Databases & SQL

Python Course · Module 7

ElnurBDa

Module 7 — Databases & SQL

In this module we connect **real-world programs** to **persistent, structured storage**.

We will learn:

- What databases are and when to use them
- Main database types
- How to use **SQLite** with Python
- Essential **SQL** syntax
- Building a simple database application

Files vs. Databases

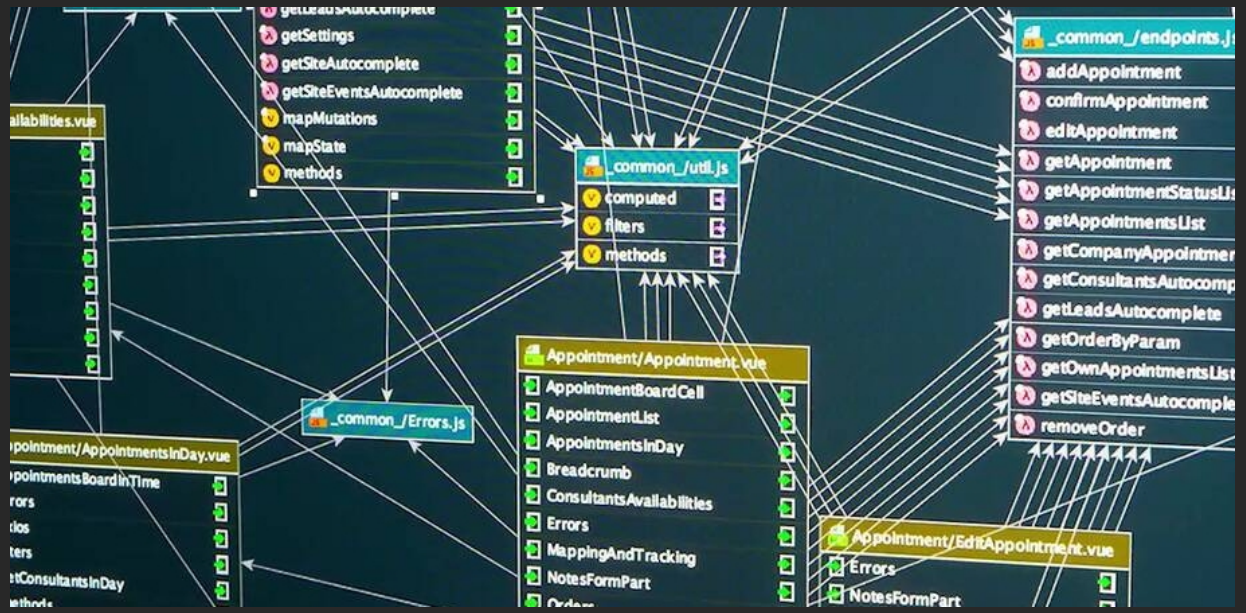
Programs normally store data in **RAM** (temporary).
Files and databases store data on **disk** (persistent).

Files are good for:

- Save logs
- Export reports
- Config files
- Simple data exchange

Databases are good for:

- Large data sets
- Structured relations
- Many users at once
- Fast searching / filtering



What is a Database?

A **database** is an organized collection of data.

Key features:

- Structured storage (tables / documents / graphs)
- Fast querying (indexes)
- Concurrency (many users at once)
- Security (permissions, access control)

Compared to raw files:

- Databases **understand** what the data means
- Easier to enforce rules (constraints)
- Optimized for large datasets

Main Database Types

SQL / Relational

- Tables: rows & columns
- Strong schema
- Uses SQL language
- Examples: PostgreSQL, MySQL, SQLite

Best for:

- Transactions
- Clear structure
- Reporting

NoSQL

- Flexible schema
- Designed for scale

Flavors:

- Key-value (Redis)
- Document (MongoDB)
- Graph (Neo4j)
- Wide-column (Cassandra)

SQLite – Lightweight SQL

SQLite is a **serverless** database engine.

Characteristics:

- Everything in **one file** (e.g. `app.db`)
- Excellent for desktop / small tools / testing
- Comes built-in with Python (`sqlite3` module)
- No separate install, no server process

Understanding Relational Databases — Tables, Rows & Columns

Think of a **table** like a **spreadsheet**:

id	username	email	score
1	alice	alice@example.com	100
2	bob	bob@example.com	85
3	charlie	charlie@example.com	92

- **Table** — the whole collection (like a sheet named "users")
- **Columns** — vertical categories (id, username, email, score)
- **Rows** — horizontal records (one per user)
- **Cell** — intersection of row & column (a single value)

Relational Databases — Why Relations?

Relational means tables are connected.

