Herramientas de Programación en Bioinformática y Biología Computacional

Resumen

Esta es una asignatura introductoria de los conceptos más importantes que se irán extendiendo a lo largo de todo el máster. Se divide en tres módulos: Linux, Python y Bases de Datos.

Sandra Mingo Ramírez

UAM - 2024/25

19 de septiembre de 2024 17:15

Universidad Autónoma de Madrid
Bioinformática y Biología Computacional

Código en Github

Índice general

Capítulo I

Linux - Bash

Todos los comandos se lanzan desde la terminal de Linux. Se pueden explorar los ficheros del árbol de directorios con un explorador que es equivalente al explorador de Windows. En cuanto se maneje con ficheros grandes, o se requieran acciones mínimas, las herramientas de la interfaz gráfica no van a ser suficientes - de ahí que tengamos que trabajar con la terminal.

man

Con el comando `man comando`, se accede al manual para buscar el comando con su nombre, descripción y posibles parámetros tanto en su versión larga como corta.

I.1. Tratamiento de ficheros y directorios

ls

El comando `ls` muestra la lista de la información del directorio en el que nos encontremos. Muestra tanto los ficheros como los directorios. En Ubuntu, distingue los directorios en azul oscuro y los ficheros en verde, pero el código de colores puede cambiar en la configuración y puede no ser igual siempre, por lo que no nos deberíamos fijar en eso. El comando se puede completar con una serie de **parámetros detrás de un guion que modifican el comportamiento del comando inicial**. `ls -l` lista el contenido del directorio con información adicional que incluye la localización del fichero, el tipo de fichero, el tamaño en bytes, los permisos, el inode y cuándo se modificó por última vez. Algunos tamaños pueden ser difíciles de leer a simple vista. Para ello, se emplea `ls -lh` que lista el contenido con el tamaño largo, de forma que indica las unidades de forma más comprensible a simple vista. De hecho, la h viene de "human". El parámetro `-t` ordena la lista por orden de creación o modificación de más reciente a más antiguo, y `-p` añade una barra invertida a los directorios para poder distinguirlos de los ficheros. Si se escribe `-lp`, la primera columna tiene una d al inicio de todos los directorios, mientras que los ficheros tienen un guion como primer caracter. Así, todos los parámetros se pueden concatenar en `ls -lhtp`. El parámetro `ls -a` lista el contenido del directorio en el que nos encontremos, incluyendo los ficheros ocultos cuyo nombre empieza por un punto (por ejemplo, ".config"). Estos ficheros no aparecen a no ser que se añada el parámetro a para protegerlos y que no se modifiquen sin querer. "Si no quieres que alguien meta la pata, no le enseñes dónde puede meter la pata". En función de los permisos que se tenga, luego se pueden modificar esos ficheros o no. El parámetro `-i` añade un inode que será importante cuando seamos administradores,

ya que es un índice que único para cada fichero que permite modificar un fichero sin modificar otro aunque se llamen igual. Aunque no se puedan tener dos ficheros en el mismo directorio que se llamen igual porque el path sería el mismo, sí puede haber dos ficheros con el mismo nombre en directorios distintos, ya que el path sería diferente, al igual que el inode.

mkdir

Para crear nuevas carpetas o directorios, en el explorador es click derecho y crear nueva carpeta. En la terminal, esto se traduce en el comando `mkdir nombre-carpeta-nueva`. El tamaño de un directorio no es el tamaño de lo que tenga dentro el directorio; al crear una carpeta nueva, ya tiene un tamaño de 4K, y si se tuvieran muchos ficheros muy pesados, no tiene por qué ser la suma de todos los tamaños que contenga ese directorio.

cd

Para cambiar de directorio, se emplea el comando `cd nuevo-directorio` Así, el siguiente prompt incluye el directorio en el que se encuentra. Al entrar en un directorio que acabamos de crear, con el comando `ls -lht` no aparece nada, pero con `ls -lhta` aparecen dos directorios ocultos llamados "." y "..". Los dos puntos son un **enlace al directorio anterior o padre**, de forma que con `cd ..` se va al directorio anterior. El punto simple es el **directorio actual**.

pwd

El comando `pwd` muestra dónde se encuentra el **directorio actual**, es decir, el **directorio de trabajo o working directory**. El primer símbolo es una barra (/) que indica el **directorio raíz**. La virgulilla (∼) lleva a la "carpeta personal", que es una traducción de **"home directory"**. Se puede mover entre directorios ya sea por path directo mediante el uso del directorio raíz o mediante el árbol del directorio con los dobles puntos o la virgulilla.

cp

Para copiar un archivo, se emplea el comando `cp archivo-original archivo-nuevo`. Hay comandos que no necesitan **argumentos**, y hay comandos (como este) que sí necesitan. En este caso, requiere de dos argumentos: el archivo que se quiere copiar y el nombre que se le quiere dar a la copia. El orden es siempre **comando -parámetro argumento**. Si se quiere copiar un directorio con todo su contenido a otro directorio, es necesario añadir el parámetro `cp -r directorio-original directorio-nuevo` (r de recursivo, es decir, todo lo que haya dentro). La fecha de los ficheros copiados es de cuando se crea la copia, no la del fichero original. Solo se podrán copiar archivos en aquellos directorios en los que tengamos los permisos apropiados por el administrador. Antes de realizar el comando para copiar, la terminal realiza una serie de comprobaciones, de forma que si el nombre del archivo a copiar tiene alguna errata, salta un error al no poder realizarse el comando stat, que forma parte de las comprobaciones previas. Si se copia un directorio en otro preexistente, se copia el primer directorio en el segundo. Esto resulta en que el segundo directorio incluye los ficheros que ya estaban y un nuevo directorio nuevo dentro - no se sobreescribe todo el directorio ni se borra lo que hubiese ahí. Aun así, sigue siendo recomendable proteger los directorios sensibles añadiendo un punto antes del nombre o directamente cambiando los permisos.

mv

Para mover un archivo, se utiliza el comando `mv`. Se utiliza de forma similar a cp, pudiendo mover un fichero a otro directorio o al mismo con el mismo u otro nombre siempre que el nuevo directorio ya exista previamente. Si se mueve un directorio entero, se mueve con todo el contenido que tenga en el nuevo directorio, y también se le puede cambiar el nombre.

rm
rmdir

A la hora de querer borrar, se utiliza `rm` para los archivos y `rmdir` para los directorios. Para esto último, es necesario que el directorio esté vacío. En caso de que no, se puede utilizar `rm -r directorio/` o `rm -r ./directorio/` para borrar tanto el directorio como su contenido.

ln

En lugar de copiar un fichero y tenerlo doble, se puede crear un link simbólico. Esto quiere decir que se crea un fichero que no tiene el contenido del fichero original (y por tanto pesa menos, solo 11 bytes), si no que redirige a ese archivo, permitiendo enlazar distintos directorios o ficheros en ordenadores en remoto. Para crear un link simbólico, el comando es `ln -s fichero-original fichero-simbolico`. Si se borra el fichero original, el enlace simbólico ya no tiene utilidad. Para borrar un enlace, se puede borrar como un fichero cualquiera con `rm`.

> Todos los ficheros tienen un nombre y una extensión, y siempre se debe escribir completo. Los directorios no llevan nunca extensión. Se recomienda no utilizar espacios en blanco en los nombres tanto de los ficheros como de los directorios porque dificulta la navegación.

I.2. Mostrar contenidos en pantalla

clear

El comando `clear` borra los comandos de la pantalla para dejarla vacía.

cat

Para mostrar en pantalla el contenido de un fichero, se puede emplear el comando `cat fichero`. No obstante, este comando no diferencia cambio de página, si no que muestra todo el contenido por pantalla aunque ocupe más. También se puede utilizar este comando con varios ficheros para que se muestren uno detrás de otro.

less

Para mostrar el contenido de un fichero en la pantalla por partes, se emplea `less`. De esa forma no hay que subir y bajar en la pantalla, si no que se desplaza mediante el uso del espacio para ir por páginas enteras, enter para ir línea a línea, y la tecla q para salir (quit).

head
tail

El comando `head` muestra en pantalla las primeras 10 líneas de un fichero por defecto. Se le puede añadir un parámetro para indicar el número de líneas que se desea ver: `head -15 fichero` muestra las primeras 15 líneas. De forma similar, `tail` muestra las últimas 10 líneas de un fichero.

I.3. Buscar contenidos de ficheros

grep

El comando `grep` es fundamental al utilizarse mucho con los pipes y las redirecciones (véase más abajo). Se utiliza `grep cadena-buscada fichero`, y el resultado son todas las líneas que incluyen la cadena de caracteres que se ha buscado, tanto como palabra suelta como dentro de otra palabra. Utilizando `grep -w`, se filtran aquellas líneas en las que la cadena forma palabras individuales (si se busca with, descarga within, por ejemplo). Este comando es sensible a las mayúsculas y minúsculas. Para ignorar eso, se añade el parámetro `-i`. El parámetro `-v` muestra aquellas líneas

que no incluyan la cadena, `-n` incluye al inicio de cada línea el número de dicha línea, y `-c` el número de líneas que se deberían mostrar.

wc

El comando `wc` cuenta distintas cosas en un fichero: líneas, palabras, y bytes. Para obtener solo una de esas mediciones, se pueden añadir los parámetros `-l`, `-w`, `-c` respectivamente. Tradicionalmente, la cantidad de bytes indica la cantidad de caracteres, pero puede haber pequeñas diferencias por la codificación. Para obtener la cantidad de caracteres, se emplea `grep -m`.

