Python and SQL: intro / SQL platforms

Ewa Weychert

Class 10: SQL

Uniwersytet Warszawski | Wydział Nauk Ekonomicznych

Why Use SQL in Python?

- Combine the power of **Python** with the structure of **SQL**.
- Use Python for:
  - data cleaning and transformation,
  - automation and scripting,
  - analysis and visualization.
- Use SQL for:
  - querying structured data,
  - filtering, aggregations, and joins,
  - working with relational schemas.
- Together they form a flexible and powerful data workflow.

SQLite in Python: `sqlite3`

- **SQLite** is a lightweight, file-based relational database.
- Python has a built-in module: `sqlite3`.
- Great for:
  - small to medium datasets,
  - teaching and experimentation,
  - local analytics pipelines.
- No separate server needed: database is a single `.db` file.

Connecting to SQLite in Python

```
import sqlite3

# Connect to (or create) a database file
conn = sqlite3.connect("example.db")

# Create a cursor for executing SQL commands
cur = conn.cursor()
```

- `connect(...)` opens the database.
- `cursor()` gives you an object to execute SQL.
- Always remember to call `conn.commit()` and `conn.close()` when done.

Creating Tables

```
cur.execute("""
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    age INTEGER
);
""")
conn.commit()
```

- `CREATE TABLE IF NOT EXISTS` avoids errors.
- Use standard SQL DDL (Data Definition Language).

Inserting Data (Parameterized)

```
users = [
    ("Alice", 30),
    ("Bob",   25),
    ("Carol", 35)
]

cur.executemany("""
INSERT INTO users (name, age)
VALUES (?, ?);
""", users)

conn.commit()
```

- Use **placeholders** (`?`) to avoid SQL injection.
- `executemany` inserts multiple rows efficiently.

Querying Data

```
cur.execute("SELECT id, name, age FROM users;")
rows = cur.fetchall()

for row in rows:
    print(row)
```

- `execute(...)` sends a SQL query.
- `fetchall()` retrieves all results as a list of tuples.

Relational Joins

- Joins combine rows from two tables based on matching keys.
- Common types:
  - **INNER JOIN** – only matching rows in both tables.
  - **LEFT JOIN** – all rows from left table, matching from right.
  - **RIGHT JOIN** – all rows from right table, matching from left.
  - **FULL OUTER JOIN** – all rows from both, match when possible.
- SQLite natively supports:
  - `INNER JOIN`
  - `LEFT JOIN`
- Other joins can be *emulated*.

Example: Two Tables

Table A

```
key | val_A
----+------
1   | A1
2   | A2
3   | A3
4   | A4
```

Table B

```
key | val_B
----+------
3   | B3
4   | B4a
4   | B4b
5   | B5
```

INNER JOIN in SQLite

```
SELECT A.key, A.val_A, B.val_B
FROM A
INNER JOIN B
  ON A.key = B.key;
```

- Returns only rows where `A.key = B.key`.
- Keys present in only one table are ignored.

LEFT JOIN in SQLite

```
SELECT A.key, A.val_A, B.val_B
FROM A
LEFT JOIN B
  ON A.key = B.key;
```

- All rows from `A`, matching rows from `B`.
- Non-matching rows in `B` become `NULL`.

Emulating FULL OUTER JOIN in SQLite

```
SELECT A.key, A.val_A, B.val_B
FROM A
LEFT JOIN B ON A.key = B.key

UNION

SELECT B.key, A.val_A, B.val_B
FROM B
LEFT JOIN A ON B.key = A.key
WHERE A.key IS NULL;
```

- Combines:
  - all rows from `A` (left join),
  - plus rows from `B` that do not match `A`.

Bridging SQL and pandas

- Often we want to:
  - Run SQL queries in a database.
  - Load results into a `pandas.DataFrame`.
  - Continue analysis using pandas.
- pandas provides convenient helpers:
  - `pd.read_sql_query(...)`
  - `pd.read_sql_table(...)`

Reading SQL Results into pandas

```
import pandas as pd
import sqlite3

conn = sqlite3.connect("example.db")

df_users = pd.read_sql_query(
    "SELECT id, name, age FROM users;",
    conn
)

print(df_users)
conn.close()
```

- `read_sql_query` executes SQL and returns a DataFrame.

pandas Joins with `merge()`

```
import pandas as pd

A = pd.DataFrame({
    "key":   [1, 2, 3, 4],
    "val_A": ["A1", "A2", "A3", "A4"]
})

B = pd.DataFrame({
    "key":   [3, 4, 4, 5],
    "val_B": ["B3", "B4a", "B4b", "B5"]
})

inner_join = A.merge(B, on="key", how="inner")
```

- `how="inner"` corresponds to SQL `INNER JOIN`.
- Other options: `"left"`, `"right"`, `"outer"`.

Anti-Join in pandas

```
left_anti = (
    A.merge(B, on="key", how="left", indicator=True)
      .query("_merge == 'left_only'")
      .drop(columns="_merge")
)
```

- Selects rows that exist *only* in `A`.
- This pattern is useful for:
  - data validation,
  - finding unmatched records,
  - implementing set differences.

Comparing SQL Joins vs pandas `merge()`

- Conceptually the same operations:
  - match rows based on keys,
  - control which non-matching rows are kept.
- SQL:
  - Declarative: describes *what* result you want.
  - Great for databases and large datasets on disk.
- pandas:
  - Imperative/programmable: part of Python code.
  - Great for in-memory analytics and quick experiments.

Summary

- Python integrates smoothly with SQL through `sqlite3` and other libraries.
- SQLite is perfect for lightweight, local relational databases.
- SQL joins and pandas `merge()` express the same relational ideas with different syntax.
- pandas can both:
  - *consume* SQL query results,
  - and perform additional joins and transformations.
- Knowing both SQL and pandas gives you flexible options for data work.

