

Acceso a Bases de datos SQLite con Python

Curso de Data y Desarrollo en Python
Arturo Bernal Mayordomo - 2023



Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro



GOBIERNO
DE ESPAÑA

MINISTERIO
DE INDUSTRIA, COMERCIO
Y TURISMO



Escuela de
organización
industrial

¿Qué vamos a Aprender?

- Acceder a SQLite mediante el módulo de Python **sqlite3**
 - SQLite es una base de datos ligera que no necesita un servidor de BBDD (la base de datos se guarda en un archivo)
 - Útil para estructuras simples y con pocos datos. O para prototipado, migrando luego a otros sistemas (MySQL, PostgreSQL, etc.)
- Ejecutar consultas básicas CRUD → Select, Insert, Update y Delete
- Navegar por los resultados, transformandolos a Objetos de Python (dataclass)
- Estructurar la aplicación para facilitar las consultas

Limitaciones de SQLite

- Los tipos de datos que admite SQLite son:
 - TEXT (texto)
 - NUMERIC (Enteros o decimales)
 - INTEGER (Enteros)
 - REAL (Decimales)
 - BLOB (Binario)
- Los valores booleanos se almacenan como enteros → 0 (FALSE), 1 (TRUE)
- Dependiendo la implementación, soporta relaciones entre tablas:
 - <https://www.sqlitetutorial.net/sqlite-foreign-key/>
- Para probarlo desde la línea de comandos, necesitamos instalarlo
 - <https://www.sqlite.org/download.html> (o en linux: `sudo apt install sqlite3`)

Crear una tabla

- Lo primero que haremos será abrir el archivo que contiene base de datos. Si el archivo no existe lo crea automáticamente
 - A partir de la conexión obtenemos un cursor, que permite ejecutar consultas
 - Ejecutamos la sentencia de creación de la tabla
 - Para evitar problemas si intentamos crear la tabla más de 1 vez, primero la vamos a borrar si ya existiera
- Si queremos ejecutar más de una instrucción en un único string (por ejemplo, creación de varias tablas de la base de datos), usaremos **executescript** en lugar de **execute**

Crear una tabla (ejemplo)

```
import sqlite3

database = sqlite3.connect('ejemplo.db')
cursor = database.cursor()

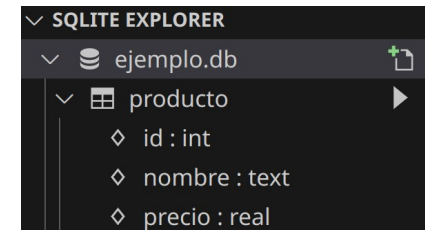
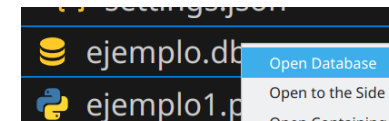
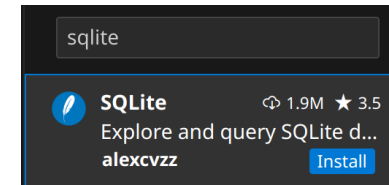
drop_table_query = "DROP TABLE IF EXISTS producto"

create_table_query = """
CREATE TABLE producto (
    id INTEGER PRIMARY KEY,
    nombre TEXT,
    precio REAL
)
"""

cursor.execute(drop_table_query)
cursor.execute(create_table_query)
```

Soporte para SQLite en VSCode

- Para no tener que usar la línea de comando para inspeccionar las bases de datos SQLite, podemos instalar la extensión de SQLite para VSCode
- Después seleccionamos el archivo con botón derecho y hacemos clic en “Open Database”
- Debajo del todo se nos abrirá una nueva pestaña llamada “SQLite Explorer”, que nos permitirá ver el contenido de las tablas, etc.



Insertar un registro

- Para insertar un registro en una tabla, ejecutaremos la instrucción INSERT con el comando **execute**
 - Las instrucciones INSERT, UPDATE y DELETE no generan cambios permanentes hasta que llamamos al método **database.commit()**, momento en el cual se guardan los cambios
 - Es recomendable utilizar “placeholders” para los valores. Es decir, en lugar de poner valores, ponemos un símbolo ‘?’. A continuación, pasamos una tupla con los valores correspondientes en orden correcto
 - Así evitamos la tristemente famosa [Inyección SQL](#)
 - Se pueden hacer varias inserciones con una única instrucción utilizando **executemany**, donde le pasamos una lista de tuplas con los valores a insertar
 - Con execute (1 INSERT) se puede consultar la id generada con **cursor.lastrowid**

Insertar un registro (ejemplo)

```
# La id no hace falta porque es autogenerada por defecto al ser INTEGER
cursor.execute("INSERT INTO producto(nombre, precio) VALUES(?, ?)",
               ("Silla", 25))
print(cursor.lastrowid) # 1 (id generada)

cursor.executemany( # Inserción múltiple
    "INSERT INTO producto(nombre, precio) VALUES(?, ?)",
    [
        ("Mesa", 90),
        ("Lámpara", 14.5),
        ("Estantería", 90),
    ],
)

database.commit()
```