I.4. Redireccionamientos y pipes

Tras ejecutar un comando, el resultado que sale en pantalla es la **salida estándar**. Sin embargo, se puede **redireccionar la salida** a otro lugar mediante el símbolo >.

> > >

Linux trata igual la salida estándar y un fichero, pero la salida estándar es la opción por defecto, por lo que nosotros debemos redireccionar la salida cuando lo queremos guardar en un fichero. Hay que tener cuidado porque si se redirecciona una salida a un fichero que ya existe, se sobreescribe el contenido del mismo. Si lo que queremos es que el contenido nuevo se añada detrás del contenido ya presente en un fichero, se deben emplear los dos símbolos > >.

También existe un **redireccionamiento de entrada**, siendo la entrada estándar lo que se escribe por teclado. Para redireccionar la entrada, se emplea el símbolo <.

<

Nunca se debe emplear el mismo fichero de entrada y de salida. Al leer un fichero, el programa va línea a línea, por lo que si se procesa la primera línea y se guarda, se sobreescribe el fichero que se usaba de entrada, afectando a la continuidad del programa.

gedit

El comando `gedit fichero` abre un editor de texto plano que nos permite crear un fichero y escribir su contenido. Para guardar el contenido que hemos escrito, pulsa `ctrl s` y para cerrar `ctrl q`. Aunque el editor de texto también muestre el contenido de un fichero que ya existe previamente, si el tamaño del fichero es muy grande, no es recomendable leer su contenido mediante este comando, si no mediante otros mencionados previamente como `less`.

sort

El comando `sort` ordena una entrada por teclado o un fichero alfanuméricamente. Si un fichero está separado en líneas, compara todos los caracteres de cada línea y empieza a ordenar por el primero. A este comando se le puede pasar directamente ficheros o se le puede redireccionar por entrada. El parámetro `-r` muestra el resultado en orden inverso, y `-u` filtra las líneas repetidas en un archivo. Para ordenar una sola parte, el parámetro `-k` permite elegir los campos a comparar.

|

Hay ocasiones que para obtener un resultado paso a paso, hay que crear ficheros temporales que modificar y a los que acceder. Esto se puede evitar mediante el uso de **pipes** o barras verticales (|), que conectan los comandos antes y después del pipe. De esta forma, se omite el paso intermedio y se utiliza la salida del primer comando como entrada del segundo.

I.5. Wildcards y convenciones en nombres de archivo

*
?

Los wildcards en informática son como comodines.Existen dos tipos: el asterisco y el símbolo de interrogación. El **asterisco** va a representar cualquier número de caracteres en el nombre de un fichero o directorio. Por ejemplo, si ponemos `prueba*`, puede resultar en el directorio pruebas y en los ficheros prueba.txt y prueba.pdf. Por el contrario, el **interrogante** representa exactamente un caracter. Así, al poner `prueba?`, el resultado será exclusivamente el directorio pruebas, pero no los ficheros.

A la hora de nombrar ficheros y directorios, se deben **evitar los símbolos especiales** tales como /, *, & y %. También es muy recomendable **evitar espacios entre caracteres**. En resumen, a la hora de poner un nombre a un archivo, se deberían usar solo caracteres alfanuméricos junto a barras bajas y puntos. Tradicionalmente, los nombres empiezan en minúcula y pueden terminar en un punto seguido de un grupo de letras que indican el contenido de un archivo (por ejemplo, poner al final de todos los archivos en código Python .py).

I.6. Seguridad del sistema de archivos (permisos de acceso

Cada archivo y directorio tiene **permisos de acceso asociados**, que se pueden comprobar en la primera columna al realizar `ls -lht`. Se trata de una cadena de 10 símbolos d, r, w, x, -. La d solo estará presente en primera posición e indica que se trata de un directorio. Si se trata de un fichero, en primera posición hay un guion. Los 9 símbolos restantes se agrupan en grupos de 3 en 3 y representan los **permisos del usuario, del grupo al que pertenece el usuario (y no es el usuario) y de todos los demás** respectivamente. Las opciones son:

- r (read): permisos de lectura y copiado de un archivo y de listar el contenido de un directorio.
- w (write): permisos de escritura y modificado de un archivo y de crear y borrar los archivos del directorio o mover archivos a él.
- x (execution): permisos de ejecución de un archivo cuando sea apropiado, es decir, cuando sea ejecutable. Por ejemplo, los comandos de linux como ls o mv son ficheros que se encuentra en /usr/bin/ y que se pueden ejecutar por todos a la hora de escribir esos comandos en la terminal.

chmod

Si en lugar del caracter aparece un guion, significa que ese permiso no está dado. Si eres el propietario de un fichero, se pueden cambiar los permisos mediante el comando `chmod`. Para indicar a quién se le quiere cambiar el permiso, se pone u (usuario), g (grupo), o (otros), a (todos). Para quitar un permiso, se escribe un guion (-) seguido del permiso que se quiere quitar y del fichero; para dar un permiso, se escribe un más (+) y el permiso seguido del nombre del fichero. Un igual (=) expresa los permisos que se quieran. Para poner permisos diferentes, se escriben separados por comas (pero

sin espacios), por ejemplo `chmod g+wx,o+w fichero`. También se puede ejecutar el comando `chmod 777 fichero` para darle todos los permisos a todos. Es importante tener en cuenta también los permisos de los directorios, ya que para poder modificar un fichero, no solo se deben tener los permisos para modificarlo, si no que a su vez debe estar en un directorio para el que se tengan los permisos para modificarlo.

A la hora de crear ficheros, Bash proporciona al usuario autor los permisos de lectura y escritura, pero no de ejecución. Esto es importante a la hora de crear archivos ejecutables con extensión .sh.

I.7. Procesos y trabajos

Un **proceso** es un programa que se ejecuta y que recibe un **identificador (PID)** por parte del sistema operativo. Hay procesos que los lanzamos nosotros, pero hay otros que funcionan por detrás como por ejemplo buscar conexiones Wifi.

top

El comando `top` muestra todos los procesos que se están llevando a cabo en tiempo real. La terminal es un proceso, y cada vez que nosotros lanzamos un proceso desde la terminal, por lo que se genera un proceso con un proceso padre (la terminal). Los procesos no pueden ejecutarse a la vez (van secuencialmente), pero el sistema operativo va gestionando los procesos y sus tiempos para que parezca que van en paralelo. Los sistemas operativos modernos pueden contener varios núcleos o kernel que permite su funcionamiento a la vez, pero dentro de cada núcleo o kernel los procesos van de forma secuencial. Para salir de top, se debe ejecutar `ctrl c`.

ps

El comando `ps` muestra todos los procesos que se han lanzado desde esa terminal. Añadiendo el parámetro `-ef`, se incluyen los procesos padres (lanzados por root) con el usuario que lanzó el proceso, el PID y el PID padre.

sleep

El proceso `sleep` está parado el tiempo que se le indique. En realidad, va a estar más tiempo debido a la gestión del tiempo del sistema operativo que va alternando distintos procesos de forma intermitente. Por ello, aunque ese proceso sí dure el tiempo determinado, el usuario tiene que esperar algo más. Si se escribe `sleep 10s &`, se estará ejecutando de fondo (en el background), y lo que se devuelve a la terminal es el PID. Esto permite seguir utilizando la terminal para otros procesos mientras tanto. Si se va enviado un proceso en el foreground cuando se quería en el background, se puede abortar (con ctrl c) y volver a ejecutar bien o se puede detener (con ctrl z) y poner bg para enviarlo al background. Si por el contrario se quiere reiniciar un proceso suspendido, se pone `fg`. Esto reinicia el último proceso suspendido. Para especificar uno concreto, se debe poner su número de trabajo (no el PID): `fg $1`.

kill

Para matar un proceso de forma eficiente, se puede usar el comando `kill -9` seguido del identificador del proceso. De esta forma se asegura que también se eliminen los subprocesos generados por ese proceso y se cierren todos los recursos. Sin embargo, no es posible matar los procesos de otros usuarios.

I.8. Otros comandos útiles de UNIX

- **df**: informa del espacio libre del sistema.
- **du**: saca los kilobytes utilizados por cada subdirectorio. Esto puede ser útil cuando se ha superado el almacenamiento para ver qué directorio tiene más archivos.
  - -s: muestra solo un resumen
- **gzip y gunzip**: `gzip` reduce el tamaño de un fichero utilizando el compresor Zip, resultando en un fichero con extensión .gz. `gunzip` descomprime el fichero .gz. Para esto, es necesario tener permisos de lectura.
- **tar**: combina varios archivos en uno único que puede o no estar comprimido.
  - -c: crea un archivo.
  - -x: extrae un archivo.
  - -v: muestra el progreso de un archivo.
  - -f: nombre de archivo.
  - -t: ver el contenido del archivo.
  - -j: comprime el archivo mediante bzip2.
  - -z: comprime el archivo mediante gzip.
  - -r: añade o actualiza archivos o directorios a archivos existentes.
  - -C: directorio especificado
  - –exclude=: excluye los archivos que vayan a continuación de la orden principal.
- **zcat**: lee archivos de extensión .gz sin la necesidad de descomprimirlos previamente.
- **cut**: extrae porciones de texto de un fichero al seleccionar columnas o caracteres.
  - -d: delimitador en comillas simples (por ejemplo, `-d '|'`).
  - -f: campo o columna que se quiere extraer (`-f1` o, si se quieren varias, `-f1,2`).
- **echo**: muestra en la salida estándar lo que se ponga a continuación. Aunque funcione directamente, es recomendable escribir las cadenas de caracteres entre comillas dobles.
- **tr**: reemplaza o borra caracteres de una cadena. No admite un fichero de entrada, si no que debe venir por un pipe o utilizando el redireccionamiento de entrada.
  - -s: sustituye la secuencia que se ponga a continuación entre comillas simples con una única ocurrencia.
  - -d: elimina los caracteres proporcionados.

