0.

Objetivos del aprendizaje

- Saber **cuando utilizar *scripts*** para resolver tareas de programacion, identificando las ventajas e inconvenientes de los lenguajes de *scripting* y su aplicabilidad en administracion de sistemas.
- Conocer los **distintos interpretes** de ordenes en GNU/Linux y justificar el uso de `bash` para la programacion de *scripts* de administracion 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** mas habituales en `bash`.
- Utilizar las **variables intrinsecas** de `bash` para interactuar de forma mas efectiva con la terminal de comandos.
- Utilizar correctamente el comando `exit`.
- Utilizar correctamente el comando `read`.
- Aplicar correctamente la **sustitucion de comandos** en `bash`.
- Conocer y utilizar distintas alternativas para realizar **operaciones aritmeticas** en `bash`.
- Utilizar **estructuras condicionales**.
- Comparar correctamente cadenas y numeros, chequear el estado de ficheros y aplicar operadores logicos.
- Utilizar **estructuras iterativas**.
- Utilizar *arrays* en `bash`.
- Utilizar **funciones** en `bash`.
- Aplicar diversas opciones para la **depuracion** de *scripts* en `bash`.
- **Redirigir la entrada y la salida** de comandos desde y hacia ficheros.
- Interconectar distintos comandos mediante el uso de **tuberias**.
- Conocer los *here documents* y utilizarlos para hacer *scripts* mas legibles.
- Utilizar correctamente los siguientes comandos adicionales: `cat`, `head`, `tail`, `wc`, `find`, `basename`, `dirname`, `stat` y `tr`.
- Aplicar el mecanismo de **expansion de llaves** en la creacion de *arrays*.

Contenidos

Evaluacion

- Pruebas de validacion de practicas.

1. Introduccion

1.1. Justificacion

¿Linea de comandos?

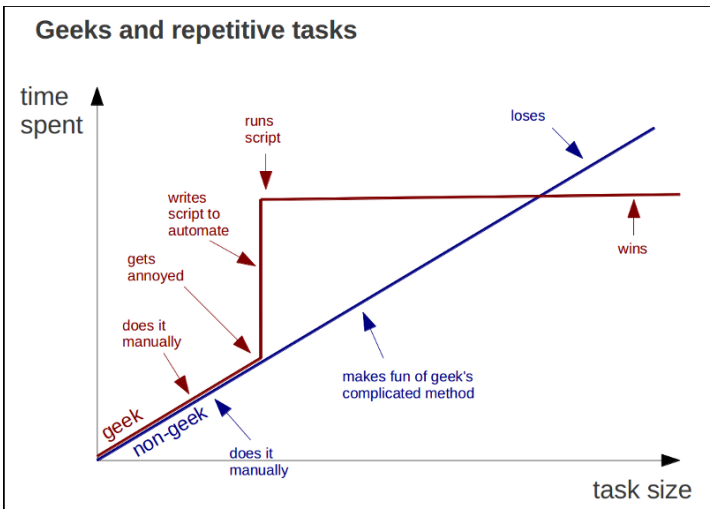
- ¿Para que necesito aprender a utilizar la linea de comandos?
- Historia real[1]:
  - Unidad compartida por cuatro servidores que esta llenandose → impedia a la gente trabajar.
  - El sistema no soportaba cuotas.
  - Un ingeniero escribe un programa en C++ que navega por los archivos de todos los usuarios, calcula cuanto espacio esta 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
```

bash

- Las interfaces graficas de usuario (GUI) son buenas para muchas cosas, pero no para todas, especialmente las mas repetitivas.

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



bash

- ¿Que es la *shell*?.
  - Programa que recoge comandos del ordenador y se los proporciona al SO para que los ejecute.
  - Antiguamente, era la unica interfaz disponible para interactuar con SO tipo Unix.
- En casi todos los sistemas GNU/Linux, el programa que actua como *shell* es `bash`.
  - *B*ourne *A*gain *SH*ell → version mejorada del `sh` original de Unix.
  - Escrito por Steve Bourne.

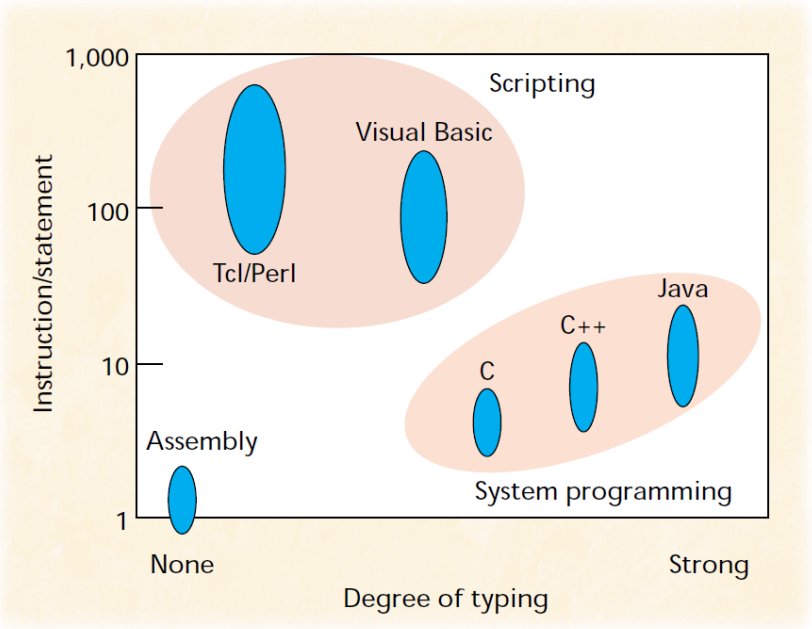
bash

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

1.2. ¿Programacion o scripting?

¿Programacion o scripting?

- `bash` no es unicamente una excelente *shell* por linea de comandos...
- Tambien es un *lenguaje de scripting* en si mismo.
- El *shell scripting* sirve para automatizar multitud de tareas que, de otra forma, requeririan multiples comandos introducidos de forma manual.
- Lenguaje de programacion (LP) vs. *scripting*:
  - Los LPs son, en general, mas potentes y mucho mas rapidos que los lenguajes de *scripting*.
  - Los LPs comienzan desde el codigo fuente, que se compila para crear los ejecutables (lo que no permite que los programas sean facilmente 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.)

¿Programacion o scripting?

- Un lenguaje de *scripting* (LS) tambien comienza por el codigo fuente, pero no se compila en un ejecutable.
- En su lugar, un interprete lee las instrucciones del fichero fuente y las ejecuta secuencialmente.
  - Programas interpretados $\rightarrow$ mas lentos que los compilados.
  - "Tipado" debil (*¿ventaja o desventaja?*).

- Ventajas:
  - En general, una linea de LS **realiza mas operacione**s que una de un LP.
  - El fichero de codigo es facilmente **portable** a cualquier SO.
  - Todo lo que yo pueda hacer con mi *shell*, lo puedo **automatizar** con un *script*.
  - Nivel de **abstraccion muy superior** en cuanto a operaciones con ficheros, procesos...

1.3. Primeros programas

Primer programa `bash`: *`holaMundo.sh`*

- Abrir un editor de textos:

