2. Programación de Scripts

2.1 Línea de comandos

2.1.1 Interprete de comandos

El shell se inicia cuando accedemos a nuestra cuenta y proporciona:

- un intérprete de comandos
- un entorno de programación
  Para salir del shell o volver al shell anterior: exit o Ctrl-D

Comandos externos o internos al shell

Comandos externos:

- son programas ajenos al shell
- cuando se lanzan inician un nuevo proceso
- se buscan en los directorios indicados en la variable PATH
- por ejemplo: `ls, cat, mkdir`, etc

Comandos internos:

- se ejecutan en el mismo proceso del shell, sin lanzar un nuevo proceso
- por ejemplo: `alias, cd, pwd, eval, exec, bg`, etc.

Para saber si un comando es externo o interno en bash usar el comando interno `type`

Comandos y parámetros

Un comando consta del nombre del comando y de cero, uno o más parémtros o argumentos. Para que interprete los espacios como parte de un parámetro usar comillas simples o dobles:

```
$ echo 'hola          amigo'
```

- Comando ⟶ `echo`
- Argumento 1 ⟶ `hola          amigo`

Gestión de comandos

- `su, sudo` - permiten ejecutar comandos con la identidad de otro usuario o como administrador
- `alias` - Permiten crear alias de comandos complejos (para eliminarlos `unalias`) `$ alias l='ls -la'`
- `history` - muestra una lista con los últimos comandos ejecutados y permite reejecutarlos

Manejo del historial de comandos

Comando	Descripción
<up-arrow>/<down-arrow>	Comando anterior/posterior
!!	Último comando ejecutado
!*n*	*n*-ésimo comando del historial
!-*n*	*n* comandos hacia atrás
!*cadena*	Último comando ejecutado que empieza por cadena
!?*cadena*	Último comando ejecutado que contiene cadena
^*cadena1*^*cadena2*	Ejecuta el último comando cambiando *cadena1* por *cadena2*
Ctrl-r	Busca hacia atrás en el historial
fc	Permite ver, editar y reejecutar comandos del historial

2.1.2 Variables de Shell

Dos tipos:

- **variables locales**: visibles sólo desde el shell actual
- **variables globales**: de entorno o exportadas: visibles en otros shells

Para ver las variables definidas:

- Para ver todas las variables definidas en nuestra shell usar `set`
- Para ver las variables de entorno definidas usar `env o printenv`

El nombre de las variables debe:

- empezar por una letra o _
- seguida por cero o mas letras, números o _ (sin espacios en blanco)

Uso de las variables

- Asignar un valor: *nombre_variable=valor*

```
$ un_numero=15
$ nombre="Pepe Pota"
```

- Acceder a las variables: *$nombre_variable* o *${nombre_variable}*

```
$ echo $nombre
Pepe Pota
```

- Número de caracteres de una variable

```
$ echo ${#un_numero}
2
```

- Eliminar una variable: *unset nombre_variable*

```
$ unset nombre
$ echo ${nombre}mo
mo
```

- Variables de solo lectura: *readonly nombre_variable*

```
$ readonly nombre
$ unset nombre
bash: unset: nombre: cannot unset: readonly variable
```

Variables globales, de entorno o exportadas

Cada shell se ejecuta en un **entorno** (environment). El entorno de ejecución especifica **aspectos del funcionamiento del shell a través de la definición de variables de entorno** (=globales=exportadas) Algunas variables de entorno predefinidas son:

Nombre	Propósito
HOME	directorio base del usuario
SHELL	shell por defecto
USER	el nombre de usuario
PS1/PS2	*prompts* primario y secundario
PWD	el directorio actual
PATH	el *path* para los ejecutables
LD_LIBRARY_PATH	el *path* para las librerías dinámicas
MANPATH	el *path* para las páginas de manual
LANG	aspectos de localización geográfica e idioma
LC_*	aspectos particulares de loc. geográfica e idioma

Para definir una nueva variable de entorno: `export`

```
$ a=1 ; b=2 ; c=3 ; d=4    # Define cuatro variables de shell
$ export d                 # Exporta la variable d
$ cat script.sh            # Ejemplo de script
  #!/bin/bash              # Dos variables serán parametros,
  echo "$1, $2, $c, $d"    # c será local y d será global
$ ./script.sh $a $b        # Ejecutamos el script
   --> 1, 2, , 4           # no tiene acceso a la variable local
```

La variable exportada será visible en los shell hijos (y para los scripts y procesos) que se creen a continuación

- el shell hijo crea una copia local de la variable y la usa
- las modificaciones de una copia no afectan a los demás shell

2.1.3 Expansiones del shell

Expansión de parámetros

Es la sustitución de las variables por su valor

```
$ A=Pepe
$ echo $A
Pepe
```

Expansión de nombres de ficheros

Es la sustitución de los comodines (wildcards), que nos permiten especificar múltiples ficheros al mismo tiempo. También se conoce como `glob`.

Carácter	Corresponde a
*	0 o más caracteres
?	1 carácter
[ ]	uno de los caracteres entre corchetes
[!] o [^]	cualquier carácter que no esté entre corchetes

`$ ls -l *html` # Lista los ficheros del directorio actual con terminación html

`$ ls -l [abd].html` # Implica `a.html`, `b.html` y `d.html`.[1]

Para referirnos a mayúsculas o minúsculas podemos usar los patrones:

- `[[:lower:]]`: corresponde a un carácter en minúsculas
- `[[:upper:]]`: corresponde a un carácter en minúsculas
- `[[:alpha:]]`: corresponde a un carácter alfabético
- `[[:digit:]]`: corresponde a un número

Expansión de comados

Permite que la salida de un comando reemplace el propio comando. Formato:

```
$(comando) o `comando`
```

```
$ echo date
date
$ echo `date`
Xov Xul 21 13:09:39 CEST 2005
$ echo líneas en fichero=$(wc -l fichero)
# wc -l cuenta el número de líneas en el fichero; el comando se
ejecuta y su salida se pasa al echo
```

Expansión de llaves

Permite generar strings arbitrarios

```
$ echo a{d,c,b}e
ade ace abe
```

Expansión de la tilde

```
cd ~              # Accedemos al nuestro HOME
cd ~root          # Accedemos al HOME de root
ls ~pepe/cosas/   # Vemos el contenido del directorio
                  cosas de pepe
```

Expansión aritmética

Permite evaluar expresiones aritméticas enteras

- se usa `$(( expresión ))` o `$[ expresión ]`
- `expresión` tiene una sintaxis similar a la del lenguaje C

  - permite operadores como ++, +=, &&,...
- También se puede usar `let` (usar comillas dobles si hay espacios)

  ```
  $ let numero=(numero+1)/2
  ```
- Ejemplos:

  ```
  $ echo $(((4+11)/3))                          --> 5
  $ numero=15 ; echo $((numero+3))              --> 18
  $ echo $numero                                --> 15
  $ echo $((numero+=4))                         --> 19
  $ echo $numero                                --> 19
  $ numero=$(((numero+1)/2)) ; echo $numero     --> 10
  ```

Eliminación del significado especial

- bash permite eliminar el significado de los caracteres especiales
- para ello se usan las comillas simples, dobles o \
- El caracter o caracteres que vengan a continuación simplemente se escriben

