Essential Guide to SQL with Python

Query, analyze, and combine data with ease: master SQL in your Python projects.

A practical and detailed guide to using SQL with Python, covering SQLite, pandas, and real-world data handling.

rubió
Metabolomics

Ibon Martínez-Arranz | imartinez@labrubiocom
Data Science Manager at Rubió Metabolomics
www.rubiometabolomics.com

Ibon Martínez-Arranz, PhD in Mathematics and Statistics, holds MSc degrees in Applied Statistical Techniques and Mathematical Modeling. He has extensive expertise in statistical modeling and advanced data analysis techniques. Since 2017, he has led the Data Science area at Rubió Metabolomics, driving predictive model development and statistical computing management for metabolomics, data handling, and R&D projects. His doctoral research in Mathematics investigated and adapted genetic algorithms to improve the classification of NAFLD subtypes.

Itziar Mincholé Canals | iminchole@labrubio.com
Data Specialist at Rubió Metabolomics
www.rubiometabolomics.com

Itziar Mincholé Canals has a degree in physical sciences (1999) from the University of Zaragoza (Spain). In 1999 she began developing her professional career as a software analyst and developer, mainly working on web development projects and database programming in different sectors. In 2009 she joined OWL Metabolomics as a bioinformatician, where she has participated in several R&D projects and has developed several software tools for metabolomics. In 2016 she completed the master's degree in applied statistics with R software. After joining the Data Science department, she also supports data analysis.

Essential Guide to SQL with Python

Ibon Martínez-Arranz

Life Feels Good

rubió Metabolomics

Contents

Introduction and Setup **1**
- What is SQL and why it's important for data science . . . 1
- Introduction to relational databases . . . 1
- Installing SQLite and pandas . . . 2
- Overview of the example datasets used in the book . . . 2
  - Load customers . . . 3
  - Load orders . . . 3

Getting Started with SQLite **5**
- Creating a SQLite database . . . 5
- Defining tables and data types . . . 5
- Inserting, updating, and deleting records . . . 6
- Basic queries with SELECT, WHERE, ORDER BY, LIMIT, and DISTINCT . . . 7

Relational Queries **9**
- Performing INNER, LEFT, and simulated RIGHT/FULL joins . . . 9

Using aggregate functions **13**

Grouping with GROUP BY and filtering with HAVING **15**

Advanced SQL Techniques **19**
- Subqueries and nested queries . . . 19
- Common Table Expressions (WITH) . . . 20
- Window functions . . . 21

Creating and querying views 22

Using SQLite with Python **25**

Working with the `sqlite3` module 25
Connect to database 25
Executing queries and retrieving results 26
Converting results to pandas DataFrames 26
Automating SQL tasks with Python 27

pandas + SQL Integration **31**

Reading and writing SQLite data with `pandas.read_sql()` and `to_sql()` 31
Read full table 31
Create a new DataFrame and write it to the database 32
Check insertion 32
Mixing SQL queries and pandas operations 33
Now use pandas for further filtering or analysis 34
Performance tips and best practices 35

Real-World Examples **37**

Full data analysis using SQLite and pandas 37
Step 1: Combine customer and order data 37
Step 2: Clean and transform data 38
Convert order_date to datetime 38
Extract month and year 38
Step 3: Total revenue over time 38
Step 4: Top customers by spending 39
Step 5: Orders by country 40

References and Resources **41**

Official Documentation 41
Recommended Books and Courses 41
Books: 41

Online Courses: 42
SQL and pandas Cheat Sheets 42

Introduction and Setup

What is SQL and why it's important for data science

Structured Query Language (SQL) is the standard language used to interact with relational databases. It allows you to query, insert, update, and manage data efficiently. In the context of data science, SQL is invaluable for working with large structured datasets that are often stored in relational databases.

Even if your final analysis is done in Python with pandas or in R, it is very common to extract and filter your data using SQL first. Understanding SQL makes it easier to work with production systems, data warehouses, and shared data platforms.

Introduction to relational databases

A relational database organizes data into tables, which consist of rows and columns. Each table represents an entity (like customers or orders), and relationships between tables are established through keys, typically using primary and foreign keys.

The most important features of relational databases include:

- Structured format with schemas
- Data integrity and consistency
- Support for complex queries and relationships

- Efficient storage and retrieval

We will use **SQLite**, a lightweight, file-based database system that is widely used for development, prototyping, and teaching purposes.

Installing SQLite and pandas

SQLite comes preinstalled with Python, so you don't need to install anything extra to use it. To interact with SQLite in Python, we'll use the built-in `sqlite3` module.