```
i72jivem@VTS3:~/PAS/p1$ gedit holaMundo.sh &
```

- Escribimos el codigo:

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

- Hacemos que el fichero de texto sea ejecutable:

```
i72jivem@VTS3:~/PAS/p1$ chmod u+x holaMundo.sh
i72jivem@VTS3:~/PAS/p1$ ls -l holaMundo.sh
-rwx------ 1 i72jivem upi0 29 feb 14 09:54 holaMundo.sh
```

Primer programa `bash`

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

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

Primer programa `bash`

- Para ejecutar el programa:

```
i72jivem@VTS3:~/PAS/p1$ holaMundo.sh
-bash: holaMundo.sh: no se encontro la orden
```

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

```
i72jivem@VTS3:~/PAS/p1$ echo $PATH
```

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

```
i72jivem@VTS3:~/PAS/p1$ /home/i72jivem/PAS/p1/holaMundo.sh
Hola Mundo
i72jivem@VTS3:~/PAS/p1$ ./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 programacion.
  - Se puede utilizar el `man` para ver sus opciones.

```
i72jivem@VTS3:~/PAS/p1$ echo "Imprimo una linea con salto de linea"
Imprimo una linea con salto de linea
i72jivem@VTS3:~/PAS/p1$ echo -n "Imprimo una linea sin salto de linea"
Imprimo una linea sin salto de lineai72jivem@VTS3:~/PAS/p1$

which echo
/usr/local/bin/echo

i72jivem@VTS3:~/PAS/p1$ echo "ho\nla"
ho\nla
i72jivem@VTS3:~/PAS/p1$ echo -e "ho\nla"
ho
la
```

2. Variables

2.1. Concepto y declaracion

Variables: concepto

- Al igual que en los LP, se pueden utilizar *variables*.
- Todos los valores son almacenados como tipo cadena de texto (*"tipado" debil*).
- ¿No puedo operar?
  - Operadores matematicos que convierten las variables en numero para el calculo.
- 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`*

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

- Asignacion: `VARIABLE="valor"`
- Resolver una variable, es decir, sustituir la variable por su valor: *`$VARIABLE`*
- *No* se pueden poner espacios antes o despues del "="

Variables: precaucion

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

```
contador=0
contador=Domingo
```

- La conmutacion del tipo de una variable puede llevar a confusion.
- Buena practica: asociar siempre el mismo tipo de dato a una variable en el contexto de un mismo *script*.

