PD4 · CEPY · UT3

De objetos Python

a bases de datos relacionales

PD4 - CEPY

Índice

1. Bases de datos relacionales

Tablas, filas, columnas — y cómo se comparan con lo que ya sabes de Python

¿Qué es una base de datos relacional?

Una **base de datos relacional** organiza la información en **tablas** con filas y columnas.

💡 Como una hoja de cálculo...

- Columnas = campos fijos con tipo
- Filas = registros individuales

✅ ...pero con garantías automáticas

- Tipos de datos obligatorios
- Restricciones de integridad
- Persiste en disco entre ejecuciones

```
── tabla: productos ──────────────────
codigo | nombre   | precio | cantidad
A1     | Agua     | 1.0    | 10
A2     | Papas    | 1.5    |  8
D1     | Refresco | 2.5    |  5
```

De diccionario Python a tabla SQL

Concepto Python	Equivalente SQL
`dict` / `list`	Tabla
Clase (`Item`)	Estructura de la tabla (columnas)
Objeto (`Item("A1", …)`)	Fila
Atributo (`item.nombre`)	Columna
`item.codigo` como clave del dict	`PRIMARY KEY`

> El diccionario desaparece al cerrar el programa. La base de datos persiste en un fichero `.db` y sobrevive entre ejecuciones.

SQLite — el motor embebido de Python

SQLite almacena toda la información en **un único fichero** `.db`. Sin servidor, sin instalación.

Característica	SQLite	MySQL / PostgreSQL
Instalación	✅ Ninguna (incluido en Python)	❌ Requiere servidor
Almacenamiento	Un fichero `.db`	Servidor con proceso dedicado
Conexión	`sqlite3.connect("bd.db")`	Host, puerto, usuario, contraseña
Casos de uso	Desarrollo, apps de escritorio	Producción, múltiples usuarios

✅ **Ideal para aprender y prototipar.** Cero configuración, portabilidad total y el mismo SQL estándar que usarás después con otros motores.

2. El módulo `sqlite3`

El ciclo conectar → ejecutar → confirmar → cerrar

Flujo de trabajo con `sqlite3`

Toda operación con SQLite sigue siempre el mismo ciclo de **5 pasos**:

```
import sqlite3

conn   = sqlite3.connect("mi_bd.db")   # 1. Conectar (crea el fichero si no existe)
cursor = conn.cursor()                  # 2. Crear el cursor

cursor.execute("SELECT * FROM tabla")  # 3. Ejecutar SQL

conn.commit()                           # 4. Confirmar cambios en disco
conn.close()                            # 5. Cerrar la conexión
```

💡 **`connect()`** abre o crea el fichero · **`cursor()`** envía SQL y recoge resultados · **`commit()`** escribe en disco · sin `commit()`, los cambios se pierden.

Transacciones y `commit()`

Los cambios de `INSERT`, `UPDATE` y `DELETE` **no se escriben en disco** hasta que llamas a `commit()`.

```
## Correcto
cursor.execute(
    "INSERT INTO productos VALUES ('A1', 'Agua', 1.00, 10)"
)
cursor.execute(
    "INSERT INTO productos VALUES ('A2', 'Papas', 1.50, 8)"
)
conn.commit()   ## ambos INSERT se guardan juntos
conn.close()
```

🚫 **Sin `commit()`, los datos no llegan al fichero `.db`.**

El programa termina sin error, pero la base de datos queda vacía o incompleta.

3. SQL básico: definir e iniciar tablas

`CREATE TABLE`, tipos de datos, restricciones e `INSERT INTO`

`CREATE TABLE` — definir la estructura

`CREATE TABLE` crea una tabla especificando columnas, tipos y restricciones.

```
CREATE TABLE IF NOT EXISTS nombre_tabla (
    columna1 TIPO RESTRICCIONES,
    columna2 TIPO RESTRICCIONES,
    ...
)
```

```
cursor.execute("""
    CREATE TABLE IF NOT EXISTS productos (
        codigo   TEXT    PRIMARY KEY,
        nombre   TEXT    NOT NULL,
        precio   REAL    NOT NULL,
        cantidad INTEGER NOT NULL
    )
""")
```

✅ `IF NOT EXISTS` evita un error si la tabla ya existe — imprescindible cuando el script se puede ejecutar varias veces.

Tipos de datos: Python vs SQL

Tipo Python	Tipo SQL	Ejemplo
`str`	`TEXT`	`"A1"`, `"Agua"`
`float`	`REAL`	`1.00`, `20.5`
`int`	`INTEGER`	`10`, `5`
`bool`	`INTEGER`	`1` (True), `0` (False)

💡 La correspondencia es directa. Declara siempre el tipo correcto para que el esquema sea legible y coherente.

