Programación con Python

Bases de datos y desarrollo web

Quedan rigurosamente prohibidas, sin la autorización escrita de los titulares del Copyright, bajo las sanciones establecidas en las leyes, la reproducción total o parcial de esta obra por cualquier medio o procedimiento, comprendidos la reprografía y el tratamiento informático, y la distribución de ejemplares de ella mediante alquiler o préstamo públicos. Diríjase a CEDRO (Centro Español de Derechos Reprográficos, www.cedro.org) si necesita fotocopiar o escanear algún fragmento de esta obra.

INICIATIVA Y COORDINACIÓN
DEUSTO FORMACIÓN

COLABORADORES
Realización:
E-Mafe E-Learning Solutions S.L.

Elaboración de contenidos:
Claudio García Martorell
Licenciado IT Telecomunicaciones especialidad Telemática.
Postgrado en Sistemas de Comunicación y Certificación en Business Intelligence TargIT University.
Concejal de Innovación y Tecnología.
Ponente y docente en distintas universidades y eventos.
Josep Estarlich Pau
Técnico de Ingeniería Informática.
Director Área de Software de la empresa Dismuntel.
Participante en proyectos con Python, C#, R y PHP orientados a *Machine Learning* y a la Inteligencia Artificial.

Supervisión técnica y pedagógica:
Grufium educación y excelencia

Coordinación editorial:
Grufium educación y excelencia

© Grufium educación y excelencia, S.L.
Barcelona (España), 2021

Primera edición: septiembre 2021

ISBN: 978-84-1300-688-8 (Obra completa)
ISBN: 978-84-1300-695-6 (Bases de datos y desarrollo web)

Depósito Legal: B 11038-2021

Impreso por:
SERVINFORM
Avenida de los Premios Nobel, 37
Polígono Casablanca
28850 Torrejón de Ardoz (Madrid)

Printed in Spain
Impreso en España

Esquema de contenido

Introducción

En este módulo vamos a ver los principios fundamentales de las bases de datos, su sintaxis, sus aplicaciones principales y, sobre todo, cómo comunicarnos con ellas desde Python. Esto nos abrirá la puerta a poder extraer o guardar información sensible que podemos obtener leyendo documentos externos en los lenguajes que se usan actualmente para la transmisión de información, como son XML o JSON.

Será necesario, pues, aprender también algo de seguridad para protegernos de posibles ataques a nuestro sistema de información, ya que hemos de asegurarnos de que nuestro software sea lo más robusto posible.

Este es sin duda el módulo que más nos va a ayudar a preparar y afianzar conceptos con la vista puesta en la práctica final.

1. Conexión con bases de datos en Python

Las bases de datos se utilizan para guardar información de forma persistente una vez que nuestra aplicación ya no se utiliza, para poder recuperarla en un futuro. Sin embargo, hay muchas formas de guardar esa información y de elegir la base de datos más adecuada para los requisitos de nuestro *software*, por lo que vamos a ver qué criterios hemos de seguir y de qué forma hemos de preparar la información en Python para guardarla con el tipo de datos óptimo para cada registro.

1.1. ¿Qué es una base de datos?

Entendemos por base de datos a un contenedor de información, un archivo o una serie de ellos que contienen información mejor o peor organizada, pero que se encuentra en un formato que persiste en la memoria. A diferencia de cuando almacenábamos valores en variables, que se mantenían hasta que el programa finalizaba, en este caso la información se queda guardada. Y cuando hablamos de formatos, nos estamos refiriendo a formatos lógicos, como archivos, tablas y sistemas, y a soportes físicos, como discos duros. Esta es la gran ventaja de las bases de datos y la mayor diferencia con todas las formas de guardado de información que hemos visto hasta ahora; por primera vez, podemos guardar información para usarla más adelante.

Por lo general, una base de datos contiene tablas; las tablas contienen campos, y cada campo contiene un valor. Esta es una forma estructurada de almacenar la información, y la calidad de la base de datos residirá en lo bien que hayamos creado la estructura de tablas y definido los campos. Por norma general, los datos se guardan en relación con un índice, es decir: estamos guardando información en una especie de diccionario de Python, a modo [clave:valor]; y lo eficiente o potente que sea la base de datos dependerá de cómo hayamos definido las claves y los valores.

En las bases de datos bien estructuradas será más fácil y rápido buscar y mostrar la información, ya que las consultas serán más simples y se ejecutarán de una manera más eficaz que en otra base de datos en la que no estén bien definidos los índices, haya duplicidad de datos, etc.

En esta última, las consultas serán más complejas, tanto de implementar por el desarrollador como de interpretar por el *software*.

Si analizamos la forma que tenemos de introducir información y usar una hoja de cálculo, podemos hacernos una representación mental de cómo es internamente una base de datos, ya que estaremos haciendo uso de conceptos como tabla, fila, columna o valor, que son básicamente los términos principales por los que se rige una base de datos, independientemente del tipo que sea.

Y es que, dependiendo de la forma en la que se guardan los datos, los tipos o la sintaxis usados para acceder a la información, podremos estar trabajando con distintos tipos de bases de datos, ya que hay muchas alternativas; cada sistema de base de datos está optimizado para un tipo de operaciones o para una manera de trabajar concreta, siempre en aras de favorecer la eficacia y la velocidad de procesamiento de órdenes. Aunque los veremos en profundidad más adelante, algunos de los sistemas más usados son:

- Microsoft SQL Server
- Oracle
- MySQL
- PLSQL
- PostgreSQL
- SQLite

Todos los datos van referenciados con un índice, es decir, un identificador único que señala ese registro de información de manera única. No puede haber dos índices iguales:

ID	NOMBRE	PROVINCIA	COCHE
01	Diego	Valencia	Renault
02	Pablo	Teruel	Seat
03	Simeón	Huesca	Ford

Sin embargo, sí pueden existir dos registros iguales, pero en índices diferentes:

ID	NOMBRE	PROVINCIA	COCHE
01	Diego	Valencia	Renault
02	Diego	Valencia	Renault
03	Simeón	Huesca	Ford

Podemos operar con las bases de datos de dos formas, principalmente:

- **De forma directa:**

  - Utilizaremos un *software* de gestión de bases de datos (SGBD) mediante el cual podremos ver toda la información contenida en la base de datos y realizar todas las operaciones que nos permita el sistema (crear estructuras y relaciones, modificar, borrar, etc.). Hoy en día existen programas de este tipo muy especializados en varias áreas:

    - En algún tipo de bases de datos (Oracle, Microsoft SQL Server...).

    - Referente a una técnica o conjunto de reglas concretas para acceder a la información (*machine learning*, *big data*...).

    - Generalistas, para acceder a bases de datos más estándar (Toad, Tora...).

- **De forma indirecta/remota:**

  - Mediante nuestro código. Es decir, podemos escribir un *software* que comunique con un tipo de base de datos y que, en función de las acciones que realice el usuario, nuestro programa traduzca esas directrices en órdenes para la base de datos. Por ejemplo:

    - Se carga la página de "Nuevo usuario". En ese momento ya nos comunicamos con la base de datos, confirmando que es posible la conexión y que está "escuchando" por si hay que hacer alguna operación.

    - El usuario escribe su nombre en un formulario y hace clic en el botón Guardar.

    - En nuestro código, si el botón de Guardar se pulsa, se lanza una orden de guardado en la base de datos/tabla/campo correspondiente.

De la misma forma, podemos poner a disposición del usuario formularios de borrado, modificación, etc.

1.2. Tipos y características

Por su flexibilidad de modificación, las bases de datos se dividen en:

- **Bases de datos dinámicas:** los datos pueden actualizarse en tiempo real. Sirven para consultar y guardar o modificar información.

- **Bases de datos estáticas:** en este tipo de bases los datos no pueden modificarse una vez introducidos. Sirven eminentemente para la consulta de información.

Por su organización, se clasifican en:

- **Jerárquicas:**

  - Los registros de este tipo de base de datos se denominan **nodos**; están estructuradas en forma de árbol.

  - Cada nodo hijo tiene un solo nodo padre.

  - Están, como indica su nombre, jerarquizadas mediante un conjunto de reglas y directrices.

  - A partir del nodo raíz, se puede llegar a cualquier nodo inferior, entendiendo una estructura de nodos-padre y nodos-hijo.

  - Resultan óptimas cuando hay grandes volúmenes de información, ya que la estructura de árbol favorece la rapidez de navegación entre registros.

  - Los datos son independientes unos de otros

  - Cuanto más distante esté un nodo-hijo del nodo-raíz, más tiempo se tardará en acceder a él.

- **En red:**

  - Similar a la base de datos jerárquica, pero aquí un nodo hijo puede tener varios nodos-padre.

  - Por esta razón, si bien guardan muchas similitudes con las bases de datos jerárquicas, las bases de datos en red son más potentes y a la vez complejas.

- **Relacionales:**

  - Son las más usadas hoy en día.

  - Permiten crear todo tipo de datos y relacionarlos entre sí.

  - Los datos se organizan en tablas, por lo que se pueden relacionar entre sí elementos de diferentes tablas a partir de sus índices.

  - Son muy fáciles de gestionar, lo que abre la posibilidad a que su uso se extienda entre la comunidad.

  - Son muy eficientes con tipos de datos simples, pero poco eficientes a la hora de guardar material multimedia o información geográfica.

- **Deductivas:**

  - Son las llamadas bases de datos lógicas.

  - Se utilizan principalmente como fuente de información para los buscadores (aunque este uso no es exclusivo).

  - Los datos son consultados a través de búsquedas que utilizan un marco de normas y reglas de acceso previamente almacenadas.

  - Permiten expresar las consultas mediante reglas lógicas.

  - Son muy eficientes para tipos de datos complejos.

  - Hacen uso de reglas matemáticas, sobre todo de las comparaciones de tipo lógico.

- **Multidimensionales:**

  - Este es posiblemente el tipo de base de datos más complejo.

  - Se basan en el concepto de los cubos de datos (varias dimensiones para representar los datos).

  - Cada dato o pieza de información viene definido por la conjunción de tres o más atributos diferentes.

  - No existe jerarquía alguna a la hora de plantear la estructura.

- Esta disposición de cubos de información facilita mucho la búsqueda y la modificación de datos.

- Es la que, en proporción, menos espacio de almacenamiento representa.

- Es susceptible de albergar grandes cantidades de información.