Example — two related tables:

USERS table:

```
id | username | score
---+----------+-------
1  | alice    | 100
2  | bob      | 85
3  | charlie  | 92
```

POSTS table:

```
id | user_id | title
---+---------+------------
1  | 1       | "Hello"
2  | 1       | "Hi again"
3  | 2       | "World"
```

The `user_id` in POSTS points to `id` in USERS.

Benefits:

- No duplicate data (alice's info stored once)
- Easy to update (change once, everywhere updates)
- Enforced connections (`FOREIGN KEY`)
- Prevent invalid references (no post from user_id 999)

Database Schema — Planning Your Data

Before creating tables, **plan your structure**:

1. **Identify entities** — What objects exist? (users, posts, comments)
2. **List attributes** — What info about each? (username, email, date)
3. **Choose types** — TEXT, INTEGER, REAL, etc.
4. **Define constraints** — Required? Unique? Default?
5. **Add relationships** — How do entities connect?

Example planning:

```
USERS:
- id (INTEGER, PRIMARY KEY, auto-increment)
- username (TEXT, NOT NULL, UNIQUE)
- email (TEXT, NOT NULL, UNIQUE)
- created_at (TEXT)

POSTS:
- id (INTEGER, PRIMARY KEY, auto-increment)
- user_id (INTEGER, FOREIGN KEY → users.id)
- title (TEXT, NOT NULL)
- content (TEXT)
- created_at (TEXT)
```

Connecting to SQLite

```python
import sqlite3

# creates file if it does not exist
conn = sqlite3.connect("app.db")

print("Connected:", conn)

conn.close()
```

———————— [finished] ————————

```
Connected: <sqlite3.Connection object at
0x7f4bcf356c50>
```

General pattern:

```python
conn = sqlite3.connect("app.db")
cur = conn.cursor()
# ... execute SQL here ...
conn.commit()
conn.close()
```

SQL Basics – Data Types

SQLite supports common data types:

```
INTEGER    whole numbers (1, -42, 1000)
REAL       floating point (3.14, 2.71)
TEXT       strings ("hello", "John")
BLOB       binary data (images, files)
NULL       no value (missing data)
```

Example:

```sql
CREATE TABLE products (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    price REAL,
    quantity INTEGER DEFAULT 0,
    description TEXT
);
```

SQL — CREATE TABLE

```sql
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL UNIQUE,
    email TEXT NOT NULL UNIQUE,
    score INTEGER DEFAULT 0,
    created_at TEXT
);
```

Key constraints:

- `PRIMARY KEY` — unique identifier
- `NOT NULL` — value must be provided
- `UNIQUE` — no duplicate values
- `DEFAULT` — default value if not provided
- `AUTOINCREMENT` — auto-generate incrementing IDs

SQL — INSERT

Insert new rows into a table.

```sql
INSERT INTO users (username, email, score)
VALUES ('alice', 'alice@example.com', 100);
```

Insert multiple rows:

```sql
INSERT INTO users (username, email, score)
VALUES
    ('bob', 'bob@example.com', 85),
    ('charlie', 'charlie@example.com', 92),
    ('diana', 'diana@example.com', 78);
```

Always specify column names for clarity and safety.

SQL — SELECT

Query data from a table.

```sql
SELECT * FROM users;
```

Select specific columns:

```sql
SELECT username, score FROM users;
```

With WHERE condition:

```sql
SELECT * FROM users WHERE score > 80;
```

With multiple conditions:

```sql
SELECT * FROM users
WHERE score > 80 AND username LIKE 'a%';
```

SQL — SELECT Advanced

ORDER BY — sort results:

```
SELECT username, score FROM users ORDER BY score DESC;
```

LIMIT — restrict number of results:

```
SELECT * FROM users LIMIT 5;
```

OFFSET — pagination:

```
SELECT * FROM users LIMIT 5 OFFSET 10;
```

COUNT — count rows:

```
SELECT COUNT(*) FROM users;
```

Aggregate functions: SUM(), AVG(), MIN(), MAX()

SQL — UPDATE

Modify existing rows.

```sql
UPDATE users SET score = 150 WHERE username = 'alice';
```

Update multiple columns:

```sql
UPDATE users
SET score = 95, email = 'newemail@example.com'
WHERE username = 'bob';
```

Update with calculations:

```sql
UPDATE users SET score = score + 10 WHERE score < 50;
```

⚠️ **Always use WHERE clause** to avoid updating all rows!

SQL — DELETE

Remove rows from a table.

```
DELETE FROM users WHERE username = 'alice';
```

Delete multiple rows:

```
DELETE FROM users WHERE score < 10;
```

Delete all rows (keep table structure):

```
DELETE FROM users;
```

⚠️ **Always use WHERE clause** — without it, all rows are deleted!

