0.

Objetivos del aprendizaje

- Saber cuándo utilizar *scripts* para resolver tareas de programación, identificando las ventajas e inconvenientes de los lenguajes de *scripting* y su aplicabilidad en administración de sistemas.
- Conocer los distintos intérpretes de órdenes en GNU/Linux y justificar el uso de `bash` para la programación de *scripts* de administración de sistemas.
- Escribir *scripts* de `bash` de la mejor forma posible y ejecutarlos correctamente.
- Declarar y utilizar correctamente variables en `bash`.
- Conocer la diferencia entre el uso de comillas dobles y comillas simples en *scripts* de `bash`.
- Diferenciar las variables locales de un *script* de las variables de entorno.
- Utilizar correctamente el comando `export`.
- Conocer las variables de entorno más habituales en `bash`.
- Utilizar las variables intrínsecas de `bash` para interactuar de forma más efectiva con la terminal de comandos.
- Utilizar correctamente el comando `exit`.
- Utilizar correctamente el comando `read`.
- Aplicar correctamente la substitución de comandos en `bash`.
- Conocer y utilizar distintas alternativas para realizar operaciones aritméticas en `bash`.
- Utilizar estructuras condicionales `if`.
- Comparar correctamente cadenas y números, chequear el estado de ficheros y aplicar operadores lógicos.
- Utilizar estructuras condicionales `case`.
- Utilizar estructuras iterativas `for`.
- Utilizar *arrays* en `bash`.
- Utilizar funciones en `bash`.
- Aplicar diversas opciones para la depuración de *scripts* en `bash`.
- Redirigir la entrada y la salida de comandos desde y hacia ficheros.
- Interconectar distintos comandos mediante el uso de tuberías.
- Utilizar correctamente el comando `tee` para la redirección de salida.

- Conocer los *here documents* y utilizarlos para hacer *scripts* más legibles.
- Utilizar correctamente los siguientes comandos adicionales: `cat`, `head`, `tail`, `wc`, `find`, `basename`, `dirname`, `stat` y `tr`.
- Aplicar el mecanismo de expansión de llaves en la creación de *arrays*.

Contenidos

Evaluación

- Entrega de prácticas.
- Pruebas de validación de prácticas.

1. Introducción

1.1. Justificación

¿Línea de comandos?

- ¿Para qué necesito aprender a utilizar la línea de comandos?
- Historia real[1]:
  - Unidad compartida por cuatro servidores que está llenándose → impedía a la gente trabajar.
  - El sistema no soportaba cuotas.
  - Un ingeniero escribe un programa en C++ que navega por los archivos de todos los usuarios, calcula cuanto espacio está ocupando cada uno y genera un informe.
  - Utilizando un entorno GNU/Linux y su *shell*:

```
1 du -s * | sort -nr > $HOME/user_space_report.txt
```

[1] `http://www.linuxcommand.org/lc3_learning_the_shell.php`

bash

- Las interfaces gráficas de usuario (GUI) son buenas para muchas cosas, pero no para todas, especialmente las más repetitivas.
- ¿Qué es la *shell*?.
  - Programa que recoge comandos del ordenador y se los proporciona al SO para que los ejecute.
  - Antiguamente, era la única interfaz disponible para interactuar con SO tipo Unix.
- En casi todos los sistemas GNU/Linux, el programa que actúa como *shell* es `bash`.
  - *B*ourne *A*gain *SH*ell → versión mejorada del `sh` original de Unix.
  - Escrito por Steve Bourne.

bash

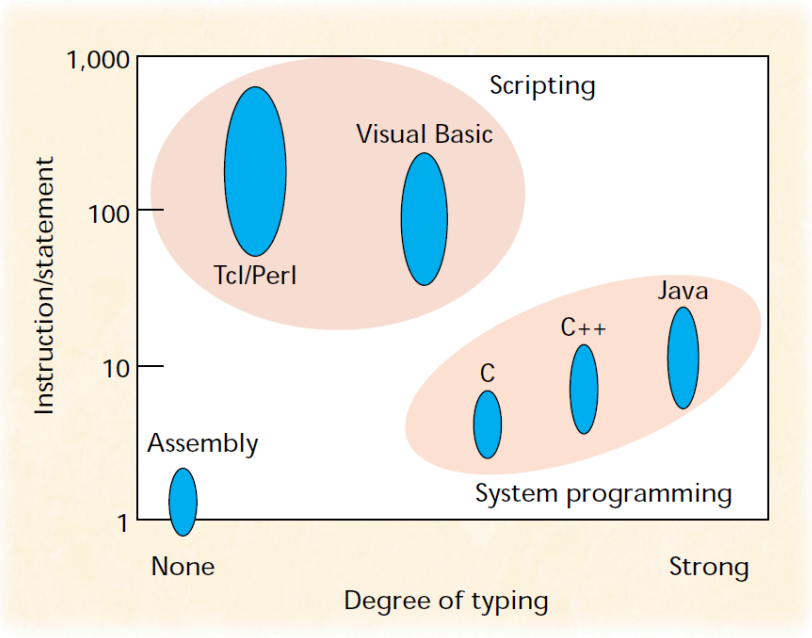
- Alternativas a bash:
  - Bourne shell (`sh`), C shell (`csh`), Korn shell (`ksh`), TC shell (`tcsh`)...
- `bash` incorpora las prestaciones más útiles de ksh y csh.
  - Es conforme con el estándar IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools.
  - Ofrece mejoras funcionales sobre la shell desde el punto de vista de programación y de su uso interactivo.
- ¿Qué es una terminal?
  - Es un programa que emula la terminal de un computador, iniciando una sesión de *shell* interactiva.
  - `gnome-terminal`, `konsole`, `xterm`, `rxvt`, `kvt`, `nxterm` o `eterm`.

1.2. ¿Programación o scripting?

¿Programación o scripting?

- `bash` no es únicamente una excelente *shell* por línea de comandos...
- También es un *lenguaje de scripting* en sí mismo.
- El *shell scripting* sirve para automatizar multitud de tareas que, de otra forma, requerirían múltiples comandos introducidos de forma manual.
- Lenguaje de programación (LP) vs. *scripting*:
  - Los LPs son, en general, más potentes y mucho más rápidos que los lenguajes de *scripting*.

- Los LPs comienzan desde el código fuente, que se compila para crear los ejecutables (lo que no permite que los programas sean fácilmente portables entre diferentes SOs).



(OUSTERHOUT, J., "Scripting: Higher-Level Programming for the 21st Century", IEEE Computer, Vol. 31, No. 3, March 1998, pp. 23-30.)

¿Programación o scripting?

- Un lenguaje de *scripting* (LS) también comienza por el código fuente, pero no se compila en un ejecutable.
- En su lugar, un intérprete lee las instrucciones del fichero fuente y las ejecuta secuencialmente.
  - Programas interpretados → más lentos que los compilados.
  - "Tipado" débil (*¿ventaja o desventaja?*).
- Ventajas:
  - En general, una línea de LS "cunde" más que una de un LP.
  - El fichero de código es fácilmente portable a cualquier SO.
  - Todo lo que yo pueda hacer con mi *shell*, lo puedo automatizar con un *script*.
  - Nivel de abstracción muy superior en cuanto a operaciones con ficheros, procesos...