We will also use pandas to handle tabular data, convert query results to DataFrames, and perform additional data manipulation.

You can install pandas using pip:

```
pip install pandas
```

Optionally, you can install SQLite command line tools to inspect `.db` files:

- Linux: sudo apt install sqlite3
- macOS: brew install sqlite
- Windows: Download from https://www.sqlite.org/download.html

Overview of the example datasets used in the book

Throughout this book, we'll use two simple example tables: `customers` and `orders`. These tables are designed to demonstrate the core features of SQL and how to use them with Python.

```
import pandas as pd
```

Load customers

```
df_customers = pd.read_csv("../data/customers.csv")
print("Customers:")
print(df_customers.head())
```

```
Customers:
   customer_id          name              email
       country
0            1    Ana García      ana@gmail.com     Españ
    a
1            2    Luis Pérez   luis@hotmail.com     Mé
    xico
2            3   María López  maria@outlook.com
    Argentina
3            4   Carlos Ruiz   carlos@gmail.com     Españ
    a
4            5  Laura Torres    laura@gmail.com
    Chile
5            6  Jorge Martín  jorge@empresa.com       Per
    ú
6            7  Isabel Gómez   isabel@yahoo.com     Españ
    a
```

Load orders

```
df_orders = pd.read_csv("../data/orders.csv")
print("\nOrders:")
print(df_orders.head())
```

```
Orders:
   order_id  customer_id  order_date  total_amount
0       101            1  2023-01-10        120.50
1       102            2  2023-01-15         75.00
2       103            1  2023-02-01        200.00
3       104            3  2023-02-10        150.25
4       105            4  2023-03-05         99.99
```

As you can see, the `customers` table contains basic information such as customer ID, name, email, and country. The `orders` table contains order-specific data and references the customer via `customer_id`.

These datasets will help us build all the SQL examples we need, including joins, aggregations, window functions, and much more.

Getting Started with SQLite

Creating a SQLite database

We willl begin by creating a SQLite database file using the `sqlite3` module in Python. This file will store all of our tables and data throughout the book.

```
import sqlite3
conn = sqlite3.connect("../data/example.db")
print("Database created and connected.")
conn.close()
```

```
Database created and connected.
```

Defining tables and data types

To define tables, we use the `CREATE TABLE` statement. Here's how we can define the `customers` and `orders` tables, specifying appropriate data types.

```
conn = sqlite3.connect("../data/example.db")
cursor = conn.cursor()
```

```
cursor.execute("""
CREATE TABLE IF NOT EXISTS customers (
    customer_id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT,
```

```
    country TEXT
);
""")
```

```
<sqlite3.Cursor at 0x7735899047a0>
```

```
cursor.execute("""
CREATE TABLE IF NOT EXISTS orders (
    order_id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    order_date TEXT,
    total_amount REAL,
    FOREIGN KEY (customer_id) REFERENCES customers(
        customer_id)
);
""")
```

```
<sqlite3.Cursor at 0x7735899047a0>
```

```
conn.commit()
print("Tables created.")
```

```
Tables created.
```

Inserting, updating, and deleting records

Now let's insert some sample records and demonstrate how to update and delete data.

```
cursor.execute("INSERT INTO customers VALUES (8, 'Anabel
    López', 'anabel@gmail.com', 'España')")
cursor.execute("UPDATE customers SET country = 'España (
    Madrid)' WHERE customer_id = 1")
cursor.execute("DELETE FROM customers WHERE customer_id =
     99")  # No effect, just an example
```

```
<sqlite3.Cursor at 0x7735899047a0>
```

```
conn.commit()
print("Insert, update, and delete operations completed.")
```

```
Insert, update, and delete operations completed.
```