Carácter	Acción
'	el shell ignora todos los caracteres especiales contenidos entre un par de comillas simples
"	el shell ignora todos los caracteres especiales entre comillas dobles excepto $, ` y \
\	el shell ignora el carácter especial que sigue a \

Ejemplos:

`ls "/usr/bin/a*"` (el nombre del fichero contiene un asterisco)
`echo '$PATH'` (escribe literalmente $PATH)
`echo "$PATH"` (escribe el contenido de la variable)
`echo I\'m Pepe` (escribe literalmente la comilla)

2.1.4 Redirección de la entrada/salida

Cada proceso tiene asociados 3 ficheros para la E/S

Nombre	Descriptor de fichero	Destino por defecto
entrada estándar (*stdin*)	0	teclado
salida estándar (*stdout*)	1	pantalla
error estándar (*stderr*)	2	pantalla

- por defecto, un proceso toma su entrada de la entrada estándar, envía su salida a la salida estándar y los mensajes de error a la salida de error estándar

Para cambiar la entrada/salida se usan los siguientes caracteres:

Carácter	Resultado
`comando < fichero`	Toma la entrada de `fichero`
`comando > fichero`	Envía la salida de `comando` a `fichero`; sobreescribe cualquier cosa de `fichero`
`comando >> fichero`	Añade la salida de `comando` al final de `fichero`
`comando << etiqueta`	Toma la entrada para `comando` de las siguientes lineas, hasta una línea que tiene sólo `etiqueta`
`comando 2> fichero`	Envía la salida de error de `comando` a `fichero` (el 2 puede ser reemplazado por otro descriptor de fichero)
`comando 2>&1`	Envía la salida de error a la salida estándar (el 1 y el 2 pueden ser reemplazado por otro descriptor de fichero, p.e. `1>&2`)
`comando &> fichero`	Envía la salida estándar y de error a `fichero`
`comando1 | comando2`	pasa la salida de `comando1` a la entrada de `comando2` (*pipe*)

Ejemplos

- `cat < lista.ficheros | more`
  Muestra el contenido de `lista.ficheros` página a página (equivale a `more lista.ficheros`)
- `ls /kaka 2> /dev/null`
  Envía los mensajes de error al dispositivo nulo (a la *basura*)
- `> kk`
  Crea el fichero `kk` vacío
- `cat > entrada`
  Lee información del teclado, hasta que se teclea `Ctrl-D`; copia todo al fichero entrada
- `cat << END > entrada`
  Lee información del teclado, hasta que se introduce una línea con `END`; copia todo al fichero `entrada`
- `ls -l /tmp /kaka > salida 2> error`
  Redirige la salida estándar al fichero `salida` y la salida de error al fichero `error`
- `ls -l /tmp /kaka > salida.y.error 2>&1`
  Redirige la salida estándar y de error al fichero `salida.y.error`; el orden es importante:

  ```
  ls -l /tmp /kaka 2>&1 > salida.y.error
  ```

  no funciona, ¿por qué?
- `ls -l /tmp /kaka &> salida.y.error`
  Igual que el anterior
- `cat /etc/passwd > /dev/tty2`
  Muestra el contenido de `/etc/passwd` en el terminal `tty2`

Comandos útiles con pipes y redirecciones

**1. Comando tee:

- Copia la entrada estándar a la salida estándar y también al fichero indicado como argumento
- `ls -l | tee lista.ficheros | less`
  - Guarda la salida de `ls -l` en el archivo `lista.ficheros` y la muestra página por página en pantalla.
  - La opción `-a` de `tee` añade la salida al archivo en vez de sobrescribirlo.

2. El comando xargs:

- Permite pasar muchos argumentos a otros comandos leyendo desde la entrada estándar y ejecutando el comando varias veces.

- Ejemplo:
  - `locate README | xargs cat`
    - Busca ficheros llamados README y muestra su contenido con `cat`.
  - `locate README | xargs -I{} cp {} /tmp/`
    - Copia todos los archivos README encontrados al directorio `/tmp/`.
    - La opción `-I{}` reemplaza `{}` por cada nombre de archivo.

3. El comando exec:

- Ejecuta un programa reemplazando el shell actual, es decir, el nuevo programa toma el PID del shell y el shell deja de existir.
- Ejemplo:
  - `echo $$` muestra el PID del shell.
  - `exec sleep 20` hace que el shell actual se convierta en el proceso `sleep`.
- También puede usarse para redireccionar la entrada/salida de todos los comandos del shell:
  - `exec > /tmp/salida` envía toda la salida estándar al archivo.
  - `exec < /tmp/entrada` toma el archivo como entrada estándar.

2.1.5 Orden de evaluación

Desde que introducimos un comando hasta que se ejecuta, el shell ejecuta los siguientes pasos, y en el siguiente orden:

1. Redirección E/S
2. Sustitución (expansión) de variables: reemplaza cada variable por su valor
3. Sustitución (expansión) de nombres de ficheros: sustituye los comodines por los nombres de ficheros

```
$ star=\* ; pipe=\|
$ ls -d $star
cuatro  dos  tres  uno
$ cat uno $pipe more
cat: |: Non hai tal ficheiro ou directorio
cat: more: Non hai tal ficheiro ou directorio
```

En el segundo caso, primero se mira si hay redirección E/S y no hay; se sustituye `$pipe` por su valor y finalmente se miran los comodines (que no hay) así `$pipe` se toma por el carácter "|", no por la redirección

Comando eval:
Evalúa la línea de comandos 2 veces:

- La primera hace todas las substituciones
- La segunda ejecuta el comando

```
$ pipe=\|
$ eval cat uno $pipe more
Este es el fichero uno
...
$
```

- En la primera pasada reemplaza `$pipe` por "|"
- En la segunda ejecuta el comando `cat uno | more`

2.1.6 Ficheros de inicialización de Bash

Bash puede iniciarse de tres maneras, y según el modo, lee diferentes ficheros de inicio donde el usuario configura variables de entorno, alias, el prompt, el path, etc.

1. **Login shell interactivo**
   - Se inicia al entrar al sistema con usuario y contraseña, usando `su -`, o ejecutando `bash --login`.
   - Archivos leídos al iniciar:
     - `/etc/profile`
     - El primero que exista de: `~/.bash_profile`, `~/.bash_login` o `~/.profile`
   - Al salir se ejecuta: `~/.bash_logout`
2. **Non-login shell interactivo**
   - Se inicia al abrir una nueva ventana de comandos sin login/password, ejecutando bash sin opciones, o usando `su`.
   - Archivos leídos:
     - `/etc/bash.bashrc`
     - `~/.bashrc`
   - Al salir no se ejecuta ningún archivo de cierre.
3. **Shell no interactivo**
   - Se inicia, por ejemplo, al ejecutar un script. La variable `$PS1` no está disponible.
   - Archivo leído:
     - El definido por la variable `BASH_ENV`

2.2 Scripts de administración

2.2.1 Programación Shell-Script

Script o programa shell: fichero de texto conteniendo comandos externos e internos, que se ejecutan línea por línea. El programa puede contener, además de comandos:

1. variables

2. constructores lógicos (if...then, AND, OR, etc.) y lazos (while, for, etc.)
3. funciones
4. comentarios