1.3. Primeros programas

Primer programa `bash`: `holaMundo.sh`

- Abrir un editor de textos:

```
pedroa@pagutierrezLaptop:~$ gedit holaMundo.sh &
```

- Escribimos el código:

```
#!/bin/bash
echo "Hola Mundo"
```

- Hacemos que el fichero de texto sea ejecutable:

```
pedroa@pagutierrezLaptop:~$ chmod u+x holaMundo.sh
pedroa@pagutierrezLaptop:~$ ls -l holaMundo.sh
-rwxr--r-- 1 pedroa pedroa 30 feb 15 19:52 holaMundo.sh
```

Primer programa `bash`

```
#!/bin/bash
echo "Hola Mundo"
```

- El carácter `#!` al principio del script se denomina *SheBang/HashBang* y es un comentario para el intérprete *shell*.
- Es utilizado por el cargador de programas del SO (el código que se ejecuta cuando una orden se lanza).
- Le indica *qué intérprete de comandos* se debe utilizar para este fichero, en el caso anterior, `/bin/bash`.

Primer programa `bash`

- Para ejecutar el programa:

```
pedroa@pagutierrezLaptop:~$ holaMundo.sh
bash: holaMundo.sh: no se encontró la orden
```

- El directorio `$HOME`, donde está el programa, no está dentro del *path* por defecto:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

- Por tanto, *¡hay que especificar la ruta completa!*:

```
pedroa@pagutierrezLaptop:~$ /home/pagutierrez/holaMundo.sh
Hola Mundo
pedroa@pagutierrezLaptop:~$ ~/holaMundo.sh
Hola Mundo
pedroa@pagutierrezLaptop:~$ ./holaMundo.sh
Hola Mundo
```

Primer programa `bash`

- Orden `echo`:
  - Imprime (manda al `stdout`) el contenido de lo que se le pasa como argumento.
  - Es un comando del sistema (un ejecutable), no una palabra reservada del lenguaje de programación.
  - Se puede utilizar el `man` para ver sus opciones.

```
pedroa@pagutierrezLaptop:~/tmp$ echo "Imprimo una línea con salto de línea"
Imprimo una línea con salto de línea
pedroa@pagutierrezLaptop:~/tmp$ echo -n "Imprimo una línea sin salto de línea"
Imprimo una línea sin salto de líneapedroa@pagutierrezLaptop:~/tmp$ which echo
/bin/echo
pedroa@pagutierrezLaptop:~$ echo "ho\nla"
ho\nla
pedroa@pagutierrezLaptop:~$ echo -e "ho\nla"
ho
la
```

Segundo programa `bash`: ***`papelera.sh`***

- Especificar los comandos para:
  - Crear una subcarpeta `papelera`.
  - Copiar todos los ficheros que hay en la carpeta `~` a la subcarpeta `papelera`.
  - Posteriormente, borrarlos.

Segundo programa `bash`

- Comandos:

```
pedroa@pagutierrezLaptop:~$ mkdir papelera
pedroa@pagutierrezLaptop:~$ cp * papelera
cp: se omite el directorio «xxx»
...
pedroa@pagutierrezLaptop:~$ rm -Rf papelera/
```

- El mensaje que nos aparece es un *warning*. Por defecto, el comando `cp` no copia carpetas, las omite y copia únicamente los ficheros.
- En lugar de tener que escribir todo esto de forma interactiva en la *shell*, escribimos un *script*.

2. Variables

2.1. Concepto y declaración

Variables: concepto

- Como en cualquier otro LP, se pueden utilizar *variables*.

- Todos los valores son almacenados como tipo cadena de texto (*"tipado" débil*).
- ¿No puedo operar?
  - Operadores matemáticos que convierten las variables en número para el cálculo.
- Como no hay tipos, no es necesario declarar variables, sino que al asignarles un valor, es cuando se crean.

Variables: primer ejemplo

- Primer ejemplo: *holaMundoVariable.sh*

```
1 #!/bin/bash
2 STR="Hola Mundo!"
3 echo $STR
```

- Asignación: `VARIABLE="valor"`
- Resolver una variable, es decir, sustituir la variable por su valor: *$*`VARIABLE`
- Probar a poner espacios antes y después del "="
  - ¿Qué sucede?

Variables: precaución

- El lenguaje de programación de la *shell* no hace un *casting* (conversión) de los tipos de las variables.
- Una misma variable puede contener datos numéricos o de texto:

```
1 contador=0
2 contador=Domingo
```

- La conmutación del tipo de una variable puede conllevar a confusión.
- Buena práctica: asociar siempre el mismo tipo de dato a una variable en el contexto de un mismo *script*.

Variables: precaución

- Carácter de escape:
  - Un carácter de escape es un carácter que permite que los símbolos especiales del LP no se interpreten y se utilice su valor literal.
  - Por ejemplo, en C:

```
1 "Esta cadena contiene el carácter \" en su interior"
```

  - En `bash`:

```
1 pedroa@pagutierrezLaptop:~$ ls \*
```

2.2. Comillas simples y dobles

Comillas simples y dobles

- Cuando el valor de la variable contenga espacios en blanco o caracteres especiales, se deberá encerrar entre comillas simples o dobles.
- Las comillas simples servirían para que la cadena se represente tal cual → como si cada carácter de la cadena tuviese un "\".
- Si son dobles, se permitirá especificar variables internas que se resolverán:

```
pedroa@pagutierrezLaptop:~$ var="cadena de prueba"
pedroa@pagutierrezLaptop:~$ nuevavar="Valor de var es $var"
pedroa@pagutierrezLaptop:~$ echo $nuevavar
Valor de var es cadena de prueba
```

- ¿Qué hubiera pasado en este caso?

```
pedroa@pagutierrezLaptop:~$ var="cadena de prueba"
pedroa@pagutierrezLaptop:~$ nuevavar='Valor de var es $var'
pedroa@pagutierrezLaptop:~$ echo $nuevavar
```

Comillas simples y dobles

- Hacer un *script* que muestre por pantalla (`comillas.sh`):

```
Valor de 'var' es "cadena de prueba"
```

2.3. Variables locales y de entorno

Variables locales y de entorno

- Hay dos tipos de variables:
  - Variables locales.
  - Variables de entorno:
    - Establecidas por el SO, especifican su configuración.
    - Se pueden listar utilizando el comando `env`.

```
pedroa@pagutierrezLaptop:~$ echo $SHELL
/bin/bash
pedroa@pagutierrezLaptop:~$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

    - Se definen en *scripts* del sistema que se ejecutan al iniciar el proceso `bash`. `/etc/profile`, `/etc/profile.d/` y `~/.bash_profile`.
    - Al salir, se ejecutan los comandos en `~/.bash_logout`.

Comando export

- El comando export establece una variable en el entorno, de forma que sea accesible por los procesos hijos.