Basic queries with SELECT, WHERE, ORDER BY, LIMIT, and DISTINCT

Here are some basic queries to retrieve data from the database:

```
for row in cursor.execute("SELECT * FROM customers"):
    print(row)
```

```
(1, 'Ana García', 'ana@gmail.com', 'España (Madrid)')
(2, 'Luis Pérez', 'luis@hotmail.com', 'México')
(3, 'María López', 'maria@outlook.com', 'Argentina')
(4, 'Carlos Ruiz', 'carlos@gmail.com', 'España')
(5, 'Laura Torres', 'laura@gmail.com', 'Chile')
(6, 'Jorge Martín', 'jorge@empresa.com', 'Perú')
(7, 'Isabel Gómez', 'isabel@yahoo.com', 'España')
(8, 'Anabel López', 'anabel@gmail.com', 'España')
```

```
print("\nCustomers from Spain:")
for row in cursor.execute("SELECT * FROM customers WHERE
    country LIKE 'España%'"):
    print(row)
```

```
Customers from Spain:
(1, 'Ana García', 'ana@gmail.com', 'España (Madrid)')
(4, 'Carlos Ruiz', 'carlos@gmail.com', 'España')
(7, 'Isabel Gómez', 'isabel@yahoo.com', 'España')
(8, 'Anabel López', 'anabel@gmail.com', 'España')
```

```
print("\nCustomers ordered by name:")
for row in cursor.execute("SELECT * FROM customers ORDER BY name ASC"):
    print(row)
```

```
Customers ordered by name:
(1, 'Ana García', 'ana@gmail.com', 'España (Madrid)')
(8, 'Anabel López', 'anabel@gmail.com', 'España')
(4, 'Carlos Ruiz', 'carlos@gmail.com', 'España')
(7, 'Isabel Gómez', 'isabel@yahoo.com', 'España')
(6, 'Jorge Martín', 'jorge@empresa.com', 'Perú')
(5, 'Laura Torres', 'laura@gmail.com', 'Chile')
(2, 'Luis Pérez', 'luis@hotmail.com', 'México')
(3, 'María López', 'maria@outlook.com', 'Argentina')
```

```
print("\nFirst 2 customers:")
for row in cursor.execute("SELECT * FROM customers LIMIT 2"):
    print(row)
```

```
First 2 customers:
(1, 'Ana García', 'ana@gmail.com', 'España (Madrid)')
(2, 'Luis Pérez', 'luis@hotmail.com', 'México')
```

```
print("\nDistinct countries:")
for row in cursor.execute("SELECT DISTINCT country FROM customers"):
    print(row)
```

```
Distinct countries:
('España (Madrid)',)
('México',)
('Argentina',)
('España',)
('Chile',)
('Perú',)
```

```
conn.close()
```

Relational Queries

Performing INNER, LEFT, and simulated RIGHT/FULL joins

In relational databases, joins are used to combine rows from two or more tables based on a related column. This is one of the most powerful features of SQL, allowing you to retrieve meaningful insights from multiple tables.

We willl start by reloading our `customers` and `orders` tables and exploring different join types:

```
import sqlite3
conn = sqlite3.connect("../data/example.db")
cursor = conn.cursor()
```

```
print("\nINNER JOIN: Customers with at least one order")
for row in cursor.execute("""
    SELECT c.customer_id, c.name, o.order_id, o.
        total_amount
    FROM customers c
    INNER JOIN orders o ON c.customer_id = o.customer_id
"""):
    print(row)
```

```
INNER JOIN: Customers with at least one order
(1, 'Ana García', 101, 120.5)
(1, 'Ana García', 103, 200.0)
(2, 'Luis Pérez', 102, 75.0)
```

```
(2, 'Luis Pérez', 106, 50.0)
(3, 'María López', 104, 150.25)
(4, 'Carlos Ruiz', 105, 99.99)
(5, 'Laura Torres', 107, 180.0)
(8, 'Anabel López', 108, 300.0)
```

```
print("\nLEFT JOIN: All customers, even those without orders")
for row in cursor.execute("""
    SELECT c.customer_id, c.name, o.order_id, o.total_amount
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
"""):
    print(row)
```

```
LEFT JOIN: All customers, even those without orders
(1, 'Ana García', 101, 120.5)
(1, 'Ana García', 103, 200.0)
(2, 'Luis Pérez', 102, 75.0)
(2, 'Luis Pérez', 106, 50.0)
(3, 'María López', 104, 150.25)
(4, 'Carlos Ruiz', 105, 99.99)
(5, 'Laura Torres', 107, 180.0)
(6, 'Jorge Martín', None, None)
(7, 'Isabel Gómez', None, None)
(8, 'Anabel López', 108, 300.0)
```

```
print("\nSimulated RIGHT JOIN: All orders, even if customer is missing")
for row in cursor.execute("""
    SELECT o.order_id, o.customer_id, c.name, o.total_amount
    FROM orders o
    LEFT JOIN customers c ON o.customer_id = c.customer_id
"""):
    print(row)
```

```
Simulated RIGHT JOIN: All orders, even if customer is
```

```
    missing
(101, 1, 'Ana García', 120.5)
(102, 2, 'Luis Pérez', 75.0)
(103, 1, 'Ana García', 200.0)
(104, 3, 'María López', 150.25)
(105, 4, 'Carlos Ruiz', 99.99)
(106, 2, 'Luis Pérez', 50.0)
(107, 5, 'Laura Torres', 180.0)
(108, 8, 'Anabel López', 300.0)
```

```
print("\nSimulated FULL OUTER JOIN (with UNION)")
for row in cursor.execute("""
    SELECT c.customer_id, c.name, o.order_id, o.
        total_amount
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
    UNION
    SELECT c.customer_id, c.name, o.order_id, o.
        total_amount
    FROM orders o
    LEFT JOIN customers c ON o.customer_id = c.
        customer_id
"""):
    print(row)
```

```
Simulated FULL OUTER JOIN (with UNION)
(1, 'Ana García', 101, 120.5)
(1, 'Ana García', 103, 200.0)
(2, 'Luis Pérez', 102, 75.0)
(2, 'Luis Pérez', 106, 50.0)
(3, 'María López', 104, 150.25)
(4, 'Carlos Ruiz', 105, 99.99)
(5, 'Laura Torres', 107, 180.0)
(6, 'Jorge Martín', None, None)
(7, 'Isabel Gómez', None, None)
(8, 'Anabel López', 108, 300.0)
```