Variables: precaucion

- Caracter de escape:
  - Un caracter de escape es un caracter que permite que los simbolos especiales del lenguaje de programacion o scripting no se interpreten y se utilice su valor literal.
  - Por ejemplo, permite incluir una comilla dentro de una cadena:

```
"Esta cadena contiene el caracter \" en su interior"
```

2.2. Comillas simples y dobles

Comillas simples y dobles

- Cuando el valor de la variable contenga espacios en blanco o caracteres especiales, se debera encerrar entre comillas.
- Las **comillas simples no permiten introducir variables** dentro de la cadena. Todos los caracteres se interpretan de forma literal.
- Si son dobles, se permitira especificar variables internas que se resolveran. Para interpretar un caracter de forma literal, se puede usar el caracter de escape "\".

```
i72jivem@VTS3:~/PAS/p1$ var="cadena de prueba"
i72jivem@VTS3:~/PAS/p1$ nuevavar="Valor de var es $var"
i72jivem@VTS3:~/PAS/p1$ echo $nuevavar
Valor de var es cadena de prueba
```

- ¿Que hubiera pasado en este caso?

```
i72jivem@VTS3:~/PAS/p1$ nuevavar='Valor de var es $var'
i72jivem@VTS3:~/PAS/p1$ echo $nuevavar
```

Comillas simples y dobles

- Hacer un *script* que muestre por pantalla usando 2 variables(`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 configuracion.
    - Se pueden listar utilizando el comando `env`.

```
1 i72jivem@VTS3:~/PAS/p1$ echo $SHELL
2 /bin/bash
3 i72jivem@VTS3:~/PAS/p1$ echo $PATH
4 /usr/local/opt/intel_composer_xe_2013/bin:/bin64:/usr/local/opt/Qt/bin:/usr/local/bin:/bin:/usr/local/
      java/bin:...
```

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

Comando export

- El comando export establece una variable en el entorno que sea accesible por los procesos hijos o permite modificar una ya existente durante una sesion de terminal (Proceso padre)

```
1  i72jivem@VTS3:~/PAS/p1$ x=hola
2  i72jivem@VTS3:~/PAS/p1$ bash          # Creamos proceso hijo
3  i72jivem@VTS3:~/PAS/p1$ echo $x
4
5  i72jivem@VTS3:~/PAS/p1$ exit          # Volvemos al proceso padre
6  exit
7  i72jivem@VTS3:~/PAS/p1$ export x      # Exportamos nuestra variable
8  i72jivem@VTS3:~/PAS/p1$ bash          # Volvemos a crear proceso hijo
9  i72jivem@VTS3:~/PAS/p1$ echo $x
10 hola
```

Comando export

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

```
1 i72jivem@VTS3:~/PAS/p1$ x=hola
2 i72jivem@VTS3:~/PAS/p1$ export x
3 i72jivem@VTS3:~/PAS/p1$ bash
4 i72jivem@VTS3:~/PAS/p1$ x=adios
5 i72jivem@VTS3:~/PAS/p1$ exit
6 exit
7 i72jivem@VTS3:~/PAS/p1$ echo $x
8 hola
```

Algunas variables importantes

- "*Home, sweet* `$HOME`":
  - `$HOME`: directorio personal del usuario, donde deberia 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* asi:

```
$ ./helloworld.sh
```

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

```
$ helloworld.sh
```

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

Algunas variables importantes

- Si modificamos el `.bash_profile`:

```
PATH=$PATH:$HOME/PAS/p1
export PATH
```

- El directorio `/home/i72jivem/PAS/p1` sera incluido en la busqueda de programas binarios a ejecutar.

```
    i72jivem@VTS3:~/PAS/p1$ echo $PATH
    /usr/local/opt/intel_composer_xe_2013/bin: ... :/home/i72jivem/PAS/p1
    i72jivem@VTS3:~/PAS/p1$ holaMundo.sh
    Hola mundo
```

- Cambios en la PATH no se guardaran al cerrar sesion de la terminal

Mas variables importantes

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

Mas variables importantes

- Ejercicio: haz un *script* que muestre la siguiente informacion(`informacion.sh`):

```
i72jivem@VTS3:~/PAS/p1$ ./informacion.sh
Bienvenido i72jivem!, tu identificador es 97710.
Esta es la shell numero 1 que lleva 108 arrancada.
La arquitectura de esta maquina es x86_64-unknown-linux-gnu y el nombre es VTS3.
```

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

```
i72jivem-i72jivem:~/PAS/p1 (hola, son las 13:19:11)
```

Variables intrinsecas

- `$#`: numero de argumentos de la linea de comandos (`argc`). Cuenta desde 0.
- `$`*n*: *n*-esimo argumento de la linea de comandos (`argv[n]`), si n es mayor que 9 utilizar `${`*n*`}`.
- `$*`: todos los argumentos de la linea de comandos (como una sola cadena).
- `$@`: todos los argumentos de la linea de comandos (como un *array*).
- `$!`: pid del ultimo proceso que se lanzo con &.
- `$-`: opciones suministradas a la *shell*.
- `$?`: valor de salida la ultima orden ejecutada (ver `exit`).

