local o por SSH. Por tanto, una *shell* no es sino un proceso más.

En **modo no interactivo**, la *shell* lee un archivo (es decir, un *script*) y ejecuta los comandos contenidos en él, línea por línea. Estos *scripts* pueden convertirse en comandos en sí mismos. Estos nuevos comandos tienen el mismo *status* que comandos del sistema en directorios como `/bin`, permitiendo que usuarios o grupos puedan construir sus propios entornos y automatizar sus tareas comunes.

En cuanto a los **comandos GNU**, la *shell* permite su ejecución de modo síncrono y asíncrono. En el primer caso, la *shell* acepta un comando, lo ejecuta y espera a que este termine antes de aceptar el siguiente comando. Por su parte, los comandos asíncronos continúan ejecutándose en paralelo con la *shell*, mientras lee y ejecuta comandos adicionales.

Las *shells* suelen incluir un pequeño **conjunto de comandos** que implementan funcionalidades que son casi imposibles de obtener por una vía diferente, por ejemplo, `cd`, `break`, `continue` y `exec`. Estos comandos no pueden ser implementados por fuera de la *shell*, porque la manipulan directamente. Otros comandos, como `history`, `getopts`, `kill` o `pwd`, pueden ser implementados por separado, pero, aun así, es más conveniente utilizarlos como comandos *built-in*. Algunos de estos comandos están orientados específicamente al uso interactivo, más que a aumentar el lenguaje de programación. Por ejemplo, la edición de línea de comando, el ya mencionado `history`, `alias` y los comandos de control de trabajo.

Funciones

Las funciones no son más que pequeñas subrutinas dentro de un *script* de *shell*. Son una forma de agrupar comandos para su ejecución posterior, usando un solo nombre para referenciar varias sentencias. Se ejecutan como un comando regular u ordinario. Las funciones de *shell* se ejecutan en el contexto de *shell* actual: no se crea ningún proceso nuevo para interpretarlos.

Parámetros

Un parámetro es una entidad que almacena valores. Puede ser un nombre, un número o un carácter especial. Una variable es un parámetro denotado por un nombre. Un parámetro se establece si se le ha asignado un valor.

Conceptos relacionados con la *shell*

- ▶ **POSIX:** su nombre es un acrónimo de *portable operating system interface* - Unix. Define una familia de estándares de sistemas abiertos basados en Unix, que intentan asegurar la portabilidad entre diferentes sistemas operativos.
- ▶ **Built-in:** es un comando que se implementa internamente por la *shell*, en lugar de por un programa ejecutable en algún lugar en el sistema de archivos. Es decir, es un comando llevado a cabo por la *shell*, como `cd`, en lugar de interpretarlo como una solicitud para cargar y ejecutar algún otro programa, como `vim`.

Esto tiene dos consecuencias principales. En primer lugar, por lo general, es más rápido, ya que el tiempo que toma cargar y ejecutar un programa es más prolongado. En segundo lugar, un comando *built-in* puede afectar el estado interno de la *shell*. Es por eso por lo que los comandos como `cd` deben ser *built-in*, dado que un programa externo no puede cambiar el directorio actual de la *shell*. Otros comandos, como `echo`, podrían ser *built-in* por un motivo de eficiencia, pero no hay ninguna otra razón intrínseca para que no puedan ser comandos externos.

- ▶ **Job control:** las características básicas del control de trabajos son la suspensión, continuación o terminación de procesos. El usuario puede ejecutarlo selectivamente, de acuerdo con sus necesidades.
- ▶ **Name:** también conocido como identificador, es una palabra que consiste únicamente de letras, números y guiones bajos y comienza con una letra o un guion bajo. Los nombres se utilizan para identificar variables y funciones.
- ▶ **Field:** una unidad de texto que es el resultado de una de las expansiones de la *shell*. Después de la expansión, al ejecutar un comando, los campos resultantes se utilizan como nombre del comando y argumentos.

- ▶ *Process group*: una colección o conjunto de procesos relacionados que tienen el mismo ID.
- ▶ *Process group ID*: identificador único que representa a un *process group* durante su ciclo de vida.
- ▶ *Token*: es una secuencia de caracteres que la *shell* considera una unidad. Puede ser una palabra o un operador.
- ▶ *Control operator*: es un *token* que tiene una función de control. Los más relevantes son:
 - (;): se pueden poner dos o más comandos en la misma línea separados por un punto y coma. Ambas series se ejecutarán secuencialmente y la *shell* esperará a que cada comando termine antes de iniciar el siguiente.
 - (&): cuando una línea termina con &, la *shell* no esperará a que termine el comando. El comando se ejecutará en segundo plano y el usuario podrá seguir introduciendo comandos.
 - (\$?): el código de salida de la orden anterior se almacena en el parámetro de shell \$? . Este parámetro se utiliza para comprobar el estado de la última orden ejecutada. Si el valor devuelto por \$? es 0, significa que el último comando acabó con éxito; de lo contrario, el comando falló. El valor concreto depende de cada utilidad, pero, en todo caso, es un número entero de 8 bits.
 - (&&): un «and» lógico. El segundo comando se ejecuta solo si el primero tiene éxito. Por ejemplo, test "\$VAR1" = "val1" && echo \$VAR1 imprime el valor de la variable de entorno VAR1, solo si está definida como "val1".
 - (||): un «or» lógico. El segundo comando se ejecuta solo cuando la primera orden ha fallado. Por ejemplo, test "\$VAR1" != "val1" || echo "ERROR" imprime el mensaje de error si VAR1 no contiene "val1".
 - (#): todo lo escrito después de # es ignorado por la *shell*. Esto es útil para escribir un comentario y que no tenga efectos en la ejecución.

3.3. Comandos habituales

Este apartado muestra algunos de los ejemplos de expresiones habituales de *shell*, tanto de comandos básicos como con ciertos parámetros útiles.

