

# UF2404\_01\_Python\_sqlite

## Módulo 1.– SQLite con Python

1. Qué es SQLite y cuándo usarlo
  2. Conexión desde Python con `sqlite3`
  3. Crear tablas y ejecutar consultas desde Python
  4. CRUD completo (crear, leer, actualizar, eliminar)
  5. Consultas avanzadas: JOIN, filtros, agregaciones
  6. Manejo de errores y transacciones
  7. Casos prácticos: apps locales y prototipos
-

# SQLite con Python

---

### ✓ 1.1. ¿Qué es SQLite y cuándo usarlo?

**SQLite** es una base de datos relacional liviana, embebida en archivos `.db` o `.sqlite`, sin necesidad de servidor. Ideal para:

- Proyectos pequeños o medianos.
- Aplicaciones de escritorio o móviles.
- Prototipos y pruebas rápidas.
- Entornos educativos o apps monousuario.

#### Ventajas:

- Sin instalación de servidor.
  - Totalmente compatible con SQL estándar.
  - Incluido por defecto en Python (`sqlite3`).
- 

### ✓ 1.2. Conexión desde Python con `sqlite3`

```
import sqlite3

# Conectar (crea el archivo si no existe)
conexion = sqlite3.connect('mi_base_de_datos.db')

# Crear un cursor
cursor = conexion.cursor()

# Cerrar conexión
conexion.close()
```

🧠 *Consejo:* Usa `with` para gestión automática de recursos.

---

# UF2404\_01\_Python\_sqlite

## ✓ 1.3. Crear tablas y ejecutar consultas desde Python

python

```
conexion = sqlite3.connect('mi_base_de_datos.db')
cursor = conexion.cursor()

cursor.execute('''
CREATE TABLE IF NOT EXISTS contactos (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nombre TEXT NOT NULL,
    telefono TEXT,
    email TEXT
)
''')

conexion.commit()
conexion.close()
```

---

## ✓ 1.4. Operaciones CRUD (Create, Read, Update, Delete)

### ■ CREATE: Insertar datos

python

```
conexion = sqlite3.connect('mi_base_de_datos.db')
cursor = conexion.cursor()

cursor.execute("INSERT INTO contactos (nombre, telefono, email)
VALUES (?, ?, ?)",
              ('Ana', '654321098', 'ana@email.com'))

conexion.commit()
conexion.close()
```

# UF2404\_01\_Python\_sqlite

## **READ: Leer datos**

python

```
conexion = sqlite3.connect('mi_base_de_datos.db')
cursor = conexion.cursor()
```

```
cursor.execute("SELECT * FROM contactos")
resultados = cursor.fetchall()
```

```
for fila in resultados:
    print(fila)
```

```
conexion.close()
```

## **UPDATE: Modificar datos**

python

```
conexion = sqlite3.connect('mi_base_de_datos.db')
cursor = conexion.cursor()
```

```
cursor.execute("UPDATE contactos SET telefono = ? WHERE nombre = ?",
('666666666', 'Ana'))
```

```
conexion.commit()
conexion.close()
```

## **DELETE: Borrar datos**

python

```
conexion = sqlite3.connect('mi_base_de_datos.db')
cursor = conexion.cursor()
```

```
cursor.execute("DELETE FROM contactos WHERE nombre = ?", ('Ana',))
```

```
conexion.commit()
conexion.close()
```

---

# UF2404\_01\_Python\_sqlite

## ✓ 1.5. Consultas avanzadas: JOIN, filtros y agregaciones

### Filtrar por condiciones:

python

```
cursor.execute("SELECT * FROM contactos WHERE nombre LIKE 'A%')"
```

### Agregaciones:

python

```
cursor.execute("SELECT COUNT(*) FROM contactos")
```

### JOIN (si hay varias tablas):

python

```
# Ejemplo si tenemos una tabla direcciones relacionada con contactos
cursor.execute('''
SELECT c.nombre, d.ciudad
FROM contactos c
JOIN direcciones d ON c.id = d.contacto_id
''')
```