Ejecución de un script

Los scripts deben empezar por el número mágico `#!` seguido del programa a usar para interpretar el script:

- `#!/bin/bash` - script de bash
- `#!/bin/sh` - script de shell
- `#!/usr/bin/env python3` - script de python3
- `#!/usr/bin/awk -f` - script de awk

Las formas usuales de ejecutar un script son:

```shell
chmod +x helloworld
./helloworld

bash helloworld

 . helloworld
source helloworld
```

Paso de parámetros

Variable	Uso
$0	el nombre del script
$1 a $9	parámetros del 1 al 9
${10}, ${11},...	parámetros por encima del 10
${a:-b}	si no existe $a se usa el valor b
$#	número de parámetros
$*, $@	todos los parámetros

El comando `shift` desplaza los parámetros hacia la izquierda el número de posiciones indicado:

```
$ cat parms2.sh
#!/bin/bash
echo $#
echo $*
echo "$1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11}"
shift 9
echo $1 $2 $3
echo $#
echo $*
$ bash parms2.sh a b c d e f g h i j k l
12
a b c d e f g h i j k l
a b c d e f g h i j k
j k l
3
j k l
```

Uso de arrays

```
#!/bin/bash
name=( "cero" "uno" "dos" "tres" )
num=( 1 2 3 4 )
echo ${name[0]}
echo ${num[0]}
for i in "${name[@]}" ; do
   echo $i
done
for i in "${num[@]}" ; do
   echo $i
done
echo "total= ${#name[@]}"
```

Salida

`exit` permite salir del script en cualquier punto

2.2.2 Entrada/Salida

Es posible leer desde la entrada estándar o desde fichero usando read y redirecciones:

```
#!/bin/bash
echo -n "Introduce algo: "
read x
echo "Has escrito $x"
echo -n "Escribe 2 palabras: "
read x y
echo "Primera palabra $x; Segunda palabra $y"
```

Si queremos leer o escribir a un fichero utilizamos redirecciones:

```
echo $X > fichero
read X < fichero
```

Este último caso lee la primera línea de fichero y la guarda en la variable X. Si queremos leer un fichero línea a línea podemos usar while:

```
#!/bin/bash
# FILE: linelist
# Usar: linelist filein fileout
# Lee el fichero pasado en filein y
# lo salva en fileout con las lineas numeradas
count=0
while read BUFFER
do
   count=$((++count))
   echo "$count $BUFFER" » $2
done < $1
```

2.2.3 Test

Los comandos que se ejecutan en un shell tienen un código de salida, que se almacena en la variable `$?`

- si `$?` es 0 el comando terminó bien
- si `$?` es > 0 el comando terminó mal

Podemos chequear la salida de dos comandos mediante los operadores && (AND) y || (OR)

```
comando1 && comando2
comando2 sólo se ejecuta si comando1 acaba bien
comando1 || comando2
comando2 sólo se ejecuta si comando1 falla
```

Comandos `true` y `false`: devuelven 0 y 1, respectivamente

2.2.4 Estructura if...then...else

```
if comando1
then
  ejecuta otros comandos
elif comando2
then
  ejecuta otros comandos
else
  ejecuta otros comandos
fi
```

Comando test

Notar que `if` sólo chequea el código de salida de un comando, no puede usarse para comparar valores: para eso se usa el comando `test`

El comando test permite:

- chequear la longitud de un string
- comparar dos strings o dos números
- chequear el tipo de un fichero
- chequear los permisos de un fichero
- combinar condiciones juntas

```
test expresión

[ expresión ][3]

[[ expresión ]] (comando test extendido)
```

Si la expresión es correcta `test` devuelve un código de salida 0, si es falsa, devuelve 1:

```
if [[ "$1" = "hola" ]]
then
  echo "Hola a ti también"
else
  echo "No te digo hola"
fi
if [[ $2 ]]
then
  echo "El segundo parámetro es $2"
else
  echo "No hay segundo parámetro"
fi
```

2.2.5 Expresiones

Expresión	Verdadero si
string	el string es no nulo ("")
-z *string*	la longitud del string es 0
-n *string*	la longitud del string no es 0
string1 = *string2*	los strings son iguales
string1 != *string2*	los strings son distintos

Expresión	Verdadero si
int1 -eq *int2*	los enteros son iguales
int1 -ne *int2*	los enteros son distintos
int1 -gt *int2*	*int1* mayor que *int2*
int1 -ge *int2*	*int1* mayor o igual que *int2*
int1 -lt *int2*	*int1* menor que *int2*
int1 -le *int2*	*int1* menor o igual que *int2*

Expresión	Verdadero si
-e *file*	*file* existe
-r *file*	*file* existe y es legible
-w *file*	*file* existe y se puede escribir
-x *file*	*file* existe y es ejecutable
-f *file*	*file* existe y es de tipo regular
-d *file*	*file* existe y es un directorio
-c *file*	*file* existe y es un dispositivo de caracteres
-b *file*	*file* existe y es un dispositivo de bloques
-p *file*	*file* existe y es un pipe
-S *file*	*file* existe y es un socket
-L *file*	*file* existe y es un enlace simbólico
-u *file*	*file* existe y es *setuid*
-g *file*	*file* existe y es *setgid*
-k *file*	*file* existe y tiene activo el *sticky bit*
-s *file*	*file* existe y tiene tamaño mayor que 0

Operadores lógicos de test

Expresión	Propósito
!	niega el resultado de una expresión
-a	operador AND
-o	operador OR
\(*expr* \)	agrupación de expresiones; los paréntesis tienen un significado especial para el shell, por lo que hay que *escaparlos*

```
$ test -f /bin/ls -a -f /bin/ll ; echo $?  -->  1
$ [ ! -w /etc/passwd ] ; echo $?  -->  0
$ [ $$ -gt 0 -a \( $$ -lt 5000 -o -w file \) ]
```

2.2.6 Control de flujo

Estructura case

```
case valor in
  patrón_1)
    comandos si value = patrón_1
    comandos si value = patrón_1 ;;
  patrón_2)
    comandos si value = patrón_2 ;;
  *)
    comandos por defecto ;;
esac
```

```
#!/bin/bash
echo -n "Respuesta:" read RESPUESTA
case $RESPUESTA in
  S* | s*)
    RESPUESTA="SI";;
  N* | n*)
    RESPUESTA="NO ";;
  *)
    RESPUESTA="PUEDE";;
esac
echo $RESPUESTA
```

Lazos for

```
for var in lista
do
  comandos
done
```

`var` toma los valores de la lista

```
LISTA="10 9 8 7 6 5 4 3 2 1"
for var in $LISTA
do
  echo $var
done
```

Bucle while

```
while comando
do
  comandos
done
```

```
while [[ $1 ]]
do
  echo $1
  shift
done
```

Bucle until

```
until comando
do
  comandos
done
```

```
until [[ "$1" = "" ]]
do
  echo $1
  shift
done
```

Break y continue

- `break` permite salir de un lazo.
- Con break n especificamos el número de lazos que queremos salir
- `continue` permite saltar a la siguiente iteración

```
# Imprime el contenido de los ficheros hasta que
# encuentra una línea en blanco
for file in $*
do
  while read buf
  do
    if [[ -z "$buf" ]]
    then
      break 2
    fi
    echo $buf
  done <$file
done
```

```
# Muestra un fichero pero no las líneas de más
# de 80 caracteres
while read buf
do
  cuenta=`echo $buf | wc -c`
  if [[ $cuenta -gt 80 ]]
  then
    continue
  fi
  echo $buf
done < $1
```

2.2.7 Funciones

```
funcion() {
  comandos
}
```

```
$ cat funcion1.sh
#!/bin/bash
funcion1()
{
  echo "Parámetros pasados a la función: $*"
  echo "Parámetro 1: $1"
  echo "Parámetro 2: $2"
}
# Programa principal
funcion1 "hola" "que tal estás" adios
$
$ bash funcion1.sh
Parámetros pasados a la función: hola que tal estás adios
Parámetro 1: hola
Parámetro 2: que tal estás
```

`return` Una función de bash puede devolver dos posibles valores: 0 y 1.