A continuación, en el vídeo *Primeros pasos en Bash*, se verá una demostración de comandos básicos en Bash: archivos y directorios, utilidades GNU, Sudo.



Accede al vídeo

► Comandos de directorio:

- `pwd`: muestra el camino completo del directorio actual.

```
$ pwd
/etc/apache2/extra
```
- `cd`: cambia de directorio.

```
$ cd /etc/apache2
```
- `cd ~` : lleva a la carpeta *home* del usuario.
- `cd -` : lleva a la última ruta.
- `cd ..` : cambia al directorio padre del directorio actual.

► Listado de ficheros:

- `ls`: listado de ficheros.

```
$ ls
extra          magic          other
httpd.conf     mime.types     users
httpd.conf.pre-update  original
```
- `ls -al`: lista ficheros, carpetas e información.

```
$ ls -al
total 128
drwxr-xr-x 10 root wheel  320 Aug 27  2018 .
drwxr-xr-x 90 root wheel 2880 Apr 18 17:22 ..
drwxr-xr-x 14 root wheel  448 Apr  4  2018 extra
-rw-r--r--  1 root wheel 21150 Aug 27  2018 httpd.conf
```

```
-rw-r--r-- 1 root wheel 21150 Apr 4 2018 httpd.conf.pre-update
-rw-r--r-- 1 root wheel 13077 Apr 4 2018 magic
-rw-r--r-- 1 root wheel 61118 Apr 4 2018 mime.types
drwxr-xr-x 4 root wheel 128 Apr 4 2018 original
drwxr-xr-x 3 root wheel 96 Apr 4 2018 other
drwxr-xr-x 3 root wheel 96 Aug 27 2018 users
```

- `ls -aR`: lista de forma recursiva.
 - `ls -aR | more`: lista de forma recursiva, usando una canalización para mostrarlo por pantalla. Es útil si la salida ocupa más líneas que las que puede mostrar la ventana de consola.
 - `ls -aR > resultado.txt`: similar a la anterior, pero vuelca el resultado de la salida a un archivo, en vez de mostrarlo por pantalla.
 - `ls *.htm`: enumera todos los archivos terminados en .htm.
 - `ls -al dir/subdir/`: muestra un listado de los ficheros en ese directorio.
- Crear, modificar y borrar ficheros y carpetas:
- `touch /home/usr/html/index.htm`: crea el fichero index.htm sin contenido.
 - `rm file.txt`: borra file.txt.
 - `rm -rf dir/`: borra el directorio de forma recursiva y sin confirmación.
 - `mkdir carpeta`: crea un directorio con el nombre carpeta.
 - `rmdir carpeta`: borra el directorio llamado carpeta.
- Archivos comprimidos:
- `zip arch.zip /home/usr/public/dir`: comprime el directorio y su contenido en el fichero arch.zip.
 - `unzip arch.zip`: descomprime arch.zip.
 - `unzip -v arch.zip`: visualiza el contenido de arch.zip.
 - `tar xvzf package.tgz`: descomprime el fichero package.tgz.
- Visualización de ficheros:
- `less fichero.log`: muestra el contenido del fichero solo en modo lectura; permite búsquedas, ir al principio y al final del fichero, pasar de página, etc.
 - `cat fichero.log`: vuelca el contenido completo del fichero en la pantalla.

- `tail fichero.log`: muestra las últimas diez líneas del fichero.
- `tail -5 fichero.log`: muestra las últimas cinco líneas del fichero.
- `tail -f fichero.log`: muestra las últimas diez líneas del fichero y se engancha al fichero, mostrando las líneas nuevas que aparezcan en este. Es muy útil para mostrar un fichero de *log*, según se actualiza.
- `head fichero.log`: muestras las diez primeras líneas del fichero.
- `tail -5 fichero.log`: muestra las últimas cinco líneas del fichero.

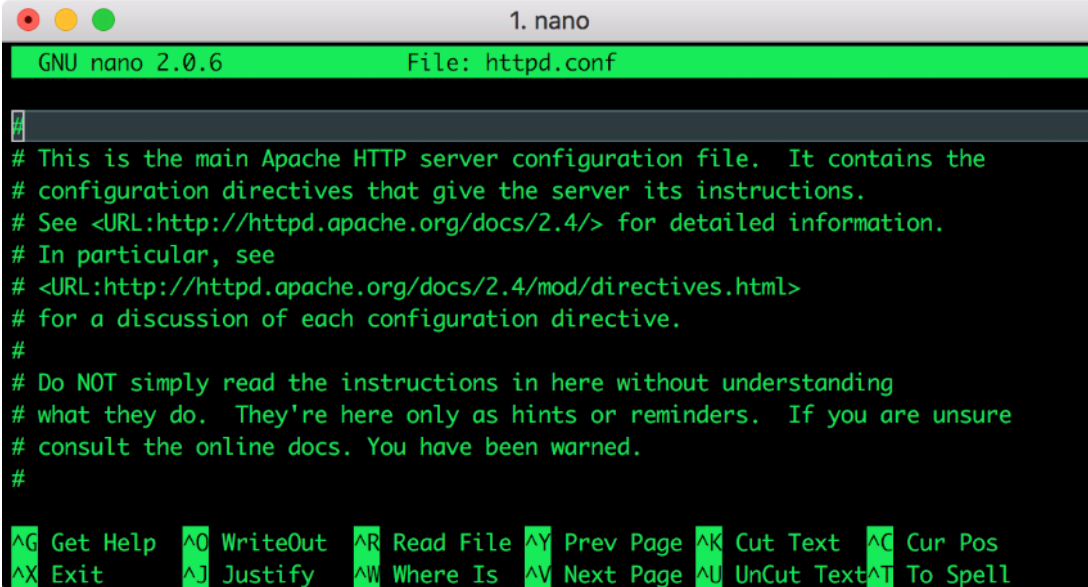
► Otros comandos:

- `cp -a /home/usr/origen/* /home/usr/destino/`: copia todos los contenidos de un directorio a otro, manteniendo sus permisos.
- `du -sh`: visualiza el espacio total ocupado por la carpeta actual.
- `du -sh *`: muestra el espacio ocupado de cada fichero.
- `whoami`: muestra el nombre del usuario.
- `cat /var/log/secure* | grep Accepted | awk '{print $9 " " $11 }' | sort | uniq -c | sort -rn | head -20`: visualiza las veinte IP con más *login* exitosos.
- `cat /var/log/secure* | grep "Failed password" | awk '{print $9 " " $11 }' | sort | uniq -c | sort -rn | head -20`: visualiza las veinte IP con más intentos de *login* incorrectos.