Using aggregate functions

Aggregate functions are used to summarize data across multiple rows. Common examples include COUNT, SUM, AVG, MIN, and MAX.

Let's use them to answer some questions:

```
print("\nTotal number of orders:")
for row in cursor.execute("SELECT COUNT(*) FROM orders"):
    print(row)
```

```
Total number of orders:
(8,)
```

```
print("\nTotal revenue:")
for row in cursor.execute("SELECT SUM(total_amount) FROM
    orders"):
    print(row)
```

```
Total revenue:
(1175.74,)
```

```
print("\nAverage order amount:")
for row in cursor.execute("SELECT AVG(total_amount) FROM
    orders"):
    print(row)
```

```
Average order amount:
(146.9675,)
```

Grouping with GROUP BY and filtering with HAVING

We use `GROUP BY` to group rows based on a column, and then apply aggregate functions to each group. The `HAVING` clause allows us to filter these grouped results.

Examples:

```
print("\nNumber of orders per customer:")
for row in cursor.execute("""
    SELECT customer_id, COUNT(*) AS num_orders
    FROM orders
    GROUP BY customer_id
"""):
    print(row)
```

```
Number of orders per customer:
(1, 2)
(2, 2)
(3, 1)
(4, 1)
(5, 1)
(8, 1)
```

```
print("\nCustomers with more than 1 order:")
for row in cursor.execute("""
    SELECT customer_id, COUNT(*) AS num_orders
    FROM orders
    GROUP BY customer_id
```

```
    HAVING num_orders > 1
"""):
    print(row)
```

```
Customers with more than 1 order:
(1, 2)
(2, 2)
```

```
print("\nTotal amount spent per customer:")
for row in cursor.execute("""
    SELECT c.name, SUM(o.total_amount) AS total_spent
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.name
    ORDER BY total_spent DESC
"""):
    print(row)
```

```
Total amount spent per customer:
('Ana García', 320.5)
('Anabel López', 300.0)
('Laura Torres', 180.0)
('María López', 150.25)
('Luis Pérez', 125.0)
('Carlos Ruiz', 99.99)
('Jorge Martín', None)
('Isabel Gómez', None)
```

```
conn.close()
```

These examples illustrate the power of combining joins with grouping and aggregation.

They allow us to answer common analytical questions such as:

- Which customers generate the most revenue?
- How many orders does each customer have?
- Are there customers without any orders?

This is the foundation for more complex SQL analysis you willll see in later chapters.

Advanced SQL Techniques

Subqueries and nested queries

Subqueries are queries nested inside other queries. They can be used in SELECT, FROM, or WHERE clauses. They allow you to write powerful filters or computed columns.

```
import sqlite3
conn = sqlite3.connect("../data/example.db")
cursor = conn.cursor()
print("Database created and connected.")
```

```
Database created and connected.
```

```
print("\nCustomers who spent more than the average order
    amount:\n")
for row in cursor.execute("""
    SELECT c.name, SUM(o.total_amount) AS total_spent
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.customer_id
    HAVING total_spent > (
        SELECT AVG(total_amount) FROM orders
    )
"""):
    print(row)
```

```
Customers who spent more than the average order amount:

```

```
('Ana García', 320.5)
('María López', 150.25)
('Laura Torres', 180.0)
('Anabel López', 300.0)
```

Common Table Expressions (WITH)