2.2.8 Otros comandos

Resumen de los puntos principales:

- **Comando** `wait`:
  Permite esperar a que procesos en segundo plano (background) finalicen antes de continuar el script.
  Ejemplo:

  ```shell
  sort $largefile > $newfile &
  wait
  # luego puedes usar $newfile
  ```

  Si lanzas varios procesos en background, puedes guardar el PID de cada uno usando `$!` y esperar a cada proceso por separado:

  ```shell
  sort $largefile1 > $newfile1 & SortPID1=$!
  sort $largefile2 > $newfile2 & SortPID2=$!
  wait $SortPID1
  wait $SortPID2
  ```

- **Comando** `trap`:
  Permite capturar señales del sistema operativo (por ejemplo, Ctrl+C) y ejecutar una función para terminar el script limpiamente (borrar temporales, mostrar mensajes, etc.).
  Ejemplo:

```shell
trap "echo 'Me has matado!!!'" SIGINT SIGQUIT
while true; do true; done
```

Las señales comunes incluyen:

- `SIGINT` (interrupción de teclado, Ctrl+C)
- `SIGTERM` (terminación)
- `SIGQUIT` (salida de teclado)
- `SIGKILL` (no puede ser capturada ni ignorada)
- `SIGHUP` (cuelgue del terminal)

- **Referencias indirectas**:
  Permiten usar el contenido de una variable como nombre de otra variable.
  Ejemplo simple:

```shell
a=letra
letra=z
# Referencia directa: $a => letra
# Referencia indirecta:
echo ${!a}   # Imprime z
```

También se puede hacer con `eval`:

```shell
dniPepe=23456789
nombre=Pepe
eval echo "DNI = \$dni${nombre}"  # Imprime DNI = 23456789
```

2.3 Expresiones regulares

Son patrones que se utilizan para buscar y manipular texto (por ejemplo, con comandos como `egrep`, `sed`, `awk`). Las expresiones regulares extendidas (ERE) añaden más capacidades a las básicas.

Comandos egrep y sed -r

- `egrep:` busca parones en ficheros de texto. Si el patrón se encuentra en una línea devuelve la línea indicando la coincidencia. En caso contrario no devuelve nada. `egrep` usa expresiones regulares extendidas, mientras que `grep` usa solo expresiones regulares básicas.

```shell
egrep 'patrón'
```

- `sed -r:` sustituye patrones por la cadena de texto que se le indica (la opción -r indica que se utilicen expresiones regulares extendidas). El formato básico de

utilización es:

```shell
sed -r 's/patrón/sustitución/g'
```

- Estos comandos se pueden usar con ficheros de texto o con cualquier otro de tipo de entrada, como la entrada estándar, tuberías, etc.
- Pueden usar comillas dobles o simples. Las comillas dobles permiten sustituir variables y ejecutar comandos, mientras que las comillas sim- ples no.

Ejemplos:

- `egrep 'unix' tmp.txt`
  busca en el fichero `tmp.txt` las líneas que contienen la palabra `unix`
- `egrep '[Uu]nix' tmp.txt`
  busca las líneas que contienen `unix` o `Unix`
  (los corchetes indican que hay varias posibilidades de coincidencia)
- `egrep 'hel.' tmp.txt`
  busca las líneas que contienen `hel` seguido de cualquier carácter
  (el punto indica cualquier caracter)
- `egrep 'ab*c' tmp.txt`
  localiza las cadenas que empiecen por `a`, que continúen con 0 o más `b`,
  y que sigan con una `c`, por ejemplo: `abbbc` o `aaacb`, pero no `axc` o `cba`
  (* indica repetición de cero, una, dos o más veces)
- `egrep 't[^aeiouAEIOU][a-zA-Z]*' tmp.txt`
  localiza las cadenas que empiecen por `t`, seguido de algún carácter no vocálico y de 0 o más apariciones de letras
- `sed -r "s/inicio/fin/g" tmp.txt`
  sustituye la cadena `inicio` por `fin`
- `sed -r "s/[Ww]indows/Linux/g" tmp.txt`
  sustituye las cadenas `Windows` y `windows` por Linux

Importante: no debemos confundir las expresiones regulares con los como- dines para la sustitución de nombres de ficheros (glob). Por ejemplo, el asterisco * como comodín indica cualquier cadena en nombres de fichero, mientras que como expresión regular es repetición

Si no especificamos fichero, egrep y sed usan la entrada estándar. Las comillas dobles permiten utilizar variables y comandos dentro de las expresiones regulares.

```
$ a=5
$ sed -r "s/x/$a/g" fichero.txt → 5
$ sed -r 's/x/$a/g' fichero.txt → $a
$ sed -r "s/x/`date`/g" fichero.txt → mié ago 3 09:15:30
$ sed -r 's/x/`date`/g' fichero.txt → `date`
```

2.3.1 Expresiones regulares básicas

ER	concuerda con
.	cualquier carácter
[ ]	cualquiera de los caracteres entre corchetes, p.e. [abc] concuerda con a, b o c; [a-z] concuerda con cualquier letra minúscula
[^]	cualquier carácter que no esté entre corchetes
^	principio de línea
$	final de línea
*	0 o más ocurrencias de la expresión regular anterior
\(\)	permite agrupar ER
\	*escapa* un metacarácter

Constructor	Propósito
\{n\}	concuerda con exactamente n ocurrencias de la ER previa
\{n,\}	concuerda con al menos n ocurrencias de la ER previa
\{n, m\}	concuerda con entre n y m ocurrencias de la ER previa

2.3.2 Expresiones regulares extendidas

- **Concordancia literal:**
  Casi todos los caracteres coinciden consigo mismos.
  Ejemplo: `egrep "a"` encuentra cualquier línea con una "a".
- **Metacaracteres:**
  Son caracteres especiales que modifican el significado del patrón:
  - `.` : cualquier carácter
  - `[ ]` : cualquiera de los caracteres entre corchetes (ej: `[abc]` coincide con "a", "b", o "c")
  - `[^ ]` : cualquier carácter que NO esté entre corchetes (ej: `[^abc]` no "a", "b" ni "c")
  - `^` : principio de línea
  - `$` : final de línea

- `*` : cero o más repeticiones del carácter anterior
- `+` : una o más repeticiones del carácter anterior
- `?` : cero o una repetición del carácter anterior
- `( )` : agrupa expresiones
- `|` : OR (alternativa)
- `{n}` : exactamente n repeticiones
- `{n,}` : al menos n repeticiones
- `{n,m}` : entre n y m repeticiones
- `\` : escapa (convierte metacaracter en literal)
- Dentro de `[ ]`, los metacaracteres pierden su significado especial.