Variables intrinsecas

- Ejercicio: escribir un *script* (`parametros.sh`) que imprima el numero de argumentos que se le han pasado por linea 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*.

```
i72jivem@VTS3:~/PAS/p1$ ./parametros.sh estudiante1 estudiante2
2; ./comillas.sh; estudiante1; estudiante2; estudiante1 estudiante2; estudiante1 estudiante2
```

Variables intrinsecas: navegar por comandos anteriores

- `!$`: ultimo argumento del ultimo comando ejecutado.
- `!:`*n*: *n*-esimo argumento del ultimo comando ejecutado.

```
i72jivem@VTS3:~/PAS/p1$ echo argumentos 2 3
argumentos 2 3
i72jivem@VTS3:~/PAS/p1$ echo !$
echo 3
3
i72jivem@VTS3:~/PAS/p1$ echo !:0
echo echo
echo
```

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

Comando `exit`

- Se puede utilizar para finalizar la ejecucion de un *script* y devolver un valor de salida $(0 - 255)$ que estara disponible para el proceso padre que invoco el *script*.
  - Si lo llamamos sin parametros, se utilizara el valor de salida del ultimo comando ejecutado (equivalente a `exit $?`).

```
1  i72jivem@VTS3:~/PAS/p1$ echo $?
2  0
3  i72jivem@VTS3:~/PAS/p1$ bash
4  i72jivem@VTS3:~/PAS/p1$ echo $?
5  0
6  i72jivem@VTS3:~/PAS/p1$ exit 2
7  exit
8  i72jivem@VTS3:~/PAS/p1$ echo $?
9  2
10 i72jivem@VTS3:~/PAS/p1$ echo $?
11 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:

```
1 #!/bin/bash
2 echo -n "Introduzca nombre de fichero a borrar: "
3 read fichero
4 rm -i $fichero # La opcion -i pide confirmacion
5 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 informacion al usuario.
  - `read -t`*T*: acepta la entrada durante un tiempo maximo de *T* segundos.

```
1 i72jivem@VTS3:~/PAS/p1$ read -s -t5 -n1 -p "si (S) o no (N)?" respuesta
2 si (S) o no (N)?S
3 i72jivem@VTS3:~/PAS/p1$ echo $respuesta
4 S
```

Sustitucion de comandos (*IMPORTANTE*)

- El acento hacia atras (`) es distinto que la comilla simple (').
- `` `comando` `` se utiliza para sustitucion de comandos. Es decir, se ejecutaria el comando `comando` y se almacenaria su salida :

```
1 i72jivem@VTS3:~/PAS/p1$ LISTA=`ls`
2 i72jivem@VTS3:~/PAS/p1$ echo $LISTA
3 comillas.sh holaMundo.sh informacion.sh
```

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

```
1 i72jivem@VTS3:~/PAS/p1$ LISTA=$(ls)    #Listar directorio actual
2 i72jivem@VTS3:~/PAS/p1$ echo $LISTA
3 comillas.sh holaMundo.sh informacion.sh
4
5 i72jivem@VTS3:~/PAS/p1$ ls $(pwd)
6 comillas.sh holaMundo.sh informacion.sh
7
8 i72jivem@VTS3:~/PAS/p1$ ls $(echo /home/i72jivem/PAS/p1)
9 comillas.sh holaMundo.sh informacion.sh
```

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

2.5. Operadores aritmeticos

Operadores aritmeticos

- Bash permite realizar operaciones aritmeticas

Operador	Significado
+	Suma
−	Resta
*	Multiplicacion
/	Division
**	Exponenciacion
%	Modulo

```
1 i72jivem@VTS3:~/PAS/p1$ a=(5+2)*3
2 i72jivem@VTS3:~/PAS/p1$ echo $a
3 (5+2)*3
4 i72jivem@VTS3:~/PAS/p1$ b=2**3
5 i72jivem@VTS3:~/PAS/p1$ echo $a+$b
6 (5+2)*3+2**3
```

Operadores aritmeticos

- Hay que utilizar la instruccion `let`:

```
1 i72jivem@VTS3:~/PAS/p1$ let X=10+2*7
2 i72jivem@VTS3:~/PAS/p1$ echo $X
3 24
4 i72jivem@VTS3:~/PAS/p1$ let Y=X+2*4
5 i72jivem@VTS3:~/PAS/p1$ echo $Y
6 32
```

- Alternativamente, las expresiones aritmeticas tambien se pueden evaluar con `$[expresion]` o `$((expresion))`:

```
i72jivem@VTS3:~/PAS/p1$ echo $((123+20))
143
i72jivem@VTS3:~/PAS/p1$ echo "$((123+20))"
143
i72jivem@VTS3:~/PAS/p1$ VALOR=$[123+20]
i72jivem@VTS3:~/PAS/p1$ echo $[123*$VALOR]
17589
i72jivem@VTS3:~/PAS/p1$ echo "$[123*$VALOR]"
17589
                              !!CUIDADO!!
i72jivem@VTS3:~/PAS/p1$ echo '$[123*$VALOR]'
$[123*$VALOR]
```

Operadores aritmeticos

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

```
i72jivem@VTS3:~/PAS/p1$ ./operaciones.sh
Introduzca un primer numero: 2
Introduzca un segundo numero : 9
Suma: 11
Resta: -7
Multiplicacion: 18
Division: 0
Modulo: 2
```

3. Estructuras de control

3.1. Condicionales if

Condicionales `if`

- La forma mas basica es:

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

- Las secciones `elif` (`else if`) y `else` son opcionales.
- *IMPORTANTE*: espacios antes y despues `[` y `]`.
- *IMPORTANTE*: No olvidar `;`

Expresiones logicas

- Expresiones logicas pueden ser:
  - Comparacion de cadenas.