- **touch**: toca el fichero, es decir, accede a él, cambiando la fecha de acceso. Si el fichero no existe, se crea el fichero sin contenido (tamaño 0).
- **date**: imprime en pantalla el día y la hora. Se puede personalizar la forma en la que se muestra.
- **file**: muestra la información de uno o varios ficheros como su codificación.
- **stat**: muestra el archivo o el estado del archivo. También es posible acceder a la meta-información del fichero mediante el parámetro `-c` seguido de una secuencia válida.
- **basename y dirname**: basename elimina las partes del path en el nombre de un archivo para dejar únicamente el nombre propio. También puede eliminar el sufijo de un nombre. Por el contrario, dirname elimina el nombre del archivo para dejar únicamente el path.
- **diff**: compara el contenido de dos ficheros y muestra las diferencias entre ambos.
- **find**: busca en los directorios ficheros y directorios con un cierto nombre, fecha, tamaño u otro atributo que se quiera especificar. Por ejemplo, para buscar todos los archivos en el directorio actual que sean de texto plano, se puede poner `find . -name "*.txt" -print`. Para buscar únicamente directorios, se debe poner `-type d` y para buscar únicamente ficheros `-type e`.
- **xargs**: se utiliza usualmente para combinar comandos.
- **history**: muestra el historial de comandos.
- **ssh**: permite acceder a un servidor de Linux remoto.
- **nohup**: hace que haya procesos de fondo al trabajar desde un servidor remoto y que no se detengan cuando cerremos sesión.
- **gnome-terminal**: crea una nueva terminal.

I.9. Variables

En bash, las únicas variables que se puede crear son con **cadenas de caracteres** (`x=ejemplo`). Si se pone un número, el intérprete interpreta el número como un caracter ("3" en Python). Aunque las comillas no sean necesarias, sí son recomendables (sí son obligatorias cuando hay espacios en blanco entre caracteres). Es importante **no dejar espacios** entre la variable, el igual y la cadena, ya que se interpretaría el nombre de la variable como un comando y puede dar un error. Para acceder al valor de la variable, se debe poner con un símbolo de dólar antes del nombre: `echo $x`. Si la variable que se accede no existe, la salida estará en blanco. Las variables que creamos se borran al apagar el ordenador. Para prevenir esto, se pueden guardar las variables en el fichero de configuración .bashrc.

El sistema crea automáticamente algunas variables denominadas **variables de ambiente** cada vez que se enciende el ordenador. Esto es importante saberlo para no cambiar el valor de estas variables. Ejemplos son:

- HOME: path completo del usuario hasta la carpeta personal o home directory.
- USER: usuario en el que nos encontramos.
- PATH: directorios separados por dos puntos donde el intérprete va a buscar los ficheros de los comandos si no se encuentran en el directorio de trabajo.
- ?: guarda un 0 si el último comando ha sido correcto y un entero diferente de 0 si la última acción no ha sido ejecutada correctamente (ha dado un error). Es importante remarcar que aquí, comando verdadero es 0, mientras que en Python y otros lenguajes de programación, True es cualquier número que no sea 0.

export

En una terminal nueva, las variables que se habían creado no existen. Para ello, se puede usar el comando `export nombre-variable` para que terminales nuevas sí puedan acceder a las variables en ese estado. Si la variable se modifica posteriormente, si no se vuelve a exportar, la terminal nueva no tiene la actualización. A su vez, como la terminal nueva es un proceso nuevo, no le puede exportar sus variables al proceso padre.

readonly

Para evitar que una variable se modifique, se puede utilizar el comando `readonly nombre-variable`.

Para guardar en una variable la salida de un comando, se debe emplear la expresión `x=$(comando)`. Entre los paréntesis y el comando sí se pueden dejar espacios para facilitar la lectura humana. Si hay una errata en el comando y da error, la asignación a la variable no se realiza y la variable se queda sin contenido.

Se puede crear una variable con el contenido de otras variables. Teniendo `x=3` y `y=4`, si se escribe `z=$x+$y`, al poner `echo $z`, el resultado es 3+4. Esto se debe a que bash no interpreta el símbolo + como suma matemática. Para poder hacer eso, la declaración de variable y la suma se debe realizar entre doble paréntesis: `((z=x+y))`.

unset

Para eliminar variables, se usa el comando `unset nombre-variable`.

I.10. Scripts

Los scripts son ficheros ejecutables de código. La extensión de estos ficheros depende del lenguaje de programación: en el caso de Python, los scripts tienen la extensión .py, y en el caso de bash, la extensión .sh. Para que un fichero se pueda ejecutar, se deben tener los permisos de ejecución. Además, la primera línea del código debe contener el directorio del intérprete que se debe utilizar. Tradicionalmente, los intérpretes se encuentran en /usr/bin o en /bin. Por ejemplo, la primera línea de un script de Python debe ser `#!/usr/bin/python3` y de Bash `#!/bin/bash`. Una vez hecho esto, desde la propia terminal se puede ejecutar el script.

I.10.1. Variables especiales en scripts

Es posible pasar argumentos a un script para personalizar la salida. Para asignar algo indeterminado a una variable, se emplea `$1 - $9`, de forma que al ejecutar el script, se deban poner los argumentos que pasará a reemplazar los huecos en ese orden. Es buena

práctica que en el script los argumentos pasados por teclado se guarden en variables con nombres que tengan sentido y luego sean esas variables las que se empleen en el resto del código. De esa forma, el código queda más claro visualmente y es más fácil de modificar sin equivocaciones.

Otras variables especiales son:

- $0: el nombre del script de bash.
- $#: la cantidad de argumentos que se han pasado al script.
- $@: todos los argumentos que se han pasado al script.
- $$: el PID del script actual.
- HOSTNAME: nombre de la máquina en la que se está ejecutando el script.
- SECONDS: la cantidad de segundos desde que el script se inició.
- RANDOM: devuelve un número aleatorio diferente cada vez que se refiera a esto.
- LINENO: devuelve el número de la línea actual en el script de bash.

I.11. Branching

Las sentencias condicionales if y case ayudan a tomar decisiones en los scripts de bash. Permiten decidir si ejecutar ciertos fragmentos de código en base a ciertas condiciones.

I.11.1. Expresiones condicionales

Una expresión condicional básica hace que, si una condición es verdadera, se ejecute el siguiente código. Si la condición resulta ser falsa, entonces esas acciones no se desarrollan.

```
if [ condición ]
then
   código a ejecutar
fi
```

Los posibles operadores que se pueden utilizar en la condición son:

- `! expresión`: la expresión es falsa
- `-z string`: la longitud de string es 0, es decir, está vacía.
- `string1 = string2`: string1 es igual que string2.
- `string1 != string2`: string1 no es igual que string2.
- `integer1 -eq integer2`: integer1 es numéricamente igual que integer2.

- `integer1 -gt integer2`: integer1 es numéricamente mayor que integer2.
- `integer1 -lt integer2`: integer1 es numéricamente menor que integer2.
- `-d fichero`: el fichero existe y es un directorio.
- `-e fichero`: el fichero existe.
- `-r fichero`: el fichero existe y tiene el permiso de lectura.
- `-s fichero`: el fichero existe y su tamaño es mayor que 0, es decir, no está vacío.
- `-w fichero`: el fichero existe y tiene el permiso de escritura.
- `-x fichero`: el fichero existe y tiene el permiso de ejecución.
- `fichero1 -nt fichero2`: es True si fichero1 es más nuevo que fichero2.
- `fichero1 -ot fichero2`: es True si fichero1 es más antiguo que fichero2.

Es posible conectar varias condiciones mediante el uso de `and` y `or`. Para ello, la forma preferida es combinar condiciones individuales con los operadores && para and y || para or.

```
if [ condición 1 ] && [ condición 2 ]; then
   código a ejecutar
fi
```

Otra opción sería escribir los operadores en la propia condición con `-a` para and y `-o` para or. Para invertir la condición (buscar que algo no sea x), se utiliza ! antes de la condición.

Las expresiones condicionales pueden requerir ciertas acciones si una condición es verdadera, y otras si es falsa. Para ello, se emplea `else`:

```
if [ condición ] ; then
   código a ejecutar si la condición es verdadera
else
   código a ejecutar si la condición es falsa
fi
```

En algunas ocasiones puede haber distintas condiciones que deriven en distintas acciones.

```
if [ condición ]
then
   código a ejecutar
elif [ otra condición ]
then
   otro código a ejecutar
else
   código diferente
fi
```

I.11.2. Bucles

Los bucles permiten repetir una serie de comandos hasta que se llega a un objetivo. Son útiles para ahorrar en código redundante y automatizar tareas repetitivas.

I.11.2.1. For loops

Los bucles for toman cada elemento de una lista en orden, lo asignan a una variable y se ejecuta el comando hasta que se acaben los objetos de la lista.

```
for variable in lista
do
   código a repetir
done
```

También es posible procesar una serie de números, pero para ello deben estar en formato C: `{inicial..final..pasos}`.