Restricciones: `PRIMARY KEY` y `NOT NULL`

`PRIMARY KEY`

- Identifica de forma **única** cada fila
- No puede repetirse ni ser `NULL`
- Equivale a la clave del diccionario Python

`NOT NULL`

- La columna **siempre debe tener** un valor
- Sin ese campo → la BD rechaza la fila con error

```
cursor.execute("""
    CREATE TABLE IF NOT EXISTS productos (
        codigo   TEXT    PRIMARY KEY,  ## identificador único, nunca nulo
        nombre   TEXT    NOT NULL,     ## obligatorio
        precio   REAL    NOT NULL,     ## obligatorio
        cantidad INTEGER NOT NULL      ## obligatorio
    )
""")
```

`INSERT INTO` — añadir filas

`INSERT INTO` añade una nueva fila a una tabla existente.

```
## Valores literales directos
cursor.execute("INSERT INTO productos VALUES ('A1', 'Agua', 1.00, 10)")
cursor.execute("INSERT INTO productos VALUES ('A2', 'Papas', 1.50, 8)")
conn.commit()
```

Con datos de variables, usa siempre **parámetros** `?`:

```
codigo   = "B1"
nombre   = "Chocolate"
precio   = 2.00
cantidad = 12
cursor.execute("INSERT INTO productos VALUES (?, ?, ?, ?)",
               (codigo, nombre, precio, cantidad))
```

`INSERT` — buena práctica vs antipatrón

🚫 No construyas SQL con f-strings

```
cursor.execute(
    f"INSERT INTO productos VALUES "
    f"('{codigo}', '{nombre}', "
    f"{precio}, {cantidad})"
)
```

Un valor malicioso en `nombre` puede manipular la sentencia SQL.

✅ Usa siempre parámetros `?`

```
cursor.execute(
    "INSERT INTO productos "
    "VALUES (?, ?, ?, ?)",
    (codigo, nombre, precio, cantidad)
)
```

SQLite gestiona el escape automáticamente.

4. SQL básico: consultar datos

`SELECT`, `fetchall()` y `fetchone()`

`SELECT` — leer filas

`SELECT` lee datos de una tabla. `SELECT *` devuelve todas las columnas.

```
SELECT * FROM nombre_tabla
SELECT columna1, columna2 FROM nombre_tabla
SELECT * FROM nombre_tabla WHERE columna = valor
```

```
cursor.execute("SELECT * FROM productos")
for fila in cursor.fetchall():
    print(fila)
```

```
('A1', 'Agua', 1.0, 10)
('A2', 'Papas', 1.5, 8)
('D1', 'Refresco', 2.5, 5)
```

Cada fila llega a Python como una **tupla**. Los elementos siguen el mismo orden que las columnas del `CREATE TABLE`.

`fetchall()` y `fetchone()`

Método	Devuelve	Cuándo usarlo
`fetchall()`	Lista de tuplas (todas las filas)	Iterar sobre todos los resultados
`fetchone()`	Una tupla o `None`	Esperas exactamente un resultado

```
cursor.execute("SELECT * FROM productos WHERE codigo = ?", ("A1",))
fila = cursor.fetchone()   ## primera fila, o None si no hay resultado
print(fila[0])             ## 'A1'   — codigo
print(fila[1])             ## 'Agua' — nombre
print(fila[2])             ## 1.0    — precio
print(fila[3])             ## 10     — cantidad
```

5. SQL básico: modificar y eliminar datos

`UPDATE`, `DELETE FROM` y el orden con claves foráneas

`UPDATE` — modificar filas existentes

`UPDATE` modifica columnas de filas ya existentes. `WHERE` selecciona qué filas.

```
cursor.execute(
    "UPDATE productos SET cantidad = ? WHERE codigo = ?",
    (9, "A1")
)
conn.commit()
```

💡 Varios campos a la vez

```
cursor.execute(
    "UPDATE productos SET precio = ?, cantidad = ? WHERE codigo = ?",
    (1.20, 15, "A1")
)
```

🚫 Sin `WHERE` → actualiza TODAS las filas

```
## Actualiza la cantidad en todos los productos
cursor.execute("UPDATE productos SET cantidad = 0")
```

`DELETE FROM` — eliminar filas

`DELETE FROM` elimina filas. Sin `WHERE` → borra **todas** las filas de la tabla.

```
cursor.execute("DELETE FROM productos WHERE codigo = ?", ("A1",))
conn.commit()
```

Con claves foráneas: **eliminar primero la tabla hija**, luego la padre:

```
cursor.execute("PRAGMA foreign_keys = ON")
cursor.execute("DELETE FROM descuentos WHERE codigo = ?", ("D1",))  # hijo primero
cursor.execute("DELETE FROM productos WHERE codigo = ?", ("D1",))   # padre después
conn.commit()
```

🚫 `DELETE FROM productos` sin `WHERE` borra **todos** los productos. Filtra siempre con `WHERE`.

6. Claves primarias, valores nulos y foráneas

`PRIMARY KEY`, `NULL` / `None`, `FOREIGN KEY` y `PRAGMA`

`PRIMARY KEY` — identificador único

La **clave primaria** (*primary key*) identifica de forma única e irrepetible cada fila de una tabla.