Editores

Aunque los editores enfocados a programación ofrecen multitud de características para facilitar la vida del administrador o del desarrollador, es habitual encontrarse en la situación de tener que **editar o crear un fichero** en un servidor sin poder usar una herramienta de entorno gráfico. Prácticamente cualquier distribución incorporará al menos un editor en modo texto, es decir, un editor de ficheros que funciona en la consola de línea de comandos. Entre estos editores, se encuentran *vi*, *vim*, *nano*, *pico* o *emacs*, por mencionar algunos.

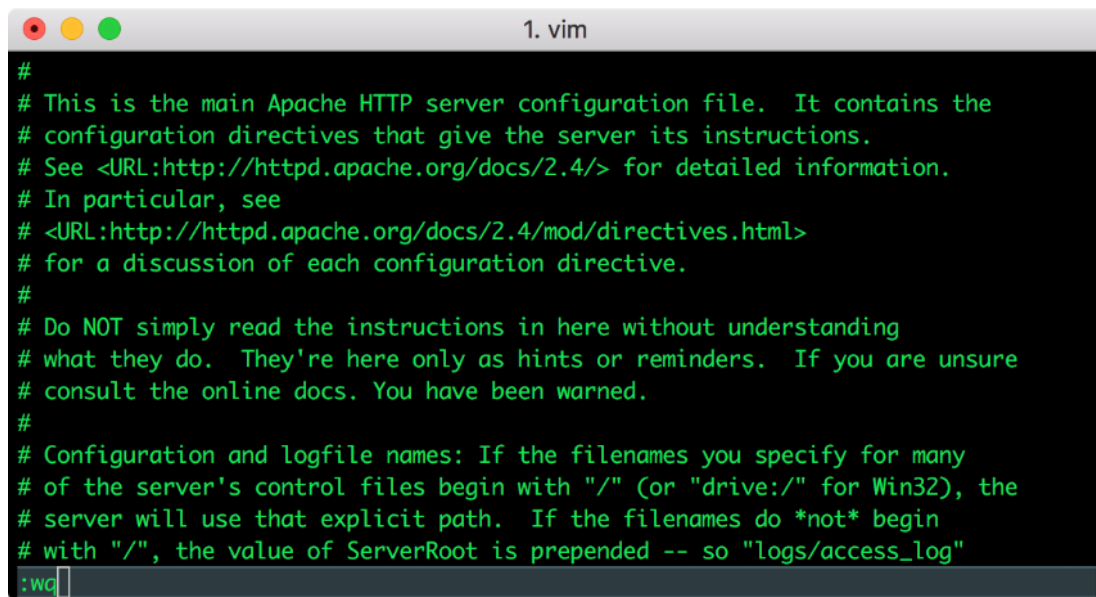
Al no disponer de entorno gráfico, estos editores se manejan con **combinaciones de teclas**. Por ejemplo, nano muestra una barra en la parte inferior con los comandos más habituales. En la Figura 1, ^O significa Ctrl+O.



```
1. nano
GNU nano 2.0.6 File: httpd.conf
# This is the main Apache HTTP server configuration file. It contains the
# configuration directives that give the server its instructions.
# See <URL:http://httpd.apache.org/docs/2.4/> for detailed information.
# In particular, see
# <URL:http://httpd.apache.org/docs/2.4/mod/directives.html>
# for a discussion of each configuration directive.
#
# Do NOT simply read the instructions in here without understanding
# what they do. They're here only as hints or reminders. If you are unsure
# consult the online docs. You have been warned.
#
^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell
```

Figura 1. Editor nano con el fichero httpd.conf. Fuente: elaboración propia.

Uno de los editores más extendidos es vi (Van Vugt, 2015, pp. 69-90). Este editor funciona con el concepto de «modo»: por defecto, arranca en modo de «comando», en el que se pueden ejecutar operaciones. Para poder escribir, hay que pulsar i para entrar en modo de «edición». Habrá que pulsar Esc para volver al modo de comando para, por ejemplo, guardar el fichero o salir de vi.



```
#
# This is the main Apache HTTP server configuration file. It contains the
# configuration directives that give the server its instructions.
# See <URL:http://httpd.apache.org/docs/2.4/> for detailed information.
# In particular, see
# <URL:http://httpd.apache.org/docs/2.4/mod/directives.html>
# for a discussion of each configuration directive.
#
# Do NOT simply read the instructions in here without understanding
# what they do. They're here only as hints or reminders. If you are unsure
# consult the online docs. You have been warned.
#
# Configuration and logfile names: If the filenames you specify for many
# of the server's control files begin with "/" (or "drive:/" for Win32), the
# server will use that explicit path. If the filenames do *not* begin
# with "/", the value of ServerRoot is prepended -- so "logs/access_log"
:wq
```

Figura 2. Editor vi con el fichero httpd.conf en modo comando con el comando wq. Fuente: elaboración propia.