1.3. Gestión de bases de datos

Hoy en día existe una gran cantidad de gestores de bases de datos, también llamados **motores de bases de datos**. Podemos decir que son los mecanismos internos que rigen la forma en la que se guarda la información en la base de datos y que establecen las reglas para que podamos acceder a dicha información. Dichas reglas, aunque básicamente van a permitirnos hacer lo mismo (guardar, leer, crear, modificar, filtrar...), presentarán ligeras diferencias sintácticas de un gestor a otro.

A continuación, enumeraremos y explicaremos los motores de bases de datos con los que, en la actualidad, se trabaja principalmente en Python.

1.3.1. MySQL

MySQL es, sin duda, el motor de bases de datos más utilizado. Permite la coincidencia de varios usuarios concurrentes, así como el lanzamiento de varios hilos simultáneos. Además, la gran mayoría de las webs actuales tienen en su *back-end* este motor. Resulta extremadamente sencillo de instalar y configurar. Ahora bien, no está pensado para el tratamiento de grandes bloques de información ni para albergar muchos registros.

1.3.2. SQLite

SQLite no deja de ser una biblioteca en C que nos da la posibilidad de hacer operaciones con datos. No necesita un servidor (y, por tanto, tampoco hace falta configuración al respecto), por lo que ocupa mucho menos espacio que otros motores de bases de datos. Cumple a la perfección con las ideas de consistencia, aislamiento, atomicidad y durabilidad.

Certificación

Es importante interiorizar sobre la gestión de bases de datos, pues en numerosas preguntas de la certificación se hace referencia a este tema.

1.3.3. MongoDB

Es el motor de base de datos que no depende del lenguaje SQL más empleado en la actualidad. Se basa en ficheros que utilizan el lenguaje BSON. Es utilizado por empresas como Facebook, Cisco o Google.

1.3.4. MariaDB

Es una evolución del motor MySQL, por lo que guarda muchas similitudes con este gestor (además de ser 100 % compatible). Se popularizó tras la compra de MySQL por parte de Oracle. Se diferencia de MySQL principalmente por la cantidad de extensiones que tiene y por el hecho de ser de código abierto.

1.4. El lenguaje SQL

El lenguaje SQL es el más usado en bases de datos. Responde a las siglas de *Structured Query Language*, o "lenguaje de consultas estructuradas". Se puede decir que es un lenguaje de programación diseñado única y exclusivamente para operar con bases de datos relacionales. La sintaxis de este lenguaje se puede considerar de alto nivel por la similitud que guardan sus instrucciones con el lenguaje humano.

El lenguaje SQL nos permite hacer actualizaciones de información en bases de datos, borrar, controlar accesos, filtrar, etc.

Sus principales características son:

- Permite crear, modificar y borrar esquemas de relación.
- Para la generación de consultas, hace uso de principios algebraicos y cálculo relacional.
- Sus reglas se basan en comandos o instrucciones, lo que le permite asegurar la integridad de los datos.
- Se integra fácilmente con otros lenguajes de programación de alto nivel como Java, C++, PHP o Python.

Certificación

Es importante interiorizar sobre el lenguaje SQL, pues en numerosas preguntas de la certificación se hace referencia a este tema.

1.5. Tipos de datos

Las bases de datos son más eficientes cuanto mejor definida está su estructura. Por ello debemos darles forma a partir de la creación de tablas y la definición de los campos y los tipos de campo de cada tabla. Esto es importante porque cuanta más información concreta tenga el motor de la base de datos sobre lo que se va a guardar, más eficientemente podrá hacer su tarea.

Campus online

Accede al Campus online para ver el videotutorial *Configuración del servidor de bases de datos.*

¿Cómo sabemos el tipo de dato que tenemos que poner en nuestra tabla? Si sabemos cuál es el valor que vamos a guardar (una fecha, una condición de verdadero o falso, un número...), podemos definir el tipo de dato con mucha precisión. Estos son los tipos de datos disponibles en las bases de datos con motor MySQL:

- **Tipos de dato numéricos:**

  - **INT:** 4 bytes con un rango de valores entre −2.147.483.648 y 2.147.483.647, o entre 0 y 4.294.967.295.

  - **SMALLINT:** 2 bytes con un rango de valores entre −32.768 y 32.767, o entre 0 y 65.535.

  - **TINYINT:** 1 byte con un rango de valores entre −128 y 127, o entre 0 y 255.

  - **MEDIUMINT:** 3 bytes con un rango de valores entre −8.388.608 y 8.388.607, o entre 0 y 16.777.215.

  - **BIGINT:** 8 bytes con un rango de valores entre −8.388.608 y 8.388.607, o entre 0 y 16.777.215.

  - **DECIMAL:** almacena los números de coma flotante como cadenas o *strings*.

  - **FLOAT (m,d):** almacena números de coma flotante, donde *m* es el número de dígitos de la parte entera y *d*, el número de decimales.

  - **DOUBLE (REAL):** almacena número de coma flotante con precisión doble. Igual que FLOAT, la diferencia es el rango de valores posibles.

  - **BIT (BOOL, BOOLEAN):** número entero con valor 0 o 1.

- **Tipos de dato con formato de fecha:**

  - **DATE:** fecha con año, mes y día. Rango entre 1000-01-01 y 9999-12-31.

  - **DATETIME:** fecha (año-mes-día) y hora (horas-minutos-segundos). Rango entre 1000-01-01 00:00:00 y 9999-12-31 23:59:59.

  - **TIME:** almacena una hora (horas-minutos-segundos). Rango entre −838-59-59 y 838-59-59. El formato almacenado es HH:MM:SS.

- **TIMESTAMP:** almacena una fecha y una hora (UTC). Rango entre 1970-01-01 00:00:01 y 2038-01-19 03:14:07.

- **YEAR:** almacena un año dado con 2 o 4 dígitos de longitud (por defecto son 4). Rango entre 1901 y 2155 con 4 dígitos. Con 2 dígitos, el rango es desde 1970 a 2069 (70-69).

• **Tipos de dato con formato de string o cadena de caracteres:**

- **CHAR:** de 1 a 255 caracteres (fijo).

- **VARCHAR:** de 1 a 255 caracteres (variable).

- **TINYBLOB:** longitud máxima de 255 caracteres. Válido para almacenar ficheros de texto, audio, vídeo o imágenes.

- **BLOB:** longitud máxima de 65.535 caracteres. Válido para almacenar ficheros de texto, audio, vídeo o imágenes.

- **MEDIUMBLOB:** longitud máxima de 16.777.215 caracteres. Válido para almacenar ficheros de texto, audio, vídeo o imágenes.

- **LONGBLOB:** longitud máxima de 4.294.967.298 caracteres. Válido para almacenar ficheros de texto, audio, vídeo o imágenes.

- **SET:** almacena desde 0 hasta 64 elementos en una lista.

- **ENUM:** igual que SET, pero solo almacena un valor.

- **TINYTEXT:** longitud máxima de 255 caracteres. Sirve para almacenar texto sin formato.

- **TEXT:** longitud máxima de 65.535 caracteres. Sirve para almacenar texto sin formato.

- **MEDIUMTEXT:** longitud máxima de 16.777.215 caracteres. Sirve para almacenar texto sin formato.

- **LONGTEXT:** longitud máxima de 4.294.967.298 caracteres. Sirve para almacenar texto sin formato.

Resumen

- Las bases de datos hacen que partes de información persistan en memoria una vez que nuestra aplicación ya ha terminado, y aunque el usuario salga de nuestro programa.

- Las bases de datos se organizan en tablas y las tablas están llenas de registros (donde está contenida la información). Cada registro tiene un identificador único.

- Hemos de saber qué tipo de datos vamos a guardar en cada tabla. El tipo de datos que enviamos desde Python ha de coincidir con el que preparamos en la tabla: si en Python pedimos un número al usuario, en la base de datos guardaremos un tipo INT; si pedimos una fecha, guardaremos un tipo DATE; etc.

- El lenguaje más usado para bases de datos, y en el que más se basan los lenguajes actuales, es el SQL.

- Hay distintos motores de bases de datos, cada uno optimizado para guardar y mostrar la información de la forma más eficiente para el uso que le vayamos a dar.

2. Operaciones principales con bases de datos en Python

Existen cuatro comandos básicos de SQL, “INSERT”, “SELECT”, “UPDATE”, y “DELETE”, que nos permiten realizar las cuatro operaciones básicas necesarias para crear y mantener datos; pero antes de poder trabajar con estas operaciones sobre los datos, hemos de definir correctamente la conexión a la base de datos y crear correctamente no solo la estructura general de la base de datos, sino también la de sus tablas y los tipos de datos que van a contener.

2.1. Conexión desde Python

Con todo lo visto hasta el momento, tenemos ya la suficiente información sobre qué es una base de datos, qué tipos de bases de datos existen, cuáles son sus operaciones comunes, etc.

Vamos a crear una base de datos sencilla desde Python basándonos en el motor SQLite. Este motor de bases de datos ya está integrado en la herramienta de trabajo interactiva con la que venimos mostrando ejemplos, Google Colab; por esa razón, no será necesario hacer ningún tipo de configuración. Además, como ya se ha dicho anteriormente, no requiere un servidor o ficheros extra, por lo que nos va a ser muy sencillo trabajar y podremos centrarnos en el código de Python. Concretamente, utilizaremos el módulo SQLite3, módulo que importaremos para poder trabajar con comandos SQL sin problemas y de forma transparente. Los pasos que deberemos seguir, a modo de pseudocódigo, serán:

1. Establecer una conexión con la base de datos.

2. Crear un curso para comunicar con los datos.

3. Interactuar con los datos de la base de datos a través de operaciones y comandos SQL.

4. Indicar que la conexión aplique las instrucciones o comandos establecidos en el punto anterior o que los anule.

5. Cerrar la conexión.

Llevando los anteriores puntos a la práctica, tenemos el siguiente código:

1. Establecer conexión:

Primero importamos el módulo SQLite3 para más tarde crear un objeto de ese tipo. Se especifica un fichero donde residirá la configuración de la base de datos. Es importante que la extensión del fichero sea ".db" (de *data base*), y podemos especificar el nombre de archivo que queramos; en este caso hemos optado por "grupos.db".

```
#creamos la conexión
import sqlite3
conn = sqlite3.connect('grupos.db')
print("Conexión establecida correctamente")
```

2. Crear cursor:

El objeto "cursor" se necesita para trabajar con la base de datos. Es un objeto que centralizará las operaciones que vayamos a realizar con la base de datos y sus registros.

```
#creamos el cursor
cursor = conn.cursor()
```

3. Manipulación de los datos:

Una vez conectados a la base de datos y con el cursor creado, podemos empezar a escribir instrucciones en SQL. En primer lugar, crearemos una tabla donde incluiremos los registros. Para ello usaremos el comando "execute()" del módulo SQLite3. En cuanto al comando, primero pondremos el nombre del campo que vamos a crear ("nombre") y el tipo de dato que va a ser guardado en ese campo ("text").

```
#creamos la tabla
cursor.execute('CREATE TABLE rock (nombre text, miembros int)')
```

Esta instrucción crea una tabla llamada "rock" con dos columnas o campos, "nombre" y "miembros".