- No pueden existir dos filas con el mismo valor en la columna clave.
- Nunca puede ser `NULL`.
- Cada tabla tiene exactamente una clave primaria.

```
cursor.execute("""
    CREATE TABLE IF NOT EXISTS productos (
        codigo   TEXT    PRIMARY KEY,  ## identifica cada producto de forma única
        nombre   TEXT    NOT NULL,
        precio   REAL    NOT NULL,
        cantidad INTEGER NOT NULL
    )
""")
```

> Equivale a usar `codigo` como clave de un diccionario Python, donde la clave tampoco puede repetirse.

Python, SQL y valores nulos

`NULL` en SQL ≠ `0` ≠ `""` ≠ `False` → significa **valor desconocido, ausente o no aplicable**.

Python	SQL	Con restricción `NOT NULL`
`""`	texto vacío `''`	✅ Se acepta — no es `NULL`
`None`	`NULL`	🚫 `IntegrityError`
`0` / `False`	`0`	✅ Se acepta — tiene valor

- `NOT NULL` solo prohíbe `NULL`. Una cadena vacía `""` lo cumple.
- En Python, `None` se mapea a `NULL`. Si la columna tiene `NOT NULL`, la BD rechaza la inserción.
- SQLite acepta `""` incluso en columnas numéricas — valida los datos en Python antes de enviarlos.

Python, SQL y valores nulos — ejemplos

✅ `""` **no es** `NULL` → se guarda sin error

```
cursor.execute(
    "INSERT INTO productos (codigo, nombre) VALUES (?, ?)",
    ("A1", "")
)
```

🚫 `None` → `NULL` → `IntegrityError` si `NOT NULL`

```
cursor.execute(
    "INSERT INTO productos (codigo, nombre) VALUES (?, ?)",
    ("A1", None)
)
```

`IntegrityError: NOT NULL constraint failed: productos.nombre`

`FOREIGN KEY` — relaciones entre tablas

Una **clave foránea** (*foreign key*) referencia la `PRIMARY KEY` de otra tabla. Garantiza la **integridad referencial**: ninguna fila puede apuntar a un registro inexistente.

```
cursor.execute("""
    CREATE TABLE IF NOT EXISTS descuentos (
        codigo     TEXT PRIMARY KEY,
        porcentaje REAL NOT NULL,
        FOREIGN KEY (codigo) REFERENCES productos(codigo)
    )
""")
```

💡 `FOREIGN KEY (codigo) REFERENCES productos(codigo)` establece la regla: `descuentos.codigo` debe existir previamente en `productos.codigo`. Si no existe → `IntegrityError`.

`PRAGMA foreign_keys` y orden de inserción

SQLite tiene las claves foráneas **desactivadas por defecto**. Actívalas al inicio de cada conexión:

```
cursor.execute("PRAGMA foreign_keys = ON")

## 1. Primero el padre — tabla referenciada
cursor.execute("INSERT INTO productos VALUES ('D1', 'Refresco', 2.50, 5)")

## 2. Después el hijo — tabla con FK, D1 ya existe en productos
cursor.execute("INSERT INTO descuentos VALUES ('D1', 20)")
conn.commit()
```

🚫 **Insertar el hijo antes que el padre →** `IntegrityError`

`INSERT INTO descuentos VALUES ('D1', 20)` falla si D1 aún no existe en `productos`.

7. Mapeo de objetos Python a tablas SQL

De la clase al esquema, de la instancia a la fila

De clase a tabla

Cada clase Python se mapea directamente a una tabla SQL: atributos → columnas, instancias → filas.

Elemento Python	Elemento SQL
Clase `Item`	Tabla `productos`
Atributo `codigo`	Columna `codigo TEXT PRIMARY KEY`
Atributo `nombre`	Columna `nombre TEXT NOT NULL`
Atributo `precio`	Columna `precio REAL NOT NULL`
Atributo `cantidad`	Columna `cantidad INTEGER NOT NULL`
`Item("A1", "Agua", 1.00, 10)`	Fila `('A1', 'Agua', 1.0, 10)`

Herencia con dos tablas

`ItemConDescuento` hereda de `Item` y añade `porcentaje_descuento` → **dos tablas relacionadas**.

`Item` normal

→ 1 `INSERT` en `productos`

```
cursor.execute(
    "INSERT INTO productos VALUES "
    "('A1', 'Agua', 1.00, 10)"
)
```

`ItemConDescuento`

→ 2 `INSERT`: primero `productos`, luego `descuentos`

```
cursor.execute(
    "INSERT INTO productos VALUES "
    "('D1', 'Refresco', 2.50, 5)"
)
cursor.execute(
    "INSERT INTO descuentos VALUES ('D1', 20)"
)
```

La tabla `descuentos` solo tiene filas para los productos con descuento. Un `Item` normal no aparece en `descuentos`.

¿Preguntas?

BD relacional

SQLite · sqlite3

CREATE TABLE · INSERT · SELECT · UPDATE · DELETE

PRIMARY KEY · NULL

FOREIGN KEY

PD4 · CEPY · UT3 — De objetos Python a bases de datos relacionales