---

## ✓ 1.6. Manejo de errores y transacciones

### Manejo de errores:

try:

```
    conexion = sqlite3.connect('mi_base_de_datos.db')
    cursor = conexion.cursor()
    cursor.execute("INSERT INTO contactos (nombre) VALUES (?)",
    (None,))
    conexion.commit()
except sqlite3.Error as e:
    print("Error:", e)
finally:
    conexion.close()
```

# UF2404\_01\_Python\_sqlite

## Transacciones:

python

```
with sqlite3.connect('mi_base_de_datos.db') as conn:
    cursor = conn.cursor()
    cursor.execute("UPDATE contactos SET telefono = ? WHERE nombre = ?")
    conn.commit()
```

---

## ✓ 1.7. Casos prácticos

### 🎯 Caso 1: Agenda de contactos

- CRUD completo
- Búsqueda por nombre
- Exportación a CSV

### 🎯 Caso 2: Gestor de tareas

- Tabla `tareas`: título, descripción, fecha, estado
  - Marcar como completada
  - Listar tareas por fecha o estado
- 

## ✓ Buenas prácticas

- Usar consultas parametrizadas para evitar **inyecciones SQL**.
- Verificar siempre el **estado de la conexión**.
- Usar `with` para manejar automáticamente la conexión.
- Separar la lógica de negocio de la lógica de acceso a datos (crear funciones).
- Hacer backups periódicos del archivo `.db`.



# Ejercicio Dirigido: Agenda de Contactos con SQLite y Python

---



## Objetivo

Crear una aplicación de consola que permita gestionar una agenda de contactos utilizando una base de datos SQLite. El usuario podrá:

- Añadir nuevos contactos
  - Listar todos los contactos
  - Buscar por nombre
  - Editar un contacto
  - Eliminar un contacto
- 



## Paso 1: Crear el archivo Python e importar librerías

python

```
import sqlite3
```

---

# UF2404\_01\_Python\_sqlite



## Paso 2: Conexión a la base de datos y creación de la tabla

python

```
def conectar():
    conexion = sqlite3.connect("agenda.db")
    cursor = conexion.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS contactos (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            nombre TEXT NOT NULL,
            telefono TEXT,
            email TEXT
        )
    ''')
    conexion.commit()
    conexion.close()
```

conectar()

---



## Paso 3: Función para añadir contactos

python

```
def agregar_contacto(nombre, telefono, email):
    conexion = sqlite3.connect("agenda.db")
    cursor = conexion.cursor()
    cursor.execute("INSERT INTO contactos (nombre, telefono, email)
VALUES (?, ?, ?)",
                    (nombre, telefono, email))
    conexion.commit()
    conexion.close()
```

---



## Paso 4: Función para listar todos los contactos

python



# UF2404\_01\_Python\_sqlite

```
def listar_contactos():
    conexion = sqlite3.connect("agenda.db")
    cursor = conexion.cursor()
    cursor.execute("SELECT * FROM contactos")
    contactos = cursor.fetchall()
    conexion.close()

    for contacto in contactos:
        print(f"ID: {contacto[0]}, Nombre: {contacto[1]}, Teléfono: {contacto[2]}, Email: {contacto[3]}")
```

---



## Paso 5: Buscar contactos por nombre

python

```
def buscar_contacto(nombre):
    conexion = sqlite3.connect("agenda.db")
    cursor = conexion.cursor()
    cursor.execute("SELECT * FROM contactos WHERE nombre LIKE ?",
('%' + nombre + '%',))
    resultados = cursor.fetchall()
    conexion.close()

    for contacto in resultados:
        print(f"ID: {contacto[0]}, Nombre: {contacto[1]}, Teléfono: {contacto[2]}, Email: {contacto[3]}")
```

