



**Software Engineering  
Bootcamp**

Hyperiondev

**SQLite**

# Lecture – Housekeeping

- ❑ The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
- ❑ No question is daft or silly - **ask them!**
- ❑ There are Q/A sessions at the end of the session, should you wish to ask any follow-up questions.
- ❑ For all non-academic questions, please submit a query:  
[www.hyperiondev.com/support](http://www.hyperiondev.com/support)
- ❑ Report a safeguarding incident:  
<http://hyperiondev.com/safeguardreporting>

# Learning Outcomes

- Recall the importance of connecting Python with SQL for data manipulation and analysis.
- Explain the basic concepts of database connectivity in Python.
- Implement Python scripts to interact with SQL databases.
- Execute SQL queries from Python and retrieve data.

# Polls

1. What is the purpose of connecting Python with SQL?
  - A. Data visualisation
  - B. Data manipulation and analysis
  - C. Front-end development
  - D. Server administration

# Polls

2. What is a SQL database commonly used for?
  - A. Storing programming code
  - B. Managing web server configurations
  - C. Managing structured data
  - D. Running Python scripts

# Understanding Database Connectivity

- Database connectivity refers to the ability of an application or software system to establish a connection and interact with a database management system (DBMS) to perform various operations such as data retrieval, insertion, update, and deletion.

# Importance of Database Connectivity: Data Access

- Database connectivity enables applications to access and retrieve data stored in databases.
- This is crucial for data-driven applications that rely on accessing and analysing large volumes of structured or unstructured data to perform various functions.

# Importance of Database Connectivity: Data Manipulation

- With database connectivity, applications can manipulate data within the database, such as inserting new records, updating existing
- This allows for dynamic management and maintenance of the dataset.



# Importance of Database Connectivity: Real-Time Updates

- Many data-driven applications require real-time updates to reflect changes in the underlying data.
- Database connectivity facilitates the synchronisation of application data with the database, ensuring that users always have access to the most current information.

# Importance of Database Connectivity: Scalability

- Data-driven applications often need to **scale to handle increasing data volumes** and user loads.
- Enables applications to **connect to scalable database systems** that can efficiently manage large datasets and handle concurrent user requests.

# Importance of Database Connectivity: Security

- Database connectivity allows applications to **implement security measures** such as authentication, authorisation, and encryption to ensure the confidentiality, integrity, and availability of data.
- This is **essential for protecting sensitive information** from unauthorised access or malicious activities.

# Importance of Database Connectivity: Integration

- Many applications need to integrate with multiple data sources and systems to aggregate and analyse data from different sources.
- Enables seamless integration between the application and various database systems, allowing for efficient data exchange and interoperability.

# Importance of Database Connectivity: Performance

- Database connectivity allows applications to optimise performance by **executing queries directly on the database server**, leveraging the database's processing power and indexing capabilities.
- This **reduces the need for data transfer** between the application and the database, improving overall performance and response times.

# Database Connectivity: Benefits of Using Python

- Using Python for database connectivity offers numerous benefits, including:
  - ease of use,
  - extensive library support,
  - cross-platform compatibility,
  - integration with data analysis and machine learning,
  - strong community support
  - rapid development capabilities.
- These advantages make Python a compelling choice for building database-driven applications across a wide range of domains and industries.

# Database Interaction: SQLite

- SQLite is a **lightweight, self-contained SQL database engine** that requires minimal setup and configuration. It is often used for smaller-scale projects or applications where simplicity and ease of use are prioritised.

# SQLite: Key Features

- **Zero Configuration:** SQLite databases are **self-contained**, meaning they require no external server or setup. They are simply files that can be accessed by the application directly.
- **Single File:** The entire database is stored in a single file, making it **easy to distribute and manage**.
- **SQL Support:** SQLite supports standard SQL syntax for querying and manipulating data, making it **compatible with existing SQL-based applications and tools**.



# Database Interaction: SQLAlchemy

- SQLAlchemy is a powerful SQL toolkit and Object-Relational Mapping (ORM) library for Python. It provides a high-level interface for interacting with SQL databases, abstracting away many of the complexities involved in database interaction.
- SQLAlchemy allows developers to define database models using Python classes, which are then mapped to database tables.
- To use SQLAlchemy in Python, you need to install the library using pip.
- Further details on SQLAlchemy is beyond the scope of this lesson. Please feel free to do some research on how to implement SQLAlchemy.