Redirección

Bash soporta la redirección de la salida de un comando. Por un lado, puede volcar la salida estándar en un fichero:

```
$ ls -la > listado.txt
```

Este comando no muestra nada por pantalla. El contenido se escribe en el fichero listado.txt, sobrescribiendo cualquier contenido anterior. También es posible **añadir** contenido al final del contenido actual del fichero:

```
$ ls -la >> listado.txt
```

Por otro lado, se puede usar el resultado de un comando como entrada de otro usando la funcionalidad de **tubería** o *pipe*. Por ejemplo, el siguiente comando muestra el contenido del fichero de configuración del servidor SSH, pero muestra solo las líneas no comentadas y ordenadas por orden alfabético:

```
$ cat sshd_config | grep -v "#" | sort | uniq
```

La redirección en sentido contrario también está soportada, es decir, se puede leer el contenido de un fichero y pasarlo por la entrada estándar a un comando. Por ejemplo, la siguiente línea lee el fichero `input.txt` y lo entrega por la entrada estándar a `cat`. El resultado es idéntico a ejecutar `cat input.txt` directamente, pero la redirección de entrada se puede usar con comandos que no estén preparados para leer ficheros directamente.

```
$ cat < listado.txt
```

Sudo

No todos los usuarios tienen los mismos privilegios en Linux: hay un usuario administrador inicial, `root`, capaz de realizar cualquier tarea. Otros usuarios solo pueden acceder a sus ficheros y ejecutar comandos no administrativos, a menos que reciban privilegios adicionales, bien directamente o a través de un grupo.

Para ejecutar operaciones administrativas, no hay más que iniciar sesión como `root`. Sin embargo, esto conlleva ciertos riesgos de seguridad, como que el usuario teclee un comando sintácticamente válido, pero que no hace lo que debe, borrando datos importantes por error (Van Vugt, 2015, pp. 179-196). Es habitual que la cuenta de `root` esté deshabilitada por defecto y que ni siquiera tenga contraseña, por lo que no se puede iniciar sesión con ella en muchas distribuciones.

La alternativa es usar el mecanismo de `Sudo`. La idea es que tareas de administración concretas pueden asignarse a usuarios específicos. Si un usuario necesita ejecutar una tarea administrativa como, por ejemplo, instalar un paquete nuevo con `apt-get install`, debe ejecutarlo con `Sudo`:

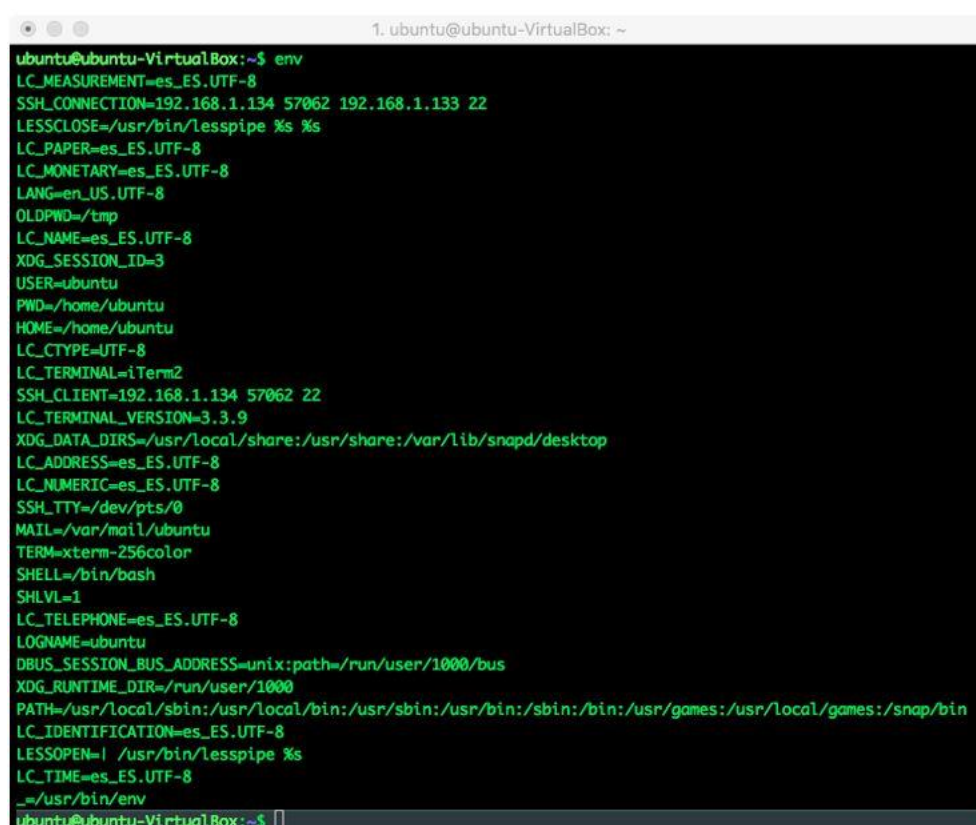
```
sudo apt-get install openssh-server
```

El usuario `root` podría ejecutar el mismo comando sin `Sudo`, pero es obligatorio para cualquier otro usuario. `Sudo` solicita la contraseña del usuario y rechaza el comando

si el usuario no tiene permiso para ejecutarlo. El fichero `sudoers` contiene los permisos asignados a los usuarios y grupos del sistema. Debe editarse con `visudo`: este comando abre el fichero `sudoers` con el editor del sistema (por ejemplo, `vi` o `nano`) y lo valida al guardarlo. De esta manera, se evita que un usuario cometa un error de sintaxis al guardarlo, rompiendo la funcionalidad de `Sudo`.