Una vez que hemos creado la tabla y su estructura, vamos a insertar datos. Primero lo haremos de forma individual:

```
#insertamos 1 registro en la tabla
#usamos el comando .execute
cursor.execute("INSERT INTO rock VALUES ('KISS', 4)")
```

Esta instrucción viene a decir "inserta en la tabla 'rock' los valores [nombre] y [miembros]".

Sin embargo, insertar registros uno a uno es un trabajo muy pesado, por lo que podemos optar por incluir varios de una sola vez, almacenándolos en un *array*, a modo de diccionario, cambiando el comando; en vez de ".execute()", usaremos ".executemany()":

```
#insertamos varios registros a la vez
#usamos el comando .executemany
valores = [('Bon Jovi', 5), ('ACDC', 4), ('Muse', 3), ('The Cult', 4)]
cursor.executemany('INSERT INTO rock values(?,?)', valores)
```

4. Una vez que hemos manipulado los datos en SQL según nuestro criterio, es hora de aplicar los cambios. En nuestro ejemplo, hemos creado una tabla, hemos introducido un registro y luego cuatro registros más. Para validar los cambios, hacemos uso del comando "commit()".

```
#aplicar los cambios
conn.commit()
```

5. Cerrar la conexión:

```
#cerrar la conexión
conn.close()
```

Por lo tanto, aunando todos estos trozos de código, nos debería quedar un pequeño programa con esta forma:

```
#creamos la conexión
import sqlite3
conn = sqlite3.connect('grupos.db')
print("Conexión establecida correctamente")

#creamos el cursor
cursor = conn.cursor()

#creamos la tabla
cursor.execute('CREATE TABLE rock (nombre, miembros)')
```

```
#insertamos 1 registro en la tabla
#usamos el comando .execute
cursor.execute("INSERT INTO rock VALUES ('KISS', 4)")

#insertamos varios registros a la vez
#usamos el comando .executemany
valores = [('Bon Jovi',5), ('ACDC', 4), ('Muse', 3), ('The Cult', 4)]
cursor.executemany('INSERT INTO rock values(?,?)', valores)

#aplicamos los cambios
conn.commit()

#cerramos la conexión
conn.close()
```

Al ejecutarlo, obtendremos un mensaje por pantalla que nos comunicará que se ha podido establecer la conexión con la base de datos. Además, en nuestro directorio de Colab se habrá creado un archivo llamado "grupos.db".

Pero ¿cómo sabemos si la información se ha insertado correctamente? Podemos crear otro pequeño código para consultar la tabla "rock".

```
#creamos la conexión
import sqlite3
conn = sqlite3.connect('grupos.db')
print("Conexión establecida correctamente")

#creamos el cursor
cursor = conn.cursor()

#creamos la consulta para leer los registros de la tabla
cursor.execute('SELECT * FROM rock')
lineas = cursor.fetchall()

#sacamos la información por pantalla
for registro in lineas:
  print(registro)

#cerramos la conexión
conn.close()
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Por pantalla veremos la siguiente información:

```
Conexión establecida correctamente
```

```
('KISS', 4)
('Bon Jovi', 5)
('ACDC', 4)
('Muse', 3)
('The Cult', 4)
```

2.2. Operaciones de lectura y filtrado

Para leer y seleccionar (filtrar) datos de la base de datos, tenemos varios comandos que pueden usarse a la vez. Lo más sencillo sería utilizar el comando "SELECT", que simplemente indica que queremos seleccionar algo de algún sitio.

```
SELECT <dato> FROM <nombre_de_la_tabla>
```

Esa es su forma más sencilla. Llevado a la práctica, quedaría algo de la siguiente forma:

```
'SELECT * FROM rock'
```

El carácter asterisco ("*") indica que se quieren seleccionar todos los campos de la tabla "rock". Si, por el contrario, quisiéramos saber solo el nombre de la banda, quedaría algo así:

```
'SELECT nombre FROM rock'
```

Vamos a afinar más la búsqueda introduciendo una condición de filtrado. De momento queremos saber el nombre de todos los registros de la tabla "rock"; pero ahora vamos más allá: queremos saber el nombre de todos los registros de la tabla "rock" que tengan 3 miembros. Para ello, añadimos la condición "WHERE", acompañada del criterio de filtrado que deseamos.

```
'SELECT nombre FROM rock WHERE miembros=3'
```

Hay que destacar el detalle de que la cláusula "WHERE" utiliza el símbolo de igual ("=") para añadir una condición de comparación; en esto se diferencia de Python, ya que hasta ahora habíamos visto que la comparación de igualdad (ha de ser igual a tres, en nuestro caso) se realizaba con dos símbolos de igual ("=="). Hay que ser cuidadosos con estos detalles, puesto que estamos mezclando dos tipos de lenguaje (y dos sintaxis diferentes) en el mismo *software*: Python y SQL.

Más allá de eso, y en lenguaje humano, nuestra consulta de arriba vendría a significar: "Selecciona el nombre de todos los registros de la tabla 'rock' cuyos miembros sean 3". El resultado sería el siguiente:

```
#creamos la conexión
import sqlite3
conn = sqlite3.connect('grupos.db')
print("Conexión establecida correctamente")

#creamos el cursor
cursor = conn.cursor()

#creamos la consulta para leer los registros de la tabla
cursor.execute('SELECT nombre FROM rock WHERE miembros=3')
lineas = cursor.fetchall()

#sacamos la información por pantalla
for registro in lineas:
  print(registro[0])

#cerramos la conexión
conn.close()
```

Resultado:

```
Conexión establecida correctamente
Muse
```

Profundizando un poco más, podemos llegar a ordenar los resultados que nos dé nuestra consulta mediante el criterio que nos convenga. Se hace uso del comando "ORDER BY" ("ordenar por"), con el que podemos añadir un criterio de ordenación. Por ejemplo, si queremos que se ordene por el campo "nombre" del grupo, le especificaremos "ORDER BY nombre". Podemos incluso decirle que lo ordene de forma ascendente ("ASC") o descendente ("DESC").

Ejercicios resueltos

Practica el **ejercicio 1** con los conocimientos adquiridos hasta esta unidad.

Esta vez vamos a especificar que queremos que saque la lista de grupos de 4 miembros ordenada ascendentemente y luego descendentemente, para ver la diferencia:

```
#creamos la conexión
import sqlite3
conn = sqlite3.connect('grupos.db')
print("Conexión establecida correctamente")
```

```
#creamos el cursor
cursor = conn.cursor()

#creamos la consulta para leer los registros de la tabla
print ("Ordenados por nombre de forma ascendente")
cursor.execute('SELECT * FROM rock WHERE miembros=4 ORDER BY nombre ASC')
lineas = cursor.fetchall()

#sacamos la información por pantalla
for registro in lineas:
  print(registro[0])

print ("------------------")
print ("Ordenados por nombre de forma descendente")
cursor.execute('SELECT * FROM rock WHERE miembros=4 ORDER BY nombre DESC')
lineas = cursor.fetchall()

#sacamos la información por pantalla
for registro in lineas:
  print(registro[0])

#cerramos la conexión
conn.close()
```

Por pantalla:

```
Conexión establecida correctamente
Ordenados por nombre de forma ascendente
ACDC
KISS
The Cult
------------------
Ordenados por nombre de forma descendente
The Cult
KISS
ACDC
```

Por otra parte, podemos poner varias condiciones en el "WHERE", es decir, que la consulta devuelva la información que cumpla diferentes condiciones. Esto lo conseguiremos con el comando "AND", que concatenará tantas condiciones de "WHERE" como queramos. Por ejemplo, vamos a querer todos los nombres de los grupos cuyo número de miembros sea superior a 3 pero inferior a 5.

```
#creamos la conexión
import sqlite3
conn = sqlite3.connect('grupos.db')
```

```
print("Conexión establecida correctamente")

#creamos el cursor
cursor = conn.cursor()

#creamos la consulta para leer los registros de la tabla
print ("Ordenados por nombre de forma ascendente")
cursor.execute('SELECT * FROM rock WHERE miembros>3 AND miembros<5 ORDER BY nombre ASC')
lineas = cursor.fetchall()

#sacamos la información por pantalla
for registro in lineas:
  print(registro[0])

#cerramos la conexión
conn.close()
```

Por pantalla:

```
Conexión establecida correctamente
Ordenados por nombre de forma ascendente
ACDC
KISS
The Cult
```

2.3. Operaciones de escritura, modificación y borrado

Para la inserción de nuevos datos se va a usar el comando "INSERT". Esto añadirá datos a la tabla que le especifiquemos. A modo de ejemplo, vamos a añadir algunos grupos más a nuestra tabla. Como ya vimos anteriormente, se pueden crear registros de uno en uno o varios a la vez; nosotros optaremos por la segunda opción.