I.11.2.2. While loops

Los bucles while permiten repetir una lista de comandos mientras que la condición sea cierta y se detiene cuando la condición pasa a ser falsa:

```
while [ condición ]
do
   código a repetir
done
```

I.11.2.3. Until loops

El bucle until es similar al while, pero repite la lista de comandos hasta que la condición sea cierta, es decir, mientras sea falsa y se detiene cuando se hace verdadera.

```
until [ condición ]
do
   código a repetir
done
```

Capítulo II

Programación en Python

Los lenguajes de programación son la forma de comunicarnos con los ordenadores. Algunos lenguajes tienen bajo nivel de abstracción que trabajan en binario y por ello son más similares al propio sistema. Por otro lado, están los lenguajes de alto nivel de abstracción que son más cercanos al idioma humano. Python es el lenguaje con el mejor **balance entre potencia y simplicidad**.

La máquina ejecuta un programa ejecutable (.exe), que está escrito en ensamblador, el lenguaje del ordenador. Para algunos lenguajes de programación, se requiere de un compilador para que genere a partir de nuestro código un ejecutable que entienda la máquina, como por ejemplo C. En el caso de Python, lo que requiere es un **intérprete**, que genera a tiempo real las instrucciones del ordenador para ejecutar nuestro código. El intérprete solía ser más lento que el compilador al tener que estar traduciendo a tiempo real el código. No obstante, a día de hoy, Python puede alcanzar un 90 % del tiempo que tardaría un programa escrito en C en ejecutarse. Esto se debe a que hay problemas paralelizables, es decir, que se pueden dividir en partes y realizar al mismo tiempo. El rendimiento de un programa depende de entrada-salida, es decir, la lectura de los datos que se encuentran en el disco duro, y eso depende de la tecnología del disco duro, no del lenguaje de programación.

Python tiene un **gran soporte de librerías**, que son programas escritos por otras personas y que se pueden utilizar por los demás. Los comandos de Python se pueden escribir directamente desde la terminal (escribiendo primero `python3` y luego salir con ctrl d), pero eso no es cómodo. Es mejor crear primero un fichero fuente con extensión .py que luego pueda utilizar el intérprete. Para ello, se escribe `pico nombre.py`. Esto abre un editor de texto donde podemos escribir el código. Se guarda con ctrl s y se cierra con ctrl x. Para que el intérprete lo lea, se debe poner en la consola `python3 nombre.py`. En caso de que haya alguna errata o algún error en el código, a la hora de ejecutar el código por el intérprete se va a indicar en qué línea del código fuente se encuentra y qué problema hay. También existen **programas o IDEs** que son entornos donde poder editar el código fuente de forma más intuitiva. Ejemplos son Visual Studio Code, PyCharm, Spider, etc.

El código de Python se va ejecutando secuencialmente comando a comando.

II.1. Variables

Las **variables** son contenedores donde guardar un contenido al que poder referirse mediante el nombre de la variable. El contenido puede ser desde números a cadenas de texto, lo que define el tipo de variable (el tipo de contenido que contiene una variable). En otros lenguajes, es necesario declarar una variable antes de poder usarla (se crea antes de meterle un contenido). Esto en Python no es necesario; la **creación de tipos es implícita por el intérprete en función del primer valor que se le dé a la variable**. Así, si se escribe directamente age = 35, el intérprete le da el tipo integer (número entero) a la variable age. Los **tipos en Python son implícitos y dinámicos** (pueden cambiar), lo que puede ser muy cómodo, pero también puede dar lugar a errores. Los tipos son:

- Integer: número entero
- Float: número con decimales
- String: cadena de caracteres, ya sean letras o números, que van entre comillas simples o dobles. Una vez que están definidas, ya no se pueden modificar.
- Bool: booleanos, es decir, True o False.
- List: lista
- Tuple: tupla
- Dictionary: diccionario

En cuanto a los nombres de las variables, no pueden contener espacios en blanco ni empezar por un número. Python es case-sensitive, por lo que hay una distinción entre mayúsculas y minúsculas.

II.2. Primeras nociones

El comando `print` permite imprimir algo por pantalla.

El comando `input` le pide información al usuario que debe aportar por teclado.

```
name = input('Enter your name: ')
print('Hello, ', name)
```

Ese código está dividido en dos líneas de código. La primera línea es bloqueante, ya que le pide al usuario insertar por teclado su nombre. Una vez dado, ese nombre se asigna a la variable name que se emplea para imprimir por pantalla Hello y el nombre que se le ha insertado.

Aunque los tipos de variables sean implícitos, se puede realizar un **casting**, es decir, especificar el tipo de la variable:

- str(variable): convierte en tipo string.

- int(variable): convierte en tipo integer.
- float(variable): convierte en tipo float.

Los comentarios en Python, marcados con #, sirven para poner anotaciones para otras personas o nosotros mismos y que el intérprete va a omitir. Es buena práctica poner comentarios para explicar las partes del código.

```
#Código de prueba
print('Esto es una prueba') #print
```

II.3. Expresiones condicionales

En Python, se pueden escribir condicionales: si algo es verdadero/falso, ocurre algo. Esto se hace mediante `if` y `else` y sirve para cambiar el flujo del programa o tomar decisiones.

```
num_pensado = 4
num = input("Elige un número: ")

if num_pensado == num:
   print("Enhorabuena, has acertado")
else:
   print("Sigue intentando")
```

Si se cumple la condición del if, el código que se ejecuta es el que se encuentra indentado y se continúa sin ejecutarse el bloque else. Si no se cumple el if, el código que se ejecuta es el que se encuentra indentado tras else, y se omite el bloque del if.

En el caso de que haya más condicionales, se puede utilizar `elif` seguido del siguiente condicional, siempre y cuando la condición sea de verdadero o falso:

```
num_pensado = 4
num = input("Elige un número: ")

if num_pensado == num:
   print("Enhorabuena, has acertado")
elif num_pensado < num:
   print("No has acertado, el número es más bajo")
else:
   print("No has acertado, el número es más grande")
```

II.3.1. Operadores condicionales

Los operadores condicionales son aquellos que resultan en True o False. Los operadores son:

- x == y: x es igual a y. Un solo igual asigna el valor de una variable a otra, dos iguales comprueban si son lo mismo.
- x != y: x no es igual a y.
- x < y: x es menor que y.
- x <= y: x es menor o igual que y.
- x > y: x es mayor que y.
- x >= y: x es mayor o igual que y.

II.3.2. Operadores lógicos

Para poder incluir varios condicionales, se utilizan los operadores lógicos **and** y **or**.

El operador `and` implica que las dos condiciones que se unen deben ser True para que el resultado sea True. Para el operador `or`, basta con que una condición sea True para que el resultado sea True.

X	Y	X and Y	X or Y
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

II.4. Bucles o loops

Los bucles permiten repetir ciertas sentencias para ahorrar código redundante.

II.4.1. For loops

Los for loops permiten repetir un código un **número determinado de veces**. La estructura típica de un for loop es:

```
for variable in range(número inicial, número final, incremento):
   código que se quiere repetir

for variable in otra_variable:
   código a repetir
```

Es importante remarcar que el código que se quiere repetir debe estar indentado. Si no se pone número inicial, automáticamente empieza en 0. De igual forma, el incremento es 1 por defecto. El número final no está incluido. Por ello, si sólo se pone `range(número)`, se podría interpretar que se repite el bucle se repite el número indicado de veces. Por ejemplo, utilizando el ejemplo anterior, se podría escribir:

```
num_pensado = 4

for i in range(5):
   num = input("Elige un número: ")
   if num_pensado == int(num):
      print("Enhorabuena, has acertado")
      break
   else:
      print("Sigue intentando")
```

La i es una variable que empieza en 0 y con cada iteración aumenta en uno hasta llegar a 4. Esto se hace automáticamente por el intérprete. Si se acierta el número antes, se sale del bucle mediante `break`.

II.4.2. While loops

Los bucles while se repiten un **número indeterminado de veces**. Este bucle depende de una condición:

```
while condición:
   código que se quiere repetir
```

En estos bucles, es importante que la condición se modifique dentro del bucle para no crear un bucle infinito.

```
num_pensado = 4
fallo = True

while fallo == True:
   num = input("Elige un número: ")
   if num_pensado == num:
      print("Enhorabuena, has acertado")
      fallo = False
   else:
      print("Sigue intentando")
```

Así, cuando el usuario acierta el número, la variable fallo pasa a ser falsa y el bucle se interrumpe.