---



## Paso 6: Editar un contacto

python

```
def editar_contacto(id, nuevo_nombre, nuevo_telefono, nuevo_email):
    conexion = sqlite3.connect("agenda.db")
    cursor = conexion.cursor()
    cursor.execute("UPDATE contactos SET nombre = ?, telefono = ?,
email = ? WHERE id = ?",
(nuevo_nombre, nuevo_telefono, nuevo_email, id))
    conexion.commit()
```

# UF2404\_01\_Python\_sqlite

```
conexion.close()
```

---

## Paso 7: Eliminar un contacto

python

```
def eliminar_contacto(id):
    conexion = sqlite3.connect("agenda.db")
    cursor = conexion.cursor()
    cursor.execute("DELETE FROM contactos WHERE id = ?", (id,))
    conexion.commit()
    conexion.close()
```

---

## Paso 8: Menú principal

python

```
def menu():
    while True:
        print("\n📅 Agenda de Contactos")
        print("1. Agregar contacto")
        print("2. Listar contactos")
        print("3. Buscar contacto")
        print("4. Editar contacto")
        print("5. Eliminar contacto")
        print("6. Salir")

        opcion = input("Selecciona una opción: ")

        if opcion == "1":
            nombre = input("Nombre: ")
            telefono = input("Teléfono: ")
            email = input("Email: ")
            agregar_contacto(nombre, telefono, email)
        elif opcion == "2":
            listar_contactos()
        elif opcion == "3":
            nombre = input("Buscar nombre: ")
            buscar_contacto(nombre)
```

## UF2404\_01\_Python\_sqlite

```
elif opcion == "4":
    id = int(input("ID del contacto a editar: "))
    nombre = input("Nuevo nombre: ")
    telefono = input("Nuevo teléfono: ")
    email = input("Nuevo email: ")
    editar_contacto(id, nombre, telefono, email)
elif opcion == "5":
    id = int(input("ID del contacto a eliminar: "))
    eliminar_contacto(id)
elif opcion == "6":
    break
else:
    print("Opción no válida.")
```

menu()

## Diferencias entre MySQL y SQLite

Característica	SQLite	MySQL
 <b>Instalación</b>	No requiere instalación (incluido en Python)	Requiere instalación de servidor
 <b>Tipo de base de datos</b>	Embebida (archivo <code>.db</code> )	Cliente-servidor
 <b>Multiusuario</b>	❌ No (uso monousuario o en apps locales)	✅ Sí (varios usuarios y conexiones simultáneas)
 <b>Acceso remoto</b>	❌ No	✅ Sí (conexión por red/local)
 <b>Rendimiento</b>	Bueno para apps pequeñas o medianas	Excelente en entornos de producción escalables
 <b>Seguridad y permisos</b>	Mínimos, a nivel de archivo	Gestión avanzada de usuarios y roles
 <b>Transacciones</b>	✅ Soportadas (ACID)	✅ Soportadas (ACID, mejor controlado)
 <b>Relaciones complejas</b>	Soportadas, pero sin enforcement total	Totalmente soportadas con claves foráneas
 <b>Escalabilidad</b>	Limitada	Alta
 <b>Almacenamiento</b>	Un solo archivo <code>.db</code>	Múltiples archivos gestionados por el servidor
 <b>Soporte a características avanzadas</b>	Limitado (no hay stored procedures, triggers limitados)	Completo: procedimientos almacenados, triggers, vistas, replicación
 <b>Herramientas de gestión</b>	Básicas (CLI o scripts)	Completo ecosistema: Workbench, phpMyAdmin, etc.