```
pedroa@pagutierrezLaptop:~$ x=hola
pedroa@pagutierrezLaptop:~$ bash    # Ejecutar una shell hija
pedroa@pagutierrezLaptop:~$ echo $x # No aparece nada

pedroa@pagutierrezLaptop:~$ exit    # Volver al padre
exit
pedroa@pagutierrezLaptop:~$ export x # También se podría export x=hola
pedroa@pagutierrezLaptop:~$ bash
pedroa@pagutierrezLaptop:~$ echo $x # Ahora si
hola
```

Comando export

- Si el proceso hijo modifica la variable, no se modifica la del padre:

```
pedroa@pagutierrezLaptop:~$ x=hola
pedroa@pagutierrezLaptop:~$ export x
pedroa@pagutierrezLaptop:~$ bash
pedroa@pagutierrezLaptop:~$ x=adios
pedroa@pagutierrezLaptop:~$ exit
pedroa@pagutierrezLaptop:~$ echo $x
hola
```

Algunas variables importantes

- "*Home, sweet* `$HOME`":
  - `$HOME`: directorio personal del usuario, donde debería almacenar todos sus archivos.
  - `$HOME` ≡ `~` ≡ `/home/usuario`
  - Argumento por defecto del comando `cd`.
- `$PATH`: carpetas que contienen los comandos.
  - Es una lista de directorios separados por ":".
  - Normalmente, ejecutamos *scripts* así:

```
$ ./trash.sh
```

  - Pero si antes hemos establecido `PATH=$PATH:~`, podríamos ejecutar los scripts que haya en el `$HOME` de la siguiente forma:

```
$ trash.sh
```

- `$LOGNAME` o `$USER`: ambas contienen el nombre de usuario.

Algunas variables importantes

- Si creamos una carpeta:

```
$ mkdir ~/bin
```

- Y modificamos el `.bash_profile`:

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

- El directorio `/home/usuario/bin` será incluido en la búsqueda de programas binarios a ejecutar.
- ¿Qué hubiera pasado si no hubiese incluido el `export`?

Más variables importantes

- `$HOSTNAME`: contiene el nombre de la máquina.
- `$MACHTYPE`: arquitectura.
- `$PS1`: cadena que codifica la secuencia de caracteres monstrados antes del *prompt*
  - `\t`: hora.
  - `\d`: fecha.
  - `\w`: directorio actual.
  - `\h`: nombre de la máquina.
  - `\W`: última parte del directorio actual.
  - `\u`: nombre de usuario.
- `$UID`: contiene el id del usuario que no puede ser modificado.
- `$SHLVL`: contiene el nivel de anidamiento de la *shell*.
- `$RANDOM`: número aleatorio.
- `$SECONDS`: número de segundos que `bash` lleva en marcha.

Más variables importantes

- Ejercicio: haz un *script* que muestre la siguiente información:

```
pedroa@pagutierrezLaptop:~/tmp$ ./informacion.sh
Bienvenido pedroa!, tu identificador es 1000.
Esta es la shell número 1, que lleva 107 arrancada.
La arquitectura de esta máquina es x86_64-pc-linux-gnu y el cliente de terminal es
    xterm
```

- Ejercicio: personaliza el *prompt* para que adquiera este aspecto:

```
pagutierrezLaptop:~(hola, son las 07:32:30)&
```

Variables intrínsecas

- `$#`: número de argumentos de la línea de comandos (`argc`).
- `$n`: *n*-ésimo argumento de la línea de comandos (`argv[n]`), si n es mayor que 9 utilizar `${n}`.
- `$*`: todos los argumentos de la línea de comandos (como una sola cadena).
- `$@`: todos los argumentos de la línea de comandos (como un *array*).
- `$!`: pid del último proceso que se lanzó con &.
- `$-`: opciones suministradas a la *shell*.
- `$?`: valor de salida la última orden ejecutada (ver `exit`).

Variables intrínsecas

- Ejercicio: escribir un *script* (`parametros.sh`) que imprima el número de argumentos que se le han pasado por línea de comandos, el nombre del *script*, el primer argumento, el segundo argumento, la lista de argumentos como una cadena, y la lista de argumentos como un *array*.

```
pedroa@pagutierrezLaptop:~/tmp$ ./parametros.sh estudiante1 estudiante2
2; ./parametros.sh; estudiante1; estudiante2; estudiante1 estudiante2; estudiante1
    estudiante2
```

Variables intrínsecas: navegar por comandos anteriores

- `!$`: último argumento del último comando ejecutado.
- `!:n`: *n*-ésimo argumento del último comando ejecutado.

```
pedroa@pedroa-laptop ~ $ echo argumentos 2 3
argumentos 2 3
pedroa@pedroa-laptop ~ $ echo !$
echo 3
3
pedroa@pedroa-laptop ~ $ echo !:0
echo echo
echo
```

- Comandos interactivos de consola:
  - Buscar un comando en el historial de la consola: `Ctrl+R` (en lugar de pulsar ↑ *n* veces).
  - Navegar por los argumentos del último comando: `Alt+.`.

Comando `exit`

- Se puede utilizar para finalizar la ejecución de un *script* y devolver un valor de salida $(0 - 255)$ que estará disponible para el proceso padre que invocó el *script*.
  - Si lo llamamos sin parámetros, se utilizará el valor de salida del último comando ejecutado (equivalente a `exit $?`).

```
pedroa@pagutierrezLaptop:~$ echo $?
0
pedroa@pagutierrezLaptop:~$ bash
pedroa@pagutierrezLaptop:~$ exit 1
exit
pedroa@pagutierrezLaptop:~$ echo $?
1
pedroa@pagutierrezLaptop:~$ echo $?
0
pedroa@pagutierrezLaptop:~$ echo $?
0
pedroa@pagutierrezLaptop:~$ bash
pedroa@pagutierrezLaptop:~$ exit
exit
pedroa@pagutierrezLaptop:~$ echo $?
0
```

2.4. Dando valor a variables

Comando `read`

- El comando `read` permite leer un comando del usuario por teclado y almacenarlo en una variable.
  - Ejemplo:

```
#!/bin/bash
echo -n "Introduzca nombre de fichero a borrar: "
read fichero
rm -i $fichero # La opción -i pide confirmación
echo "Fichero $fichero borrado!"
```

Comando `read`

- Opciones del comando `read`:
  - `read -s`: no hace *echo* de la entrada.
  - `read -n`*N*: solo acepta *N* caracteres de entrada.
  - `read -p "`*mensaje*`"`: muestra el mensaje *mensaje* al pedir la información al usuario.
  - `read -t`*T*: acepta la entrada durante un tiempo máximo de *T* segundos.