Why SQL useful in Web App

1. **Why SQL inside a Streamlit app?**

   Without a database, a Streamlit app is essentially "forgetful":

   - Any information stored only in variables disappears after refresh or rerun.
   - Multiple users cannot share or persist state.

   With SQLite + Streamlit:

   - User inputs (e.g. form submissions) are saved in a persistent database file.
   - Data can later be queried, filtered, grouped, or aggregated using SQL.
   - Results may be loaded into pandas for further analysis or visualization.
   - Setup is extremely simple, requires no separate database server.

   Streamlit provides the interface; SQLite provides the memory and structure.

2. **Basic architecture**

   Conceptually:

   - `app.py` (Streamlit):
     - Displays UI elements such as forms and buttons
     - Calls helper functions that interact with the database
   - `my_database.db` (SQLite file):
     - Holds relational tables (e.g. employees)
     - Stores and retrieves data using SQL operations

   Instead of writing Python manually, users interact with forms and SQL handles the persistence layer.

Why SQL is Useful in a Web App

Collecting user data with a form:

- The UI includes text inputs, dropdowns, and number fields.
- When the user saves an employee, the application passes the data to a database helper.

Persisting the data with SQLite:

- The helper stores the submitted information in the database.
- This ensures data remains available after restart or refresh.

Displaying & filtering data:

- The app retrieves data from the database and displays it in an interactive table.
- Filtering by department or other attributes becomes a simple SQL query.

Easy to extend:

- Add update/delete actions for each employee.
- Add more related tables and join them.
- Add summary charts computed from SQL aggregations.

Next steps:

- Enable editing and deleting employees directly from the UI.
- Build analogous apps for your court or forest inventory examples.

What app_sql_1.py Does

- **Login / Register** Credentials are stored in a simple `users` table.
- **Dashboard** Displays key metrics derived from the `user_actions` table.
- **Click logging** Every click (e.g. liking the dashboard or downloading a report) adds a row to `user_actions` with a timestamp.
- **History views**
  - A personal activity log filtered by username.
  - A system-wide activity log for administrative oversight.

This structure can be adapted to your own domain—such as court cases or forest inventory.

Goal of the App

- Build an interactive **Streamlit** application that supports:
  - User authentication (login & registration)
  - Logging user actions into SQL
  - Exploratory Data Analysis (EDA) for uploaded CSVs
- Use a small **SQLite** database (`app.db`) to store:
  - User account information
  - User action logs
- Provide a structured EDA interface with:
  - Overview panel
  - Missingness and data-type inspection
  - Analysis of numerical variables
  - Analysis of categorical variables

High-Level Architecture

- **Streamlit front-end**
  - Login / register interface
  - Multi-tab Data Explorer
- **SQLite back-end**
  - Stores user accounts
  - Stores all logged user interactions
  - Optional details column for extra metadata
- **EDA layer**
  - Uses pandas for data handling
  - Uses matplotlib for visualizations
  - Organized into 4 tabs, each with 4 logical sections

SQL / Auth Layer: Conceptual Overview

- The application opens a connection to a local SQLite database file.
- On startup, the database is initialized:
  - A table for users is created if it doesn't exist.
  - A table for user actions is created if it doesn't exist.
  - Older databases are updated to include newer fields (e.g. details column).
- The **users** table contains:
  - a unique username,
  - a password (demo only – no hashing),
  - an auto-incrementing ID.
- The **user_actions** table tracks:
  - who did what,
  - when they did it,
  - optional extra information.

Authentication Helpers (Conceptual)

- A function creates a new user account, ensuring usernames remain unique.
- A verification function checks whether a username and password match an existing record.
- These helpers allow Streamlit to switch between logged-in and logged-out states.

Logging User Actions (Conceptual)

- Throughout the application, user actions are recorded in the database.
- Examples include:
  - successful logins,
  - file uploads,
  - saving analysis snapshots,
  - switching tabs or pressing buttons.
- This creates a full audit trail of user behavior for later analysis.

Login / Register Screen

- The login page includes two tabs: one for signing in, one for registration.
- Successful login updates the session state and logs the event.
- The registration tab checks whether the username is free and provides feedback.

EDA Layout: Main Idea

- After login, the user sees the **Data Explorer**.
- The user uploads a CSV file, which is analyzed in four tabs:
  1. Overview
  2. Missing & Types
  3. Continuous Variables
  4. Categorical Variables
- Each tab uses a four-block layout for structured display.
- Sidebar controls allow adjusting:
  - number of rows to preview,
  - histogram bin count,
  - number of categories to display.

EDA Panels (Conceptual Descriptions)

Overview Panel

- Shows dataset shape, first rows, and column names.
- Provides numerical and categorical summaries.

Missing Values & Data Types

- Displays missing value counts and percentages.
- Shows data types and number of unique values.

Continuous Variables

- Histograms generated for all numerical columns.
- Display arranged in two-column format.

Categorical Variables

- Bar plots for top-N categories per column.
- Rotated labels for readability.

Main Application Flow

- On startup, the database tables are created or updated.
- The sidebar indicates login status and includes a logout button.
- The main area shows:
  - the login/registration page if no user is logged in,
  - the full EDA dashboard if a user is authenticated.
- All major interactions log an appropriate event to the database.

Summary

- The app demonstrates how to:
  - combine **Streamlit** with **SQLite**,
  - build a minimal authentication system,
  - track user behavior using SQL logs,
  - create a structured, tabbed EDA workflow.
- This architecture can be extended to:
  - more advanced authentication,
  - richer logging,
  - domain-specific dashboards (e.g. courts, forestry, finance).