# UF2404\_01\_Python\_sqlite



## Uso típico

Prototipos, apps  
locales, móviles,  
educativas

Aplicaciones web, ERPs,  
servicios profesionales

---



## ¿Cuándo usar cada uno?



### SQLite

- Aplicaciones de escritorio o móviles (Android, iOS)
- Prototipos rápidos y pruebas locales
- Apps monousuario o sin alta concurrencia
- Proyectos educativos o personales



### MySQL


- Aplicaciones web o empresariales
- Sistemas multiusuario (APIs, CRMs, ERPs)
- Entornos en producción que requieren rendimiento y seguridad
- Casos donde se necesita alta disponibilidad y control de usuarios

## Sintaxis SQL: Comparativa entre SQLite y MySQL

---

### 1. Crear una base de datos

SQLite	MySQL
(No es necesario)	<code>CREATE DATABASE nombre;</code>

 SQLite trabaja directamente con un archivo `.db`.

---

### 2. Crear una tabla

sql

```
CREATE TABLE contactos (  
    id INTEGER PRIMARY KEY AUTOINCREMENT,  
    nombre TEXT NOT NULL,  
    telefono TEXT,  
    email TEXT  
);
```

SQLite	MySQL
Usa <code>INTEGER PRIMARY KEY AUTOINCREMENT</code>	Usa <code>INT AUTO_INCREMENT PRIMARY KEY</code>
Tipos de datos más flexibles	Tipos de datos más estrictos (VARCHAR, INT, DATE, etc.)

---

# UF2404\_01\_Python\_sqlite

## 3. Insertar datos

sql

```
INSERT INTO contactos (nombre, telefono, email)
VALUES ('Ana', '123456789', 'ana@email.com');
```

☒ Mismo en ambos.

---

## 4. Leer datos (SELECT)

sql

```
SELECT * FROM contactos;
SELECT nombre, email FROM contactos WHERE telefono LIKE '6%';
```

☒ Igual en ambos. Soporta **WHERE**, **ORDER BY**, **LIMIT**, **LIKE**, etc.

---

## 5. Actualizar datos

sql

```
UPDATE contactos
SET telefono = '987654321'
WHERE nombre = 'Ana';
```


☒ Igual en ambos.

---

# UF2404\_01\_Python\_sqlite

## 6. Eliminar datos

```
DELETE FROM contactos WHERE nombre = 'Ana';
```

 Igual en ambos.

---

## 7. Relaciones (FOREIGN KEY)

```
CREATE TABLE ciudades (  
    id INTEGER PRIMARY KEY AUTOINCREMENT,  
    nombre TEXT  
);
```


```
CREATE TABLE contactos (  
    id INTEGER PRIMARY KEY AUTOINCREMENT,  
    nombre TEXT,  
    ciudad_id INTEGER,  
    FOREIGN KEY (ciudad_id) REFERENCES ciudades(id)  
);
```

SQLite	MySQL
Requiere <code>PRAGMA foreign_keys=ON</code>	Soporta claves foráneas por defecto
No hace enforcement por defecto	Valida integridad automáticamente

---

## 8. JOIN entre tablas

```
SELECT c.nombre, ci.nombre AS ciudad  
FROM contactos c  
JOIN ciudades ci ON c.ciudad_id = ci.id;
```

 Idéntico en ambos motores.

---



# UF2404\_01\_Python\_sqlite



## 9. Funciones de agregación

sql

```
SELECT COUNT(*) FROM contactos;  
SELECT AVG(edad) FROM empleados;
```

✅ Compatibles: SUM, AVG, MIN, MAX, COUNT, etc.

---



## 10. Ordenar y limitar resultados

sql

```
SELECT * FROM contactos ORDER BY nombre ASC LIMIT 5;
```

✅ Igual en ambos.

💡 En MySQL también puedes usar OFFSET:

sql

```
LIMIT 5 OFFSET 10;
```

---



## 11. Procedimientos almacenados y triggers

SQLite

MySQL

❌ No soporta  
procedimientos

✅ Soporta CREATE PROCEDURE

✅ Triggers simples

✅ Triggers avanzados con lógica  
compleja

---



## 12. Gestión de usuarios y permisos

# UF2404\_01\_Python\_sqlite

## SQLite

✗ Sin gestión de usuarios

Seguridad basada en archivo `.db`

## MySQL

✓ `CREATE USER`, `GRANT`, `REVOKE`, etc.