II.5. Cadenas o strings

Las cadenas son cualquier secuencia de caracteres o números que se encuentran entre comillas simples o dobles. También hay cadenas vacías que no tienen ninguna secuencia entre las comillas. Las cadenas se pueden concatenar mediante el +. También se puede repetir el contenido de una cadena al multiplicarlo con las veces que se quiere repetir.

```
'AB' + 'cd' # ABcd
'Hi' * 2 # HiHi
```

len

Se puede calcular la longitud de una cadena mediante `len()`.

```
cadena = "Hola"
len(cadena) #4
```

in
not in

Para comprobar si una cadena contiene algo, se emplea el operador `in`. Para ver si no contiene algo, se usa `not in`. Esto realmente es una expresión booleana, ya que el resultado es verdadero o falso.

```
secuencia = "ATC"
if "G" in secuencia:
   print("La secuencia contiene guanina".)
else:
   print("La secuencia no contiene guanina.")
```

II.5.1. F-strings

A la hora de imprimir texto, es común alternar cadenas con valores de variables. Esto se puede hacer intercalando las cadenas entre comillas con las variables o con los denominados f-strings (format-strings). Los f-strings son expresiones con una sintaxis más fluida que permite darle formato a cadenas. Para imprimir el valor de variables, basta con poner el nombre de la variable entre llaves ({}). En caso de querer imprimir una llave en un f-string, se debe poner doble.

```
variable = 1
print(f"El valor de la variable es {variable}.")
# Output: El valor de la variable es 1.
print(f"El valor de la variable es {{{variable}}}.")
# Output: El valor de la variable es {1}.
```

II.5.2. Indexación y slicing

Las cadenas se pueden indexar, es decir, referir a una posición en la misma cadena. Esto se realiza escribiendo la cadena (o la variable que contiene la cadena) seguido de la posición que se quiere entre corchetes. La primera posición es un 0, y las subsiguientes van incrementando en uno. A esto se le conoce como **indexación positiva**.

```
cadena = "hola"
cadena[0] #h
cadena[1] #o
cadena[2] #l
cadena[3] #a
```

La **indexación negativa** se refiere a acceder a la posición desde el final de la cadena. El último carácter es -1, y los siguientes van incrementando en 1.

```
cadena = "hola"
cadena[-1] #a
cadena[-2] #l
cadena[-3] #o
cadena[-4] #h
```

En algunas ocasiones, queremos acceder a un fragmento de una cadena. A esto se le conoce como **slicing**, y se emplea con las posiciones: `[primero:último:paso]`. La última posición no está incluida. Los pasos son opcionales, ya que tienen un valor por defecto de 1. Si se omite el primer valor, se empieza desde el primer caracter por defecto. Si se omite el último valor, se termina en el último caracter. Así, si se pone [:], se incluye todo el contenido.

```
cadena = "hola"
cadena[1:3] #ol
cadena[::2] #hl
```

II.5.3. Métodos de cadenas

- lower(): devuelve la cadena con todas las letras en minúscula.
- upper(): devuelve la cadena con todas las letras en mayúscula.
- replace(s1, s2): reemplaza la primera cadena con la segunda.
- count(x): cuenta el número de veces que se encuentra x.
- index(x): devuelve la posición del primer x.
- isalpha(): devuelve True si todos los caracteres de una cadena son letras.
- eval(): evalúa la expresión matemática dentro de la cadena cuando el resultado es numérico.

II.6. Estructuras de datos

En Python hay cuatro estructuras de datos: listas, diccionarios, tuplas y conjuntos o sets.

II.6.1. Listas

La lista es un **conjunto ordenado de objetos**. Pueden ser números, cadenas, booleanos, y se pueden mezclar los distintos tipos. El contenido de una lista se escribe entre corchetes y separando los distintos objetos por comas.

```
lista = [False, 1, 2, "tres"]
```

Las listas guardan algunas similitudes con las cadenas:

- Son indexables y pueden sufrir slicing.
- `len` muestra la cantidad de elementos que se encuentran en cadena.
- El operador `in` muestra si la lista contiene algo.
- Se puede concatenar listas mediante + y multiplicarlas con *.
- Se pueden crear listas vacías con [].
- Se puede iterar sobre ellas.

```
codones = ['AAA', 'GCT', 'CCA']
for codon in codones:
   for nucleotido in codon:
      print(nucleotido)
```

Algunos métodos específicos para trabajar con listas son:

- `append(x)`: añade x al final de la lista.
- `sort()`: ordena la lista de forma alfabética si solo contiene letras y de pequeño a grande si solo contiene números.
- `count(x)`: devuelve el número de veces que aparece x en la lista.
- `index(x)`: devuelve el índice del primer valor de x.
- `reverse()`: revierte el orden de la lista.
- `remove(x)`: elimina el primer x en la lista.
- `pop(i)`: elimina el item en el índice i y devuelve su valor.
- `insert(i,x)`: inserta x en el índice i de la lista.

II.6.2. Diccionarios

Los diccionarios son **conjuntos no ordenados de parejas de clave y valor**. Las claves deben ser únicas e inmutables de cadenas o números, ya que sirven para acceder al valor que tengan asignado.

```
dic_vacio = {} #Diccionario vacío
year ={'Enero':31, 'Febrero':28} #Diccionario de dos elementos
```

Se puede iterar sobre los diccionarios, pero el elemento que se devuelve es la clave. Para que se devuelva el valor, se debe **indexar por la clave**:

```
for month in year:
   print(month) # Imprime las claves
   print(year[month]) #Imprime los valores
```

Se pueden añadir o cambiar el valor de elementos de un diccionario mediante indexación de la clave. Si la clave no existe previamente en el diccionario, se crea con el valor que se le asigna. Si la clave ya existía, se actualiza su valor. Para eliminar elementos de un diccionario, se emplea `del()` y se indexa la clave que se quiere eliminar.

Se puede obtener una lista de todas las claves y de todos los valores con los métodos `.keys()` y `.values()` respectivamente.

II.7. Funciones

Las funciones son una agrupación de líneas de código que se pueden ejecutar llamando el nombre de la funciones un número ilimitado de veces. De esa forma, las funciones permiten simplificar programas largos y mantenerlos. Además, eliminan código redundante. Es muy recomendable que las funciones se encuentren definidas antes de su llamada. Por convenio, todas las funciones se ponen en la parte superior del código, aunque sean llamadas solo al final. La estructura de una función es la siguiente:

```
def nombre_funcion(argumento1, argumento2):
   código que ejecuta la función
```

El número de argumentos tiene que ser el mismo en la definición y en la llamada. No es necesario que los argumentos tengan el mismo nombre en la llamada y en la definición. Las **variables de la función son locales** y sólo se puede acceder a ellas dentro de la misma (en el ámbito de la función).

La mayoría de las funciones devuelven un valor. Esto significa que una variable dentro de la función pase a estar disponible fuera de la misma. Para ello, se emplea `return`, el cual indica qué valor se devuelve. Cuando el código ejecuta un return, en ese momento termina la ejecución de la función, aunque haya más código posterior.

```
def celsius_a_fahrenheit(temperatura_c):
   temperatura_f = temperatura_c * 9/5 + 32
   return temperatura_f

temp_c = 23
temp_f = celsius_a_fahrenheit(temp_c)
#el valor de temp_c se pasa a la función y el valor de temperatura_f se
    le asigna a temp_f
```

Se pueden devolver varios valores en una función separándolos por comas. En la llamada, se pueden poner varias variables que se irán asignando en el mismo orden. Si alguna de las variables que se devuelven no se quieren guardar en una variable global, a la hora de asignar las variables se puede poner un guion bajo.

```
def funcion(arg1, arg2, arg3):
   return arg3, arg2, arg1

v1, v2, v3 = funcion(1, 2, 3) #v1 = 3, v2 = 2, v3 = 1
_, _, variable = funcion (1, 2, 3) # variable = 1
```

Las funciones no deberían escribirse para imprimir nada en pantalla, ya que entonces no sería reusable. Las funciones deberían ser modulares y autocompletadas: que reciban un valor y devuelvan un valor. Se puede hacer que una función imprima algún dato auxiliar en pantalla, pero esa no debería ser su función principal.

II.7.1. Variables locales vs globales

Las variables que se encuentran dentro de una función son variables locales, ya que no existen fuera del ámbito de la función. Por el contrario, variables globales están presentes en el código principal y existen en cualquier parte. Aunque se pueda tener una variable local que se llame igual a otra global sin que se afecten mutuamente, no es buena práctica hacer esto.

II.7.2. Argumentos por defecto

Es posible definir funciones que tengan argumentos por defecto. Esto hace que el argumento sea opcional, es decir, que en caso de no obtener un valor en la llamada, el argumento va a tener un valor predeterminado. Esto se utiliza en aquellas funciones en las que la mayoría de las veces un argumento tiene un valor. Así, solo en los casos excepcionales se debe indicar el valor, y en la mayoría de casos no se debe incluir en la llamada. Los parámetros opcionales se deben definir en la función siempre después de los parámetros obligatorios.

II.8. Librerías/módulos

Muchas veces, vamos a necesitar importar librerías o módulos para nuestro código. Las librerías y los módulos son lo mismo; en informática, se solía hablar de librerías, pero en Python reciben el nombre de módulos. Para importar una librería, se emplea la siguiente estructura:

```
import libreria
import libreria as lib
```

La diferencia entre la segunda línea y la primera es que en la primera, siempre se debe poner `libreria` cuando queramos usar una función que esté ella, mientras que en la segunda se le asigna el alias lib para no tener que escribir el nombre completo. Si solo se quiere importar una función de una librería, se puede escribir:

```
from libreria import funcion
```

```
from libreria import funcion1, funcion2

libreria.funcion1()
```

Esto puede ser beneficioso si de una librería sólo se quiere importar una función, ya que actualmente hay librerías que ocupan mucho espacio y, dependiendo del código, puede ser ineficiente cargarla entera.

> Un script en Python está organizado en tres bloques: la importación de librerías, la definición de funciones, y el código principal.

II.8.1. Instalar nuevos módulos

pip install

Desde la terminal, se puede instalar nuevos módulos de Python mediante el comando `pip install`. Las librerías tienen su propia documentación en la que se detalla las funciones que tienen y cómo usarlas. Esta documentación puede estar disponible en internet o en un repositorio de GitHub. Hay que tener cuidado a la hora de instalar nuevos módulos, ya que pueden darse problemas de seguridad importantes.

Las librerías tienen distintas versiones, por lo que se recomienda utilizar librerías que hayan sido actualizadas recientemente y haya mucha gente colaborando y mejorándola en el repositorio GitHub. Esto asegura que no esté anticuada y que tenga un número mínimo de errores, ya que la mayoría de errores se habrán solucionado ya por todos los usuarios.