```
pedroa@pagutierrezLaptop:~$ read -s -t5 -n1 -p "si (S) o no (N)?" respuesta
si (S) o no (N)?S
pedroa@pagutierrezLaptop:~$ echo $respuesta
S
```

Substitución de comandos (*IMPORTANTE*)

- El acento hacia atrás (`) es distinto que la comilla simple (').
- `` `comando` `` se utiliza para sustitución de comandos, es decir, se ejecuta `comando`, se recoge lo que devuelve por consola y se sustituye por el comando:

```
pedroa@pagutierrezLaptop:~/tmp$ LISTA=`ls`
pedroa@pagutierrezLaptop:~/tmp$ echo $LISTA
d1 ecj-read-only simbolico tar
```

- También se puede utilizar `$(comando)`:

```
pedroa@pagutierrezLaptop:~/tmp$ LISTA=$(ls)
pedroa@pagutierrezLaptop:~/tmp$ echo $LISTA
d1 ecj-read-only simbolico tar
pedroa@pagutierrezLaptop:~/tmp$ ls $(pwd)
d1  ecj-read-only  simbolico  tar
pedroa@pagutierrezLaptop:~/tmp$ ls $(echo /bin)
```

- Antes de ejecutar una instrucción, `bash` sustituye las variables de la línea (empiezan por $) y los comandos ($( ) o ` `).

2.5. Operadores aritméticos

Operadores aritméticos

- Bash permite realizar operaciones aritméticas

Operador	Significado
+	Suma
−	Resta
*	Multiplicación
/	División
**	Exponenciación
%	Módulo

```
pedroa@pagutierrezLaptop:~/tmp$ a=(5+2)*3
pedroa@pagutierrezLaptop:~/tmp$ echo $a
(5+2)*3
pedroa@pagutierrezLaptop:~/tmp$ b=2**3
pedroa@pagutierrezLaptop:~/tmp$ echo $a+$b
(5+2)*3+2**3
```

Operadores aritméticos

- Hay que utilizar la instrucción `let`:

```
pedroa@pagutierrezLaptop:~/tmp$ let X=10+2*7
pedroa@pagutierrezLaptop:~/tmp$ echo $X
24
pedroa@pagutierrezLaptop:~/tmp$ let Y=X+2*4
pedroa@pagutierrezLaptop:~/tmp$ echo $Y
32
```

- Alternativamente, las expresiones aritméticas también se pueden evaluar con `$[expresión]` o `$((expresión))`:

```
pedroa@pagutierrezLaptop:~/tmp$ echo "$((123+20))"
143
pedroa@pagutierrezLaptop:~/tmp$ VALOR=$[123+20]
pedroa@pagutierrezLaptop:~/tmp$ echo "$[123*$VALOR]"
17589
pedroa@pagutierrezLaptop:~/tmp$ echo $[123*$VALOR]
17589
pedroa@pagutierrezLaptop:~/tmp$ echo '$[123*$VALOR]'
$[123*$VALOR]
```

Operadores aritméticos

- Ejercicio:
  - Implementar un *script* (`operaciones.sh`) que lea dos números y aplique todas las operaciones posibles sobre los mismos.

```
pedroa@pagutierrezLaptop:~/tmp$ ./operaciones.sh
Introduzca un primer número: 2
Introduzca un segundo número : 9
Suma: 11
Resta: -7
Multiplicación: 18
División: 0
Módulo: 2
```

3. Estructuras de control

3.1. Condicionales if

Condicionales `if`

- La forma más básica es:

```
if [ expresión ];
then
  instrucciones
elif [ expresión ];
then
  instrucciones
else
  instrucciones
fi
```

- Las secciones `elif` (`else if`) y `else` son opcionales.
- *IMPORTANTE*: espacios antes y después `[` y `]`.

Expresiones lógicas

- Expresiones lógicas pueden ser:
  - Comparación de cadenas.
  - Comparación de números.
  - Chequeo de ficheros.
  - Combinación de los anteriores mediante operadores lógicos.
- Las expresiones se encierran con corchetes `[ expresion ]`.
- En realidad, se está llamando al programa `/usr/bin/[`.

```
1  pedroa@pagutierrezLaptop:~$ /usr/bin/[ 3 = 4 ]
2  pedroa@pagutierrezLaptop:~$ echo $?
3  1
4  pedroa@pagutierrezLaptop:~$ /usr/bin/[ 4 = 4 ]
5  pedroa@pagutierrezLaptop:~$ echo $?
6  0
7  pedroa@pagutierrezLaptop:~$ /usr/bin/[ 'asa' == 'asa' ]
8  pedroa@pagutierrezLaptop:~$ echo $?
9  0
10 pedroa@pagutierrezLaptop:~$ /usr/bin/[ 'asa' == 'asaa' ]
11 pedroa@pagutierrezLaptop:~$ echo $?
12 1
```

Comparación de cadenas

Operador	Significado
`s1 == s2`	Igual a
`s1 != s2`	Distinto a
`-n s`	Longitud mayor que cero
`-z s`	Longitud igual a cero

- Ejemplos:
  - `[ s1 == s2 ]`: `true` si `s1` es igual a `s2`, sino `false`.
  - `[ s1 != s2 ]`: `true` si `s1` no es igual a `s2`, sino `false`.
  - `[ s1 ]`: `true` si `s1` no está vacía, sino `false`.
  - `[ -n s1 ]`: `true` si `s1` tiene longitud > 0, sino `false`.
  - `[ -z s2 ]`: `true` si `s2` tiene longitud 0, sino `false`.
- Los dobles corchetes permiten usar expresiones regulares:
  - `[[ s1 == s2* ]]`: `true` si `s1` empieza por `s2`, sino `false`.

Comparación de cadenas

- Implementar un *script* que pregunte el nombre de usuario y devuelva un error si el nombre no es correcto:

```
pedroa@pagutierrezLaptop:~/tmp$ ./saludaUsuario.sh
Introduzca su nombre de usuario: pedroa
Bienvenido "pedroa"
pedroa@pagutierrezLaptop:~/tmp$ ./saludaUsuario.sh
Introduzca su nombre de usuario: pagutierrez
Eso es mentira!
```

Comparación de números

Operador	Significado
`n1 -lt n2`	Menor que
`n1 -gt n2`	Mayor que
`n1 -le n2`	Menor o igual que
`n1 -ge n2`	Mayor o igual que
`n1 -eq n2`	Igual
`n1 -ne n2`	Distinto

Comparación de números

- Implementar un *script* que pida un número en el rango $[1, 10)$ y compruebe si el número introducido está o no fuera de rango:

```
pedroa@pagutierrezLaptop:~/tmp$ ./numeroRango.sh
Introduzca un número (1 <= x < 10): 1
El número 1 es correcto!
pedroa@pagutierrezLaptop:~/tmp$ ./numeroRango.sh
Introduzca un número (1 <= x < 10): 0
Fuera de rango!
pedroa@pagutierrezLaptop:~/tmp$ ./numeroRango.sh
Introduzca un número (1 <= x < 10): 10
Fuera de rango!
```

Chequeo de ficheros

Operador	Significado
`-e f1`	¿Existe el fichero `f1`?
`-s f1`	¿`f1` tiene tamaño mayor que cero?
`-f f1`	¿Es `f1` un fichero normal?
`-d f1`	¿Es `f1` un directorio?
`-l f1`	¿Es `f1` un enlace simbólico?
`-r f1`	¿Tienes permiso de lectura sobre `f1`?
`-w f1`	¿Tienes permiso de escritura sobre `f1`?
`-x f1`	¿Tienes permiso de ejecución sobre `f1`?

Chequeo de ficheros

- Ejemplo: *script* que comprueba si el archivo `/etc/fstab` existe y si existe, lo copia a la carpeta actual.