Ejemplos de uso

- `a..c` : "a" seguido de dos caracteres cualquiera y luego "c" (ejemplo: "azxc")
- `[abc]` : "a", "b" o "c"
- `[^abc]` : cualquier carácter menos "a", "b", "c"
- `[a-z]` : cualquier minúscula
- `^abc` : líneas que empiezan por "abc"
- `abc$` : líneas que acaban en "abc"
- `ab*c` : "a" seguido de cero o más "b" y luego "c": "ac", "abc", "abbc", etc.
- `b[cq]*e` : "b" seguido de cero o más "c" o "q", seguido de "e"
- `.*` : cualquier cadena
- `abc.*` : cadenas que contienen "abc" seguido de cualquier cosa
- `x(abc)*x` : "x", seguido de cero o más "abc", seguido de "x"
- `(a|b)c` : "ac" o "bc"
- `ab+c` : "a", una o más "b", "c": "abc", "abbc"
- `ab?c` : "a", cero o una "b", "c": "ac", "abc"
- `a{5}` : cinco "a" seguidas
- `.{5,}` : al menos cinco caracteres
- `^#.*\.$` : línea que empieza por "#" y termina por "."

Etiquetado y referencias

- Si usas paréntesis `( )`, puedes referenciar lo que has encontrado con `\1`, `\2`, etc.
  - Ejemplo:
    - `(.)oo\1` encuentra "moom" o "noon" (lo que está en `(.)` se repite después de "oo")
    - `sed -r 's/(.)(.)/\2\1/g'` cambia "ab" por "ba", "xy" por "yx", etc.
- El carácter `&` en `sed` representa la cadena reconocida.
  - Ejemplo: `sed -r 's/a.*a/[reconocido: &]/g'` sustituye "axyza" por "[reconocido: axyza]".

Concordancia más larga

- Las ER buscan la coincidencia más larga posible.
  - Ejemplo: `egrep --color "a(.*)a"` encuentra "axaxa" en "yyyaxaxayyy".

Otros metacaracteres útiles

- `\n`, `\r`, `\t` : salto de línea, retorno de carro, tabulador
- `[:space:]` : cualquier espacio en blanco
- `[:alnum:]` : cualquier letra o número (`\w`)
- `[:digit:]` : cualquier número
- `[:alpha:]` : cualquier letra
- `\<`, `\>` : inicio y fin de palabra
- `\b`, `\B` : borde/punto interno de palabra

Ejemplos con estos metacaracteres

- `[[:upper:]]bc` : "Abc", pero no "abc"
- `\babc\b` : "abc" como palabra completa
- `\Babc\B` : "abc" dentro de una palabra

2.3.3 Opciones de los comandos grep y sed

- **grep** y **egrep** buscan patrones en archivos.
- **egrep** (o `grep -E`) permite usar expresiones regulares extendidas.

Opciones útiles:

- `-E` o `egrep`: expresiones regulares extendidas.
- `-R` o `rgrep`: búsqueda recursiva en directorios.
- `-i`: ignora mayúsculas/minúsculas.
- `-n`: muestra el número de línea.

sed (stream editor)

Formatos básicos:

- **Sustitución:**
  `sed -r 's/ER/reemplazo/flag' [fichero]`
- **Borrar línea:**
  `sed -r '/ER/d' [fichero]`
- **Reemplazar línea completa:**
  `sed -r '/ER/c\reemplazo' [fichero]`
- **Insertar antes/después:**
  - Antes: `sed -r '/ER/i\reemplazo' [fichero]`
  - Después: `sed -r '/ER/a\reemplazo' [fichero]`

Opciones importantes:

- `-i`: modifica el fichero original (edición in-place).

Flags comunes:

- `g`: aplica el cambio a todas las apariciones en la línea.
- Un número (ej. `5`): solo cambia la aparición número 5.
- `w fichero`: guarda líneas modificadas en otro archivo.

Ejemplos prácticos

- Reemplaza la quinta aparición de "stop" por "STOP":

```shell
sed 's/stop/STOP/5' fichero
```

- Sustituye "stop" por "STOP" y guarda las líneas modificadas en otro archivo:

```shell
sed -r 's/stop/STOP/w fich2' fichero
```

- Borra las líneas que contienen "jaime":

```shell
sed -r '/jaime/d' amigos
```

- Cambia líneas que contienen "jaime" por "CAMBIADO":

```shell
sed -r '/jaime/c\CAMBIADO' amigos
```

- Inserta una línea antes de las que contienen "jaime":

```shell
sed -r '/jaime/i\APARICION' amigos
```

Comandos desde fichero (-f):
Puedes agrupar varios comandos de sed en un archivo y aplicarlos todos de una vez:

```shell
cat file.sed
3,10 {
  s/[Ll]inux/GNU\/Linux/g
  s/samba/Samba/g
}
sed -f file.sed fichero
```

2.4 Procesamiento de textos

Existe una serie de comandos simples para realizar operaciones concretas sobre ficheros de texto. Tºambién se conocen como filtros: obtienen su entrada de la entrada estándar (o un fichero) y envían la salida a la salida estándar:

head, tail	muestran el principio/final de un fichero
tac, rev	muestran el fichero al revés por líneas/caracteres de la línea
wc	cuenta el número de líneas, palabras y bytes de un fichero
sort	ordena las líneas alfabéticamente
tr	borra o reemplaza caracteres
uniq	elimina líneas sucesivas repetidas
cut	selecciona campos o columnas de un fichero
paste	combina texto de varios ficheros por líneas
join	combina varios ficheros por campos
split	divide un fichero en ficheros más pequeños
nl	añade números de línea
expand	convierte TABs en espacios
fmt	formatea párrafos
od	muestra un fichero en diferentes formatos

1) head, tail

- **head**: muestra el principio de un fichero.
- **tail**: muestra el final de un fichero.
- Opciones comunes:
  - `-n N` o `-N`: muestra las primeras/últimas N líneas.
  - `-c N`: muestra los primeros/últimos N bytes.
  - `-v`: añade una cabecera con el nombre del fichero.
- Opciones adicionales de **tail**:
  - `-f`: sigue mostrando nuevas líneas conforme se añaden al archivo (ideal para logs).
  - `--retry`: con `-f`, sigue intentando abrir el fichero si no existe o es inaccesible.
- Ejemplo:

```shell
head -n 2 -v quijote.txt
tail -n 2 -v quijote.txt
```

2) tac, rev

- **tac**: imprime el fichero desde la última línea a la primera (opuesto a `cat`).
- **rev**: invierte el contenido de cada línea del fichero.
- Ejemplo:

```shell
tac quijote.txt
rev quijote.txt
```

3) wc

- Muestra estadísticas de un fichero: número de líneas, palabras y bytes.