```
#creamos la conexión
import sqlite3
conn = sqlite3.connect('grupos.db')
print("Conexión establecida correctamente")

#creamos el cursor
cursor = conn.cursor()

#insertamos varios registros a la vez
#usamos el comando .executemany
valores = [('Nirvana',3), ('Ramones', 4), ('Police', 3), ('Scorpions', 5)]
cursor.executemany('INSERT INTO rock values(?,?)', valores)
```

```
#aplicar los cambios
conn.commit()

#cerrar la conexión
conn.close()
```

Si consultamos ahora la base de datos, obtenemos el siguiente listado:

```
#creamos la conexión
import sqlite3
conn = sqlite3.connect('grupos.db')
print("Conexión establecida correctamente")

#creamos el cursor
cursor = conn.cursor()

#creamos la consulta para leer los registros de la tabla
cursor.execute('SELECT * FROM rock')
lineas = cursor.fetchall()

#sacamos la información por pantalla
for registro in lineas:
  print(registro)

#cerramos la conexión
conn.close()
```

Por pantalla:

```
Conexión establecida correctamente
('KISS', 4)
('Bon Jovi', 5)
('ACDC', 4)
('Muse', 3)
('The Cult', 4)
('Nirvana', 3)
('Ramones', 4)
('Police', 3)
('Scorpions', 5)
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Es posible actualizar un registro mediante el comando "UPDATE". Especificaremos primero la tabla donde está el campo que queremos modificar; después, mediante el comando "SET", especificaremos qué campo en concreto queremos modificar y el valor que queremos darle. En este

caso, queremos cambiar dentro de la tabla “rock” el campo “nombre” que cumpla la condición de que sea igual a “Police” por el nuevo valor “Sting & The Police”.

```
#creamos la conexión
import sqlite3
conn = sqlite3.connect('grupos.db')
print("Conexión establecida correctamente")

#creamos el cursor
cursor = conn.cursor()

#orden para modificar un registro
cursor.execute('UPDATE rock SET nombre = "Sting & The Police" WHERE nombre = "Police"')

#creamos la consulta para leer los registros de la tabla
cursor.execute('SELECT * FROM rock')
lineas = cursor.fetchall()

#sacamos la información por pantalla
for registro in lineas:
  print(registro)

#cerramos la conexión
conn.close()
```

Por pantalla:

```
Conexión establecida correctamente
('KISS', 4)
('Bon Jovi', 5)
('ACDC', 4)
('Muse', 3)
('The Cult', 4)
('Nirvana', 3)
('Ramones', 4)
('Sting & The Police', 3)
('Scorpions', 5)
```

De igual forma, si queremos eliminar un registro que cumpla una condición en concreto, lo haremos con el comando “DELETE FROM”, especificándole de qué tabla es el registro que queremos eliminar y las condiciones concretas que identifican a este registro. Por ejemplo, nosotros queremos eliminar de la tabla “rock” al grupo que tenga en su campo

“nombre” el valor “Nirvana” (porque no queremos clasificarlo como rock, sino como *grunge*).

```
#creamos la conexión
import sqlite3
conn = sqlite3.connect('grupos.db')
print("Conexión establecida correctamente")

#creamos el cursor
cursor = conn.cursor()

#orden para eliminar un registro
cursor.execute('DELETE FROM rock WHERE nombre = "Nirvana"')

#creamos la consulta para leer los registros de la tabla
cursor.execute('SELECT * FROM rock')
lineas = cursor.fetchall()

#sacamos la información por pantalla
for registro in lineas:
  print(registro)

#cerramos la conexión
conn.close()
```

Por pantalla:

```
Conexión establecida correctamente
('KISS', 4)
('Bon Jovi', 5)
('ACDC', 4)
('Muse', 3)
('The Cult', 4)
('Ramones', 4)
('Police', 3)
('Scorpions', 5)
```

2.4. Otros comandos y consultas

A continuación, veremos otros comandos que resultan de gran utilidad:

- **Saber si una tabla existe:** lo comprobaremos mediante el comando “IF NOT EXISTS” después del “CREATE TABLE”. Si existe, no se creará. En caso contrario, se creará con el nombre y los campos que le especifiquemos.

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

```
#creamos la conexión
import sqlite3
conn = sqlite3.connect('grupos.db')
print("Conexión establecida correctamente")

tabla = conn.cursor()

tabla.execute('CREATE TABLE IF NOT EXISTS discos(id int, nombre text, canciones int)')

conn.commit()
```

- **Listar tablas:** muy similar a cuando consultamos un dato de una tabla, pero en este caso consultamos el campo "name" a sqlite_master, que es el contenedor de las tablas de la base de datos.

```
tabla.execute('SELECT name FROM sqlite_master where type= "table"')
```

- **Eliminar una tabla:** con el comando "DROP" podremos eliminar tablas. Podremos hacerlo directamente con "DROP TABLE <nombre_de_tabla>", y también podremos combinarlo con "IF EXISTS" (es decir, borrar la tabla XXX si existe); siguiendo el ejemplo anterior:

```
tabla.execute('DROP TABLE IF EXISTS discos')
```

- **Excepciones:**
  - **DatabaseError:** cualquier error relacionado con la base de datos.
  - **IntegrityError:** subclase de "DatabaseError". Se genera cuando hay un problema de integridad de los datos.
  - **ProgrammingError:** se produce cuando hay errores de sintaxis, se llama a una función con los parámetros incorrectos o no existe la tabla a la que se hace referencia.
  - **OperationalError:** ocurre cuando fallan las operaciones de la base de datos; por ejemplo, una desconexión inusual.
  - **NotSupportedError:** ocurre cuando se utilizan métodos no definidos o no compatibles con el motor de la base de datos.

Ejercicios resueltos

Practica el **ejercicio 2** con los conocimientos adquiridos hasta esta unidad.

Resumen

- Los comandos de lectura, filtrado y ordenación son “SELECT”, “WHERE” y “ORDER BY”.
- Con “SELECT” podemos seleccionar un campo, varios de ellos o todos los de la tabla.
- El comando “WHERE” se utiliza para poner una condición de filtrado. Se pueden encadenar varios mediante “AND”.
- El comando “ORDER BY” ordena los resultados en función del criterio que se le especifique.
- Los comandos de escritura, modificación y borrado son “INSERT”, “UPDATE” y “DELETE”.
- El comando “INSERT” crea un nuevo registro en la tabla.
- El comando “UPDATE” modifica el valor anterior por uno nuevo.
- El comando “DELETE” borra el registro de información.

3. Desarrollo web con Python

Vamos a ver las distintas partes implicadas en el desarrollo web. No es solo lo que vemos, la apariencia y el *look and feel* de lo que tenemos delante de la pantalla; suele haber parte de guardado e infraestructura interna, así como comunicaciones entre servicios web que son completamente trasparentes para el usuario... y, en cierta manera, cuanto más trasparentes sean esas tareas, mejor experiencia de usuario le estaremos brindando a quien está al otro lado de la pantalla.

3.1. *Front-end*, *back-end* y API

Ya sea por trabajo o por ocio, pasamos muchísimo tiempo delante de un ordenador, una tableta, un *smartphone*... interactuando con webs, aplicaciones o servicios web que, en su gran mayoría, tienen tres partes muy diferenciadas:

- Interfaz humano-máquina, que es lo que vemos y con lo que interactuamos.
- Guardado, modificación y presentación de datos.
- Comunicación con otras webs o servicios web.

Son tres partes bien diferenciadas de una web en las que se puede trabajar por separado y que no tienen por qué estar presentes en todas las webs. De la misma forma, no todas las partes han de ser igual de complejas (ni de simples), ya que cada una se desarrollará en mayor o menor medida en función de los requisitos del proyecto o del uso que se le vaya a dar. Podemos referirnos al conjunto que forman como a una estructura bastante general de lo que es una web con comunicaciones con otras plataformas, guardado de datos y parte visual para para interactuar con los humanos.

Por la parte que nos interesa, que es la del programador, podemos distinguir muy claramente entre los programadores de *front-end* y de *back-end*: mientras que los primeros son los que se encargan de la parte visual de las aplicaciones, los segundos son los que se encargan de la parte estructural de *software*. Estas dos partes no comparten código ni, muchas veces, tecnología; es decir, pueden estar hechas con lenguajes

distintos, con *frameworks* distintos, con criterios de programación distintos... pero ambas han de estar coordinadas para dar al usuario una buena experiencia de navegación, usabilidad y funcionamiento.

Un programador de *front-end* se preocupará de que la aplicación, programa o web resulte claro y entendible por el usuario, y de que este pueda usarlo sin ningún tipo de limitación. Por otra parte, un programador de *back-end* se preocupará de que los datos que introduzca el usuario se guarden de manera correcta en una base de datos, y de que cuando pida un informe, este tome la información de los registros correctos y se genere de forma rápida.

Dentro de un equipo de desarrollo, los programadores *back-end* y *front-end* están condenados a entenderse, coordinarse y desarrollar de forma que después todo tenga sentido. ¿Y puede un programador estar especializado en ambos campos? Sí; a la programación de ambas (*front-end* y *back-end*) en conjunto, se le llama **programación *full-stack***. Así, a un programador especializado en ambas áreas, se le llamará programador *full-stack*. Alguien con este tipo de capacidad está legitimado para llevar el control o la coordinación de un proyecto de desarrollo, puesto que tiene una visión más global de todo el proyecto y no solo de una parte.

No se puede decir que un área sea más importante que la otra. Tan esencial es la consistencia de los datos y su estructura como la forma en la que se le presenta al usuario y hace que este use nuestro software de forma correcta. El objetivo de la combinación de ambas podría resumirse diciendo que el *front-end* debe recoger los datos, y el *back-end*, procesarlos.

3.1.1. *Front-end*

Como ya hemos dicho, cuando mencionamos la parte de front-end de una web, una aplicación o un programa, nos estamos refiriendo a la parte con la que interactúa el usuario, la parte visual. Está, obviamente, enfocada y orientada a que el usuario la pueda usar de forma correcta, sencilla y clara. Nosotros mismos sabemos cuándo una interfaz nos es fácil de usar y cuándo no.

Las principales tecnologías usadas para el desarrollo de la parte *front-end* de un proyecto web son:

- **HTML:** es un lenguaje de programación de marcado con etiquetas (similar a XML, que veremos más adelante). Los elementos incrustados

Para saber más

Un *full-stack developer* es un programador con conocimientos técnicos avanzados en *front-end* y *back-end*. Suelen ser los mejores candidatos a dirigir un proyecto de desarrollo y están muy cotizados, ya que no es común tener especialización en ambas áreas.

dentro de las etiquetas se comportan conforme a las reglas que especifique la etiqueta en cuestión. Forma el esqueleto de la web, la estructura principal.

- **CSS:** no se considera un lenguaje de programación, ya que hace referencia a las reglas de estilo que han de seguir los elementos de HTML, definiendo colores, tamaños, alineaciones, posiciones o espacios, entre otras características de naturaleza eminentemente gráfica.

- **JavaScript:** lenguaje de programación interpretado encargado del comportamiento dinámico dentro de una web, así como de elementos interactivos para ofrecer una mejor experiencia de uso al cliente o usuario.