CTEs (Common Table Expressions) allow you to define temporary named result sets that you can reuse in a query. They are especially useful for breaking complex queries into readable parts.

```
print("\nUsing a CTE to show customers with their total orders and revenue:\n")
for row in cursor.execute("""
    WITH customer_orders AS (
        SELECT customer_id, COUNT(*) AS num_orders, SUM(total_amount) AS total_spent
        FROM orders
        GROUP BY customer_id
    )
    SELECT c.name, co.num_orders, co.total_spent
    FROM customers c
    LEFT JOIN customer_orders co ON c.customer_id = co.customer_id
    ORDER BY co.total_spent DESC
"""):
    print(row)
```

```
Using a CTE to show customers with their total orders and revenue:

('Ana García', 2, 320.5)
('Anabel López', 1, 300.0)
('Laura Torres', 1, 180.0)
('María López', 1, 150.25)
('Luis Pérez', 2, 125.0)
('Carlos Ruiz', 1, 99.99)
```

```
('Jorge Martín', None, None)
('Isabel Gómez', None, None)
```

Window functions

Window functions allow you to perform calculations across a set of rows that are related to the current row. Unlike aggregate functions, they do not collapse the result set.

```
print("\nUsing ROW_NUMBER to rank customer orders:\n")
for row in cursor.execute("""
    SELECT customer_id, order_id, order_date,
        total_amount,
           ROW_NUMBER() OVER (PARTITION BY customer_id
               ORDER BY order_date) AS rn
    FROM orders
"""):
    print(row)
```

```
Using ROW_NUMBER to rank customer orders:

(1, 101, '2023-01-10', 120.5, 1)
(1, 103, '2023-02-01', 200.0, 2)
(2, 102, '2023-01-15', 75.0, 1)
(2, 106, '2023-03-15', 50.0, 2)
(3, 104, '2023-02-10', 150.25, 1)
(4, 105, '2023-03-05', 99.99, 1)
(5, 107, '2023-04-01', 180.0, 1)
(8, 108, '2023-04-15', 300.0, 1)
```

```
print("\nUsing RANK to compare order amounts across all
    orders:\n")
for row in cursor.execute("""
    SELECT order_id, customer_id, total_amount,
           RANK() OVER (ORDER BY total_amount DESC) AS
               rank
    FROM orders
```

```
"""):
    print(row)
```

```
Using RANK to compare order amounts across all orders:

(108, 8, 300.0, 1)
(103, 1, 200.0, 2)
(107, 5, 180.0, 3)
(104, 3, 150.25, 4)
(101, 1, 120.5, 5)
(105, 4, 99.99, 6)
(102, 2, 75.0, 7)
(106, 2, 50.0, 8)
```

Creating and querying views

A view is a virtual table based on a SQL query. It can simplify access to complex queries and help organize your logic. Views are useful for reuse and for encapsulating business logic.

```
cursor.execute("DROP VIEW IF EXISTS high_spenders")
cursor.execute("""
    CREATE VIEW high_spenders AS
    SELECT c.name, SUM(o.total_amount) AS total_spent
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.customer_id
    HAVING total_spent > 200
""")

print("\nQuerying the high_spenders view:\n")
for row in cursor.execute("SELECT * FROM high_spenders"):
    print(row)
```

```
Querying the high_spenders view:

('Ana García', 320.5)
```

```
('Anabel López', 300.0)
```

```
conn.close()
```

```
conn.close()
```

This chapter introduced several powerful SQL features:

- Subqueries to add flexibility to filtering and comparisons
- CTEs for structuring complex queries
- Window functions for row-wise computations
- Views for encapsulation and reuse

These tools are essential when working with large databases and need for clarity and maintainability.

Using SQLite with Python

Working with the `sqlite3` module

Python provides a built-in module called `sqlite3` to work with SQLite databases. You can connect to a database file (or create one if it doesn't exist), execute SQL commands, and fetch results using cursors.

```
import sqlite3
```

```
import sqlite3
```

Connect to database

```
conn = sqlite3.connect("../data/example.db")
cursor = conn.cursor()

print("Connected to SQLite using sqlite3 module.")
```

```
conn = sqlite3.connect("../data/example.db")
cursor = conn.cursor()

print("Connected to SQLite using sqlite3 module.")
```

```
Connected to SQLite using sqlite3 module.
```