# What is SQL?

- SQL stands for Structured Query Language
- SQL is a database language that is composed of commands that enable users to create databases or table structures, perform various types of data manipulation and data administration as well as query the database to extract useful information.

# Aspects of SQL?

- Data Definition Language (DDL):
  - Defines databases
  - Defines views
  - Defines access rights
- Data Manipulation Language (DML):
  - INSERT
  - UPDATE
  - DELETE
  - SELECT

# Why SQL?

- SQL is **easy to learn** since its vocabulary is relatively simple.
- Its basic command set has a vocabulary of fewer than 100 words.
- It is also a **non-procedural language**, which means that the **user specifies what must be done** and not how it should be done. This aligns with our declarative approach towards programming.
- Users do not need to know the physical data storage format or the complex activities that take place when a SQL command is executed in order to issue a command.

# SQL: Important Keywords

- **CREATE TABLE** : Creates a new table
- **NOT NULL** : Ensures that a column doesn't contain null values
- **UNIQUE** : Ensures that there are no repetitions
- **PRIMARY KEY** : Defines a primary key
- **FOREIGN KEY** : Defines a foreign key
- **DROP TABLE** : Deletes a table entirely

# SQL: Creating a Table

- To create new tables in SQL, you use the **CREATE TABLE** statement.
- Pass all the columns you want in the table, as well as their data types and constraints, as arguments to the CREATE TABLE function.
- The syntax of the CREATE TABLE statement:

```
CREATE TABLE table_name (  
    column1_name datatype constraint,  
    column2_name datatype constraint,  
    ...  
);
```

# SQL: CREATE TABLE Example

- Table names use the snake\_case convention with plural nouns.
- Columns name use the snake\_case convention with singular nouns.

```
CREATE TABLE employees (  
    employee_id int NOT NULL,  
    last_name varchar(255) NOT NULL,  
    first_name varchar(255),  
    address varchar(255),  
    phone_number varchar(255),  
);
```

# SQL: More Keywords

- **INSERT** : Inserts rows into a table
- **SELECT** : Select attributes from a row
- **WHERE** : Restricts the selection of rows based on a conditional expression.
- **ORDER BY** : Orders the selected rows by a specific column. Can specify ASCENDING or DESCENDING
- **UPDATE** : Modifies an attribute's values in a table
- **DELETE** : Deletes one or more rows from a table



# SQL: Inserting Records

- There are two ways to write the INSERT command:
  1. Specify both the column names and the values to be inserted.

```
INSERT INTO employees (column1, column2, column3, ...)  
VALUES (value1, value2, value3, ...)
```

2. You do not have to specify the column names if you are adding values for all of the columns of the table. However, you should ensure that the order of the values is in the same order as the columns in the table.

```
INSERT INTO table_name  
VALUES (value1, value2, value3,...);
```

# SQL: INSERT Example

1. Specify both the column names and the values to be inserted.

```
INSERT INTO employees (employee_id, last_name, first_name,  
address, phone_number)  
VALUES (1234, 'Smith', 'John', '25 Oak Rd', '0837856767');
```

2. Specify the values only.

```
INSERT INTO employees  
VALUES (1, 'Smith', 'John', '25 Oak Rd', '0837856767');
```

# SQL: Retrieving Data

- The **SELECT** statement is used to fetch data from a database. The data returned is stored in a result table, known as the result-set.
- To select all the columns in a table:

```
SELECT * FROM table_name;
```

- To select specific columns from a table:

```
SELECT column1, column2,...  
      FROM table_name;
```

# SQL: SELECT, WHERE Example

- To **select all** the columns in a table:

```
SELECT * FROM employees;
```

- To **select specific columns** from a table:

```
SELECT first_name, last_name,...  
FROM employees;
```

- To **select** columns from **specific records** only:

```
SELECT * FROM employees  
WHERE first_name = 'John';
```

# SQL: Ordering Data

- You can use the ORDER BY command to sort the results returned in ascending or descending order. The ORDER BY command sorts the records in ascending order by default.
- You need to use the **DESC** keyword to sort the records in descending order. **ASC** is the default sorting order.

```
SELECT * FROM table_name  
        ORDER BY column1 DESC;
```

- Ordering Data Example:

```
SELECT * FROM employees  
        ORDER BY last_name, first_name DESC;
```