Asimismo, los componentes más comunes que podemos encontrar en un *front-end* se podrían resumir en:

- Pruebas de accesibilidad
- Pruebas de usabilidad
- Diseño gráfico con herramientas de edición de imágenes
- Posicionamiento en buscadores (SEO)
- Rendimiento de carga y pruebas de estrés
- Compatibilidad de navegadores
- Compatibilidad de *plugins*

Por otro lado, podemos enumerar una serie de *frameworks* que están destinados al desarrollo de *front-end*:

- Foundation
- AngularJS
- Backbone
- Bootstrap

Existen reglas de programación que aseguran que una interfaz está programada correctamente desde el punto de vista de la usabilidad y de

la experiencia del usuario. Estamos hablando de la UI (*User Interface*) y la UX (*User eXperience*), que son directrices que todo programador de *front-end* debe tener en cuenta a la hora de desarrollar este tipo de proyectos.

3.1.2. *Back-end*

El *back-end* es la parte que está por detrás del *front-end*. Esta área no la ve el usuario, es invisible para él, aunque la usa sin saberlo. Es toda la parte que hace referencia al tratamiento de los datos, a su estructura, su ordenación y su presentación; es decir, es donde se definen las reglas de cómo se van a guardar los datos que el usuario introduzca por el *front-end* (a través de un formulario, por ejemplo). Aquí están implicadas bases de datos, tipos de datos, tablas, motores de bases de datos, búsquedas, etc.

Más allá de cómo se guardan y filtran los datos, en el *back-end* está contenida toda la parte del servidor, manejo de sesión, autenticación, seguridad, etc.

Las principales tecnologías usadas para el desarrollo de la parte *front-end* de un proyecto web son:

- J2EE
- Python
- NodeJS
- Ruby
- PHP
- .NET

Asimismo, los componentes más comunes que podemos encontrar en un *back-end* se podrían resumir en:

- Lenguajes de *scripting*
- Escalabilidad de infraestructuras de red
- Bases de datos

- Seguridad, cotejamiento y cohesión de los datos
- *Backup* o respaldo
- Ciberseguridad
- Pruebas, *pentesting*, estrés de comunicaciones

Por norma general, los *back-ends* más complejos a la hora de desarrollar un proyecto web suelen ser los que tienen que ver con la generación de contenidos en tiempo real, como pueden ser redes sociales, geolocalizaciones o mensajería. Ese requisito de inmediatez tiene que estar reflejado no solo en un *front-end* fácil e intuitivo, sino también en un *back-end* muy bien modelizado y estructurado para soportar muchas peticiones en un lapso muy breve (e incluso simultáneas), mientras que también tiene que hacerse cargo del procesamiento de esas peticiones, buscar la información concreta y enviarla a la parte de *front-end* para que sea mostrada al usuario.

Como ya hemos advertido al inicio de este punto, no se pude decir que un área sea más importante que la otra, porque, por muy bien hecho que esté un *front-end*, si el *back-end* no está optimizado y bien estructurado, va a ocasionar un retraso en la obtención de la información por parte del usuario que a la postre podrá resultar en que se rechace nuestro *software*.

NodeJS es un lenguaje que también está basado en JavaScript, que ya incluimos entre los lenguajes de *front-end*; este, sin embargo, es una variación especializada y completamente orientada a la parte de *back-end*.

Por otra parte, encontramos las bases de datos en las que puede estar contenida la información. En cuanto a la tecnología, las herramientas que se utilizan en el *back-end* son:

- MariaDB
- MongoDB
- MySQL

En lo que respecta a tecnología, existe un *stack* de tecnologías agrupadas que es muy usado como *back-end* "estándar" para ciertos servicios muy comunes hoy en día. Nos referimos al LAMP, cuyo nombre corresponde a las siglas de cuatro tecnologías:

- Linux (sistema operativo, *back-end*)
- Apache (tipo de servidor, *back-end*)
- MySQL (base de datos, *back-end*)
- PHP (lenguaje de programación, *front-end*)

Muchos de los CRM más usados actualmente, como Wordpress, Prestashop o Drupal, utilizan esta combinación de tecnologías.

Puede existir alguna ligera variación en función de los requisitos de los servicios que se le vayan a ofrecer al usuario, como que el servidor, en lugar de ser una distribución de Apache, sea NginX; o que la base de datos sea PostgreSQL en lugar de MySQL. Otro cambio muy común es usar Ruby on Rails para la parte que va a comunicar con el *front-end*, en lugar de usar PHP.

Existe otro *stack* muy de moda estos días y que cada vez va ganando más seguidores. Su objetivo es procesar y servir grandes cantidades de información en el menor tiempo posible, y está basado en su mayoría en el uso de JavaScript; se trata del *stack* llamado MEAN, que responde a las siglas de:

- MongoDB
- Express
- Angular
- NodeJS

Por otro lado, hay un *stack* propio de Microsoft, basado completamente en herramientas de esa empresa, y está compuesto por:

- Windows
- Microsoft IIS
- .NET
- SQL Server

Para saber más

Un *stack* ("pila de cosas", en inglés) de desarrollo es la pila de tecnologías que se utilizan en un entorno de desarrollo determinado (en este caso sería la suma del sistema operativo del servidor, el lenguaje de programación del *back-end* y el *software* de base de datos).

3.1.3. API

Cuando hablamos de una API, nos estamos refiriendo a un conjunto de reglas y protocolos que rigen sobre un conjunto de datos generados por una aplicación para integrar esos datos en otras aplicaciones. Responde a las siglas de *Application Programming Interface*, o "interfaz de programación de aplicaciones".

Las API posibilitan que distintos servicios web se comuniquen entre sí. Pongamos por ejemplo un escenario donde existen dos aplicaciones, la aplicación A y la aplicación B. El uso de una API por parte de la aplicación A posibilita que la aplicación B pueda "leer" e importar a su plataforma B los datos generados por A mediante la API de A. Esto simplifica enormemente el desarrollo de muchas aplicaciones, hace las comunicaciones de datos mucho más flexibles y favorece principios de optimización como, por ejemplo, la no duplicidad de datos.

La definición de una API, es decir, la definición de los datos que han de ser compartidos entre dos o más aplicaciones, tiene un alto componente técnico y obviamente ha de ser un programador quien la lleve a cabo, pero dicha definición no está exenta de un cierto carácter o componente comercial, ya que al fin y al cabo, es una "cesión" de acceso a unos datos, y muchas veces las condiciones y las reglas de acceso se establecen desde la redacción de un contrato en el departamento comercial.

Si por cuestiones de contrato o de relaciones comerciales es necesario restringir o ampliar la cantidad de datos que se comparten vía API, será el equipo de desarrollo el que modifique la definición, por lo que, en lo referente a esta área de las aplicaciones web, ambos departamentos deben estar coordinados. Cuando la empresa o aplicación que quiere leer los datos envía una consulta, esta ha de tener una estructura muy particular que debe ajustarse a la que se ha definido en la aplicación web de la empresa productora de los datos.

Las API permiten a una empresa compartir el acceso de lectura a sus datos por parte de clientes y usuarios externos; esto representa un valor añadido y favorece el desarrollo de aplicaciones de terceros que usan un servicio en concreto, lo que aumenta la visibilidad y la presencia de ese servicio. Un ejemplo claro lo tenemos hoy en día con Google Maps o Twitter, que ponen a disposición de quien quiera una API pública para que sea usada para la extracción, lectura y explotación de datos... no todos, lógicamente, sino solo los que las compañías quieren (Figura 3.1).

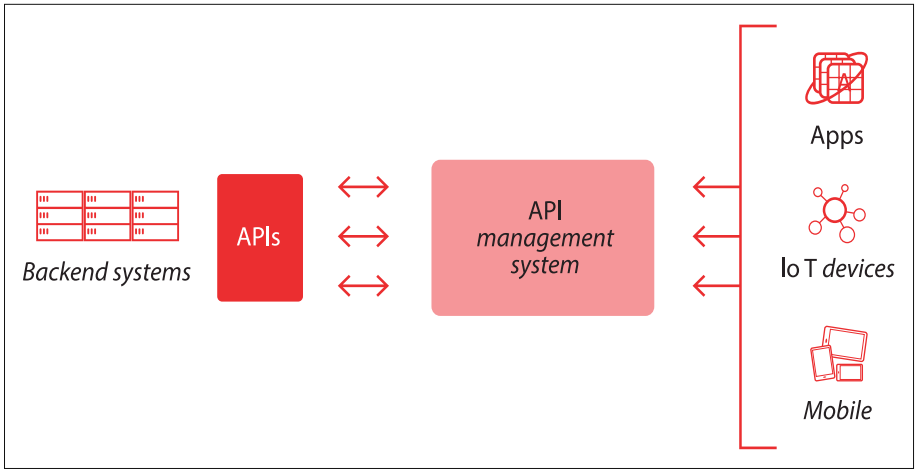

Figura 3.1
Esquema de APIs.

Existen tres planteamientos para la generación y uso de las API, y vienen determinados por el componente de la privacidad en cuanto a los permisos y la jerarquía:

- **API privadas:** de uso interno por parte de la propia empresa.

- **API de *partners*:** solo son accesibles por parte de partners específicos. Suelen ser API sobre datos de logística, catálogos, lotes, etc., para favorecer la vinculación de los *partners* con la empresa, pero se limitan los datos a criterio de esta.

- **API públicas:** todos tienen acceso a la API y, por tanto, a los datos, lo que favorece el desarrollo de aplicaciones y servicios por parte de terceros y aumenta la presencia de la empresa productora de los datos.

Las API usan en su gran mayoría el protocolo HTTP para la solicitud y respuesta de los datos. Por lo general, el formato es XML o JSON, dependiendo de la naturaleza de los datos, ya que estos dos formatos son los más fáciles de transmitir y leer cuando hay grandes volúmenes de información... y uno de los lenguajes más usados para estas tareas de lectura y procesamiento es Python.

En el ámbito de la seguridad, el uso de las API debe tener una dedicación especial, mucho más aún si estamos hablando de las API públicas, puesto que, de no estar bien definidas, pueden suponer una brecha de seguridad importante para la empresa. Lo que hace la empresa que ofrece la API a un tercero es darle una "clave API"; así, puede tener el control de quién entra a leer datos y qué datos son los que se leen. Se pueden configurar varias reglas para evitar riesgos, como, por ejemplo, permitir el acceso a la API durante un tiempo determinado o restringir a un número máximo de veces el acceso a los datos.