  - Comparacion de numeros.
  - Chequeo de ficheros.
  - Combinacion de los anteriores mediante operadores logicos.
- Las expresiones se encierran con corchetes `[ expresion ]`.
- En realidad, se esta llamando al programa `/usr/bin/[`.

```
i72jivem@VTS3:~/PAS/p1$ /usr/bin/[ 3 -eq 4 ]
i72jivem@VTS3:~/PAS/p1$ echo $?
1
i72jivem@VTS3:~/PAS/p1$ /usr/bin/[ 4 -eq 4 ]
i72jivem@VTS3:~/PAS/p1$ echo $?
0
i72jivem@VTS3:~/PAS/p1$ /usr/bin/[ 'asa' == 'asa' ]
i72jivem@VTS3:~/PAS/p1$ echo $?
0
i72jivem@VTS3:~/PAS/p1$ /usr/bin/[ 'asa' == 'asaa' ]
i72jivem@VTS3:~/PAS/p1$ echo $?
1
```

Comparacion 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 esta vacia, 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`.

Comparacion de cadenas

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

```
vi72jivem@VTS3:~/PAS/p1$ ./saludaUsuario.sh
Introduzca su nombre de usuario: Isa
Bienvenido "Isa"
i72jivem@VTS3:~/PAS/p1$ ./saludaUsuario.sh
Introduzca su nombre de usuario: Javi
Eso es mentira!
```

Comparacion de numeros

Operador	Significado
`n1 -lt n2`	(***L**ess **T**han*) Menor que
`n1 -gt n2`	(***G**reater **T**han*) Mayor que
`n1 -le n2`	(***L**ess or **E**qual*) Menor o igual que
`n1 -ge n2`	(***G**reater or **E**qual*) Mayor o igual que
`n1 -eq n2`	(***Eq**ual*) Igual
`n1 -ne n2`	(***N**ot **E**qual*) Distinto

Comparacion de numeros

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

```
i72jivem@VTS3:~/PAS/p1$ ./numeroRango.sh
Introduzca un numero (1 <= x < 10): 1
El numero 1 es correcto!
i72jivem@VTS3:~/PAS/p1$ ./numeroRango.sh
Introduzca un numero (1 <= x < 10): 0
Fuera de rango!
i72jivem@VTS3:~/PAS/p1$ ./numeroRango.sh
Introduzca un numero (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 simbolico?
`-r f1`	¿Tienes permiso de lectura sobre `f1`?
`-w f1`	¿Tienes permiso de escritura sobre `f1`?
`-x f1`	¿Tienes permiso de ejecucion 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
```

Operadores logicos

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

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

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

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

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 numero 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 "Numero 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?:

```
i72jivem@VTS3:~/PAS/p1$ ./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:

```
i72jivem@VTS3:~/PAS/p1$ ./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 informacion del numero de nodo
for x in *
do
  ls -i $x
done

# Listar todos los ficheros del directorio /bin
for x in /bin
do
  ls -i $x
done
```

Estructuras iterativas `for`

- Comando `find`:

```
i72jivem@VTS3:~/PAS/p1$ find -name "*.sh"
./ejemploForArg.sh
./holaMundoVariable.sh
...
```

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

```
#!/bin/bash

# Imprimir todos los ficheros que se encuentren
# con extension .sh
for x in $(find -name "*.sh")
do
  echo $x
done
```

Estructuras iterativas `for`

- Comando util: *seq*.

```
#!/bin/bash
for i in $(seq 8)
do
  echo $i
done
```

Estructuras iterativas `for`

- `for` tipo C:

```
for (( EXPR1; EXPR2; EXPR3 ))
do
  instrucciones;
done
```

- Ejemplo (*ejemploForTipoC.sh*):

```
#!/bin/bash

echo -n "Introduzca un numero: "; read x;
let sum=0
for (( i=1; $i<$x; i=$i+1 ))
do
  let "sum=$sum + $i"
done
echo "La suma de los primeros $x numeros 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[@]}`.

```
i72jivem@VTS3:~/PAS/p1$ miNuevoArray[0]="Gran"
i72jivem@VTS3:~/PAS/p1$ miNuevoArray[1]="Array"
i72jivem@VTS3:~/PAS/p1$ miNuevoArray[2]="Triunfador"
i72jivem@VTS3:~/PAS/p1$ echo ${miNuevoArray[2]}
Triunfador
i72jivem@VTS3:~/PAS/p1$ miNuevoArray=( "Gran" "Array" "Triunfador" )
i72jivem@VTS3:~/PAS/p1$ echo ${miNuevoArray[1]}
Array
i72jivem@VTS3:~/PAS/p1$ 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 numero: "; 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 numeros es: $sum"
```