# SQL: Modifying Data

- The **UPDATE** statement is used to modify rows in a table.
  - Choose the table where the row you want to change is located.
  - Set the new value(s) for the desired column(s).
  - Choose which of the rows you want to update using the WHERE statement. NB: If you omit the WHERE, all rows in the table will change.

```
UPDATE table_name  
    SET column1 = value1, column2 = value2, ...  
    WHERE condition;
```

# SQL: UPDATE Example

- From the table below:

customer_id	first_name	last_name	address	city
1	Maria	Anderson	23 York Street	New York
2	Jackson	Peters	124 River Road	Berlin
3	Thomas	Hardy	455 Hanover Square	London
4	Kelly	Martins	55 Loop Street	Cape Town

```
UPDATE customers
  SET address = '78 Oak St', city= 'Los Angeles'
 WHERE customer_id = 1;
```

# SQL: Removing Records

- The **DELETE** statement is used to remove existing rows from a table.
- Removing a row is a simple process. All you need to do is select the right table and row that you want to remove.

```
DELETE FROM table_name  
      WHERE condition;
```

- Removing Row/s Example:

```
DELETE FROM customers  
      WHERE customer_id = 4;
```



# SQL: Special Operations (Used in Conditional Statements)

- **BETWEEN** : Checks if a value is within range
- **IS NULL** : Checks if a value is null
- **LIKE** : Checks if a string matches a given pattern (regular expression)
- **IN** : Checks if a value is in a given list
- **EXISTS** : Checks if a query returns any rows
- **DISTINCT** : Limits to unique values

# Using IN and BETWEEN keywords

- **IN** Example:

```
SELECT * FROM albums  
WHERE genre IN ('pop', 'soul');
```

- **BETWEEN** Example:

```
SELECT * FROM albums  
WHERE released BETWEEN 1975 AND 1985;
```

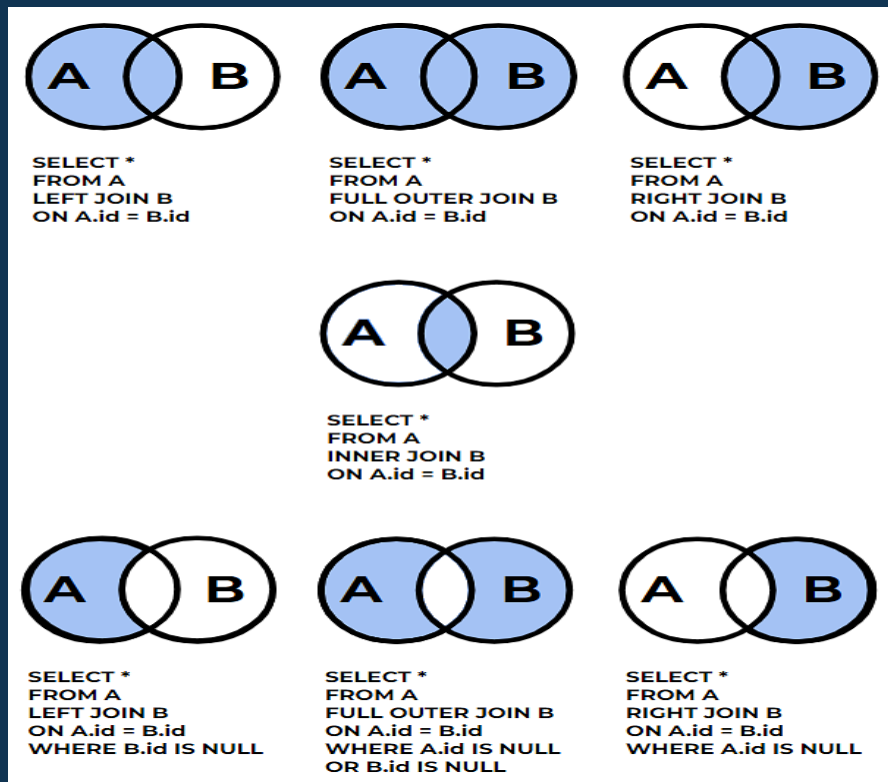
# SQL: Aggregate Functions (Used when specifying columns)

- **COUNT** : Number of rows with non-null values for a given column
- **MIN** : Returns minimum value for a given column
- **MAX** : Returns maximum value for a given column
- **SUM** : Returns the sum of a given column
- **AVG** : Returns the average of a given column
- **COUNT** Example:

```
SELECT COUNT(student_id) FROM students;
```

# SQL: Accessing Multiple Tables

- **INNER JOIN** - Records match in both tables
- **LEFT JOIN** - All values in A, and matching values in B
- **FULL OUTER JOIN** - All values in both tables