Un caso destacado en el ámbito de seguridad de las API lo tenemos en la tecnología OAuth. Esta es una tecnología muy usada para la autorización de accesos a API (y a otros servicios); hay muchas librerías OAuth gratuitas, por lo que se facilita mucho el poder escribir una implementación para gestionar los accesos mediante este método. Su función principal es que una aplicación pueda acceder a servicios de terceros (API) sin que el otro usuario tenga que darle a la aplicación las credenciales. Está basada en el uso de *token* de sesión. Es importante conocer las diferentes partes implicadas en el uso de esta tecnología:

- **Consumidor:** el servicio al que el usuario quiere acceder usando una credencial externa.
- **Proveedor de servicio:** el que ofrece la información o el servicio al que el consumidor quiere acceder.
- **Usuario final:** el que disfruta de la información.

El flujo de información debería seguir un esquema similar al siguiente:

- El consumidor pide un token al proveedor de servicio. Esta operación es transparente para el usuario final.
- El consumidor del servicio lleva al usuario final a una página segura del proveedor de servicio, pasándole como parámetro un *token* que lo identifica.
- El usuario final se autentica en la página del proveedor del servicio, habiendo validado el token anterior.
- El proveedor de servicio redirige al usuario de vuelta a la página del consumidor, que queda especificada en un parámetro llamado ***oauth_callback***.
- El consumidor recoge al usuario final en la dirección de *callback* junto con el *token* de confirmación de identidad.

3.2. Análisis de XML

El nombre del lenguaje **XML** viene de las siglas *eXtensible Markup Language*, lo que viene a significar que es un lenguaje marcado que define una serie de reglas para la codificación de documentos. Con lenguaje

marcado se refiere a un conjunto de códigos que pueden ser aplicados para el análisis o la lectura de textos. Con XML se crea, pues, una definición de elementos bajo un formato bien estructurado.

Antes de entrar en materia y analizar un archivo XML, vamos a diferenciar sus partes:

- ***Prolog*:** metadatos y definiciones, tales como la declaración XML, las instrucciones de procesamiento, las declaraciones de tipo de documento o los comentarios. Digamos que es la cabecera o definición del resto del contenido del documento.

- ***Body*:**

  - Estructural: es la parte donde se define la estructura o esqueleto que va a tener el documento.

  - **Contenido:** son los valores que van a estar presentes en la estructura.

El diseño de un XML se centra en la simplicidad y la facilidad de lectura y uso de la información contenida en él, ya que establece una base de trabajo para el transporte de información entre sistemas web.

Por su aspecto puede recordar a HTML, si bien su estructura es mucho más definida y rígida.

```
<grupo>
        <nombre>Nirvana</nombre>
        <miembros type=”intl”>
                3
        </miembros>
        <discos />
</grupo>
```

La información está encapsulada entre dos etiquetas, una de apertura y una de cierre.

```
<nombre>Nirvana</nombre>
```

Se puede dar el caso de que una etiqueta no contenga datos, que no haya información contenida dentro de ella. En este caso, sería correcto usar una sola etiqueta autocerrada.

```
<discos />
```

Este sería un ejemplo de un código sencillo de XML y unas líneas en Python para extraer la información.

```
import xml.etree.ElementTree as ET

datos = '''
<grupo>
  <nombre>Nirvana</nombre>
  <miembros type="intl">3</miembros>
  <discos type="intl">10</discos>
  <activo />
</grupo>'''

arbol = ET.fromstring(datos)
print('Nombre:', arbol.find('nombre').text)
print('Miembros:', arbol.find('miembros').text)
print('Discos:', arbol.find('discos').text)
print('En activo:', arbol.find('activo').text)
```

Por pantalla:

```
• Nombre: Nirvana
• Miembros: 3
• Discos: 10
• En activo: None
```

Cuando importamos el módulo "ELEMENTTREE", disponemos de una serie de validaciones de XML que nos permiten obviar todas las reglas de comprobación de sintaxis de los elementos XML.

Cuando se llama a la función "FROMSTRING", se representa la cadena donde está contenido el XML en un árbol de nodos XML. Una vez que están aislados los nodos de esa forma, existen diversos métodos para extraer la información contenida en ellos, como, por ejemplo, el método "FIND".

En el ejemplo anterior se ha analizado un solo nodo, por lo que no era necesario usar ningún tipo de bucle. Vamos a ampliar el número de nodos, y usaremos un bucle "FOR" para recorrer todos los nodos del XML y extraer su información.

```
import xml.etree.ElementTree as ET

datos = '''
<concierto>
```

```
  <cartel>
    <grupo>
      <nombre>Nirvana</nombre>
      <miembros type="intl">3</miembros>
      <discos type="intl">10</discos>
    </grupo>
    <grupo>
      <nombre>Foo Fighters</nombre>
      <miembros type="intl">6</miembros>
      <discos type="intl">10</discos>
    </grupo>
    <grupo>
      <nombre>Pearl Jam</nombre>
      <miembros type="intl">6</miembros>
      <discos type="intl">11</discos>
    </grupo>
  </cartel>
</concierto>'''

concierto = ET.fromstring(datos)
lista = concierto.findall('cartel/grupo')
print('Total de grupos:', len(lista))

for item in lista:
    print("--------------------------")
    print('Nombre', item.find('nombre').text)
    print('Miembros', item.find('miembros').text)
    print('Discos', item.find('discos').text)
```

Por pantalla:

```
Total de grupos: 3
--------------------------
Nombre Nirvana
Miembros 3
Discos 10
--------------------------
Nombre Foo Fighters
Miembros 6
Discos 10
--------------------------
Nombre Pearl Jam
Miembros 6
Discos 11
```

El método "FINDALL" devuelve un tipo lista con todos los subnodos "grupo" por debajo del nivel "cartel". De ahí podemos sacar el número de elementos de esa lista y averiguar cuántos nodos hay, que es lo que representamos en la línea:

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

```
print('Total de grupos:', len(lista))
```

El bucle "FOR" recorre esa lista identificando el nombre de los registros de cada nodo de último nivel, independientemente de la cantidad de estos que haya.

3.3. Análisis de JSON

El formato JSON se basa en el formato usado por parte de JavaScript con objetos y *arrays*, y su nombre viene de las singlas en inglés de *JavaScript Object Notation*, o "notación de objeto de JavaScript". Curiosamente, como Python es anterior a JavaScript, la sintaxis que se definió en su día para que Python tratase los diccionarios y listas influyó directamente en la definición de las sintaxis de JSON. Por tanto, la sintaxis con la que se trabaja en JSON es muy similar a la combinación que se haría en Python de listas y diccionarios.

Haciendo un símil entre el ejemplo visto anteriormente en XML y el formato JSON, tendríamos el siguiente código:

```
{
        "nombre" : "ACDC",
        "pais" : "Australia",
        "genero" : "Heavy Rock"
}
```

Como podemos ver, la definición es muchísimo más sencilla. Obviamente, al no ser un lenguaje de etiquetas, no tenemos aperturas ni cierres, y el nodo de información viene marcado por la apertura y el cierre de llaves. Esto se debe a que, como hemos dicho al inicio, la sintaxis es propia del mapeo o definición de diccionarios y listas en Python, por lo que es muy intuitivo y sencillo definir un registro de información en JSON. Por ello está ganando cada vez más presencia en detrimento de XML como formato de intercambio de información entre plataformas,.

Para proceder al análisis de un código en JSON, vamos a obtener una lista que, como elementos, tendrá parejas "clave" : "valor", por lo que podemos decir que vamos a tener una lista de diccionarios. Si bien la simplicidad del lenguaje es una ventaja y ahorramos mucho código respecto a XML, una de sus desventajas, posiblemente, sea justamente esa misma:

que perdemos de vista cierta información definitoria como pueden ser los nombres de las etiquetas para saber el tipo o nombre de campo.

```
import json

datos = '''
[
{ "nombre" : "ACDC",
"pais" : "Australia",
"genero" : "Heavy Rock"
} ,
{ "nombre" : "Nirvana",
"pais" : "EEUU",
"genero" : "Grunge"
},
{ "nombre" : "Héroes del Silencio",
"pais" : "España",
"genero" : "Rock"
}
]'''

concierto = json.loads(datos)

print('Total de grupos:', len(concierto))

for elemento in concierto:
  print("-----------------")
  print('Nombre', elemento['nombre'])
  print('País', elemento['pais'])
  print('Género', elemento['genero'])
```

Por pantalla:

```
Total de grupos: 3
-----------------
Nombre ACDC
País Australia
Género Heavy Rock
-----------------
Nombre Nirvana
País EEUU
Género Grunge
-----------------
Nombre Héroes del Silencio
País España
Género Rock
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

El código para extraer información es ligeramente distinto al presentado en el ejemplo anterior referente a XML, pero la idea del flujo de información del algoritmo es exactamente la misma. Una de las principales diferencias, y que ofrece una dimensión del tipo de dato con el que se está tratando, es que hemos pasado de analizar la información con un:

```
lista = concierto.findall('cartel/grupo')
```

...A cargar toda la información directamente con una instrucción del tipo:

```
concierto = json.loads(datos)
```

Y otra de las grandes diferencias es que no hemos de echar mano de una librería específica de JSON para crear objetos y estructuras de análisis del formato JSON, puesto que la sintaxis que usamos para analizar tipos de dato lista y diccionario es la propia de Python, ya que las estructuras que devuelve la carga del JSON son nativas del lenguaje Python.

Resumen

- El f*ront-end* es la parte visible de las webs, todo lo que tiene que ver con el diseño, las imágenes, la maquetación, etc.; es decir, aquello con lo que interactúa el usuario.

- El *back-end* es la parte interna o trasera de las webs, donde se especifican bases de datos, procesos de guardado, seguridad, etc. No es accesible por el usuario.

- La API es la forma más usada para intercambiar información entre servicios web. Las API pueden ser privadas, de partner o públicas, y establecen una serie de normas para que un tercero pueda consultar información.

- El formato XML es uno de los más usados para intercambiar información. Utiliza etiquetas.

- El formato JSON le está ganando terreno a XML como lenguaje de intercambio de información porque es más sencillo y permite transmitir una mayor cantidad de datos.

4. Seguridad en la programación web con Python

La seguridad es uno de los fundamentos que hay que tener en cuenta en todos y cada uno de los bloques de código que desarrollemos. El más mínimo fallo, despiste, caso sin cubrir o puerta trasera que no tengamos protegida puede dar como resultado que los ciberdelincuentes se puedan infiltrar en nuestro sistema; por eso hemos de conocer las amenazas y las formas en las que pueden hacerse con el control para tomar conciencia, proteger y poner más atención al detalle en partes concretas de nuestro código.

4.1. Herramientas de seguridad en Python

Como ya hemos visto hasta, a través de los años Python se ha convertido en un lenguaje muy extendido entre la comunidad informática para el desarrollo de *software*, y muy especialmente en la industria de la seguridad informática debido a su simplicidad, a su eminente orientación práctica y al hecho de que se puede trabajar con él bien sea como lenguaje de *scripting* o bien como lenguaje interpretado.

Python es uno de los lenguajes que muchas distribuciones de Linux traen ya integrado en el sistema. Sin duda, eso ha favorecido la expansión del uso de este lenguaje entre la comunidad, como ya vimos anteriormente.

Por eso hoy en día se pueden encontrar muchísimas herramientas, parches y *scripts* desarrollados con Python, concretamente dentro del ámbito de la ciberseguridad. Donde más peso tiene es en el área del análisis de datos, ya que, como hemos visto, tiene una serie de módulos de análisis matemáticos que pueden ser de mucha ayuda en diferentes tipos de análisis. Pero ¿qué tiene que ver el análisis de datos con Python y la ciberseguridad? Python es una ayuda que debe tenerse muy en cuenta para desarrollar, por ejemplo, programas de análisis de vulnerabilidades de sistemas (*pentesting*) o como analizador de tráfico para medir flujos de datos y optimización de sistemas (análisis de redes). En un plano un poco más elevado, y utilizando los dos usos anteriormente citados, Python resulta un lenguaje muy potente para analizar sistemas una vez que ya se ha penetrado en ellos (*hacking*) y para justo lo contrario, esto es: proteger a estos sistemas de las vulnerabilidades detectadas.

Esa sería la vertiente constructiva o de *white-hat*; porque, claro está, también se puede usar con otros fines menos lícitos como, por ejemplo, la extracción, modificación o borrado de información. Debido a su integración con infinidad de librerías de terceros, se puede usar para desarrollar algoritmos maliciosos y conseguir armar diferentes tipos de ataques:

- Ataque por fuerza bruta
- Ataque de denegación de servicio (DoS)
- SQL *injection*
- LDAP *injection*
- Etc.

Por otro lado, Python es el lenguaje preferido y más usado por los investigadores en ámbitos como *big data*, *machine learning*, inteligencia artificial o para trabajos de *data science*. Es en esta última área donde más auditores de seguridad trabajan con Python, para generar sistemas de defensa contra los ciberdelincuentes. Como es lógico, todo buen programador de sistemas de defensa ha de ser también un gran conocedor de las vulnerabilidades y ha de saber explotarlas para desarrollar sistemas de defensa efectivos (Figura 4.1).

Sin duda una de las piedras de toque de todo programador de *black-hat* es el conocimiento y el desarrollo de *exploits*, que son fragmentos de *software* que se aprovechan de puertas traseras, vulnerabilidades o simplemente de información pública que debiera estar mejor protegida, y que toman el control del sistema total o parcialmente.

Figura 4.1
Python puede resultar de gran ayuda para analizar vulnerabilidades de sistemas.

Gracias a sus librerías, con Python se pueden desarrollar programas que funcionen en este sistema y que podemos usar o bien para fines de protección y preservación de la información, o bien para lo contrario y tratar de penetrar en los sistemas operativos de los dispositivos y sus aplicaciones. Para impedir esto, Python ofrece la posibilidad de desarrollar *software* para el cifrado, la codificación y la protección de archivos y carpetas.

A continuación, ofrecemos un listado de las principales herramientas usadas para la detección de vulnerabilidades en sistemas informáticos y que están desarrolladas en Python. Su potencia se basa en la capacidad de desarrollar algoritmos precisos, ligeros y robustos, y de elegir correctamente los módulos y librerías de los que depende el código, ya que, en parte, esto determinará con qué sistemas se va a poder integrar de manera efectiva.

- **sqlmap:** destinada a la inyección de código SQL (código de bases de datos). Detecta y aprovecha vulnerabilidades debidas a este tipo de ataque en aplicaciones web.

- **SET (*Social-Engineer Toolkit*):** de los mejores *softwares* con finalidades de ingeniería social, ya que permite la automatización de tareas, alertas, suplantación de número de teléfono, clonación de código web y herramientas de *phishing*.

- **Glastopf Honeypot:** monta un pequeño servidor web que emula una gran cantidad de vulnerabilidades.

- **Jsunpack-n:** hace una simulación de nuestro navegador y analiza *exploits* y vulnerabilidades del propio navegador y de los *plugins* instalados en él.

- **Sparta:** aplicación dotada de una interfaz gráfica de usuario (o *Graphic User Interface*, GUI en inglés) que se encarga de rastrear las vulnerabilidades de la red.

- **The Harvester:** ayuda a analizar el riesgo de la información pública.

- **W3af:** *framework* de test de intrusión web, diseñado con la idea de ser una plataforma de testeo de intrusiones.

- **Fimap:** microherramienta destinada a auditar sitios web, analizando sobre todo la inclusión remota de archivos en ellos. Pese a que aún

está en fase de desarrollo, el paquete que se ha liberado es 100 % funcional. El objetivo principal es optimizar la seguridad y la calidad de las páginas web.

- **BlindElephant:** orientada al proceso de captación de información relativa al sistema operativo del ordenador que se quiere auditar o analizar. Esta técnica también es conocida como ***fingerprinting***.

4.2. *Pentesting*

Podríamos empezar explicando en qué consiste el ***pentesting***: su nombre viene de *penetration* y *testing*, lo que deja bastante clara la finalidad de esta técnica. Es sin duda una de las prácticas o técnicas más en boga hoy en día y consiste en atacar uno o más entornos o sistemas con el fin de probar su fiabilidad y encontrar y prevenir posibles fallos o brechas de seguridad. Las características y técnicas de esta práctica están diseñadas para determinar el alcance de los fallos de seguridad, pero no para aprovecharlos y extraer, modificar o borrar información del sistema.

Es sin duda una de las prácticas más demandadas hoy en día en el ámbito laboral. Para las empresas es imprescindible tener un recurso formado en este tipo de disciplina; sabiendo la importancia de las comunicaciones y la presencia *online*, junto con la innumerable cantidad de sistemas informáticos en la nube... es impensable no poner el foco en asegurar o blindar al máximo todos los sistemas de la empresa, por lo que esta figura profesional ha de actuar de manera continua analizándolos, bien sea de forma interna o externa.

Existen varias maneras de hacer *pentesting* contra un sistema, en función del tipo de información que se posee a la hora de realizar la batería de test:

- ***Pentesting* de caja blanca (o *white-box*):** se conocen todos los datos del sistema (IP, máscaras, diseño de red, infraestructura, cortafuegos...). Es muy común que parte de los técnicos de la empresa analizada tome parte en este análisis. Al conocer toda la información de antemano, se pueden plantear ataques o análisis más enfocados y resulta más fácil determinar el posible alcance en forma de cambios o modificaciones que puede sufrir el sistema.

- ***Pentesting* de caja negra (o *black-box*):** este tipo es el que más se parece a la realidad, ya que el analista *pentester* no conoce de antemano

nada del sistema. Por ello, puede llegar a asemejarse mucho a lo que podría pasar en un caso real, ya que se deben ir descubriendo las vulnerabilidades a ciegas. Es sin duda el tipo de *pentesting* más lento de realizar.

- ***Pentesting* de caja gris (o *grey-box*):** nace como híbrido de los dos anteriores, ya que en este caso el *pentester* conoce parte de la información del sistema pero no toda. Suele ser el más recomendado, puesto que, si bien puede tener un comienzo relativamente rápido y enfocado, nunca se poseerá toda la información previamente, por lo que los ataques para descubrir vulnerabilidades implicarán el uso de medios y técnicas similares a los de un escenario real.

Tras una batería de análisis del sistema, un buen *pentester* ha de ser capaz de proporcionar cierta información:

- **Determinar auditoría:** a partir de los datos iniciales que posea, el *pentester* debe elaborar un punto inicial de partida a modo de auditoría para poder plantear una serie de test y ataques que realizará sobre el sistema.

- **Recogida de información:** resultados de las pruebas, en los que se ha de poner de manifiesto todo aquello relacionado con las fugas de información, brechas de seguridad, vulnerabilidades, etc.

- **Acceso al sistema:** tras analizar toda la información previa, se organizan y secuencian los ataques, y se procede a ejecutarlos.

- **Elaboración de informes:** recomendaciones sobre cómo acotar o paliar los distintos ataques que han resultado ser positivos, estimando el alcance tanto del propio ataque como de la defensa propuesta.

Como hemos visto, no solo se actúa intentando descubrir vulnerabilidades del sistema, sino también trabajando en la defensa de estos, planteando reglas y sistemas que minimicen los ataques. Por eso, profesionalmente, los *pentesters* se dividen en equipos, según el cometido o la especialización de cada uno:

- ***Red team*:** parte ofensiva, atacantes del sistema.

- ***Blue team*:** parte defensiva, protectores del sistema.

El *pentesting* es una práctica completamente legal siempre que los ataques tengan un fin "profesional", sea contra nuestros propios equipos o

contra los de un cliente (deberemos tener un documento de consentimiento previo). En caso de no ser así, estaríamos hablando de *hacking*, que, entre otras cosas, está penado por ley.

Una vez que empecemos a desarrollar o utilizar herramientas, hay que conocer un par de términos de este ámbito y saber diferenciarlos correctamente para que no nos sean extraños y tengamos todas las características bien identificadas, ya que no podemos empezar a atacar nuestros sistemas (o los del cliente) "a lo loco".