- Opciones:
  - `-l`: solo líneas.
  - `-w`: solo palabras.
  - `-c`: solo bytes.
  - `-L`: longitud de la línea más larga.
- Ejemplo:

```shell
wc quijote.txt
wc -l quijote.txt
wc -w quijote.txt
wc -c quijote.txt
```

4) sort

- Ordena líneas de texto.
- `-n`: ordena numéricamente.
- `-k POS1[,POS2]`: ordena por campos (el primero es el campo 1).
- `-f`: ignora mayúsculas/minúsculas.
- Ejemplo:
  - `sort nombres.txt`: ordena alfabéticamente.
  - `sort -f -k 2,2 nombres.txt`: ordena por el segundo campo, ignorando mayúsculas.

5) tr

- Borra o reemplaza caracteres.
- `-d`: borra caracteres.
- `-s`: sustituye grupos repetidos por uno solo.
- Ejemplos:
  - `tr 'a-z' 'A-Z' < file`: convierte minúsculas en mayúsculas.
  - `tr -d ' ' < file`: elimina espacios.

6) uniq

- Elimina líneas consecutivas idénticas.
- `-d`: muestra solo las duplicadas.
- `-u`: muestra solo las únicas.
- `-c`: cuenta ocurrencias.
- `-i`: ignora mayúsculas/minúsculas.
- Ejemplo:
  - `uniq nombres.txt`: elimina duplicados sucesivos.
  - `uniq -c nombres.txt`: muestra cada línea y cuántas veces aparece.

7) cut

- Selecciona partes de líneas (por bytes, caracteres o campos).
- `-b`, `-c`, `-f`: corta por bytes, caracteres o campos respectivamente.
- `-d`: define delimitador (por defecto, TAB).
- Ejemplo:
  - `cut -c 1-7 file.txt`: muestra los 7 primeros caracteres de cada línea.
  - `cut -d ' ' -f 1 file.txt`: muestra el primer campo (palabra) de cada línea.

8) paste

- Une líneas de varios archivos.
- `-s`: pega secuencialmente.
- `-d`: define delimitador (por defecto, TAB).
- Ejemplo:
  - `paste file1 file2`: une línea a línea.
  - `paste -d ' ' file1 file2`: une usando espacio como separador.

9) join

- Une líneas de dos archivos por campos comunes (como una "join" de SQL).
- `-j`, `-1`, `-2`: especifica campo de unión.
- `-t`: define separador.
- `-o`: formato de salida.
- `-v N`, `-a N`: muestra líneas no unidas.
- Ejemplo:
  - `join -j 2 file1 file2`: une por el segundo campo de ambos archivos.

10) split

- Divide un archivo en partes más pequeñas.
- `-l n`: cada parte tiene n líneas.
- `-b n`: cada parte tiene n bytes.
- `-d`: nombres de salida con números en vez de letras.
- Ejemplo:
  - `split -l 2 file.txt prefijo`: partes de 2 líneas.

11) nl

- Añade números de línea.
- `-s STRING`: separador entre número y texto.
- `-v n`: empieza la numeración en n.
- `-i n`: incrementa en n.
- `-b`, `-h`, `-f`: estilos de numeración (todas, solo no vacías, solo cabecera, etc.).
- Ejemplo:

    - `nl -s 'q ' file.txt`: añade números con "q" como separador.

12) expand

- Convierte TABs en espacios.
- `-t n`: cada TAB pasa a n espacios.
- `-i`: solo TABs al inicio de línea.
- Ejemplo:
    - `expand -t 2 file.c`: TABs se convierten en 2 espacios.

`unexpand` hace la operación contraria (espacios a TABs).

13) fmt

- Formatea párrafos, ajustando el ancho de línea.
- `-w n`: ancho de línea.
- `-u`: espaciado uniforme.
- `-c`: mantiene indentación.
- Ejemplo:
    - `fmt -w 45 -u file.txt`: líneas de máximo 45 caracteres, espaciado uniforme.

14) od

- Muestra archivos en formato octal, hexadecimal, decimal, ASCII, etc.
- `-t TIPO`: tipo de formato.
- `-A TIPO`: formato del offset.
- `-w BYTES`: bytes por línea.
- Ejemplo:
    - `od -t x -A x file.txt`: muestra en hexadecimal, offset en hexadecimal.

2.5 awk

Es un lenguaje diseñado para procesar datos basados en texto; el nombre AWK deriva de los apellidos de los autores

2.5.1 Funcionamiento básico

`awk` lee el fichero que se le pase como entrada (o la entrada estándar) línea a línea, y sobre cada línea ejecuta una serie de operaciones. El programa más sencillo de `awk` consiste en imprimir un mensaje de texto por cada línea que recibe de entrada.

Formas de ejecutar awk

Usando la entrada estándar:

```shell
awk Programa

awk Programa fichero_entrada

awk -f Fichero_Programa entrada/fichero_entrada

./fichero.awk  # hay que poner #!/usr/bin/awk -f al principio del fichero y hacer chmod
```

Estructura de un programa awkç

Un programa de `awk` tiene tres secciones:

1. Parte inicial, que se ejecuta sólo una vez, antes de empezar a procesar la entrada:
2. Parte central, con instrucciones que se ejecutan para cada una de las líneas de la entrada
   - Las operaciones se realizan sólo sobre las líneas que verifiquen la expresión regular indicada en el patrón
   - Si no ponemos ningún patrón las operaciones se realizan sobre todas las líneas
   - Si ponemos !patrón se ejecutan en las líneas que no concuerden con el patrón
   - La parte central es la que toma por defecto el intérprete de awk
3. Parte final, se efectúa sólo una vez, después de procesar la entrada

```
#!/usr/bin/awk -f
BEGIN{ print "Inicio" }
{ print "Hola mundo!" }
END{ print "Final" }
```

2.5.2 Manejo de ficheros de texto

`awk` divide las líneas de la entrada en campos:

- la separación entre campos la determina la variable `FS` (por defecto, uno o más blancos y TABs)
- las variables `$1, $2, ..., $N` contienen los valores de los distintos campos. `$0` contiene la línea completa

```shell
awk '{ print $1, $3 }'
uno dos tres cuatro → uno tres
cinco seis siete ocho → cinco siete
```

Variables predefinidas

`awk` tiene un conjunto de variables predefinidas, por ejemplo, nos permite especificar el separados de campos. Esas variables son:

Nombre	Significado
FS	Carácter separador entre campos de entrada (por defecto, blanco o tabulado)
NR	Número de registros de entrada (líneas)
NF	Número de campos en el registro de entrada
RS	Carácter separador entre registros de entrada (por defecto, nueva línea)
OFS	Carácter separador entre campos en la salida (por defecto, un espacio en blanco)
ORS	Carácter separador entre registros de salida (por defecto, nueva línea)
FILENAME	Nombre del fichero abierto

SHELL

```
cat usuarios.awk
BEGIN { FS = ":"; OFS = "-->"; ORS = "\n==========\n"; }
{ print NR, $1, $5 }

 awk -f usuarios.awk /etc/passwd
 ...
 37 -->tomas -->Tomás Fernández Pena„,
 ==========
 38 -->caba -->José Carlos Cabaleiro Domínguez„,
 ==========
 ...
```

2.5.3 Otras características

`awk` es un lenguaje completo:

- permite definir variables de usuario
- permite realizar operaciones aritméticas sobre las variables
- permite utilizar condiciones, lazos, etc.
- permite definir funciones