SQL — Relationships & Foreign Keys

Create related tables:

```
CREATE TABLE posts (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    title TEXT NOT NULL,
    content TEXT,
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

FOREIGN KEY ensures:

- Only valid user IDs can be referenced
- Data integrity
- Prevents orphaned posts

SQL — JOIN

Combine data from multiple tables.

```sql
SELECT users.username, posts.title
FROM posts
JOIN users ON posts.user_id = users.id;
```

Different JOIN types:

- `INNER JOIN` — matching rows only
- `LEFT JOIN` — all from left table
- `RIGHT JOIN` — all from right table
- `FULL OUTER JOIN` — all from both tables

Example with filtering:

```sql
SELECT users.username, COUNT(posts.id) as post_count
FROM users
LEFT JOIN posts ON users.id = posts.user_id
GROUP BY users.id
HAVING COUNT(posts.id) > 0;
```

SQLite CLI

SQLite has a **command-line tool** for quick testing.

Open a database:

```
sqlite3 app.db
```

You'll see:

```
SQLite version 3.x.x
sqlite>
```

Common commands:

```
.tables              -- list tables
.schema              -- show CREATE statements
.exit                -- quit
.mode column         -- prettier output
.headers on          -- show column names
```

SQLite CLI — Running SQL

```
sqlite> CREATE TABLE users (
...>     id INTEGER PRIMARY KEY AUTOINCREMENT,
...>     username TEXT NOT NULL UNIQUE,
...>     score INTEGER DEFAULT 0
...> );

sqlite> INSERT INTO users (username, score) VALUES ('alice', 100);
sqlite> INSERT INTO users (username, score) VALUES ('bob', 85);

sqlite> SELECT * FROM users;
id|username|score
1|alice|100
2|bob|85

sqlite> SELECT * FROM users WHERE score > 80 ORDER BY score DESC;
id|username|score
1|alice|100
2|bob|85

sqlite> UPDATE users SET score = 95 WHERE username = 'bob';

sqlite> .exit
```

Tip: Press ↑ to recall previous commands!

Using SQL in Python

```python
import sqlite3

conn = sqlite3.connect("app.db")
cur = conn.cursor()

sql = (
    "CREATE TABLE IF NOT EXISTS users ("
    " id INTEGER PRIMARY KEY AUTOINCREMENT,"
    " username TEXT NOT NULL UNIQUE,"
    " score INTEGER DEFAULT 0"
    ")"
)
cur.execute(sql)

conn.commit()
conn.close()
print("Table created.")
```

———————————— [finished] ————————————

```
Table created.
```

Notes:

- IF NOT EXISTS avoids errors
- PRIMARY KEY gives unique id

CRUD — Insert & Select

```python
import sqlite3

conn = sqlite3.connect("app.db")
cur = conn.cursor()

cur.execute(
    "INSERT INTO users (username, score) VALUES (?, ?)",
    ("admin", 100),
)

cur.execute("SELECT id, username, score FROM users")
rows = cur.fetchall()

for row in rows:
    print(row)

conn.commit()
conn.close()
```

———————————— [finished with error] ————————————

```
Traceback (most recent call last):
  File "/tmp/nix-shell-61594-0/.presentermnBl8gK/snippet.py", line 6, in
<module>
    cur.execute(
    ~~~~~~~~~~~^
        "INSERT INTO users (username, score) VALUES (?, ?)",
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        ("admin", 100),
        ^^^^^^^^^^^^^^^
    )
    ^
sqlite3.IntegrityError: UNIQUE constraint failed: users.username
```

? placeholders prevent SQL injection.

CRUD — Update & Delete

```python
import sqlite3

conn = sqlite3.connect("app.db")
cur = conn.cursor()

cur.execute(
    "UPDATE users SET score = ? WHERE username = ?",
    (150, "admin"),
)

cur.execute(
    "DELETE FROM users WHERE username = ?",
    ("guest",),
)

conn.commit()
conn.close()
```

```
──────────────────── [finished] ────────────────────
```

Always `commit()` after changes.

Putting It Together

```python
import sqlite3

def get_conn():
    return sqlite3.connect("app.db")

def add_user(username, score):
    with get_conn() as conn:
        cur = conn.cursor()
        cur.execute(
            "INSERT INTO users (username, score) VALUES (?, ?)",
            (username, score),
        )

def list_users():
    with get_conn() as conn:
        cur = conn.cursor()
        cur.execute("SELECT id, username, score FROM users")
        return cur.fetchall()
```

Pattern:

- Small helper functions
- Use `with` for connection lifetime

Mini Task

Create a mini **score tracker**:

1. Create SQLite DB & users table
2. Ask user for username and score
3. Insert using **parameterized** query
4. Select and print all users
5. Add option to **update** score

Bonus:

- Create a posts table
- Link posts to users with FOREIGN KEY
- Show user + their posts