II.8.2. Módulo random

random
randint

El módulo random cuenta con varias funciones que permiten obtener valores pseudoaleatorios. Por ejemplo, la función `randint()` saca un valor entero entre los valores inicial y final, ambos incluidos.

```
import random
print(random.randint(1, 10))
```

II.8.3. Módulo requests

El módulo requests permite interactuar con HTTP, es decir, con el formato de navegadores y páginas web. Probablemente lo vayamos a usar en el futuro para acceder a bases de datos biomédicas publicadas en la web.

II.8.4. Módulo sys

El módulo sys permite acceder a algunas funciones del sistema. Por ejemplo, `sys.exit()` termina la ejecución de un script.

II.9. Ficheros y directorios

Trabajar con ficheros tiene un ciclo muy concreto: abrirlo, leerlo o escribirlo, y cerrarlo. Si se abren muchos ficheros y no se cierran, el ordenador se puede quedar sin memoria, por lo que es importante cerrarlos cuando ya no sean necesarios.

open()
readlines()
write()
close()

Para **abrir un fichero** en Python, se emplea `open(path/del/fichero, "modo")`. Modo hace referencia a cómo queremos abrir el fichero (el modo de lectura): para leer ("r"), escribir ("w"), leer y escribir ("rw") o append ("a"). Así, si solo queremos leer los datos de un fichero, lo abrimos con permisos de lectura y, al intentar escribir, daría un error. En la práctica, no hay diferencia entre los modos w y rw. Cuando se escribe en ficheros, se reescriben por completo; si queremos añadir texto al final del fichero, debemos usar el modo a que no sobreescribe el fichero. Para poder (sobre)escribir un fichero, se necesitan los permisos del sistema operativo para poder modificar el archivo. Para obtener las líneas que tiene el fichero, se utiliza el método `readlines()`. También se puede iterar sobre el fichero para obtener las líneas. Si queremos escribir en un fichero, se utiliza `write("contenido")`. Para cerrar un fichero una vez leído y modificado, se utiliza `close()`.

```
fichero = open(ejemplo.txt) #el fichero se encuentra en el mismo
    directorio

contenido = fichero.readlines()
for line in fichero:
   print(line)

fichero.write("Nuevo contenido")

fichero.close()
```

II.9.1. Ejercicio

Escribir una función que escriba en un fichero una secuencia de ADN aleatoria de una longitud que reciba por parámetro.

```
import random # alternativa: from random import choice

def escribe_secuencia_fichero(secuencia, nombre_fichero):
   fichero = open(nombre_fichero, "w")
   fichero.write(secuencia)
   fichero.close()

def generar_secuencia(longitud):
   bases = ["A", "T", "C", "G"]
   secuencia_aleatoria = ""
   for i in range(longitud):
      secuencia_aleatoria += random.choice(bases)
   return secuencia_aleatoria
```

```
filename = "secuencia_aleatoria.txt"
num_bases = 10
secuencia = generar_secuencia(num_bases)
escribe_secuencia_fichero(secuencia, filename)
```

II.10. Excepciones

Una excepción es un **error irrecuperable**. Algunos errores en el programa son semánticos, es decir, son nuestros de programación. Estos son los peores porque aparentemente el programa funciona. Los errores irrecuperables hacen que el programa se detenga, como por ejemplo acceder a un fichero que no exista. Sabiendo que un código puede generar un error irrecuperable, se puede encapsular ese código dentro de una estructura especial:

```
try:
   código que debe intentar
except Error: #ZeroDivisionError, PermissionError, IndexError, ...
   código que hacer entonces
```

Así, se captura la excepción y se da la oportunidad de hacer otra cosa. Para saber qué funciones pueden producir excepciones, se debe mirar la documentación, en la que se especifica también el tipo de error que se da. Se pueden dar varias excepciones con distintos errores para hacer el código más robusto.

Hay una excepción que recibe el nombre de Exception. Si se captura esta excepción, se captura cualquier tipo de error irrecuperable. El problema de esto es que aporta muy poca información de error y no se gestiona. Así, el programa no se para, pero tampoco se repara. Por tanto, hacer esto no es una buena práctica.

Las excepciones pueden tener un else, el cual se ejecutará si no se lanza la excepción. De esa forma, no es necesario poner sys.exit(). Por último, existe la cláusula finally, la cual se ejecuta siempre, se lance o no una excepción. Esto es muy útil para cerrar ficheros (sobre todo si son grandes) y que no se queden en la memoria RAM cuando se da alguna excepción no capturada.

```
try:
   fichero = open("fichero.txt", "r")
except FileNotFoundError:
   print("El fichero no existe")
else:
   #código que podría fallar en algún momento
finally:
   fichero.close()
```

II.11. Programación orientada a objetos (POO)

Tradicionalmente, el código era lineal y se ejecutaba de forma lineal. La programación orientada a objetos (POO) presenta otro paradigma, ya que encapsula el código en objetos y clases para poder manipularlos directamente. La diferencia con la programación procedural es que la programación orientada a objetos se basa en modelizar las entidades con las que trabaja el programa en forma de clases y encapsular la funcionalidad en los objetos. Así, un objeto en Python es una colección única de datos (atributos) y comportamiento (métodos).

La programación orientada a objetos permite abordar problemas complejos de una forma más sencilla, pudiendo implementar más funciones sin comprometer la estabilidad de un proyecto. Además, permite reutilizar el código mediante la implementación de la abstracción, que hace que el código sea más conciso y legible. La clase es como una plantilla que permite crear objetos personalizados basados en los atributos y métodos que se definan. Por otro lado, las instancias son objetos individuales de una clase, que tiene una dirección de memoria única.

Ejemplo: Coche

Un vehículo tiene una serie de atributos: ruedas, tipo de motor, color, velocidad máxima, carburante. También tiene una serie de métodos, es decir, acciones que puede realizar: arrancar, repostar, acelerar. Las clases son el concepto abstracto, mientras que el objeto es la personificación de la clase. El concepto de vehículo sería una clase. El objeto concreto instanciado sería un Tesla, una moto de marca X, etc.

Para crear una clase en Python, se utiliza la palabra clave `class` seguida del nombre de la clase, cuya convención dicta que se escriba en mayúsculas para poder diferenciarlas de las variables. Tras haber creado la clase, se pueden añadir los atributos de la clase y que serán iguales para todos los objetos de esa clase que se construyan. A continuación, se definen los métodos de la clase. El primer método siempre debe ser el método constructor (`__init__`), el cual es llamado por Python cada vez que se instancie un objeto. El constructor es el que crea el estado inicial del objeto con una serie de parámetros. Todos los métodos deben incluir el parámetro self, que se refiere a la instancia de la clase (el objeto en sí). Después, se le pueden añadir los parámetros que se deseen como atributos de la instancia y que serán diferentes para cada objeto. Se deberán incluir como argumentos a la hora de crear los objetos, y para acceder a ellos una vez creada una instancia, se utiliza la notación de puntos. También se pueden crear más métodos de forma similar a como se creaban las funciones y se invoca igual.

```
class Dog:

    kind = 'canine' # atributo de la clase

    def __init__(self, name): #constructor
        self.name = name

    def makeNoise(self): #método
        print('Guau!')
```

```
#Uso de la clase
toby = Dog('Toby') #creación de un objeto
toby.makeNoise() #acceder al método del objeto
print('Toby is a', toby.kind) #acceder al atributo del objeto
```

El método __str__ devuelve una cadena que puede contener la información que se quiera sobre un objeto.

II.11.1. Herencia

La herencia permite definir múltiples subclases a partir de una clase ya definida. Las subclases heredan todas las características (atributos) y algunos comportamientos comunes (métodos). Los atributos creados en la subclase no estarán presentes en la clase padre.

II.12. Persistencia de datos

La persistencia de datos es la forma de guardar y leer datos para que no se pierdan ni se dañen. Hay tres maneras de persistir datos en un ordenador:

- Ficheros: pueden ser de texto o binarios. La diferencia es que los ficheros de texto solo pueden tener algunos de los caracteres ASCII, mientras que los ficheros binarios pueden guardar cualquier tipo de texto. Se recomienda abrir los ficheros como binarios (utilizando "rb"), ya que también permiten leer el fichero de texto, mientras que si se abre un fichero como si fuera de texto, no se pueden leer los ficheros binarios. Esto es adecuado para ficheros pequeños y medianos, ya que se irán leyendo poco a poco. Los ficheros grandes también se pueden abrir así, pero pueden tardar más.
- Serialización de objetos: sirve para guardar objetos pequeños y restaurarlos mediante la librería pickle. Así, cuando se trabaja con objetos, cada x líneas se puede guardar en un fichero el estado del objeto a modo de backup mediante `pickle.dump(objeto, fichero)`. En caso necesario, se puede cargar el fichero para leer su contenido con `pickle.load(fichero)`. Normalmente se guarda con extensión .bin, que es la extensión de un fichero binario.
- Bases de datos relacionales: Python permite crear una tabla de base de datos para cada clase. Esto se hace mediante la librería MySQLdb. Normalmente, las bases de datos se obtienen desde un servidor externo al que nos tenemos que conectar primero para acceder. Esto se hace mediante `MySQLdb.connect(servidor, usuario, contraseña, base de datos)`. Así, desde Python se pueden escribir programas que hablen con la base de datos. Todas las operaciones que se pueden hacer en SQL se pueden hacer desde Python.

```
import MySQLdb
db = MySQLdb.connect("localhost", "testuser", "test123", "TESTDB")
cursor = db.cursor()
sql = "INSERT INTO employee (first_name, last_name, age, sex, income)
    VALUES ('%s', '%s', '%d', '%c', '%d')" % ('Marc', 'Mohan', 20, 'M',
    2000)

try:
   cursor.execute(sql)
   db.commit()
except:
   db.rollback
   #vuelve al estado previo en caso de que no fuese posible actualizar
       todos los datos a la base de datos por una caída o así

db.close()
```

Capítulo III

Bases de datos

III.1. Introducción a las bases de datos relacionales

En cualquier empresa o laboratorio, se va a necesitar acceder a una gran cantidad de datos. Los **sistemas de manejo de los datos (DBMS por sus siglas en inglés)** permiten tener todos los datos juntos y relacionados para facilitar su acceso, estando de forma conveniente y eficiente de usar. Las bases de datos pueden ser muy largas y tocan todos los aspectos de nuestras vidas, desde las transacciones bancarias al registro de notas de la universidad.