Modificación y borrado

- Funcionan igual que la instrucción INSERT, pero ejecutando una sentencia SQL del tipo UPDATE o DELETE
 - Consultando después **cursor.rowcount**, se pueden saber cuantas filas han sido insertadas, borradas o modificadas

```
cursor.execute(  
    "UPDATE producto SET nombre = ?, precio = ? WHERE id = ?",  
    ("Modificado", 99, 1)  
)  
cursor.execute("DELETE FROM producto WHERE id = ?", (4,))  
print(cursor.rowcount) # 1 (borrados)  
database.commit()
```

Obtener datos (SELECT)

- Funciona igual que las anteriores pero con 2 diferencias
 - No se necesita un commit (si solo se ejecutan este tipo de consultas), ya que la SELECT no produce cambios
 - Devuelve datos → Llamando a **fetchall** te devuelve una lista de tuplas con las filas obtenidas. Si llamamos a **fetchone** te devuelve una tupla con el único (o el primer) resultado

```
productos = cursor.execute("SELECT * FROM producto").fetchall()
print(productos) # [(1, 'Modificado', 99.0), (2, 'Mesa', 90.0), (3, 'Lámpara', 14.5)]

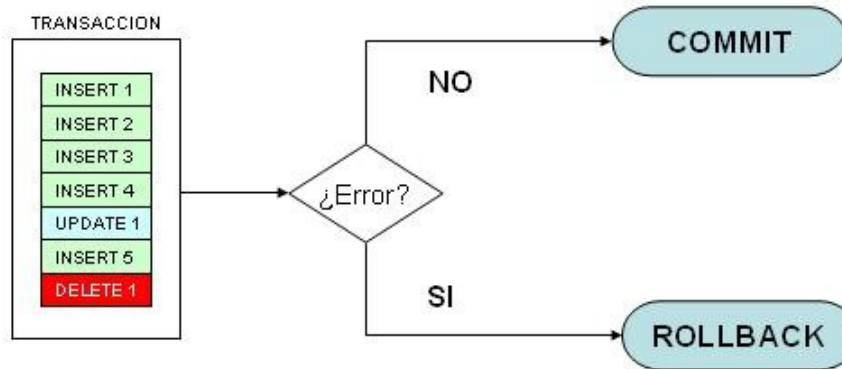
producto1 = cursor.execute("SELECT * FROM producto WHERE id = ?", (1,)).fetchone()
print(producto1) # (1, 'Modificado', 99.0)
```

Transacciones

- Una transacción es un conjunto de consultas que deben hacerse secuencialmente.
- La operación es atómica (si falla una consulta, falla toda la transacción).
 - Ejemplo: Insertar un empleado y una dirección asociada (tablas diferentes)
 - Ejemplo 2: Restar una cantidad al saldo de una cuenta bancaria que luego sumaremos a otra cuenta (transferencia de dinero)

Commit y rollback

- Al iniciar una transacción, las operaciones no se harán realmente hasta ejecutar **commit** (al final).
- Si alguna operación intermedia falla, ejecutaremos **rollback** para cancelar la transacción entera.



Controlando transacciones

- En el módulo de sqlite3, las transacciones las controlamos de manera manual, haciendo `database.commit()` cuando todo va bien
 - Debemos hacer `database.rollback()` en caso de error en alguna operación. Por ejemplo, usando **try..except**
- Otra opción es que Python haga el commit y el rollback automáticamente. Para ello metemos las consultas dentro de un bloque **with database:**
 - Si todo va bien, se hará un commit al final del bloque
 - En caso de error, el rollback es automático también

```
try:
    with database:
        cursor = database.cursor()
        cursor.execute("INSERT INTO producto VALUES(?, ?, ?)", (1, "Id Repetida", 0))
except sqlite3.IntegrityError:
    print("La id del producto ya existe")
```

Usando dataclasses para mapear datos

- Trabajar con datos genéricos como listas de tuplas puede traer confusión, especialmente en programas más grandes
- Si queremos relacionar cada valor devuelto en una fila con un nombre, tenemos 2 opciones, diccionarios y objetos. Estos últimos dan mayor flexibilidad y potencia
- Para convertir los resultados en objetos de la clase elegida, tenemos que crear una función (**row_factory**), que a partir de cada fila devuelva un objeto en nuestro caso
 - Tenemos que asignar la función correspondiente a la propiedad **database.row_factory** antes de la SELECT correspondiente. Por defecto es **sqlite3.Row**.