Executing queries and retrieving results

Once connected, you can execute SQL queries using `cursor.execute()`. You can fetch results with `fetchone()`, `fetchall()`, or iterate over the cursor.

```
cursor.execute("SELECT * FROM customers LIMIT 5")
rows = cursor.fetchall()

print("\nFirst 5 customers:\n")
for row in rows:
    print(row)
```


```
cursor.execute("SELECT * FROM customers LIMIT 5")
rows = cursor.fetchall()

print("\nFirst 5 customers:\n")
for row in rows:
    print(row)
```


```
First 5 customers:

(1, 'Ana García', 'ana@gmail.com', 'España (Madrid)')
(2, 'Luis Pérez', 'luis@hotmail.com', 'México')
(3, 'María López', 'maria@outlook.com', 'Argentina')
(4, 'Carlos Ruiz', 'carlos@gmail.com', 'España')
(5, 'Laura Torres', 'laura@gmail.com', 'Chile')
```

Converting results to pandas DataFrames

To make further analysis easier, we often convert SQL query results into a pandas DataFrame. This is especially useful for filtering, aggregating, and visualizing the data in Python.

```
import pandas as pd
```

```
import pandas as pd
```

```
df = pd.read_sql_query("SELECT * FROM orders", conn)
print("\nOrders as a pandas DataFrame:\n")
print(df.head())
```


```
df = pd.read_sql_query("SELECT * FROM orders", conn)
print("\nOrders as a pandas DataFrame:\n")
print(df.head())
```


```
Orders as a pandas DataFrame:

   order_id  customer_id  order_date  total_amount
0       101            1  2023-01-10        120.50
1       102            2  2023-01-15         75.00
2       103            1  2023-02-01        200.00
3       104            3  2023-02-10        150.25
4       105            4  2023-03-05         99.99
```

Automating SQL tasks with Python

Python allows you to automate many tasks: generating reports, loading data, or updating databases. You can wrap your SQL logic in functions, or even integrate it in workflows or pipelines. Here's a simple example that prints a revenue report.

```
def print_revenue_report(min_amount=100):
    query = """
        SELECT c.name, SUM(o.total_amount) AS total_spent
        FROM customers c
        JOIN orders o ON c.customer_id = o.customer_id
        GROUP BY c.customer_id
        HAVING total_spent > ?
        ORDER BY total_spent DESC
    """
    df_report = pd.read_sql_query(query, conn, params=(
        min_amount,))
```

```
    print("\nRevenue report (customers with total spent >
        ", min_amount, "):\n")
    print(df_report)

print_revenue_report(150)
```

```
def print_revenue_report(min_amount=100):
    query = """
        SELECT c.name, SUM(o.total_amount) AS total_spent
        FROM customers c
        JOIN orders o ON c.customer_id = o.customer_id
        GROUP BY c.customer_id
        HAVING total_spent > ?
        ORDER BY total_spent DESC
    """
    df_report = pd.read_sql_query(query, conn, params=(
        min_amount,))
    print("\nRevenue report (customers with total spent >
        ", min_amount, "):\n")
    print(df_report)

print_revenue_report(150)
```

```
Revenue report (customers with total spent > 150 ):

           name  total_spent
0    Ana García       320.50
1  Anabel López       300.00
2  Laura Torres       180.00
3   María López       150.25
```

```
print_revenue_report(200)
```

```
Revenue report (customers with total spent > 200 ):

           name  total_spent
0    Ana García        320.5
1  Anabel López        300.0
```

```
conn.close()
```

```
conn.close()
```

This chapter has shown how easily Python and SQLite integrate:

- Use `sqlite3` to execute queries and manage the database
- Use pandas to analyze and manipulate data from SQL
- Build reusable and automatable scripts for data workflows

In the next chapter, we'll go deeper into combining pandas and SQL seamlessly.

pandas + SQL Integration

Reading and writing SQLite data with `pandas.read_sql()` and `to_sql()`

One of the strengths of pandas is its ability to interact directly with SQL databases. Using `read_sql()` you can read tables or custom queries directly into a DataFrame. Using `to_sql()` you can store any DataFrame as a table in your SQLite database.

```
import sqlite3
import pandas as pd

conn = sqlite3.connect("../data/example.db")
print("Database created and connected.")
```

```
Database created and connected.
```

Read full table

```
df_customers = pd.read_sql("SELECT * FROM customers",
    conn)
print("\nCustomers:\n")
print(df_customers.head())
```

```
Customers:
```

```
   customer_id          name              email
       country
0            1    Ana García      ana@gmail.com  España (
    Madrid)
1            2    Luis Pérez   luis@hotmail.com
             México
2            3   María López  maria@outlook.com
    Argentina
3            4   Carlos Ruiz   carlos@gmail.com
             España
4            5  Laura Torres    laura@gmail.com
              Chile
```