El uso de bases de datos tiene muchas ventajas, como tener un acceso más sencillo a los datos que desde múltiples ficheros e integrar los datos. Utilizar ficheros aislados puede conllevar a datos redundantes e inconsistentes, aislamiento de los datos, atomicidad de las actualizaciones, concurrencia de los accesos y problemas de seguridad.

Existen bases de datos relacionales, en las cuales los modelos de datos son entidad-relación. Hay otros paradigmas como jerárquicas, no relacionales o basados en objetos.

Las arquitecturas suelen ser cliente-servidor, pero también puede ser paralela, distribuida o centralizada. En esta asignatura utilizaremos PostgreSQL, que es de arquitectura cliente-servidor.

A partir de la década de 1950 se empezó a hablar del almacenamiento de datos en cintas magnéticas. A finales de 1960, se empezó a guardar los datos, pero no fue hasta 1980 cuando se creó el lenguaje SQL. En 1990 empiezan a aparecer las redes, la web y el data mining. A principios de los 2000 comenzaron XML y XQuery. En el siglo actual hay millones de datos (big data).

III.2. Modelo de entidad relacional

III.2.1. Esquema entidad-relación (ER)

El esquema o diagrama entidad-relación (ER) es básicamente un gráfico que muestra de forma sencilla cómo se modela la problemática que se quiere abordar con una base de datos. Va a estar compuesto por un conjunto de entidades que pueden tener atributos que los describas. Entre las entidades hay relaciones. En un esquema ER, los rectángulos representan las entidades, los diamantes las relaciones, las elipses los atributos, las líneas unen los atributos con las entidades y las entidades entre sí, y el subrayado indica los atributos clave primarios.

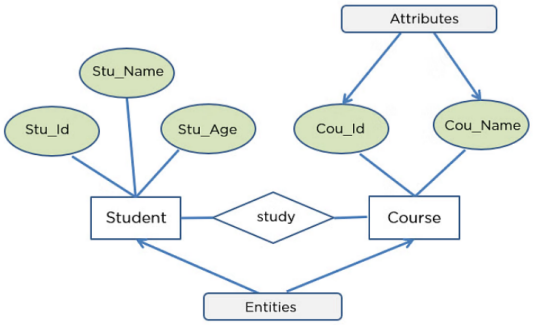

Figura III.1: *Esquema ER.*

Una **entidad** es un objeto o algo que pueda contener muchas instancias. Tiene atributos que le caracterizan y se deben poder distinguir de otras entidades mediante el contenido de los atributos. Normalmente, las entidades reciben un nombre en singular (por una sola palabra) y se representan con rectángulos. Las entidades débiles son aquellas cuya supervivencia depende de otra identidad.

Los **atributos** son propiedades de las entidades, y deben estar asignados a una entidad. El conjunto de valores para cada atributo se le conoce como dominio del atributo. Normalmente, los valores de los atributos son atómicos, es decir, indivisibles.

Las **relaciones** son asociaciones entre las distintas entidades. Puede haber atributos en las relaciones. Las relaciones pueden ir también a la misma entidad en forma de bucle. En ese caso, se especifican los roles.

Es muy importante no poner en una entidad un atributo de otra entidad con la que esté relacionada.

III.2.2. Claves primarias

Una clave primaria permite identificar de manera única cada identidad. Puede ser uno o varios atributos. La clave candidata es la clave mínima primaria. Aunque puedan existir varias claves candidatas, solo una de ellas debería ser clave primaria.

III.2.3. Mapa de cardinalidades

El mapa de cardinalidades expresa el número de entidades a los que se les puede asociar a otra entidad por una relación. Se representan por una flecha cuando es una relación individual o por una línea cuando son muchas. Hay tres cardinalidades: uno-a-uno (un cliente sólo puede pedir un préstamo, y un préstamo puede ser de un solo cliente), uno-a-muchos (un préstamo puede ser de un solo cliente, pero un cliente puede tener varios préstamos) o muchos-a-muchos (un cliente puede tener varios préstamos, y cada préstamo puede ser de varios clientes).

III.2.4. Especialización, jerarquía o generalización

ISA se conoce como especialización, jerarquía o generalización. Viene del inglés "is a", y permite que una entidad se especialice en otras entidades. Aunque todas las entidades tengan los mismos atributos, después de la especialización las subentidades van a tener otros atributos y heredan los anteriores. Cuando en el esquema se va de arriba a abajo, se trata de una especialización, mientras que si se va de abajo a arriba se trata de una generalización. Sólo se heredan los atributos conforme se va especializando.

III.2.5. Notación ER

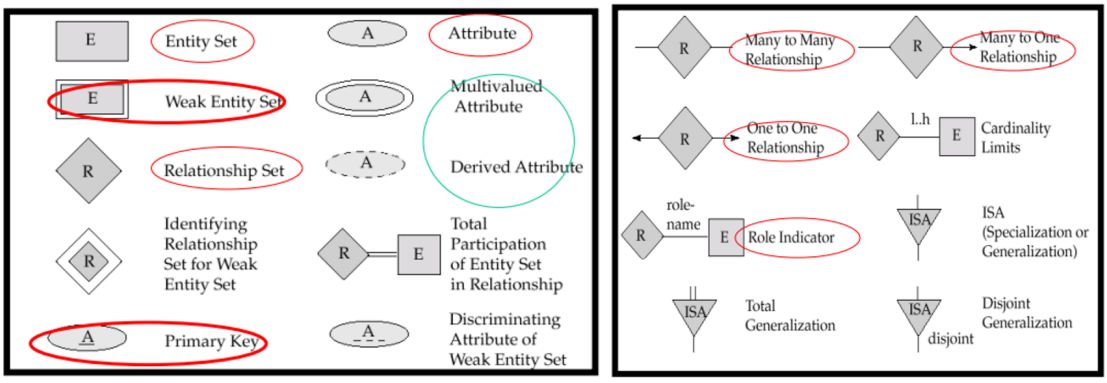

Figura III.2: *Resumen de la notación de un esquema ER. Los conceptos rodeados en rojo son los importantes. Los que están rodeados en azul no se recomiendan y los deberíamos evitar.*

III.2.6. Ejercicio 1: sistema de reserva de aulas para la universidad

Vamos a hacer un sistema de reserva de salas para una universidad. Debe ser posible acceder al usuario que ha reservado cada sala, a las salas reservadas en un día concreto, o aulas concretas. Los profesores pueden reservar cualquier sala, pero los estudiantes solo pueden reservar las salas de propósito general o salas de seminario. Los usuarios se deben identificar por usuario y contraseña.

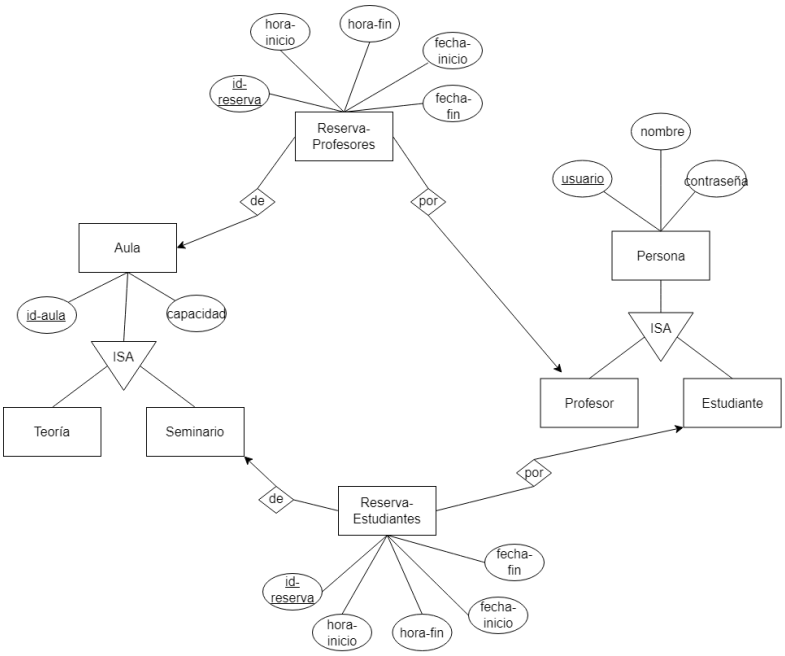

Figura III.3: *Solución del ejercicio.*

III.2.7. Modelo relacional: esquema ER a tablas

Desde el esquema ER, se debe convertir al modelo relacional o a un formato tabla. Para cada entidad hay una tabla única que tiene un número de columnas, que suele corresponder con los atributos, que tienen nombres únicos. Las relaciones n-n (muchos a muchos) tienen tablas separadas que consisten en las claves primarias de las dos identidades que relaciona. Las relaciones n-1 (muchos a uno) y 1-n (uno a muchos) pueden representar añadiendo un atributo extra en la parte de muchos que contenga la clave primaria de la parte uno.