```
#!/bin/bash
if [ -f /etc/fstab ];
then
  cp /etc/fstab .
  echo "Hecho."
else
  echo "Archivo /etc/fstab no existe."
  exit 1
fi
```

Chequeo de ficheros

- Ejercicio: escribir un *script* `bash` que haga lo siguiente:
  - Acepta un nombre de fichero.
  - Comprueba si el fichero existe.
  - Si existe, hace una copia del mismo poniéndole como nombre `nombreOriginal.bak_Fecha`, donde `Fecha` la podéis conseguir a partir del comando "`date +%d-%m-%y`"[2].

```
pedroa@pagutierrezLaptop:~/tmp$ date +%d-%m-%y
17-02-13
pedroa@pagutierrezLaptop:~/tmp$ ./backup.sh
El uso del programa es ./backup.sh nombreFichero
pedroa@pagutierrezLaptop:~/tmp$ ./backup.sh copiaFstab.sh
pedroa@pagutierrezLaptop:~/tmp$ ls copiaFstab* -l
-rwxr--r-- 1 pedroa pedroa 130 feb 17 15:02 copiaFstab.sh
-rwxr--r-- 1 pedroa pedroa 130 feb 17 15:14 copiaFstab.sh.bak_17-02-13
```

Operadores lógicos

Operador	Significado
!	No
`&&` o `-a`	Y
`||` o `-o`	O

- *Ojo*: uso distinto de las dos versiones de los operadores:

```
if [ $n1 -ge $n2 ] && [ $s1 = $s2 ];
...
if [ $n1 -ge $n2 -a $s1 = $s2 ];
...
```

- Ejercicio: implementar el *script* `numeroRango.sh` utilizando un solo `if`.

[2]Consulta `man date` para más información

3.2. Condicionales case

Condicionales `case`

- Evitar escribir muchos `if` seguidos:

```
case $var in
  val1)
    instrucciones;;
  val2)
    instrucciones;;
  *)
    instrucciones;;
esac
```

- El $\star$ agrupa a las instrucciones por defecto.
- Se pueden evaluar dos valores a la vez `val1 | val2 )`.

Condicionales `case`

- Ejemplo:

```
#!/bin/bash
echo -n "Introduzca un número t.q. 1 <= x < 10: "
read x
case $x in
  1) echo "Valor de x es 1.";;
  2) echo "Valor de x es 2.";;
  3) echo "Valor de x es 3.";;
  4) echo "Valor de x es 4.";;
  5) echo "Valor de x es 5.";;
  6) echo "Valor de x es 6.";;
  7) echo "Valor de x es 7.";;
  8) echo "Valor de x es 8.";;
  9) echo "Valor de x es 9.";;
  0 | 10) echo "Número incorrecto.";;
  *) echo "Valor no reconocido.";;
esac
```

3.3. Estructura iterativa `for`

Estructuras iterativas `for`

- Se utiliza para iterar a lo largo de una lista de valores de una variable:

```
for var in lista
do
  instrucciones;
done
```

- Las instrucciones se ejecutan con todos los valores que hay en lista para la variable var.
- *ejemploFor1.sh*:

```
#!/bin/bash
let sum=0
for num in 1 2 3 4 5
do
  let "sum = $sum + $num"
done
echo $sum
```

Estructuras iterativas `for`

- *ejemploFor2.sh*:

```
#!/bin/bash
for x in papel lapiz boligrafo
do
  echo "El valor de la variable es $x"
  sleep 5
done
```

¿y si queremos esta salida?:

```
pedroa@pagutierrezLaptop:~$ ./ejemploFor2Bis.sh
El valor de la variable es papel dorado
El valor de la variable es lapiz caro
El valor de la variable es boligrafo barato
```

Estructuras iterativas `for`

- Si eliminamos la parte de `in lista`, la lista sobre la que se itera es la lista de argumentos (`$1`, `$2`, `$3`...), *ejemploForArg.sh*:

```
#!/bin/bash
for x
do
  echo "El valor de la variable es $x"
  sleep 5
done
```

produce la salida:

```
pedroa@pagutierrezLaptop:~$ ./ejemploForArg.sh estudiante1 estudiante2
El valor de la variable es estudiante1
El valor de la variable es estudiante2
```

Estructuras iterativas `for`

- Iterando sobre listas de ficheros (*ejemploForListarFicheros.sh*):

```
#!/bin/bash

# Listar todos los ficheros del directorio actual
# incluyendo información del número de nodo
for x in *
do
  ls -i $x
done
```

```
9
10 # Listar todos los ficheros del directorio /bin
11 for x in /bin
12 do
13   ls -i $x
14 done
```

Estructuras iterativas `for`

- Comando `find`:

```
1 pedroa@pagutierrezLaptop:~$ find -name "*.sh"
2 ./ejemplos/ejemploForArg.sh
3 ./ejemplos/holaMundoVariable.sh
4 ...
```

- Listar ficheros que tengan extensión `.sh` (*ejemploForImpFichScripts.sh*):

```
1 #!/bin/bash
2
3 # Imprimir todos los ficheros que se encuentren
4 # con extensión .sh
5 for x in $(find -name "*.sh")
6 do
7   echo $x
8 done
```

Estructuras iterativas `for`

- Comando útil: *seq*.

```
1 #!/bin/bash
2 for i in $(seq 8)
3 do
4   echo $i
5 done
```

Estructuras iterativas `for`

- `for` tipo C:

```
1 for (( EXPR1; EXPR2; EXPR3 ))
2 do
3   instrucciones;
4 done
```

- Ejemplo (*ejemploForTipoC.sh*):

```
1 #!/bin/bash
2
3 echo -n "Introduzca un número: "; read x;
4 let sum=0
5 for (( i=1; $i<$x; i=$i+1 ))
6 do
7   let "sum=$sum + $i"
8 done
9 echo "La suma de los primeros $x números naturales es: $sum"
```

Arrays

- Para crear *arrays*: `miNuevoArray[i]=Valor`.
- Para crear *arrays*: `miNuevoArray=(Valor1 Valor2 Valor3)`.
- Para acceder a un valor: `${miNuevoArray[i]}`.
- Para acceder a todos los valores: `${miNuevoArray[*]}`.
- Para longitud: `${#miNuevoArray[@]}`.

```
pedroa@pagutierrezLaptop:~$ miNuevoArray[0]="Gran"
pedroa@pagutierrezLaptop:~$ miNuevoArray[1]="Array"
pedroa@pagutierrezLaptop:~$ miNuevoArray[2]="Triunfador"
pedroa@pagutierrezLaptop:~$ echo ${miNuevoArray[2]}
Triunfador
pedroa@pagutierrezLaptop:~$ miNuevoArray=( "Gran" "Array" "Triunfador" )
pedroa@pagutierrezLaptop:~$ echo ${miNuevoArray[1]}
Array
pedroa@pagutierrezLaptop:~$ echo ${miNuevoArray[*]}
Gran Array Triunfador
```

Arrays

- Combinar *arrays* y `for` (*arrayFor.sh*).