Tampoco es una disciplina en la que abunde la práctica del "gran botón rojo", es decir, que, pulsando un botón, la herramienta se ponga a desplegar ataques y nosotros solo obtengamos resultados; esta es una técnica que contiene mucho de configuración, de prueba y error, de ingeniería social...

- **Vulnerabilidad:** fallo de seguridad en el *hardware* o *software* de un sistema. Puede ser una puerta trasera, una brecha de seguridad, una condición no contemplada, etc. Por normal general, suele ser un fallo en la programación del *software* del sistema, como contraseñas fáciles, permisos en carpetas o el hecho de no contar con una recogida efectiva de excepciones y que se pueda llegar a provocar un desbordamiento de *buffer*.

- ***Exploit*:** pequeño trozo de código que ataca una vulnerabilidad del sistema. Suele tener instrucciones muy orientadas a hacerse con el control del sistema o a provocar funcionamientos indebidos. Podemos diferenciar tres tipos de *exploits*, según desde dónde se ejecuten:

  - ***Exploit* local:** para ejecutarlo, antes tendremos que habernos hecho con el control del sistema o, al menos, haber penetrado en él con permisos de ejecución en alguna de sus carpetas. También puede ejecutarse tras acceder a la máquina con un *exploit* remoto.

  - ***Exploit* remoto:** se ejecuta desde fuera de la máquina objetivo del ataque, bien sea a través de internet o de la red local donde se encuentre el equipo.

  - ***Exploit* del lado del cliente:** el más extendido, ya que aprovecha vulnerabilidades de las aplicaciones que están instaladas en los equipos de los usuarios. Para introducirlos, basta con correos electrónicos que parezcan legítimos, *pendrives* o aprovechar la navegación insegura de los usuarios.

Para saber más

Para tener un listado de *exploits* actualizado y con avisos de brechas de seguridad, disponemos de la herramienta Metasploit. Para el caso de los *payloads*, dentro de Metasploit tenemos el subproyecto Meterpreter.

- ***Payload*:** aplicación de tamaño reducido (en la mayoría de los casos) que, aprovechando que un *exploit* ha conseguido afectar a una de las vulnerabilidades del sistema objetivo del ataque, se hace con el control de este; permite desde acceder al sistema de archivos del equipo víctima hasta ver en nuestra pantalla lo que muestra la pantalla del ordenador atacado.

De todo lo expuesto anteriormente se puede deducir que una de las medidas que hay que tomar de manera preventiva es la actualización continua de nuestro sistema: parches, *service packs*, nuevas versiones, etc., ya que podrán corregir agujeros de seguridad y fallos menores que se pueden convertir (y se convertirán) en vulnerabilidades que alguien podría explotar para entrar en nuestro sistema.

4.3. Herramientas de Python para la detección de vulnerabilidades

Ya hemos visto qué es el *pentesting*, quién lo lleva a cabo y por qué, y la información que se puede llegar a extraer. Ahora tenemos que relacionarlo con Python, y es que muchas (por no decir la gran mayoría) de las herramientas dedicadas a este fin están desarrolladas en Python, a menudo con código abierto y, por lo tanto, modificables para ajustarlas más a nuestras necesidades. Dependiendo del ámbito en que vayamos a utilizarlas, hay una serie de herramientas desarrolladas en Python recomendadas o más útiles.

4.3.1. Redes

Entendemos por red una red de datos, de sistemas o de comunicaciones a un número de emisores-receptores conectados entre sí, bien sea por cable o de manera inalámbrica, por los que fluye la información de un punto a otro regidos por un protocolo de comunicaciones. Las herramientas para redes más destacadas en Python son:

- **scapy:** sirve para enviar, interceptar y analizar paquetes de red. Se puede usar por separado de manera individual o como librería de Python.

- **pypcap, Pcapy y pylibpcap:** diferentes librerías y módulos para Python también enfocados a la captura y análisis de paquetes.

- **libdnet:** rutinas de red de bajo nivel, incluidas las referentes al análisis y captura de tramas Ethernet.

4.3.2. Ingeniería inversa

Cuando hablamos de ingeniería inversa nos referimos, más que a una técnica, a una capacidad muy ligada a una mente extremadamente analítica. Por definición, es la habilidad de reconstrucción de un producto ya existente; en el mundo del *software* se usa para recrear un programa que ya existe, solventando así posibles errores, o, en caso de que el programa sea de la competencia, para desarrollar nuevos productos y funcionalidades a partir de este.

En el área del *software*, la ingeniería inversa también puede emplearse para analizar los productos de la competencia. Aunque muchas empresas prohíben la ingeniería inversa de sus productos y lo incluyen en sus condiciones de licencia, la vía legal no se aplica al análisis de protocolos. Esto es así porque el mismo *software* no es objeto de análisis de las herramientas de ingeniería inversa. Además, por lo general muchas de esas cláusulas de licencia no tienen validez en otros países. Los usuarios que adquieren un *software* tienen derecho a someterlo a una ingeniería inversa para comprobar la seguridad de la aplicación y solucionar fallos.

Las herramientas más destacadas en Python para la ingeniería inversa son:

- **Paimei:** es un *framework* para trabajar con ingeniería inversa. Incluye los paquetes PyDBG, PIDA, pGRAPH.

- **Immunity Debugger:** interfaz gráfica para la generación de *scripts*. Cuenta con una línea de comandos dedicados al *debug*.

- **mona.py:** lo más parecido a una navaja suiza en lo referente a *exploits* enfocados a la ingeniería inversa.

4.3.3. *Fuzzing*

El ***fuzzing*** es una técnica por la cual se automatiza la escritura de valores de manera indiscriminada en un sistema, tanto los valores que tengan sentido como los que no. Así se descubren muchas brechas de seguridad debido a casos de entrada de datos no contemplados.

Dependiendo de lo grande que sea el sistema objetivo sobre el que trabajamos, el testeo mediante escritura de valores puede durar desde horas hasta días. En este caso, y a partir de los informes, el *blue-team* suele

tener bastante trabajo reescribiendo métodos y redefiniendo los algoritmos para ser tolerante con todo tipo de variables de entrada.

Las herramientas más destacadas en Python para el *fuzzing* son:

- **python-afl:** módulo todavía experimental que habilita el servidor de bifurcación American Fuzzy Lop y la instrumentación para código Python puro.

- **Sulley:** desarrollo de condiciones de *fuzzing* y también de testing consistente en disponer múltiples modelos extensibles.

- **Peach Fuzzing Platform:** es un *framework* ampliable que se usa para la generación de condiciones de *fuzzing*, así como de automatismos para mutar estas condiciones y conseguir mejores resultados.

4.3.4. WEB

Las herramientas más destacadas son:

- **Requests:** biblioteca HTTP elegante y simple, construida para seres humanos.

- **lxml:** biblioteca para procesar XML y HTML fácil de usar; similar a Requests.

- **HTTPie:** cliente HTTP con líneas de comando similar a cURL y amigable.

4.3.5. Informática forense

Hoy en día tenemos a nuestra disposición una serie de herramientas que protegen nuestra información personal en la red: antivirus, *software* variado de seguridad y otros tipos de sistemas informáticos, todas estas alternativas forman parte de una disciplina que conocemos como informática forense. Esta se encarga de adquirir, preservar y proteger datos procesados de forma electrónica y almacenados en un medio físico. Los sistemas de información son investigados de forma periódica para detectar cualquier pequeña vulnerabilidad que pueda poner en peligro la enorme cantidad de datos que se procesan y almacenan cada segundo.

El carácter de la informática forense es preventivo. Mediante diversas técnicas, realiza pruebas para comprobar que los sistemas de seguridad implementados son los adecuados. Además, también se encarga de ela-

borar las políticas de seguridad y de definir qué sistemas son los idóneos para cada caso.

La informática forense es una ciencia indispensable para todas las empresas, pues garantiza que la información confidencial y los datos de todas las personas que interaccionan con la compañía estén debidamente protegidos y fuera del alcance de los delincuentes de la Red. Las herramientas de informática forense más destacadas son:

- **Volatility:** extrae artefactos digitales de muestras de memoria volátil (RAM).

- **Rekall:** *framework* de análisis de memoria desarrollado por Google.

- **LibForensics:** biblioteca para desarrollar aplicaciones de análisis forense digital.

4.3.6. Análisis de *malware*

Malware o *software* malicioso es un término amplio que describe cualquier programa o código malicioso que es dañino para los sistemas.

El *malware* hostil, intrusivo e intencionadamente desagradable intenta invadir, dañar o deshabilitar ordenadores, sistemas informáticos, redes, tabletas y dispositivos móviles, a menudo asumiendo el control parcial de las operaciones de un dispositivo.

La intención del *malware* es conseguir ilícitamente dinero del usuario. Aunque el *malware* no puede dañar el *hardware* de los sistemas o el equipo de red —con una excepción que se conozca (véase la sección Android de Google)—, sí puede robar, cifrar o borrar datos, alterar o secuestrar funciones básicas del ordenador y espiar la actividad del usuario en el ordenador sin su conocimiento o permiso. Las herramientas más destacadas son:

- **Pyew:** editor y desensamblador de líneas de comando hexadecimal, se usa principalmente para analizar *malware*.

- **Exefilter:** filtra formatos de archivo en correos electrónicos, páginas web o archivos. Detecta muchos formatos de archivo comunes y puede eliminar contenido activo.

- **PyClamAV:** añade capacidad de detección de virus al *software* de Python.

4.3.7. Análisis de PDF

El **PDF** (siglas en inglés de *Portable Document Format*, o "formato de documento portátil", en su traducción al español) es un formato de almacenamiento para documentos digitales. Su principal virtud es facilitar el intercambio de documentación digital de manera fiable, independientemente del *software*, el *hardware* o el sistema operativo que haya generado el archivo original, y de la plataforma que lo reciba para su lectura.

El documento PDF puede contener vínculos, botones que permiten interactuar, campos de formulario que podemos rellenar, audio, vídeo e incluso lógica empresarial. Además, puede ser firmado electrónicamente para dar validez legal a la documentación online presentada, por ejemplo, a la Administración Pública.

Las herramientas más destacadas son:

- **peepdf:** herramienta de Python que analiza y explora archivos PDF para averiguar si pueden ser dañinos.
- **PDFMiner:** extrae texto de archivos PDF.
- **Didier Stevens' PDF Tools:** analiza, identifica y crea archivos PDF (incluye PDFiD, pdf-parser, make-pdf y mPDF).

Resumen

- El *pentesting* es legal y se basa en el testeo de penetración en sistemas con el ánimo de saber qué partes son más vulnerables y tratar de robustecerlas desde el punto de vista de la seguridad.

- Existen muchas herramientas de código abierto para propósitos de testeo de la seguridad en los sistemas. La gran mayoría están desarrolladas en Python.

- Pese a lo que pueda pensarse, es mucho más fácil infiltrarse en un sistema mediante ingeniería social y recabando los datos necesarios para entrar que mediante ataques informáticos.

- Los *exploits* son pequeños trozos de código que se centran en una vulnerabilidad concreta y pueden ser usados de manera aislada para un ataque muy dirigido.

Índice