Usando dataclasses para mapear datos

```
from dataclasses import dataclass

@dataclass
class Producto:
    id: int
    nombre: str
    precio: float

def product_factory(cursor, row):
    return Producto(row[0], row[1], row[2])

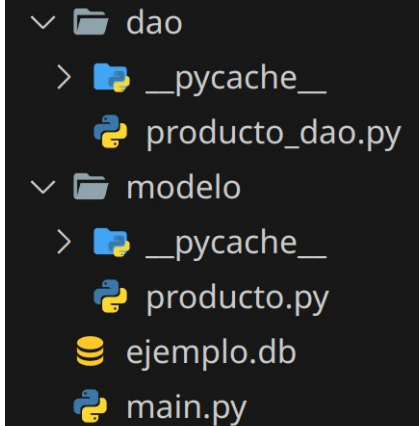
# Resto de código

database.row_factory = product_factory
cursor = database.cursor() # Debemos crear un cursor nuevo para que tenga efecto lo anterior
productos = cursor.execute("SELECT * FROM producto").fetchall()
print(productos) # [Producto(id=1, nombre='Modificado', precio=99.0), Producto(id=2,
nombre='Mesa', precio=90.0), Producto(id=3, nombre='Lámpara', precio=14.5)]

producto1 = cursor.execute("SELECT * FROM producto WHERE id = ?", (1,)).fetchone()
print(producto1) # Producto(id=1, nombre='Modificado', precio=99.0)
```

Estructurar código (patrón DAO)

- Es un patrón de diseño de software referente a como estructurar las clases para un acceso a base de datos
 - Se crean las clases que representan las tablas de nuestra base de datos. Tendrán el mismo o similar nombre que la tabla y los mismos atributos que campos tenga la tabla.
 - Estas clases se denominan entidades o modelo de datos
 - Se crean clases con los métodos necesarios para hacer operaciones con una tabla (select, insert, delete, ...). Se les llama clases DAO



```
▼ dao
  > __pycache__
    producto_dao.py
▼ modelo
  > __pycache__
    producto.py
    ejemplo.db
    main.py
```


Patrón DAO (Producto)

```
from dataclasses import dataclass

@dataclass
class Producto:
    id: int
    nombre: str
    precio: float

    @staticmethod # Cuando el producto no tiene id (no insertado)
    def crea_nuevo(nombre, precio):
        return Producto(0, nombre, precio)
```

Patrón DAO (ProductoDAO)

```
from dataclasses import dataclass
from sqlite3 import Connection
from modelo.producto import Producto

@dataclass
class ProductoDAO:
    database: Connection

    @staticmethod
    def product_factory(cursor, row):
        return Producto(row[0], row[1], row[2])

    def getAll(self) -> list[Producto]:
        self.database.row_factory = self.product_factory
        cursor = self.database.cursor()
        return cursor.execute("SELECT * FROM producto").fetchall()

    def getOne(self, id: int) -> Producto:
        self.database.row_factory = self.product_factory
        cursor = self.database.cursor()
        return cursor.execute("SELECT * FROM producto WHERE id = ?", (id, )).fetchone()

    def insert(self, p: Producto) -> None:
        with self.database: # Autocommit
            cursor = self.database.cursor()
            cursor.execute("INSERT INTO producto(nombre, precio) VALUES(?, ?)", (p.nombre, p.precio))
            p.id = cursor.lastrowid
```

Patrón DAO (main)

```
import sqlite3
from dao.producto_dao import ProductoDAO, Producto

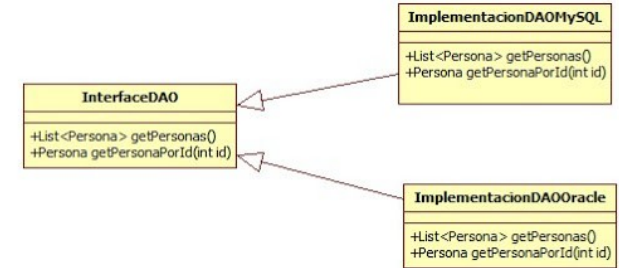
database = sqlite3.connect("ejemplo.db")

productoDAO = ProductoDAO(database)
pInsert = Producto.crea_nuevo("Prueba", 33)
productoDAO.insert(pInsert)
print(pInsert) # Producto(id=6, nombre='Prueba', precio=33) -> Incluye id

productos = productoDAO.getAll()
print(productos)
producto = productoDAO.getOne(2)
print(producto) # Producto(id=2, nombre='Mesa', precio=90.0)
```

DAO avanzado (simular interfaces)

- El patrón DAO define que se crearán interfaces que definirán los métodos que debe tener la clase DAO, y después la claseDAO que implemente dicha interfaz (los métodos definidos en ella)
 - Se puede crear una clase para acceder a SQLite, otra para MySQL, otra para usar un ORM como SQLAlchemy, etc
- En Python no existe el concepto de interfaz. Lo más cercano es usar una clase con todos los métodos abstractos
 - Nos aseguramos que todas las clases DAO derivadas de la misma implementan los mismos métodos. Esto simplifica mucho la migración de un tipo de base de datos a otro, por ejemplo






DAO Avanzado (ProductoDAO abstracto)

```
from abc import ABC, abstractclassmethod
from modelo.producto import Producto

class ProductoDAO(ABC):
    @abstractclassmethod
    def getAll(self) -> list[Producto]:
        pass

    @abstractclassmethod
    def getOne(self, id: int) -> Producto:
        pass

    @abstractclassmethod
    def insert(self, p: Producto) -> None:
        pass
```

```
>  __pycache__
    producto_dao_sqlite.py
    producto_dao.py
```