```
#!/bin/bash
elArray=("pelo" "pico" "pata")
for x in ${elArray[*]}
do
   echo "--> $x"
done
```

3.4. Estructuras iterativas while y until

Estructura iterativa `while`

```
while expresion_evalua_a_true
do
  instrucciones
done
```

Ejemplo (`while.sh`):

```
#!/bin/bash
echo -n "Introduzca un número: "; read x
let sum=0; let i=1
while [ $i -le $x ]; do
  let "sum = $sum + $i"
  let "i = $i + 1"
done
echo "La suma de los primeros $x números es: $sum"
```

Estructura iterativa `until`

```
until expresion_evalua_a_true
do
  instrucciones
done
```

Ejemplo (`until.sh`):

```
#!/bin/bash
echo -n "Introduzca un número: "; read x
until [ "$x" -le 0 ]; do
  echo $x
  x=$(($x-1))
  sleep 1
done
echo "TERMINADO"
```

4. Otras características

4.1. Funciones

Funciones en `bash`

- Las funciones hacen que los *scripts* sean más fáciles de mantener.
- El programa se divide en piezas de código más pequeñas.
- Función simple (`funcionHola.sh`):

```
#!/bin/bash
hola()
{
echo "Estás dentro de la función hola() y te saludo."
}

echo "La próxima línea llama a la función hola()"
hola
echo "Ahora ya has salido de la funcion"
```

Funciones en `bash`

- Los argumentos *NO* se especifican, sino que se usan las variables intrínsecas (`funcionCheck.sh`):

```
#!/bin/bash
function chequea() {
  if [ -e "$1" ]
  then
    return 0
  else
    return 1
  fi
}

echo -n "Introduzca el nombre del archivo: "
read x
if chequea $x
then
  echo "El archivo $x existe !"
```

```
16 else
17   echo "El archivo $x no existe !"
18 fi
```

4.2. Depuración

Depuración en bash

- Antes de ejecutar una instrucción, `bash` sustituye las variables de la línea (empiezan por $) y los comandos ($( ) o ` `).
- Para depurar los *scripts*, `bash` ofrece la posibilidad de:
  - Argumento *−x*: muestra cada línea completa del *script* antes de ser ejecutada, con sustitución de variables/comandos.
  - Argumento *−v*: muestra cada línea completa del *script* antes de ser ejecutada, tal y como se escribe.
- Introducir el argumento en la línea del *SheBang*.
- Ejemplo (`bashDepuracion.sh`):

```
1 #!/bin/bash -x
2 echo -n "Introduzca un número: "
3 read x
4 let sum=0
5 for (( i=1 ; $i<$x ; i=$i+1 )) ; do
6   let "sum = $sum + $i"
7 done
8 echo "La suma de los $x primeros números es: $sum"
```

Depuración en bash

```
1  pedroa@pagutierrezLaptop:~$ ./bashDepuracion.sh
2  + echo -n 'Introduzca un número: '
3  Introduzca un número: + read x
4  5
5  + let sum=0
6  + (( i=1  ))
7  + (( 1<5  ))
8  + let 'sum = 0 + 1'
9  + (( i=1+1  ))
10 + (( 2<5  ))
11 + let 'sum = 1 + 2'
12 + (( i=2+1  ))
13 + (( 3<5  ))
14 + let 'sum = 3 + 3'
15 + (( i=3+1  ))
16 + (( 4<5  ))
17 + let 'sum = 6 + 4'
18 + (( i=4+1  ))
19 + (( 5<5  ))
20 + echo 'La suma de los 5 primeros números es: 10'
21 La suma de los 5 primeros números es: 10
```

4.3. Redireccionamiento y tuberías

Redireccionamiento de entrada/salida

- Existen diferentes descriptores de ficheros:
  - `stdin`: entrada estándar (descriptor número 0) ⇒ Por defecto, teclado.
  - `stdout`: salida estándar (descriptor número 1) ⇒ Por defecto, consola.
  - `stderr`: salida de error (descriptor número 2) ⇒ Por defecto, consola.

Redireccionamiento de salida

- Operadores (cambiar los por defecto):
  - `comando > salida.txt`: la salida estándar de `comando` se escribirá en `salida.txt` y no por pantalla. Sobreescribe el contenido del fichero.
  - `comando >> salida.txt`: igual que `>`, pero añade el contenido al fichero sin sobrescribir.
  - `comando 2> error.txt`: la salida de error de `comando` se escribirá en `error.txt` y no por pantalla. Sobreescribe el contenido del fichero.
  - `comando 2>> error.txt`: igual que `2>`, pero añade el contenido al fichero sin sobrescribir.

```
1 ls -la > directorioactual.txt
2 date >> fechasespeciales.txt
3 ls /root 2> ~/quefalloocurrio.txt
4 cp ~/archivo.txt /root 2>> ~/logdefallos.txt
```

Redireccionamiento de salida

- `comando 2>&1`: redirecciona la salida de error de `comando` a la salida estándar.
- `comando 1>&2`: redirecciona la salida estándar de `comando` a la salida de error.
- `comando &> todo.txt`: redirecciona tanto la salida estándar como la de error hacia el fichero `todo.txt`, sobreescribiendo su contenido, y no se muestra por pantalla.
- `comando &>> todo.txt`: redirecciona tanto la salida estándar como la de error, lo añade al contenido de `todo.txt` y no se muestra por pantalla.

Redireccionamiento de entrada

- Es posible redireccionar la entrada estándar (`stdin`): `comando < ficheroConDatos.txt`.
- `comando` tomará como datos de entrada el contenido del fichero `ficheroConDatos.txt`
- Esto incluye los saltos de líneas, por lo que, por cada salto de línea se alimentará un `read`.

Tuberías

- Hasta ahora, redireccionamos entrada/salida comandos a partir de ficheros.
- Tuberías: redireccionar entrada/salida comandos entre si, sin usar ficheros.
- Sintaxis: `comando1 | comando2` la entrada de `comando2` será tomada de la salida de `comando1` (salida estándar o de error)
- Se pueden encadenar más de dos comandos.
- Mismo resultado:
  - `cat archivoConDatos.txt | grep -i prueba`
  - `grep -i prueba < archivoConDatos.txt`

Redireccionamiento de salida: `tee`

- A veces queremos redirigir la salida de forma que aparezca por consola y al mismo tiempo se vuelque a fichero.
- Para esto, podemos usar el comando `tee`:

```
pedroa@pagutierrezLaptop:~$ echo "Esto es una prueba"
Esto es una prueba
pedroa@pagutierrezLaptop:~$ echo "Esto es una prueba" > f1
pedroa@pagutierrezLaptop:~$ cat f1
Esto es una prueba
pedroa@pagutierrezLaptop:~$ echo "Esto es una prueba" | tee f1
Esto es una prueba
pedroa@pagutierrezLaptop:~$ cat f1
Esto es una prueba
pedroa@pagutierrezLaptop:~$ echo "Esto es una prueba" | tee -a f1
Esto es una prueba
pedroa@pagutierrezLaptop:~$ cat f1
Esto es una prueba
Esto es una prueba
```

Redireccionamiento de entrada: *Here documents*

- Los denominados *Here documents* son una manera de pasar datos a un programa de forma que el usuario pueda introducir más de una línea de texto. La sintaxis es la siguiente:

```
pedroa@pagutierrezLaptop:~$ cat << secuenciaSalida
> hola
> que
> tal
> secuenciaSalida
hola
que
tal
```

- Características:

- La entrada se va almacenando. Se van creando nuevas líneas pulsando la tecla *Intro*.
- Se acaban de recibir datos cuando se detecta la cadena de texto que se seleccionó para indicar la salida, en este caso `secuenciaSalida`.

Redireccionamiento de entrada: *Here documents*

- `ejemploHereDocument.sh`:

```
#!/bin/bash

# Sin here documents
echo "****************"
echo "* Mi script V1 *"
echo "****************"
echo "Introduzca su nombre"

# Usando here documents
cat << EOF
****************
* Mi script V1 *
****************
Introduzca su nombre
EOF
```

4.4. Comandos interesantes

Comando `cat`

- *cat*:
  - Visualiza el contenido de uno o más ficheros de texto.

```
pedroa@pagutierrezLaptop:~$ cat informacion.sh
#!/bin/sh
echo "Bienvenido $USER!, tu identificador es $UID."
echo "Esta es la shell número $SHLVL, que lleva $SECONDS arrancada."
echo "La arquitectura de esta máquina es $MACHTYPE y el cliente de terminal es
    $TERM"
pedroa@pagutierrezLaptop:~$ cat informacion.sh parametros.sh
#!/bin/sh
echo "Bienvenido $USER!, tu identificador es $UID."
echo "Esta es la shell número $SHLVL, que lleva $SECONDS arrancada."
echo "La arquitectura de esta máquina es $MACHTYPE y el cliente de terminal es
    $TERM"
#!/bin/bash
echo "$#; $0; $1; $2; $*; $@"
pedroa@pagutierrezLaptop:~$ cat < f1 > f2
# ¿Qué hacemos?
```

Comandos `head`, `tail` y `wc`

- *head* y *tail*:
  - Muestran las primeras o las últimas n líneas de un fichero.

```
pedroa@pagutierrezLaptop:~$ head -2 informacion.sh
#!/bin/sh
echo "Bienvenido $USER!, tu identificador es $UID."
pedroa@pagutierrezLaptop:~$ tail -1 informacion.sh
echo "La arquitectura de esta máquina es $MACHTYPE y el cliente de terminal es $TERM"
```

- *wc*: muestra el número de líneas, palabras o caracteres de uno o varios ficheros:

```
pedroa@pagutierrezLaptop:~$ wc -l informacion.sh
4 informacion.sh
pedroa@pagutierrezLaptop:~$ wc -c informacion.sh
219 informacion.sh
pedroa@pagutierrezLaptop:~$ wc -w informacion.sh
34 informacion.sh
pedroa@pagutierrezLaptop:~$ wc -w numero*.sh
 37 numeroRango1If.sh
 46 numeroRango.sh
 83 total
```

Comandos `more`, `cmp` y `sort`

- *more* `fichero`: muestra ficheros grandes, pantalla a pantalla.
- *cmp* `f1 f2`: compara dos ficheros y dice a partir de qué carácter son distintos.

```
pedroa@pagutierrezLaptop:~$ cmp numeroRango.sh numeroRango1If.sh
numeroRango.sh numeroRango1If.sh son distintos: byte 95, línea 5
```

- *sort* `[fichero]`: ordena la entrada estándar o un fichero.
  - *sort*: ordena entrada estándar por orden alfabético.
  - *sort -r*: ordena entrada estándar por orden inverso.
  - *sort -n*: ordena entrada estándar por orden numérico.
  - *sort -t c*: cambia el caracter separador al caracter `c`.
  - *sort -k 3*: cambia la clave de ordenación a la tercera columna (por defecto, primera columna).

Comando `sort`

```
pedroa@pagutierrezLaptop:~$ echo -e "18\n017\n9" | sort
017
18
9
pedroa@pagutierrezLaptop:~$ echo -e "18\n017\n9" | sort -r
9
18
017
pedroa@pagutierrezLaptop:~$ echo -e "18\n017\n9" | sort -n
9
017
18
pedroa@pagutierrezLaptop:~$ echo -e "18\n017\n9" | sort -nr
18
017
9
```

```
pedroa@pagutierrezLaptop:~$ echo -e "18 1\n017 2\n9 3" | sort -n -k 2
18 1
017 2
9 3
pedroa@pagutierrezLaptop:~$ echo -e "18 1\n017 2\n9 3" | sort -n -k 1
9 3
017 2
18 1
pedroa@pedroa-laptop ~ $ echo -e "1\t2\n2\t-1" | sort -t $'\t' -nk1
1       2
2       -1
pedroa@pedroa-laptop ~ $ echo -e "1\t2\n2\t-1" | sort -t $'\t' -nk2
2       -1
1       2
```

Comando `grep`

- *`grep [opciones] patron [fichero(s)]`*: filtra el texto de un(os) fichero(s), mostrando únicamente las líneas que cumplen un determinado patrón.
  - `-c`: cuenta el número de líneas con el patrón.
  - `-l`: muestra el nombre de los ficheros que contienen el patrón.
  - `-i`: *case insensitive* (no sensible a mayúsculas).
  - También admite la entrada estándar (`stdin`).

```
pedroa@pagutierrezLaptop:~$ grep ^c *
case.sh:case $x in
pedroa@pagutierrezLaptop:~$ grep -l ^c *
case.sh
pedroa@pagutierrezLaptop:~$ grep -c ^c *
arrayFor.sh:0
backup.sh:0
case.sh:1
...
pedroa@pagutierrezLaptop:~$ ls * | grep ^c
case.sh
comillas.sh
copiaFstab.sh
```

Comando `grep`

- *`grep [opciones] patron [fichero(s)]`*:
  - `patron`: "`^`" significa comienzo de la línea, "`$`" significa fin de la línea, "`.`" significa cualquier carácter.

```
pedroa@pagutierrezLaptop:~$ ls * | grep s\.sh$
comillas.sh
ejemploFor2Bis.sh
ejemploForImpFichScripts.sh
ejemploForListarFicheros.sh
operaciones.sh
parametros.sh
pedroa@pagutierrezLaptop:~$ ls * | grep ^ejemplo.or
ejemploFor1.sh
```

```
ejemploFor2Bis.sh
ejemploFor2.sh
ejemploForArg.sh
ejemploForImpFichScripts.sh
ejemploForListarFicheros.sh
ejemploForTipoC.sh
```

Comando `find`

- `find [carpeta] -name "patrón"`: busca ficheros cuyo nombre cumpla el patrón y que estén guardados a partir de la carpeta `carpeta` (por defecto `.`).