Create a new DataFrame and write it to the database

```
df_new = pd.DataFrame({
    "customer_id": [99],
    "name": ["Test User"],
    "email": ["test@example.com"],
    "country": ["Testland"]
})

df_new.to_sql("customers", conn, index=False, if_exists="
    append")
```

```
1
```

Check insertion

```
print("\nCustomer table after insertion:\n")
print(pd.read_sql("SELECT * FROM customers WHERE
    customer_id = 99", conn))
```

```
Customer table after insertion:

   customer_id       name             email   country
0           99  Test User  test@example.com  Testland
```

Mixing SQL queries and pandas operations

You can combine SQL queries with pandas for powerful and flexible data workflows. For example, retrieve filtered data using SQL and continue transforming it with pandas.

```
query = "SELECT * FROM orders WHERE total_amount > 100"
df_filtered = pd.read_sql_query(query, conn)
df_filtered
```

order_id

customer_id

order_date

total_amount

0

101

1

2023-01-10

120.50

1

103

1

2023-02-01

200.00

2

104

3

2023-02-10

150.25

3

107

5

2023-04-01

180.00

4

108

8

2023-04-15

300.00

Now use pandas for further filtering or analysis

```
high_value_recent = df_filtered[df_filtered["order_date"]
    > "2023-02-01"]
print("\nHigh value recent orders:\n")
print(high_value_recent)
```

```
High value recent orders:

   order_id  customer_id  order_date  total_amount
2       104            3  2023-02-10        150.25
3       107            5  2023-04-01        180.00
4       108            8  2023-04-15        300.00
```

Performance tips and best practices

Some tips to optimize the use of pandas and SQL together:

- Use SQL for filtering and joins to reduce data size early
- Load only the columns and rows you need (`SELECT column1, column2 ... WHERE ...`)
- Use indexes in your SQLite tables if you're running repeated queries
- Avoid using `to_sql(..., if_exists="replace")` in loops—it will slow things down and drop indexes
- For large datasets, chunk inserts and queries using `chunksize` parameter

Example:

```
print("\nReading orders in chunks of 3 rows:\n")
for chunk in pd.read_sql_query("SELECT * FROM orders",
    conn, chunksize=3):
    print(chunk)
```

```
Reading orders in chunks of 3 rows:

   order_id  customer_id  order_date  total_amount
0       101            1  2023-01-10         120.5
1       102            2  2023-01-15          75.0
2       103            1  2023-02-01         200.0
   order_id  customer_id  order_date  total_amount
0       104            3  2023-02-10        150.25
1       105            4  2023-03-05         99.99
```

```
2       106            2  2023-03-15          50.00
   order_id  customer_id  order_date  total_amount
0       107            5  2023-04-01         180.0
1       108            8  2023-04-15         300.0
```

```
conn.close()
```

This chapter demonstrated how to:

- Seamlessly connect pandas with SQLite
- Leverage the strengths of both SQL and pandas
- Optimize your data workflows using a hybrid approach

In the final chapter, we'll apply everything in a real-world scenario.

Real-World Examples

Full data analysis using SQLite and pandas

In this chapter, we'll apply everything we've learned in a realistic analysis scenario. We'll combine data from multiple tables, clean and filter it, and produce a simple revenue report.

The goal: find top customers by spending, orders per month, and total revenue over time.

```
import sqlite3
import pandas as pd
import matplotlib.pyplot as plt

conn = sqlite3.connect("../data/example.db")
```

Step 1: Combine customer and order data

```
query = """
    SELECT c.name, c.country, o.order_date, o.
        total_amount
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
"""
df = pd.read_sql_query(query, conn)
print("\nCombined data:\n")
print(df.head())
```

```
Combined data:

           name          country  order_date  total_amount
0   Ana García  España (Madrid)  2023-01-10        120.50
1   Ana García  España (Madrid)  2023-02-01        200.00
2   Luis Pérez           México  2023-01-15         75.00
3   Luis Pérez           México  2023-03-15         50.00
4  María López        Argentina  2023-02-10        150.25
```

Step 2: Clean and transform data

Convert order_date to datetime

```
df["order_date"] = pd.to_datetime(df["order_date"])
```

Extract month and year

```
df["year_month"] = df["order_date"].dt.to_period("M")
```

Step 3: Total revenue over time

```
revenue_over_time = df.groupby("year_month")["
    total_amount"].sum()
revenue_over_time.plot(kind="bar", title="Total Revenue
    Over Time")
plt.xlabel("Year-Month")
plt.ylabel("Revenue")
plt.tight_layout()
plt.show()
```