A la hora de representar la especialización como tablas, hay dos opciones, y cada gestor lo hace de una manera: las especializaciones adquieren solo la clave primaria o adquiere todos los atributos de la generalización.

Los pasos para crear tablas son:

1. Identificar las claves primarias
2. Identificar entidades
3. Identificar los atributos redundantes de las entidades y especializar
4. Identificar relaciones n-n
5. Todas las entidades producen una tabla
6. Todas las relaciones n-n producen una tabla
7. Todas las relaciones n-1 añaden una columna a la entidad n.

III.2.8. Resumen

El primer paso es crear el esquema ER identificando las entidades con sus atributos y relaciones, las cardinalidades y las especializaciones. Después, se debe reducir el esquema a tablas, identificando las claves primarias, entidades y relaciones n-n. Las entidades y las relaciones n-n producen una tabla. Todas las relaciones n-1 añaden columnas a la entidad de muchos.

III.2.9. Ejercicio 2: gestión de mercancías

Una empresa de gestión de mercancías desea tener almacenados los datos de sus clientes, los productos y los proveedores relacionados con los distintos pedidos que realizan los clientes. También interesa llevar un control sobre los tipos de los productos.

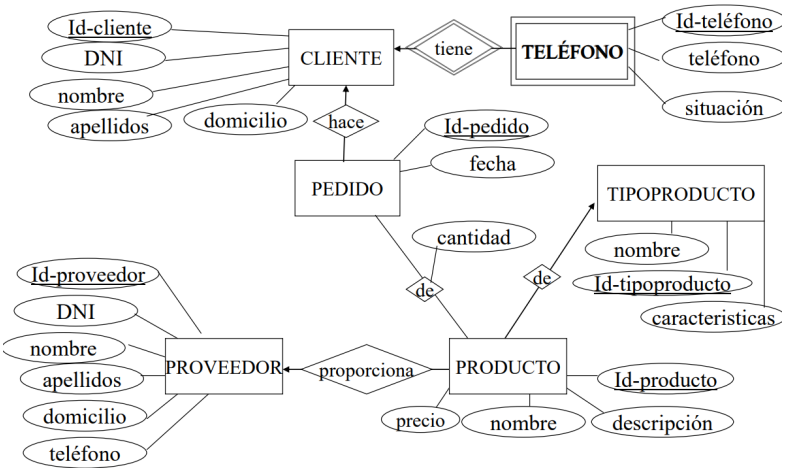

Figura III.4: *Solución del ejercicio.*

- Cliente (id-cliente, DNI, nombre, apellidos, domicilio)
- Teléfono (id-telefono, id-cliente ↑ , telefono, situación)
- TipoProducto (id-tipoProducto, nombre, características)
- Proveedor (id-proveedor, DNI, empresa, CIF, teléfono)
- Producto (id-producto, id-tipoProducto ↑, id-proveedor↑, nombre, descripción, precio)
- Pedido (id-pedido, id-cliente ↑, fecha)
- PedidoProducto / Factura (id-pedido ↑, id-producto ↑, cantidad)

III.3. SQL: Structured Query Language

SQL (Structured Query Language) es el lenguaje estándar de las bases de datos relacionales. Es un lenguaje declarativo que permite especificar diversos tipos de

operaciones sobre estas. Es capaz de conjugar las operaciones del álgebra y el cálculo relacional con operadores adicionales, y definir así consultas para recuperar o modificar información de bases de datos, así como hacer cambios en ellas.

Los tipos de comandos en SQL se agrupan en dos categorías o sub-lenguajes:

- DDL (Definition Data Language): permite definir el esquema de bases de datos, creando relaciones (tablas), campos e índices, o modificando las definiciones existentes.
- DML (Data Manipulation Language): permiten generar consultas para ordenar, filtrar y extraer datos de la base de datos, así como insertar, modificar y eliminar registros de las tablas.

Todas las queries o consultas en SQL deben terminar en punto y coma (;). Los comentarios se ponen con dos guiones.

III.3.1. SQL-Data Definition Language (DDL)

SQL-DDL proporciona comandos para definir relaciones y esquemas, borrar relaciones y modificarlas. Permite especificar las relaciones y la información de cada información: el esquema de cada relación, los valores asociados con cada atributo, restricciones de integridad, seguridad y autorización, y el conjunto de índices mantenidos en cada relación.

Los distintos tipos de datos en SQL son:

- **char(n):** cadena de caracteres de longitud fija n.
- **varchar(n):** cadena de caracteres de longitud variable, siendo máximo n.
- **integer**: número entero
- **smallint**: pequeño entero, utiliza menos números y emplea menos memoria
- **numeric(p,d)**: número con p dígitos y d decimales.
- **float(n):** número flotante con al menos n dígitos.
- **null**: valor nulo
- **date**: muestra la fecha en formato año-mes-día
- **time**: muestra la hora en formato hora, minuto y segundo.
- **timestamp**: muestra la fecha y la hora
- **interval**: periodo de tiempo

La sintaxis básica para crear una tabla y eliminarla es:

```
CREATE TABLE nombre-tabla (nombre-columna tipo-columna, nombre-columna2
    tipo-columna2);
DROP TABLE nombre-tabla;
```

Se puede elegir el nombre para las tablas, pero hay algunos **nombres reservados** que no pueden adoptar: oid, tableoid, xmin, cmin, xmax, cmax, ctid.

Si un valor no se asigna a un atributo, se asigna `null` por defecto excepto si se ha predefinido un valor por defecto a la hora de crear la relación. Esto es útil porque, aunque nosotros creemos la base de datos, ésta será utilizada por muchas otras personas, de forma que los campos con valores predeterminados sirven para evitar errores.

```
CREATE SEQUENCE product_no_seq;
CREATE TABLE product(
   product_no integer PRIMARY KEY DEFAULT nextval('product_no_seq'),
   name text, --a comment
   price numeric(10,2) DEFAULT 9.99
);
CREATE UNIQUE INDEX product_no ON product ( product_no );
```

Las **restricciones de integridad** son:

- check(P): fuerza que el resultado de una condición sea True o Null.
- not null: atributo que no acepta un valor nulo
- unique (atributos): si se pone tras un atributo, no puede haber valores repetidos en ese atributo. También se puede escribir debajo añadiendo entre paréntesis los atributos que no se pueden repetir en combinación, es decir, se puede repetir un atributo mientras que el otro sea diferente, pero no los dos juntos.
- primary key: un atributo se establece como clave primaria. Cuando se utiliza detrás de un atributo, solo se puede poner una vez. Si se quiere poner más de un campo como clave primaria, se debe poner como tupla, es decir, primary key (atributo1, atributo2). Las claves primarias no se pueden repetir, pero jamás pueden ser valores nulos. Esto lo diferencia con unique, ya que ahí sí puede haber valores nulos.
- foreign key references r: se utiliza un atributo presente en otra tabla a la que se referencia. El valor de la foreign key debe existir en la tabla principal. De igual forma, no se puede borrar en la tabla original si está referenciado. Si se borra en la referencia, la tabla original no se modifica a no ser que se especifique que se borre en cascada (lo cual no se recomienda).

```
CREATE TABLE product(
   product_no integer NOT NULL,
   name text,
   id_product_type REFERENCES product_type,
   price numeric(10,2) CONSTRAINT precio_positivo CHECK (price > 0)
```

```
);
-- Crear una restricción con nombre permite una mejor trazabilidad de
    errores
```

III.3.2. SQL-Data Manipulation Language (DML)

SQL define cuatro sentencias de manipulación de datos principales:

- **Insert:** para insertar registros en la base de datos.
- **Update:** encargado de modificar los valores de los campos indicados, en los registros que cumplan cierta condición.
- **Delete:** encargado de eliminar los registros de una tabla que cumplan una condición.
- **Select:** encargado de consultar registros de la base de datos que satisfagan una condición determinada. Se utiliza para indicar al motor de datos que devuelva información de las bases de datos. Utilizado con un asterisco (*), se pide que se seleccionen todos los campos de una tabla.

A estas sentencias (salvo insert) se les debe añadir modificadores para indicar a qué tupas afectan. Ciertos modificadores (cláusulas) permiten generar criterios para definir los datos a manipular o seleccionar.

- **From:** establece la tabla o tablas de la/s que seleccionar los registros.
- **Where:** condiciones que los registros a seleccionar deban cumplir.
- **Group by:** criterio para agrupar los registros seleccionados.
- **Having:** establece condiciones sobre datos calculados para los grupos generados por `group by`.
- **Order by:** ordena los registros seleccionados según el orden indicado.

Se pueden insertar valores en una tabla de dos formas:

```
-- Opción 1: sin especificar los nombres de las columnas, solo los
    valores
INSERT INTO nombre-tabla VALUES (valores respetando el orden del create
    table);
-- Opción 2: especificando el nombre de los atributos
INSERT INTO nombre-tabla (atributo1, atributo2, atributo3) VALUES
    (valor1, valor2, null)
```

Para cambiar la estructura de una base de datos ya creada, se podría borrar y volver a crear, pero cuando ya hay datos insertados, esto no es eficiente. Para ello, se puede realizar `alter table`:

```
-- Algunos ejemplos
ALTER TABLE nombre-tabla ADD CHECK (condición);
ALTER TABLE nombre-tabla ALTER COLUMN nombre-columna SET NOT NULL;
```