Variables de entorno

Cualquier proceso en Linux tiene asociado un conjunto de variables de entorno o *environment variables* (Dulaney, 2018). Las variables de entorno no son más que un nombre asociado a una **cadena de texto**. El comando `env` muestra las variables definidas en el entorno de la *shell*:



```
1. ubuntu@ubuntu-VirtualBox: ~  
ubuntu@ubuntu-VirtualBox:~$ env  
LC_MEASUREMENT=es_ES.UTF-8  
SSH_CONNECTION=192.168.1.134 57062 192.168.1.133 22  
LESSCLOSE=/usr/bin/lesspipe %s %s  
LC_PAPER=es_ES.UTF-8  
LC_MONETARY=es_ES.UTF-8  
LANG=en_US.UTF-8  
OLDPWD=/tmp  
LC_NAME=es_ES.UTF-8  
XDG_SESSION_ID=3  
USER=ubuntu  
PWD=/home/ubuntu  
HOME=/home/ubuntu  
LC_CTYPE=UTF-8  
LC_TERMINAL=iTerm2  
SSH_CLIENT=192.168.1.134 57062 22  
LC_TERMINAL_VERSION=3.3.9  
XDG_DATA_DIRS=/usr/local/share:/usr/share:/var/lib/flatpak/desktop  
LC_ADDRESS=es_ES.UTF-8  
LC_NUMERIC=es_ES.UTF-8  
SSH_TTY=/dev/pts/0  
MAIL=/var/mail/ubuntu  
TERM=xterm-256color  
SHELL=/bin/bash  
SHLVL=1  
LC_TELEPHONE=es_ES.UTF-8  
LOGNAME=ubuntu  
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus  
XDG_RUNTIME_DIR=/run/user/1000  
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin  
LC_IDENTIFICATION=es_ES.UTF-8  
LESSOPEN=| /usr/bin/lesspipe %s  
LC_TIME=es_ES.UTF-8  
_=/usr/bin/env  
ubuntu@ubuntu-VirtualBox:~$
```

Figura 3. Variables definidas en el entorno de la *shell*. Fuente: elaboración propia.

Estas variables afectan al comportamiento de diversos elementos. Por ejemplo, la variable `PATH` define las rutas en las que la *shell* buscará un ejecutable cuando se

invoca un comando que no es una expresión interna de la *shell*, mientras que editores como *vim* usan la variable *TERM* para decidir cómo mostrar los archivos en la consola. Se puede consultar el valor de estas variables con *echo*:

```
echo $PATH
```

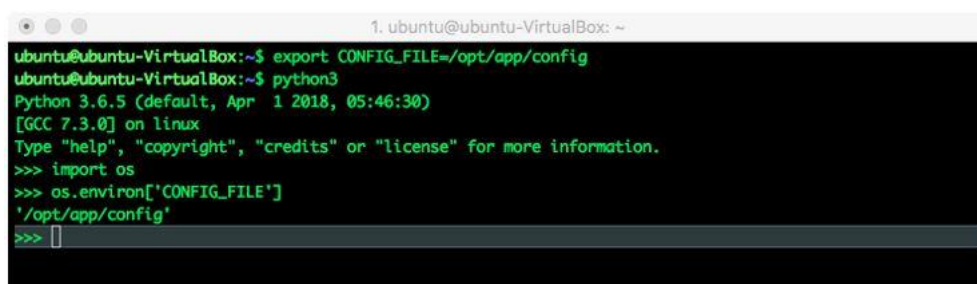
Se pueden crear nuevas variables, o cambiar valores de variables existentes, con *export*:

```
export CONFIG_FILE=/opt/app/config
```

Además de mostrarlas por pantalla, es posible usar el valor de una variable de entorno en un comando de *Bash*. La *shell* se encargará de recuperar el valor y sustituirlo.

```
$ mkdir /tmp/app
$ export CONFIG_FILE=/tmp/app/config
$ echo "DEBUG=true" > $CONFIG_FILE
$ cat $CONFIG_FILE
DEBUG=true
```

Al igual que la *shell* y los editores de línea de comandos, cualquier proceso puede acceder a sus variables de entorno. Por ejemplo, desde un proceso *Python*, el módulo *os* permite acceder a todas las variables (ver Figura 4).

A screenshot of a terminal window titled '1. ubuntu@ubuntu-VirtualBox: ~'. The terminal shows the following commands and output:

```
ubuntu@ubuntu-VirtualBox:~$ export CONFIG_FILE=/opt/app/config
ubuntu@ubuntu-VirtualBox:~$ python3
Python 3.6.5 (default, Apr 1 2018, 05:46:30)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> os.environ['CONFIG_FILE']
'/opt/app/config'
>>> 
```

Figura 4. Comandos *Python* para acceder a variables de entorno. Fuente: elaboración propia.

Las variables de entorno están muy extendidas como mecanismo de configuración, por ejemplo, en el caso de aplicaciones desplegadas como contenedores o en Kubernetes.

Heredocs

Los documentos *here*, o *heredocs*, son bloques de código que usan la redirección para alimentar un conjunto de líneas o comandos a un programa externo. Es habitual encontrarlos en *scripts* como método de insertar ficheros externos o de configuración en el propio fichero del *script*. De esta forma, es posible contener todo un *script* de automatización, incluyendo ficheros de configuración, en un único fichero.

El documento está limitado por dos cadenas de texto idénticas, la primera de ellas precedida de <<.

```
$ cat <<HEREDOC
Mensaje de varias líneas
que no podría imprimirse por pantalla
simplemente con "cat mensaje"
HEREDOC
```

Los documentos *here* se pueden combinar con tuberías y redirección.

```
$ cat > output.txt <<HEREDOC
Mensaje de varias líneas
que se guarda en un fichero
HEREDOC
```

La sustitución de variables de entorno también está soportada. Además, para hacer más legibles los *scripts*, es posible tabular el contenido del *heredoc*, pero redirigirlo a la entrada del comando sin los caracteres de tabulación, con la expresión <<-.

```
$ NAME=World
$ cat > mensaje.txt <<-HELLOALL
```

```
Hello $NAME
Heredocs rule
HELLOALL
$ cat mensaje.txt
Hello World
Heredocs rule
```

Expresiones lógicas

El **comando test** evalúa una condición lógica y devuelve un código de salida cero, si la condición es verdadera, o distinto de cero, si es falsa. Este comando tiene una forma alternativa, en la que el nombre test se sustituye por corchetes, [], y otra forma que usa corchetes dobles, [[]]; esta segunda forma no aplica expansión de ruta ni división de palabras. Ambas formas se usan habitualmente en *scripts*, ya que permiten expresar condiciones con una sintaxis similar a otros lenguajes de programación.

```
$ test 1 -eq 2; echo $?
1
$ [[ 1 -eq 1 ]]; echo $?
0
```

Este comando se puede usar en conjunción con la expresión `if` de Bash, para construir bloques condicionales como en cualquier otro lenguaje. Admite el uso de `else` y debe terminarse con `fi`.

```
$ if [[ 1 -eq 1 ]] ; then
    echo igual
else
    echo diferente
fi
```

El comando `test` soporta condiciones lógicas para comprobar ficheros, variables de entorno, comparaciones de enteros, etc.