La sintaxis de `awk` es casi igual que la de `C`

- podemos usar `printf` en lugar de `print`
- también podemos usar arrays

```shell
cat acumula.awk BEGIN{ sum=0; print "Introduce números" }
{ sum=sum+$1; }
END{ print "La suma acumulada es: ", sum}
```

2.6 Pyhton

Además de Bash existen otros lenguajes adecuados para la creación de scripts de administración. En la actualidad el lenguaje de script que domina a todos es Pyhton

- Énfasis en la legibilidad
- Uso de identación para delimitar bloques de código
- Soporte de diversos paradigmas: imperativo, orientado a objetos y funcional
- Sistema de tipos dinámico y gestión automática de memoria
- Gran librería con módulos para múltiples tareas

```python
#!/usr/bin/env python3
# Abre el fichero sólo lectura
try:
 f = open('/etc/passwd','r')
except IOError:
 print('No puedo abrir /etc/passwd')
else:
 # Lee las líneas en una lista
 lista = f.readlines()
 # Recorre e imprime la lista for
 l in lista:
  print(l, end='') # end='' elimina el retorno de línea
f.close()
```

2.6.1 Tipos de datos en Python

`Python` usa un tipado dinámico en donde son los valores, no las variables, las que definen el tipo de dato. Los enteros tienen una precisión ilimitada, mientras que los `float` suelen tener la precisión de `double` en `C`. Las cadenas pueden delimitarse por comillas simples o dobles.

- Podemos convertir de un tipo de datos a otro.
- Entre las operaciones aritméticas se incluyen +, −, * , /, // (división entera), % (resto), x ∗ ∗y (potencia).
- Las operaciones lógicas bit a bit entre enteros incluyen x|y (or), xˆy (exor), x&y (and), x << n (desplazamiento a la izquierda), x >> n (desplamiento a la derecha) y ∼ x (negación).

- Por último, las operaciones lógicas incluyen and, or y not

Colecciones

1. **Listas**
- **Mutables** (se pueden modificar).
- Permiten **mezclar tipos** (string, enteros, etc.).
- Se definen con corchetes `[]` o usando `list()`.
- Ejemplo de operaciones:
  - Añadir: `frutas.append('peras')`
  - Eliminar: `frutas.remove(123)`
  - Insertar en posición: `frutas[2:2] = ['fresas', 'pomelos']`
  - Ordenar: `frutas.sort()`
  - Acceder por índice y rangos.
- Las listas pueden contener otras listas (listas anidadas).

2. **Tuplas**
- **Inmutables** (no se pueden modificar).
- Se definen con paréntesis `()` o usando `tuple()`.
- Ejemplo: `meses = ('enero', 'febrero', ..., 'diciembre')`
- Se puede acceder por índice, pero no modificar ni añadir elementos.

3. **Conjuntos (Sets)**
- **No permiten elementos duplicados**.
- Se definen con `set()`.
- Permiten operaciones matemáticas de conjuntos: diferencia `-`, unión `|`, intersección `&`, diferencia simétrica `^`.
- Ejemplo: `frutas = set(cesta)`

4. **Diccionarios**
- Colecciones de **clave:valor**.
- Se definen con llaves `{}` o usando `dict()`.
- Ejemplo: `edad_de = {'Eva':23, 'Ana':19, 'Oscar':41}`
- Permiten modificar, añadir y eliminar elementos por clave.
- Métodos útiles: `.keys()`, `.values()`, `.items()` para recorrer claves y valores.

Referencias

La copia de una variable referencia a un objeto diferente, mientras que la copia de una colección referencia al mismo objeto. Para que las listas referencien a diferentes objetos:

PYTHON

```python
a = [1, 2] # nuevo objeto lista
b = a[:] # a y b referencian a objetos diferentes
a[0] += 5 # se modifica el objeto a
print( b ) # [1, 2], b no se ha modificado
c=list(a) # otra forma
```

Generación de listas

PYTHON

```
l = range(5) # l = [0, 1, 2, 3, 4]
l = range(2, 5) # l = [2, 3, 4]
l = range(2, 10, 3) # l = [2, 5, 8]
l = range(5, -5, -2) # l = [5, 3, 1, -1, -3]
s = sum(range(1,4)) # s = 6
# iniciar una lista
a = [0]*3 # a = [0, 0, 0]
# iniciar una lista enlazada
for i in range(3):
    a[i]=[0]*2 # a = [ [0, 0], [0, 0], [0, 0]
    ```

Matrices
Python no incluye ningún tipo para matrices. Sin embargo, podemos tra- tar una lista enlazada como una matriz. Por ejemplo:
`A = [[1, 4, 5], [-5, 8, 9]]`
Podemos referenciar a sus elementos, por ejemplo, A[0][0]=1, A[0][1]=4, etc. También podemos referenciar a las filas, por ejemplo A[0]=[1,4,5]

Además, los módulos y librerías de Python pueden extender o introducir sus propios tipos de datos. Por ejemplo, el módulo NumPy introduce el tipo de dato array con numerosas operaciones predefinidas sobre arrays y matrices.

2.6.2 Control de flujo
Lazos
```python
frutas = ['naranjas', 'uvas']

for f in frutas:
    print(f, len(f))  # naranjas, 8; uvas, 4

for i in range(len(frutas)):
    print(i, frutas[i])  # 0, naranjas; 1, uvas

nf = input('Añade otra fruta: ')
while nf:  # Si la entrada no está vacía
    frutas.append(nf)   # añádela a la lista
    nf = input('Añade otra fruta: ')
```

Condicionales

Hay 8 tipos de comparaciones en Python, que tienen todas la misma prioridad: `<` (menor), `<=` (menor o igual), `>` (mayor), `>=` (mayor o igual), `==` (igual), `!=` (distinto), `is` (identidad de objeto) y `is not` (identidad de objeto negativa).

PYTHON

```python
x = int(input('Introduce un entero: '))
if x < 0:
    x = 0
    print('Negativo cambiado a 0')
elif x == 0:
    print('Cero')
else:
    print('Positivo')
```

Funciones

Ejemplo 1:

PYTHON

```python
def opera(a, b):
    c = a + b
    d = a - b
    return (c, d)

suma, resta = opera(8, 7.5)
```

Ejemplo 2 (valor por defecto en argumento):

PYTHON

```python
def compra(fr, nf='manzanas'):
    fr.append(nf)

frutas = []  # También frutas = list()
compra(frutas, 'peras')
compra(frutas)
compra(nf='limones', fr=frutas)
print(frutas)  # peras, manzanas, limones
```

Funciones con un número arbitrario de argumentos

Si ningún argumento es un diccionario:

```python
def fun(*args):
    for arg in args:
        print(arg)

fun('peras', [1, 2, 3], 6)
# -> peras
# -> [1, 2, 3]
# -> 6
```

En general, se distinguen dos tipos de argumentos:

- *args: otros argumentos (un asterisco)
- **kwargs: diccionarios (dos asteriscos)

```python
def fun(*args, **kwargs):
    for arg in args:
        print(arg)
    for kw in kwargs.keys():
        print(kw, kwargs[kw])

fun('peras', 1, manzanas=2, limones=3)
# -> peras
# -> 1
# -> limones 3
# -> manzanas 2
```

Funciones en ficheros separados

Se utiliza la sentencia `import`

Archivo `myutils.py`:

```python
#!/usr/bin/env python3
def sqrt(x):
    return x ** 0.5

def double(x):
    return x + x

def main():
    print(sqrt(5), double(5))

if __name__ == "__main__":
    main()
```

Archivo `main.py`:

```python
#!/usr/bin/env python3
import myutils

print(myutils.sqrt(42))
```

Cuando el intérprete de Python lee un fichero, ejecuta todo el código que encuentra en él.
Si el fichero es importado desde otro programa Python, puede interesarnos que el programa principal del archivo importado **no se ejecute**, sino que solo se incluyan las funciones.

Por ello, se utiliza el condicional:

```python
if __name__ == "__main__":
    main()
```

De esta forma:

1. Si el fichero se invoca directamente, se ejecuta la función `main()`.
2. Si se invoca desde otro fichero, no se ejecuta la función `main()`.

2.6.3. Orientación a objetos

A continuación se muestra la definición de una clase:

```python
class fruteria:
    '''Ejemplo simple de clase'''
    def __init__(self, f):
        self.stock = list()
        self.stock.append(f)
    def compra(self, f):
        self.stock.append(f)
    def vende(self, f):
        if f in self.stock:
            self.stock.remove(f)
        else:
            print(f, 'no disponible')
```

El método `__init__` representa el constructor en Python.
Cuando se llama a la clase, Python crea un objeto y le pasa como primer parámetro la variable `self`, que representa la instancia del objeto en sí. A continuación se le pasa el resto de los parámetros.

Programa principal:

```python
mi_fruteria = fruteria('pera')
mi_fruteria.compra('manzana')
print(mi_fruteria.stock)           # ['pera', 'manzana']
mi_fruteria.vende('pera')
mi_fruteria.vende('platano')      # platano no disponible
print(mi_fruteria.stock)           # ['manzana']
mi_fruteria.vende('pera')         # pera no disponible
```

Métodos y atributos privados

Los métodos o atributos privados se definen con **dos guiones bajos antes del nombre** (y no pueden terminar en dos guiones bajos).

PYTHON

```python
class Ejemplo:
    def publico(self):
        print('Uno')
        self.__privado()
    def __privado(self):
        print('Dos')

ej = Ejemplo()
ej.publico()      # Imprime Uno Dos
ej.__privado()    # Da un error
```

Herencia múltiple

Se permite la herencia múltiple:

PYTHON

```python
class fruteria:
    def que_vendo(self):
        print('Vendo frutas')

class carniceria:
    def que_vendo(self):
        print('Vendo carne')

# Herencia múltiple
class tienda(carniceria, fruteria):
    pass  # operación nula

# La clase carniceria está primera en la definición de tienda
tienda().que_vendo()  # Vendo carne
```

> **Nota:**
> La sentencia `pass` es una operación nula, es decir, no hace nada cuando se ejecuta, pero ayuda a mantener el formato del lenguaje.

2.6.4. Módulos y librerías

Python dispone de una gran variedad de módulos y librerías que se utilizan con la sentencia `import`.
Ejemplos básicos:

```python
import math
print(math.cos(5))

from math import cos, sin
print(cos(5), sin(5))
```

1. Procesamiento de textos

Python incorpora muchos métodos para trabajar con cadenas:

```python
l = 'Hola que tal!'.strip('!').split()          # ['Hola', 'que', 'tal']
s = ','.join(l)                                 # 'Hola,que,tal'
c = s.count(',')                                # 2
ss = s.replace(',', '\t')                       # 'Hola\tque\ttal'
l = ss.split('\t'); l.reverse()                 # ['tal', 'que', 'Hola']
c = ss.find('tal')                              # 9
l = '''Esto es
un texto con
varias lineas'''.splitlines()                   # ['Esto es', 'un texto con',
'varias lineas']
```

2. Expresiones regulares

El módulo `re` permite trabajar con patrones de texto:

```python
import re
s = input('Introduce una palabra: ')
if re.match('^[A-Z]+$', s):
    print('Todas las letras mayusculas')

# Buscar URLs en un fichero
h = re.findall('http://([^\s]+)', l)

# Separar un string por varios caracteres
l = re.split('[:.-]', 'Uno:Dos.Tres-Cuatro')
```

3. Parámetros y funciones dependientes del sistema

El módulo `sys` permite manejar argumentos y terminar el script:

```python
import sys
sys.argv       # Lista de argumentos en línea de comandos
sys.exit(1)    # Termina el script
```

4. Subprocesos

El módulo `subprocess` permite ejecutar comandos del sistema y scripts:

```python
import subprocess
code = subprocess.call(['mkdir', '/tmp/dir'])
output = subprocess.check_output(['echo', 'Hello World!'], encoding='utf8')
p = subprocess.Popen(['df', '-h'], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
```

5. Parseo de argumentos

El módulo `argparse` sirve para procesar opciones en línea de comandos.

6. Operaciones dependientes del sistema operativo

El módulo `os` permite acceder a funcionalidades del sistema:

- `os.getlogin()`, `os.getloadavg()`, `os.getcwd()`, `os.chdir(path)`, `os.listdir(path)`
- El submódulo `os.path` ofrece utilidades para manipular rutas y ficheros
- El módulo `glob` hace "expansión de nombres"
- El módulo `shutil` permite copiar y mover ficheros
- El módulo `tempfile` para ficheros temporales

Ejemplo: Renombrar ficheros *.xml a* .html en un directorio (usando argparse, glob, shutil):

```python
import os.path, glob, shutil, sys, argparse

def main():
    parsear = argparse.ArgumentParser(description='Renombra XML a HTML')
    parsear.add_argument('directory', help='nombre de directorio')
    args = parsear.parse_args()
    dir = args.directory

    if not os.path.isdir(dir):
        print(dir + ' no es un directorio')
        sys.exit(1)
    try:
        os.chdir(dir)
        for f in glob.glob('*.xml'):
            new = os.path.splitext(f)[0] + '.html'
            shutil.move(f, new)
    except:
        print('Hubo un problema ejecutando el programa.')

if __name__ == "__main__":
    main()
```

7. Operaciones con ficheros comprimidos

Los módulos `gzip`, `bz2`, `zipfile` y `tarfile` permiten trabajar con ficheros comprimidos.

Ejemplo: Menú de acciones sobre un fichero tar:

PYTHON

```python
import tarfile, sys

while True:
    tar = tarfile.open(sys.argv[1], 'r')
    selection = input(
        '''Selecciona
        1 para extraer un fichero
        2 para mostrar información sobre un fichero
        3 para listar los ficheros
        4 para terminar\n'''
    )
    if selection == '1':
        filename = input('Indica el fichero a extraer: ')
        tar.extract(filename)
    elif selection == '2':
        filename = input('Indica el fichero a inspeccionar: ')
        for tarinfo in tar:
            if tarinfo.name == filename:
                print('\nNombre:\t', tarinfo.name, '\nTamaño:\t',
tarinfo.size, 'bytes\n')
    elif selection == '3':
        print(tar.list(verbose=True))
    elif selection == '4':
        break
    else:
        print('Selección incorrecta')
```