Estructura iterativa `until`

```
until expresion_evalua_a_true
do
  instrucciones
done
```

Ejemplo (`until.sh`):

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

4. Otras caracteristicas

4.1. Funciones

Funciones en `bash`

- Las funciones hacen que los *scripts* sean mas faciles de mantener.
- El programa se divide en piezas de codigo mas pequeñas.
- Funcion simple (`funcionHola.sh`):

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

echo "La proxima linea llama a la funcion hola()"
hola
echo "Ahora ya has salido de la funcion"
```

Funciones en `bash`

- Los argumentos *NO* se especifican, sino que se usan las variables intrinsecas (`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 !"
else
  echo "El archivo $x no existe !"
fi
```

4.2. Depuracion

Depuracion en `bash`

- Antes de ejecutar una instruccion, `bash` sustituye las variables de la linea (empiezan por $) y los comandos ($( ) o ` `).
- Para depurar los *scripts*, `bash` ofrece la posibilidad de:

- Argumento *-x*: muestra cada linea completa del *script* antes de ser ejecutada, con sustitucion de variables/comandos.
- Argumento *-v*: muestra cada linea completa del *script* antes de ser ejecutada, tal y como se escribe.

- Introducir el argumento en la linea del *SheBang*.
- Ejemplo (`bashDepuracion.sh`):

```
1 #!/bin/bash -x
2 echo -n "Introduzca un numero: "
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 numeros es: $sum"
```

Depuracion en `bash`

```
1  i72jivem@VTS3:~/PAS/p1$ ./bashDepuracion.sh
2  + echo -n 'Introduzca un numero: '
3  Introduzca un numero: + 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 numeros es: 10'
21 La suma de los 5 primeros numeros es: 10
```

4.3. Redireccionamiento y tuberias

Redireccionamiento de entrada/salida

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

Redireccionamiento de salida

- Operadores (cambiar los por defecto):
  - `comando > salida.txt`: la salida estandar de `comando` se escribira 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 escribira 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 ~/PAS/p1/archivo.txt /root 2>> ~/PAS/p1/logdefallos.txt
```

Redireccionamiento de salida

- `comando 2>&1`: redirecciona la salida de error de `comando` a la salida estandar.
- `comando 1>&2`: redirecciona la salida estandar de `comando` a la salida de error.
- `comando &> todo.txt`: redirecciona tanto la salida estandar 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 estandar 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 estandar (`stdin`): `comando < ficheroConDatos.txt`.
- `comando` tomara como datos de entrada el contenido del fichero `ficheroConDatos.txt`
- Esto incluye los saltos de lineas, por lo que, por cada salto de linea se alimentara un `read`.

Tuberias

- Hasta ahora, redireccionamos entrada/salida comandos a partir de ficheros.
- Tuberias: redireccionar entrada/salida comandos entre si, sin usar ficheros.
- Sintaxis: `comando1 | comando2` la entrada de `comando2` sera tomada de la salida de `comando1` (salida estandar o de error)
- Se pueden encadenar mas 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`:

```
i72jivem@VTS3:~/PAS/p1$ echo "Esto es una prueba"
Esto es una prueba
i72jivem@VTS3:~/PAS/p1$ echo "Esto es una prueba" > f1
i72jivem@VTS3:~/PAS/p1$ cat f1
Esto es una prueba
i72jivem@VTS3:~/PAS/p1$ echo "Esto es una prueba" | tee f1
Esto es una prueba
i72jivem@VTS3:~/PAS/p1$ cat f1
Esto es una prueba
i72jivem@VTS3:~/PAS/p1$ echo "Esto es una prueba" | tee -a f1
Esto es una prueba
i72jivem@VTS3:~/PAS/p1$ 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 mas de una linea de texto. La sintaxis es la siguiente:

```
i72jivem@VTS3:~/PAS/p1$ cat << secuenciaSalida
> hola
> que
> tal
> secuenciaSalida
hola
que
tal
```

- Caracteristicas:
  - La entrada se va almacenando. Se van creando nuevas lineas pulsando la tecla *Intro*.
  - Se acaban de recibir datos cuando se detecta la cadena de texto que se selecciono 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 mas ficheros de texto.

```
i72jivem@VTS3:~/PAS/p1$ cat informacion.sh
#!/bin/bash
echo "Bienvenido $USER!, tu identificador es $UID."
echo "Esta es la shell numero $SHLVL, que lleva $SECONDS arrancada."
echo "La arquitectura de esta maquina es $MACHTYPE y el nombre es es $HOSTNAME"
i72jivem@VTS3:~/PAS/p1$ cat informacion.sh parametros.sh
#!/bin/bash
echo "Bienvenido $USER!, tu identificador es $UID."
echo "Esta es la shell numero $SHLVL, que lleva $SECONDS arrancada."
echo "La arquitectura de esta maquina es $MACHTYPE  y el nombre es es $HOSTNAME"
#!/bin/bash
echo "$#; $0; $1; $2; $*; $@"
```