	Expresiones de test
	Significado
-e fichero	El fichero existe
-d directorio	El directorio existe y es un directorio
-f fichero	El fichero existe y es un archivo normal
-h enlace	El enlace existe y es un archivo de tipo enlace simbólico
-N fichero	El fichero existe y se modificó tras la última lectura
-O fichero	El fichero existe y el propietario es el usuario actual
-p tubería	La tubería existe y es un archivo de tipo tubería
-r fichero	El fichero existe y se puede leer
-s fichero	El fichero existe y tiene un tamaño mayor que cero
-S socket	El socket existe y es un archivo de tipo <i>socket</i>
-w fichero	El fichero existe y se puede escribir
-x fichero	El fichero existe y es ejecutable
f1 -ef f2	Los ficheros f1 y f2 enlazan al mismo fichero
f1 -nt f2	El fichero f1 es más nuevo que f2
f1 -ot f2	El fichero f1 es más antiguo que f2
texto	El texto no es nulo
-n texto	El texto tiene una longitud mayor que cero
-z texto	El texto tiene una longitud cero
t1 == t2	Las cadenas de texto son idénticas. Usado dentro de <code>[[]]</code> , t2 puede contener <i>wildcards</i>
t1 != t2	Las cadenas de texto no son idénticas. Usado dentro de <code>[[]]</code> , t2 puede contener <i>wildcards</i>
t1 =~ reg	Las cadenas de texto t1 cumple con la expresión regular reg. Solo se puede usar con <code>[[]]</code>
t1 < t2	La cadena t1 precede a t2
n1 -eq n2	Los enteros son iguales

	Expresiones de test
	Significado
n1 -ge n2	n1 es mayor o igual que n2
n1 -gt n2	n1 es mayor que n2
n1 -le n2	n1 es menor o igual que n2
n1 -lt n2	n1 es menor que n2
n1 -ne n2	Los enteros son diferentes
! condicion	Negación de la condicion
c1 && c2	«And» lógico con cortocircuito. Solo se puede usar en [[]]
c1 c2	«Or» lógico con cortocircuito. Solo se puede usar en [[]]

Tabla 1. Expresiones lógicas de test. Fuente: elaboración propia.

Las condiciones con cadenas de texto pueden dar lugar a problemas. Por ejemplo, si se quiere comprobar si una variable tiene un valor dado, este ejemplo podría fallar si \$VAR no está definido:

```
$ [ $VAR1 == yes ]; echo $?
-bash: [: ==: unary operator expected
2
```

Esto ocurre porque la sustitución de \$VAR por su valor convierte la expresión en [== yes], es decir, una expresión binaria sin el primer operador. En estos casos, es habitual rodear ambas cadenas con comillas e incluso añadir un carácter adicional en las dos cadenas, para evitar que una de las dos se convierta en una cadena vacía.

```
$ [ "$VAR"x == "yes"x ]; echo $?
1
$ VAR=yes
$ [ "$VAR"x == "yes"x ]; echo $?
0
```

Expansión y metacaracteres

Bash acepta metacaracteres para completar nombre de ficheros y directorios. Por ejemplo, `ls -l file` mostrará por consola los detalles del fichero `file`, si existe, pero `ls -l file*` mostrará los detalles de todos los ficheros que empiecen con `file`.

```
$ ls -l file*
-rw-r--r-- 1 ubuntu ubuntu 0 May 9 10:11 file
-rw-r--r-- 1 ubuntu ubuntu 0 May 9 10:12 file.log
-rw-r--r-- 1 ubuntu ubuntu 0 May 9 10:12 file1
```

Metacaracteres			
	Significado	Ejemplo	Ficheros que aceptan el ejemplo
*	Cadena de cero o más caracteres	<code>ls -l file*</code>	<code>file</code> , <code>file1</code> , <code>filenew.log</code>
?	Cadena de exactamente un carácter	<code>ls -l file?.log</code>	<code>file0.log</code> , <code>fileX.log</code>
[abc] [a-z] [0-9]	Uno, cualquiera, de los caracteres indicados. El guion indica un rango (es decir, <code>a-z</code> aceptará cualquier letra en minúscula y <code>0-9</code> cualquier dígito)	<code>file[0-9].log</code>	<code>file0.log</code> , <code>file1.log</code>
[!abc]	Ninguno de los caracteres indicados	<code>file[0-9].log</code>	<code>fileX.log</code>
~	Directorio <i>home</i> del usuario actual	<code>ls ~/Desktop</code>	
~usuario	Directorio <i>home</i> del usuario	<code>ls ~ubuntu/Desktop</code>	
~+	Directorio de trabajo actual	<code>ls ~+</code>	
~-	Directorio de trabajo anterior	<code>cp file +-</code>	

Tabla 2. Metacaracteres. Fuente: elaboración propia.