```
pedroa@pagutierrezLaptop:~$ find ~ -name "*.sh"
/home/pagutierrez/workspaces/lrws/jclec-nnep/test.sh
/home/pagutierrez/workspaces/weka_ws/weka-adaboost/test.sh
/home/pagutierrez/workspaces/weka_ws/weka-adaboost/toy-test5D3.sh
```

- `find [carpeta] -size N`: busca ficheros cuyo tamaño sea `N` (`+N`: mayor que `N`, `-N`: menor que `N`).

```
pedroa@pagutierrezLaptop:~$ find ~ -size 1024
/home/pagutierrez/.icedove/b7aw3yuu.default/addons.sqlite
/home/pagutierrez/.icedove/b7aw3yuu.default/blist.sqlite
/home/pagutierrez/.icedove/b7aw3yuu.default/cookies.sqlite
```

Comando `find`, `basename` y `dirname`

- `find [carpeta] -user usuario`: busca ficheros cuyo nombre usuario propietario sea `usuario`.

```
pedroa@pagutierrezLaptop:~$ find ~ -user pedroa
/home/pagutierrez
/home/pagutierrez/.bashrc
```

- `basename fichero [.ext]`: Devuelve el nombre de un fichero sin su carpeta [y sin su extensión].

- `dirname fichero`: Devuelve la carpeta donde se aloja un fichero.

```
pedroa@pagutierrezLaptop:~$ basename "/home/pedroa/Escritorio/recorrido.sh"
recorrido.sh
pedroa@pagutierrezLaptop:~$ basename "/home/pedroa/Escritorio/recorrido.sh" .sh
recorrido
pedroa@pagutierrezLaptop:~$ dirname "/home/pedroa/Escritorio/recorrido.sh"
/home/pedroa/Escritorio/
```

Comando `stat`

- `stat fichero`: nos muestra propiedades sobre un determinado ficheros.

```
pedroa@pedroa-laptop ~ $ stat missfont.log
  Fichero: «missfont.log»
  Tamaño: 0                       Bloques: 0          Bloque E/S: 4096   fichero
      regular vacío
Dispositivo: 803h/2051d           Nodo-i: 2938624     Enlaces: 1
Acceso: (0644/-rw-r--r--)  Uid: ( 1000/  pedroa)   Gid: ( 1000/  pedroa)
      Acceso: 2016-02-24 08:50:13.080997730 +0100
Modificación: 2016-02-24 08:50:13.080997730 +0100
      Cambio: 2016-02-24 08:50:13.080997730 +0100
    Creación: -
```

- `stat -c %a fichero`: nos permite personalizar la salida y obtener diferentes propiedades sobre un fichero[3].

```
pedroa@pedroa-laptop ~ $ stat -c "Permisos: %a. Tipo fichero: %F" missfont.log
Permisos: 644. Tipo fichero: fichero regular vacío
```

Comando `tr`

- `tr c1 c2`: reemplaza el carácter `c1` por el carácter `c2`. Trabaja en el `stdin`.

```
pedroa@pedroa-laptop ~ $ echo TIERRA | tr 'R' 'L'
TIELLA
```

- `tr -d c`: elimina el carácter `c` de la salida.

```
pedroa@pedroa-laptop ~ $ echo TIERRA | tr -d R
TIEA
pedroa@pedroa-laptop ~ $ echo TIERRA | tr -d RT
IEA
```

Expansión de llaves

- El operador *brace expansion* o expansión de llaves nos permite generar combinaciones de cadenas de texto de forma simple:

```
pedroa@pedroa-laptop ~ $ echo fichero.{pdf,png,jpg}
fichero.pdf fichero.png fichero.jpg
```

- Como se puede observar, la sintaxis es `cadena1{c1,c2,c3,...}`, de forma que se combinará `cadena1` con `c1`, `c2`, `c3`...

- {c1..c2} permite especificar todos el rango de caracteres desde `c1` hasta `c2`:

```
pedroa@pedroa-laptop ~ $ echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v w x y z
pedroa@pedroa-laptop ~ $ echo {1..8}
1 2 3 4 5 6 7 8
pedroa@pedroa-laptop ~ $ echo {1..3}{a..c}
1a 1b 1c 2a 2b 2c 3a 3b 3c
```

[3] `man stat` para más información.

Recorriendo ficheros

- Un ejemplo de redirección de comandos útil para recorrer ficheros:

```
find carpeta -name "patron" | while read f
do
 ...
done
```

- Explica qué está sucediendo.
- *Cuidado*: la entrada está redirigida durante todo el bucle (no podremos hacer `read` dentro del bucle).
- ¿Cómo lo haríamos con un `for`?

Inciso: problemas con espacios en blanco y arrays

- Cuando intentamos construir un *array* a partir de una cadena, `bash` utiliza determinados caracteres para separar cada uno de los elementos del *array*.
- Estos caracteres están en la variable de entorno IFS y por defecto son el espacio, el tabulador y el salto de línea.

```
pedroa@Laptop:~$ array=($(echo "1 2 3"))
pedroa@Laptop:~$ echo ${array[0]}
1
pedroa@Laptop:~$ echo ${array[1]}
2
pedroa@Laptop:~$ echo ${array[2]}
3
pedroa@Laptop:~$ array=($(echo -e "1\t2\n3"))
pedroa@Laptop:~$ echo ${array[0]}
1
pedroa@Laptop:~$ echo ${array[1]}
2
pedroa@Laptop:~$ echo ${array[2]}
3
```

Inciso: problemas con espacios en blanco y arrays

- Esto nos puede producir problemas si estamos procesando elementos con espacios (por ejemplo, nombres de ficheros con espacios):

```
pedroa@Laptop:~$ array=($(echo -e "El uno\nEl dos\nEl tres"))
pedroa@Laptop:~$ echo ${array[0]}
El
pedroa@Laptop:~$ echo ${array[1]}
uno
```

- *Solución*: cambiar el IFS para que solo se utilice el `\n`:

```
pedroa@Laptop:~$ OLDIFS=$IFS
pedroa@Laptop:~$ IFS=$'\n'
pedroa@Laptop:~$ array=($(echo -e "El uno\nEl dos\nEl tres"))
pedroa@Laptop:~$ echo ${array[0]}
El uno
pedroa@Laptop:~$ echo ${array[1]}
El dos
pedroa@Laptop:~$ IFS=$OLDIFS
```

5. Referencias

Referencias

Referencias

[Kochan and Wood, 2003] Stephen G. Kochan y Patrick Wood Unix shell programming. Sams Publishing. Tercera Edición. 2003.

[Nemeth et al., 2010] Evi Nemeth, Garth Snyder, Trent R. Hein y Ben Whaley Unix and Linux system administration handbook.
Capítulo 2. *Scripting and the shell*. Prentice Hall. Cuarta edición. 2010.

[Frisch, 2002] Aeleen Frisch. Essential system administration.
Apéndice. *Administrative Shell Programming*. O'Reilly and Associates. Tercera edición. 2002.