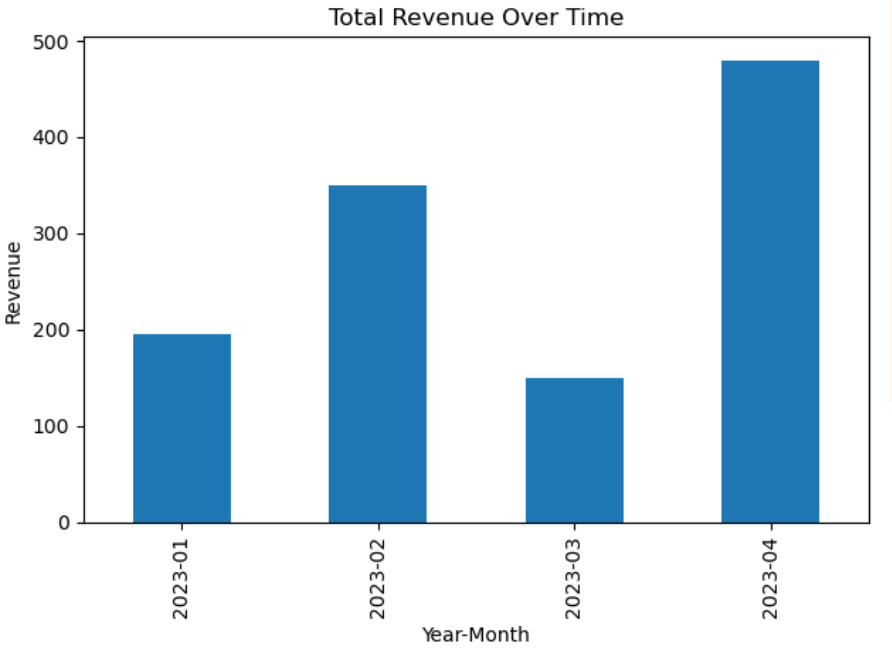

Figure 1: Image generated by the provided code.

Step 4: Top customers by spending

```
top_customers = df.groupby("name")["total_amount"].sum().
    sort_values(ascending=False)
print("\nTop customers by spending:\n")
print(top_customers.head())
```

```
Top customers by spending:

name
Ana García      320.50
Anabel López    300.00
Laura Torres    180.00
```

```
María López     150.25
Luis Pérez      125.00
Name: total_amount, dtype: float64
```

Step 5: Orders by country

```
orders_by_country = df.groupby("country")["total_amount"
    ].sum().sort_values(ascending=False)
print("\nRevenue by country:\n")
print(orders_by_country)
```

```
Revenue by country:

country
España              399.99
España (Madrid)     320.50
Chile               180.00
Argentina           150.25
México              125.00
Name: total_amount, dtype: float64
```

```
conn.close()
```

This chapter demonstrated a realistic data analysis pipeline:

- Querying and transforming relational data with SQL
- Performing analysis and visualization with pandas and matplotlib
- Producing reproducible and shareable insights

In the final chapter, we will gather references and resources for continued learning.

References and Resources

Official Documentation

For continued learning and to deepen your understanding, the official documentation is always the best place to start:

- SQLite: https://sqlite.org/docs.html
- Python sqlite3 module: https://docs.python.org/3/library/sqlite3.html
- pandas documentation: https://pandas.pydata.org/docs/
- matplotlib: https://matplotlib.org/stable/contents.html

Recommended Books and Courses

If you want to expand your skills beyond this guide, here are some well-regarded books and online courses:

Books:

- **Learning SQL** by Alan Beaulieu (O'Reilly)
- **SQL for Data Scientists** by Renee M. P. Teate (Wiley)
- **Python for Data Analysis** by Wes McKinney (O'Reilly)

Online Courses:

- **SQL for Data Science** (Coursera)
- **Joining Data in SQL** (Datacamp)
- **Mode SQL Tutorial** https://mode.com/sql-tutorial

SQL and pandas Cheat Sheets

Cheat sheets are quick references that help you remember syntax and workflows.

- SQL:
  - https://learnsql.com/blog/sql-cheat-sheet/
  - https://www.sqltutorial.org/sql-cheat-sheet/
- pandas:
  - https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf
  - https://www.datacamp.com/cheat-sheet/pandas-cheat-sheet-python

This concludes the *Essential Guide to SQL with Python*. You've learned how to:

- Create and manage databases with SQLite
- Write queries from basic to advanced
- Use SQL together with pandas in real-world scenarios
- Automate analysis and produce insights

Keep exploring, practicing, and building!


rubió
Metabolomics