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

- *head* y *tail*:
  - Muestran las primeras o las ultimas n lineas de un fichero.

```
i72jivem@VTS3:~/PAS/p1$ head -2 informacion.sh
#!/bin/bash
echo "Bienvenido $USER!, tu identificador es $UID."
i72jivem@VTS3:~/PAS/p1$ tail -1 informacion.sh
echo "La arquitectura de esta maquina es $MACHTYPE y el cliente de terminal es $TERM"
```

- *wc*: muestra el numero de lineas, palabras o caracteres de uno o varios ficheros:

```
i72jivem@VTS3:~/PAS/p1$ wc -l informacion.sh  # l neas
4 informacion.sh
i72jivem@VTS3:~/PAS/p1$ wc -m informacion.sh  # caracteres
219 informacion.sh
i72jivem@VTS3:~/PAS/p1$ wc -w informacion.sh  # palabras
34 informacion.sh
i72jivem@VTS3:~/PAS/p1$ 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 que caracter son distintos.

```
i72jivem@VTS3:~/PAS/p1$ cmp numeroRango.sh numeroRango1If.sh
numeroRango.sh numeroRango1If.sh son distintos: byte 95, linea 5
```

- *sort* `[fichero]`: ordena la entrada estandar o un fichero.
  - *sort*: ordena entrada estandar por orden alfabetico.
  - *sort -r*: ordena entrada estandar por orden inverso.
  - *sort -n*: ordena entrada estandar por orden numerico.
  - *sort -k 3*: cambia la clave de ordenacion a la tercera columna (por defecto, primera columna).

Comando `sort`

```
i72jivem@VTS3:~/PAS/p1$ echo -e "18\n017\n9" | sort
017
18
9
i72jivem@VTS3:~/PAS/p1$ echo -e "18\n017\n9" | sort -r
9
18
017
i72jivem@VTS3:~/PAS/p1$ echo -e "18\n017\n9" | sort -n
9
017
18
i72jivem@VTS3:~/PAS/p1$ echo -e "18\n017\n9" | sort -nr
18
017
9
i72jivem@VTS3:~/PAS/p1$ echo -e "18 1\n017 2\n9 3" | sort -n -k 2
18 1
017 2
9 3
i72jivem@VTS3:~/PAS/p1$ echo -e "18 1\n017 2\n9 3" | sort -n -k 1
9 3
017 2
18 1
i72jivem@VTS3:~/PAS/p1$ echo -e "1\t2\n2\t-1" | sort  -nk2
2       -1
1       2
```

Comando `grep`

- *`grep [opciones] patron [fichero(s)]`*: filtra el texto de un(os) fichero(s), mostrando unicamente las lineas que cumplen un determinado patron.
  - `-c`: cuenta el numero de lineas con el patron.
  - `-l`: muestra el nombre de los ficheros que contienen el patron.
  - `-i`: *case insensitive* (no sensible a mayusculas).
  - Tambien admite la entrada estandar (`stdin`).

```
i72jivem@VTS3:~/PAS/p1$ grep ^c *
case.sh:case $x in
i72jivem@VTS3:~/PAS/p1$ grep -l ^c *
case.sh
i72jivem@VTS3:~/PAS/p1$ grep -c ^c *
arrayFor.sh:0
backup.sh:0
case.sh:1
...
i72jivem@VTS3:~/PAS/p1$ ls * | grep ^c
case.sh
comillas.sh
copiaFstab.sh
```

Comando `grep`

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

```
i72jivem@VTS3:~/PAS/p1$ ls * | grep s\.sh$
comillas.sh
ejemploFor2Bis.sh
ejemploForImpFichScripts.sh
ejemploForListarFicheros.sh
operaciones.sh
```

```
parametros.sh
i72jivem@VTS3:~/PAS/p1$ ls * | grep ^ejemplo.or
ejemploFor1.sh
ejemploFor2Bis.sh
ejemploFor2.sh
ejemploForArg.sh
ejemploForImpFichScripts.sh
ejemploForListarFicheros.sh
ejemploForTipoC.sh
```

Comando `find`

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

```
i72jivem@VTS3:~/PAS/p1$ find ~ -name "*.sh"
/home/i72jivem/PAS/p1/saludaUsuario.sh
/home/i72jivem/PAS/p1/operaciones.sh
/home/i72jivem/PAS/p1/backup.sh
```

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

```
i72jivem@VTS3:~/PAS/p1$ find ~ -size 1024
/home/i72jivem/.mozilla/firefox/xlnw0cbr.Usuario predeterminado/cookies.sqlite
/home/i72jivem/.mozilla/firefox/fkyp01n6.isa/cookies.sqlite.bak
/home/i72jivem/.mozilla/firefox/gp04rnjj.default/cookies.sqlite
```

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

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

```
i72jivem@VTS3:~/PAS/p1$ find ~ -user i72jivem
/home/i72jivem
/home/i72jivem/.bashrc
```

- `basename fichero [.ext]`: Devuelve el nombre de un fichero sin su carpeta [y sin su extension].
- `dirname fichero`: Devuelve la carpeta donde se aloja un fichero.