Además de los metacaracteres, Bash soporta la expansión de cadenas de texto, se conoce como *brace expansion*. Al contrario de los metacaracteres, que se usan

exclusivamente para nombres de archivo, la **expansión está orientada a texto**. Las palabras generadas en una expansión no tienen por qué coincidir con ficheros.

```
$ echo Rule{A1,A2,A3,B1,C1,C2}
RuleA1 RuleA2 RuleA3 RuleB1 RuleC1 RuleC2
```

Además de la expansión del ejemplo anterior, Bash también soporta la expansión de rangos de números enteros.

```
$ echo Rule{1..5}
Rule1 Rule2 Rule3 Rule4 Rule5
```

Bucles

La expansión de Bash se puede usar, por ejemplo, en un bucle.

```
$ for r in i{1..3};
do
    echo interacion $r
done

interacion i1
interacion i2
interacion i3
```

La lista de elementos puede ser cualquier *array*, por ejemplo, la lista de parámetros de la llamada al *script*, `$@`.

```
$ cat file.sh
for p in $@; do echo $p; done
$ bash file.sh p1 param2 true
p1
param2
true
```

También es posible generar el *script* al vuelo a partir de otro comando.

```
$ ls -l file?
-rw-r--r-- 1 ubuntu ubuntu 0 May 9 11:07 file1
-rw-r--r-- 1 ubuntu ubuntu 0 May 9 11:07 file2
-rw-r--r-- 1 ubuntu ubuntu 0 May 9 11:07 filex
$ for f in $(ls file?); do echo Fichero $f; done
Fichero file1
Fichero file2
Fichero filex
```

Finalmente, también se puede iterar sobre un rango de enteros, tal como en otros lenguajes.

```
$ for ((i=1; i <= 5; i += 1))
do
    if [[ -e file$i ]]; then
        echo file$i existe;
    else
        echo file$i no existe
    fi
done

file1 existe
file2 existe
file3 no existe
file4 no existe
file5 no existe
```

3.4. Scripts de Bash

Un *script* de Bash es un fichero de texto con una secuencia de comandos de *shell* (Van Vugt, 2015, pp. 319-351). No tiene más requisitos, así que un fichero con este contenido sería un *script* válido:

```
echo "Hello World"
pwd
```

Este *script* se podría ejecutar invocándolo de la siguiente manera:

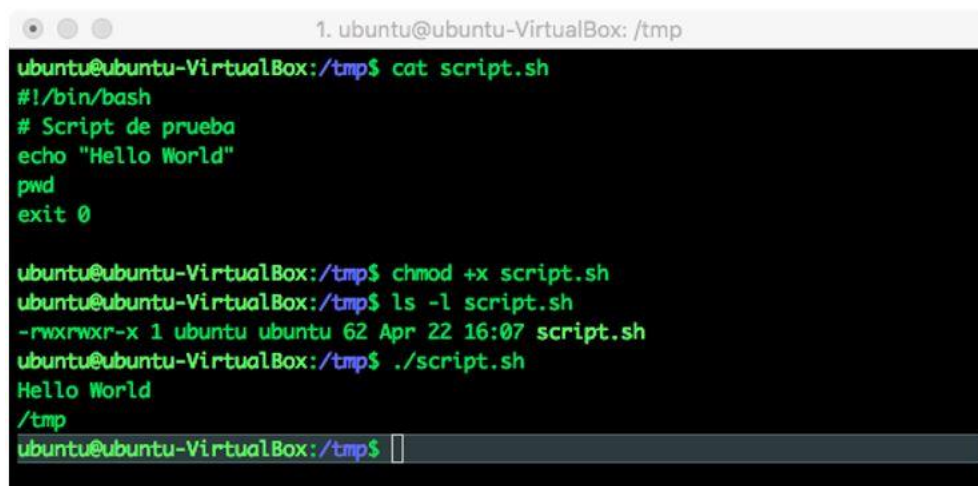
```
$ bash script.sh
Hello World
/etc/apache2
```

Sin embargo, conviene construir los *scripts* con las siguientes características:

- ▶ Deben empezar con un *shebang*, `#!/`, en la primera línea, indicando la ruta al ejecutable de la *shell*. Esta línea es un comentario especial que permite ejecutar un *script* como un ejecutable binario más. Para un *script* de Bash, la línea sería `#!/bin/bash`, mientras que un *script* de Python podría usar `#!/usr/bin/python3` - v.
- ▶ Los *scripts* pueden incluir comentarios para facilitar la lectura del código por otros administradores.
- ▶ El uso de `exit` indica explícitamente cuando termina el *script*, tanto de manera correcta con un código de salida `0` como si hay algún error.
- ▶ Los *scripts* pueden ser considerados como ejecutables, de manera que se pueden invocar en la línea de comandos como si fueran un archivo compilado más. Esto se consigue habilitando el *flag* ejecutable de archivo con, por ejemplo, `chmod +x script.sh`.

Estas ideas se pueden aplicar al *script* anterior para obtener el siguiente:

```
#!/bin/bash
# Script de prueba
echo "Hello World"
pwd
exit 0
```



```
1. ubuntu@ubuntu-VirtualBox: /tmp
ubuntu@ubuntu-VirtualBox:/tmp$ cat script.sh
#!/bin/bash
# Script de prueba
echo "Hello World"
pwd
exit 0

ubuntu@ubuntu-VirtualBox:/tmp$ chmod +x script.sh
ubuntu@ubuntu-VirtualBox:/tmp$ ls -l script.sh
-rwxrwxr-x 1 ubuntu ubuntu 62 Apr 22 16:07 script.sh
ubuntu@ubuntu-VirtualBox:/tmp$ ./script.sh
Hello World
/tmp
ubuntu@ubuntu-VirtualBox:/tmp$
```

Figura 5. Ejemplo de *script* en Bash. Fuente: elaboración propia.

Hasta ahora, se ha mostrado cómo ejecutar un *script* de dos maneras: como argumento del comando `bash` y convirtiendo el *script* en ejecutable e invocándolo directamente. Ambas opciones funcionan de manera similar: la *shell* desde la que se invoca arranca otra *shell* como un nuevo proceso y es esta la que se encarga de ejecutar los comandos del *script*.

