Una Ambiciosa Introducción a Python
Parte II

Los desarrolladores de software trabajan constantemente con datos, los cuales deben estar disponibles y accesibles para varios miembros del equipo, a menudo de forma simultánea. Para gestionar esta información de manera estructurada, se utilizan bases de datos, que organizan los datos en tablas, donde las filas representan registros y las columnas contienen atributos.
SQLite es un sistema de gestión de bases de datos ligero y embebido que permite almacenar y administrar datos sin necesidad de un servidor. Es ideal para aplicaciones que requieren un almacenamiento estructurado simple, como aplicaciones de escritorio y móviles.

Características Principales:
-Base de datos autónoma: No requiere configuraciones adicionales ni servidores en ejecución.
- **Ligero y eficiente**: Usa un solo archivo para almacenar la base de datos.
- **Soporte para SQL**: Permite crear tablas, insertar, actualizar y consultar datos.
- **Transaccional**: Garantiza integridad de datos mediante propiedades ACID.
- **Integrado en Python**: Se gestiona con el módulo *sqlite3*, sin necesidad de librerías externas.

Uso de SQLite

A continuación, explicamos su funcionamiento utilizando un ejemplo de una base de datos para almacenar información sobre películas, ver cine.py

1. Conexión a la Base de Datos
Para conectarse a una base de datos SQLite, se usa sqlite3.connect():

```
import sqlite3
CINE_DB = "cine.db"

def get_connection():
    return sqlite3.connect(CINE_DB)
```

La función get_connection() permite obtener una conexión reutilizable a la base de datos.

2. Creación de Tablas
Para definir la estructura de datos, creamos una tabla pelicula:

Lic. Franco Herrera

```
def create_tables():
    create_pelicula_statement = """
    CREATE TABLE IF NOT EXISTS pelicula (
        id INTEGER PRIMARY KEY,
        name TEXT NOT NULL,
        release_date DATE,
            category TEXT
    )
    """
    with get_connection() as conn:

        conn.execute(create_pelicula_statement)
        conn.commit()
```

Ventajas del Uso de **Context Managers**

El uso de with get_connection() as conn: es similar a trabajar con archivos (with open(...) as file:). Esto permite:
- Manejo automático de conexiones, cerrándolas correctamente al finalizar.
- Código más limpio y fácil de leer.
- Reducción de errores por conexiones abiertas innecesariamente.

3. Insertar Datos
Podemos insertar datos en la base de datos con INSERT INTO:

```
def add_pelicula(name, release_date, category):
    sql = "INSERT INTO pelicula (name, release_date, category) VALUES (?, ?, ?)"
    with get_connection() as conn:
        conn.execute(sql, (name, release_date, category))
        conn.commit()
```

Se usan **placeholders `?`** para evitar inyecciones SQL.

4. Consultar Datos
Para recuperar datos de la base, usamos SELECT:

```
def search_all():
    sql = "SELECT * FROM pelicula"
    with get_connection() as conn:
        return conn.execute(sql).fetchall()
```

5. Mostrar Películas
Podemos imprimir todas las películas almacenadas:

```
def show_movies():
    peliculas = search_all()
    for pelicula in peliculas:
        print(f"ID: {pelicula[0]} | Nombre: {pelicula[1]} | Fecha: {pelicula[2]} | Categoría: {pelicula[3]}")
```

Una Ambiciosa Introducción a Python
Parte II

Conclusión

SQLite es una herramienta poderosa y sencilla para manejar bases de datos en aplicaciones Python. Su integración con **sqlite3** permite realizar operaciones con datos de forma eficiente, y el uso de context managers facilita el manejo de conexiones. En este ejemplo, hemos visto cómo implementar un sistema de almacenamiento de películas utilizando SQLite, aplicando buenas prácticas como el uso de **with** para gestionar la base de datos de forma segura y eficiente.

referencia:

https://docs.python.org/3/library/sqlite3.html