```
i72jivem@VTS3:~/PAS/p1$ basename "/home/i72jivem/PAS/p1/recorrido.sh"
recorrido.sh
i72jivem@VTS3:~/PAS/p1$ basename "/home/i72jivem/PAS/p1/recorrido.sh" .sh
recorrido
i72jivem@VTS3:~/PAS/p1$ dirname "/home/i72jivem/PAS/p1/recorrido.sh"
/home/i72jivem/PAS/p1/
```

Comando `stat`

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

```
i72jivem@VTS3:~/PAS/p1$ stat comillas.sh
  Fichero:  comillas .sh
  Tamano: 212                   Bloques: 16         Bloque E/S: 131072 fichero regular
Dispositivo: 1dh/29d            Nodo-i: 6141310     Enlaces: 1
Acceso: (0744/-rwxr--r--)  Uid: (97710/i72jivem)   Gid: (  700/    upi0)
      Acceso: 2023-02-17 17:06:13.090143000 +0100
Modificacion: 2023-02-16 09:21:39.000000000 +0100
      Cambio: 2023-02-16 09:21:42.139536000 +0100
    Creacion: -
```

- `stat -c %a fichero`: nos permite personalizar la salida y obtener diferentes propiedades sobre un fichero[2].

```
i72jivem@VTS3:~/PAS/p1$ stat -c "Permisos: %a. Tipo fichero: %F" comillas.sh
Permisos: 744. Tipo fichero: fichero regular
```

Comando `tr`

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

```
i72jivem@VTS3:~/PAS/p1$ echo TIERRA | tr 'R' 'L'
TIELLA
```

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

```
i72jivem@VTS3:~/PAS/p1$ echo TIERRA | tr -d R
TIEA
i72jivem@VTS3:~/PAS/p1$ echo TIERRA | tr -d RT
IEA
```

Expansion de llaves

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

```
i72jivem@VTS3:~/PAS/p1$ 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 combinara `cadena1` con `c1`, `c2`, `c3`...
- {c1..c2} permite especificar el rango de caracteres desde `c1` hasta `c2`:

```
i72jivem@VTS3:~/PAS/p1$ 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
i72jivem@VTS3:~/PAS/p1$ echo {1..8}
1 2 3 4 5 6 7 8
i72jivem@VTS3:~/PAS/p1$ echo {1..3}{a..c}
1a 1b 1c 2a 2b 2c 3a 3b 3c
```

Recorriendo ficheros

- Un ejemplo de redireccion de comandos util para recorrer ficheros:

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

- Explica que esta sucediendo.
- *Cuidado*: la entrada esta redirigida durante todo el bucle (no podremos hacer `read` dentro del bucle).
- ¿Como lo hariamos con un `for` sin usar tuberias?

[2] `man stat` para mas informacion.

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 estan en la variable de entorno IFS y por defecto son el espacio, el tabulador y el salto de linea.

```
i72jivem@VTS3:~/PAS/p1$ array=($(echo "1 2 3"))
i72jivem@VTS3:~/PAS/p1$ echo ${array[0]}
1
i72jivem@VTS3:~/PAS/p1$ echo ${array[1]}
2
i72jivem@VTS3:~/PAS/p1$ echo ${array[2]}
3
i72jivem@VTS3:~/PAS/p1$ array=($(echo -e "1\t2\n3"))
i72jivem@VTS3:~/PAS/p1$ echo ${array[0]}
1
i72jivem@VTS3:~/PAS/p1$ echo ${array[1]}
2
i72jivem@VTS3:~/PAS/p1$ 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):

```
i72jivem@VTS3:~/PAS/p1$ array=($(echo -e "El uno\nEl dos\nEl tres"))
i72jivem@VTS3:~/PAS/p1$ echo ${array[0]}
El
i72jivem@VTS3:~/PAS/p1$ echo ${array[1]}
uno
```

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

```
i72jivem@VTS3:~/PAS/p1$ OLDIFS=$IFS
i72jivem@VTS3:~/PAS/p1$ IFS=$'\n'
i72jivem@VTS3:~/PAS/p1$ array=($(echo -e "El uno\nEl dos\nEl tres"))
i72jivem@VTS3:~/PAS/p1$ echo ${array[0]}
El uno
i72jivem@VTS3:~/PAS/p1$ echo ${array[1]}
El dos
i72jivem@VTS3:~/PAS/p1$ IFS=$OLDIFS
```

5. Referencias

Referencias

Referencias

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

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

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