Hay una tercera opción, ligeramente diferente: ejecutar el *script* en el mismo proceso de la *shell* actual. Este caso puede ser útil si el *script* cambia valores de variables de entorno que son necesarias en la *shell* actual (por ejemplo, para configurar la *shell* de alguna manera concreta). No obstante, puede tener efectos contraproducentes: si el *script* termina proactivamente con un comando `exit`, la *shell* actual también terminará. El comando para ejecutar los *scripts* en la propia *shell* es `source` o `.` (un punto):

```
$ source script.sh
$ . script.sh
```

Scripts especiales

La *shell* ejecuta ciertos *scripts* durante el arranque. Concretamente, intentará leer los ficheros en este orden:

- `/etc/profile`. Se ejecuta durante el inicio de sesión.
- `~/.bash_profile`, o bien `~/.bash_login`, o bien `~/.profile`, el primero que encuentre. Se ejecuta durante el inicio de sesión.
- `~/.bashrc`. Se ejecuta en una *shell* sin inicio de sesión.

Estos *scripts* suelen contener opciones de configuración de las sesiones de usuario: definición de alias, el texto del *prompt* (el texto que se muestra en cada nueva línea antes de los comandos que introduce el usuario) y cualquier otra personalización que cada usuario quiera tener disponible.

3.5. Referencias bibliográficas

Dulaney, E. (2018). *Linux All-in-one for Dummies* (6.ª ed., cap. II, pp. 203-259). John Wiley & Sons.

Robbins, A. (2010). *Bash Pocket Reference* (apartado «History», pp. 2-3). O'Reilly Media.

Van Vugt, S. (2015). *Beginning the Linux Command Line* (2.ª ed.). Apress.

Tutorial de vim

Openvim. (s. f.). *Introduction*. <https://www.openvim.com>

Aprender a manejar un editor en modo texto es fundamental para cualquier administrador. El tutorial de openvim.com empieza desde lo más básico y en unas pocas lecciones es posible aprender suficiente para manejarse con soltura. También es posible arrancar **vimtutor** en cualquier sistema con **vim**. Este tutorial interactivo abre un editor **vim** y muestra instrucciones al usuario para ir aprendiendo a manejarlo progresivamente.

Bash Reference Manual

Ramey, C. y Fox, B. (2020). *Bash Reference Manual*. Free Software Foundation. <https://www.gnu.org/software/bash/manual/bash.pdf>

El manual oficial de Bash es un compendio de todas las funcionalidades de la *Shell*, sin entrar en detalles de herramientas GNU concretas. Aunque una lectura completa puede ser muy árida, conviene echar un vistazo a los primeros capítulos para poder usarla como obra de consulta cuando surjan dudas.

1. ¿Cuál de los siguientes ejemplos hace uso de una tubería?
 - A. `grep POST /var/log/nginx/access.log | head -100 | less.`
 - B. `ping -n 5 192.168.1.1 > pings.txt.`
 - C. Todas las anteriores.
 - D. Ninguna de las anteriores.

2. ¿Cuál de los siguientes comandos lista los detalles de los archivos de un directorio, incluidos los archivos ocultos?
 - A. `ls -l.`
 - B. `ls -la.`
 - C. `ls -l > head -a.`
 - D. `ls -h.`

3. ¿Cómo crearía el usuario *user1* una carpeta nueva *app* en la ruta */home/user1*?
 - A. `mkdir ~/app.`
 - B. `mkdir /home/user1/app.`
 - C. `cd ~ && mkdir app.`
 - D. Todas las anteriores.

4. ¿Es necesaria la línea `#!/bin/bash` en cualquier *script*?
 - A. Sí, en todos.
 - B. Solo en los de Bash.
 - C. Es útil, pero no imprescindible.
 - D. Es mejor no incluirlo.

5. ¿Cuál de los siguientes comandos muestra el valor de la variable de entorno `CONFIG_FILE`?
- A. `echo $CONFIG_FILE.`
 - B. `cat $CONFIG_FILE.`
 - C. `head $CONFIG_FILE.`
 - D. `pwd $CONFIG_FILE.`
6. ¿Cuál de los siguientes comandos mostrará un error si lo ejecuta un usuario normal con permisos de Sudo para cualquier comando?
- A. `sudo apt-get install vim.`
 - B. `apt-get install vim.`
 - C. `vim /etc/hosts.`
 - D. `sudo vim /etc/hosts.`
7. ¿Cómo se podrían extraer las primeras cincuenta líneas de `file.log` a un segundo fichero, `out.log`, ordenadas alfabéticamente?
- A. `head -50 file.log | sort > out.log.`
 - B. `cat file.log | head | sort > out.log.`
 - C. `sort file.log | head -50 > out.log.`
 - D. Ninguna de las anteriores.
8. ¿En qué fichero se pueden definir opciones de personalización para las nuevas sesiones de un usuario?
- A. `/etc/profile.`
 - B. `~/.bash_profile.`
 - C. `~/.profile.`
 - D. Todos los anteriores.

9. ¿Cómo puede un usuario duplicar un fichero `fileA.conf` y editar la copia?

- A. `cp fileA.conf fileB.conf & vim fileB.conf.`
- B. `cp fileA.conf fileB.conf && vim fileB.conf.`
- C. `mv fileA.conf fileB.conf && vim fileB.conf.`
- D. `vim fileA.conf && cp fileA.conf fileB.conf.`

10. ¿Por qué es necesario el comando `sudo`?

- A. Para permitir que un usuario sin privilegios pueda ejecutar solo ciertos comandos administrativos.
- B. Para evitar trabajar continuamente con la cuenta `root` y limitar riesgos de comandos erróneos.
- C. Para evitar compartir la contraseña de `root` con demasiados administradores.
- D. Todas las anteriores.