Seguridad por usuarios, contraseñas y privilegios

---



## Conclusión

### Sentencias SQL básicas

`SELECT`, `INSERT`,  
`UPDATE`, `DELETE`

`JOIN`, `GROUP BY`, `ORDER BY`

Claves foráneas

Procedimientos, roles, usuarios

### Compatibilidad entre SQLite y MySQL

✓ 100% compatibles

✓ Compatibles

⚠ Diferencias en enforcement

✗ Solo disponibles en MySQL



## Tipos de datos en SQLite



### 1. Características generales

- **SQLite es débilmente tipado:** una columna puede almacenar cualquier tipo de dato, aunque se haya declarado con un tipo concreto.
- Pero **sí reconoce tipos principales** y los agrupa por **afinidad** (affinity).



### 2. Tipos de afinidad de SQLite

SQLite reconoce 5 afinidades principales:

Afinidad	Descripción	Ejemplos válidos de declaración
<b>TEXT</b>	Cadenas de texto, similar a <code>VARCHAR</code> , <code>CHAR</code> , etc.	<code>TEXT</code> , <code>VARCHAR(100)</code> , <code>CHAR(50)</code>
<b>NUMERIC</b>	Valores numéricos, enteros o decimales. También para fechas/hora.	<code>NUMERIC</code> , <code>DECIMAL(10,2)</code> , <code>DATE</code> , <code>DATETIME</code>
<b>INTEGER</b>	Enteros de cualquier tamaño.	<code>INTEGER</code> , <code>INT</code> , <code>TINYINT</code> , <code>BIGINT</code>
<b>REAL</b>	Números de punto flotante (decimales).	<code>REAL</code> , <code>DOUBLE</code> , <code>FLOAT</code>
<b>BLOB</b>	Datos binarios (imágenes, archivos).	<code>BLOB</code>



### 3. Regla de conversión (type affinity)

Cuando defines una columna como:

# UF2404\_01\_Python\_sqlite

sql

edad INTEGER

SQLite **no fuerza** que todos los valores sean enteros, pero **asigna internamente** la afinidad **INTEGER**.

Por ejemplo, **sí puedes insertar una cadena en una columna INTEGER**, pero no es buena práctica.

---



## 4. Ejemplos prácticos

sql

```
CREATE TABLE ejemplo (  
    id INTEGER PRIMARY KEY AUTOINCREMENT, -- entero autoincremental  
    nombre TEXT NOT NULL,                -- cadena de texto  
    sueldo REAL,                          -- número decimal  
    fecha_nacimiento NUMERIC,             -- fecha (recomendada como  
TEXT o NUMERIC)  
    imagen BLOB                           -- binario (foto,  
archivo)  
);
```

---



## 5. Comparación con MySQL

Tipo común

SQLite

MySQL

# UF2404\_01\_Python\_sqlite

Cadena de texto	TEXT	VARCHAR(n)
Entero	INTEGER	INT, BIGINT
Decimal	REAL	DECIMAL, FLOAT
Fecha y hora	NUMERIC o TEXT	DATE, DATETIME
Binario	BLOB	BLOB, LONGBLOB

---



## 6. Buenas prácticas

- Usa los tipos estándar (TEXT, INTEGER, REAL, NUMERIC, BLOB) para asegurar compatibilidad.
- Si necesitas precisión decimal (como dinero), usa NUMERIC.
- Para fechas y horas, puedes usar:
  - TEXT con formato ISO YYYY-MM-DD
  - NUMERIC con timestamp Unix
- Evita usar tipos como VARCHAR(100) si no es necesario, ya que SQLite los interpreta como TEXT igualmente.