# SQL: Removing Tables

- The **DROP TABLE** statement is used to remove every trace of a table in a database.

```
DROP TABLE table_name;
```

- **DROP TABLE** Example:

```
DROP TABLE customers;
```

# SQL vs SQLite

- SQLite is a form of SQL.
- SQL can be expressed in many ways:
  - SQLite
  - MySQL
  - PostgreSQL
- Like “UK” English vs. “US” English

# SQLite

- Native to Python (Yay! No pip installations!)
- Self-Contained
- Easy to port (Moving Database files)
- Serverless
- Doesn't require client-server architecture. Works directly with files.
- Transactional
- Atomic, Consistent, Isolated and Durable (ACID).
- Ensures data integrity.

# SQLite Syntax

```
import sqlite3

db = sqlite3.connect('data/student_db')
cursor = db.cursor()

cursor.execute("""
    CREATE TABLE student(id INTEGER PRIMARY KEY, name TEXT,
                          grade INTEGER)
""")

db.commit()
```

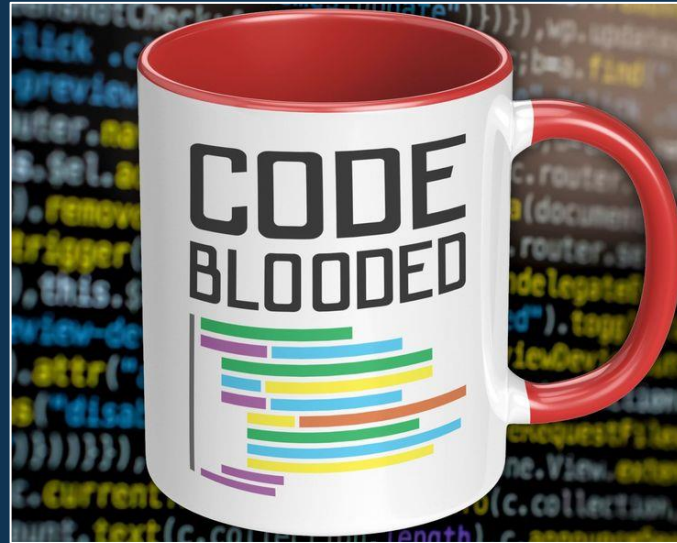


# Basic SQLite Syntax

```
cursor.execute("INSERT INTO student(name, grade)  
              VALUES(?,?)", (name1, grade1))  
db.commit()
```

```
students_ = [(name1, grade1), (name2, grade2), (name3, grade3)]  
cursor.executemany("INSERT INTO student(name, grade) VALUES(?,?)",  
students_)  
db.commit()
```

# Let's get coding!



Hyperiondev

# Q & A Section

Please use this time to ask any questions relating to the topic explained, should you have any.

# Polls

1. How can Python be used to interact with SQL databases?
  - A. By executing SQL queries and fetching data
  - B. By managing database server configurations
  - C. By generating HTML reports from database tables
  - D. By visualising database schemas

# Polls

2. What is the purpose of the sqlite library in Python?
  - A. For web development
  - B. For database connectivity
  - C. For data visualisation
  - D. For machine learning tasks

# Summary

- **Database connectivity** refers to an application's ability to **connect** and **interact** with a database management system (DBMS) to perform various operations such as data retrieval, insertion, update, and deletion.
- Python provides several **libraries** and **frameworks** for database connectivity, including built-in modules like **sqlite3** for SQLite databases and third-party libraries like **SQLAlchemy** for interacting with various SQL databases.

# Summary

- **SQL** (Structured Query Language) is a standard language for interacting with relational databases.
- SQL queries are **used to perform operations on databases**, such as querying data, inserting new records, updating existing records, and deleting records.
- Common SQL operations include **SELECT** (retrieve data), **INSERT** (add new data), **UPDATE** (modify existing data), and **DELETE** (remove data).

# Summary

- Python **integrates seamlessly** with databases through libraries and modules that provide APIs for executing SQL queries and interacting with databases.
- Libraries like **sqlite3** provide a **simple and direct interface** for interacting with SQLite databases, while ORM libraries like **SQLAlchemy** offer **higher-level abstractions and object-oriented approaches** for database interaction.





Hyperiondev

# Thank you for joining us

Take regular breaks.  
Stay hydrated.  
Avoid prolonged screen time.  
Remember to have fun :)