HyperionDev

# SQLite

## Task

Visit our website

# Introduction

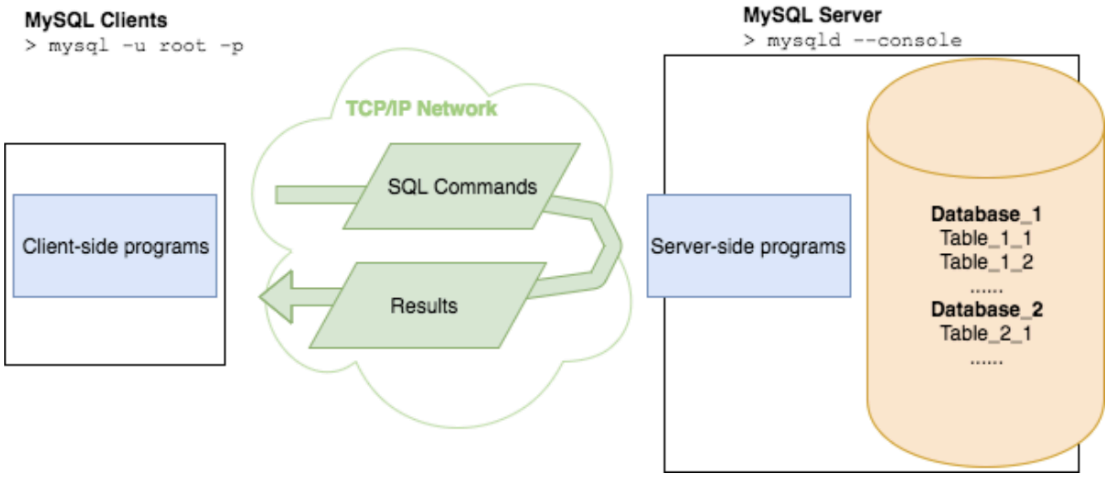
In this task, you will learn how to write code to create and manipulate a database with SQLite. SQLite is built into Python to provide a simple relational database management system (RDBMS). It is easy to set up, fast, and lightweight, which is why it's referred to as "lite".

# SQLite features

Important features to note about SQLite are that it is self-contained, serverless, transactional, and requires zero configuration to run. Let's look more closely at these properties.

- **Self-contained:** This means that SQLite does not need much support from the operating system or external libraries. This makes it suitable for use in embedded devices like mobile phones, iPods, and game devices that lack the infrastructure provided by a regular computer. In Python, you can access SQLite databases using the `sqlite3` module, which is part of Python's standard library. This means you do not need to manage SQLite source code files like **sqlite3.c** or **sqlite3.h** yourself. Instead, simply import the `sqlite3` module to interact with SQLite databases in your Python projects.

- **Serverless:** In most cases, RDBMSs require a separate server to receive and respond to requests sent from the client, as shown in the diagram below.




*SQLite as a serverless RDBMS (SQLite, n.d.)*

Such systems include MySQL, MariaDB, and the Java Database Connectivity (JDBC). These clients have to use the TCP/IP protocol to send and receive

responses. This is referred to as the client/server architecture. SQLite does not make use of a separate server and, therefore, does not utilise client/server architecture. Instead, the entire SQLite database is embedded into the application that needs to access the database.

- **Transactional:** All transactions in SQLite are atomic, consistent, isolated, and durable (**ACID**-compliant). In other words, if a transaction occurs that attempts to make changes to the databases, the changes will either be made in all the appropriate places (in all linked tables and affected rows) or not at all. This is to ensure data integrity (i.e., to avoid conflicting records in different places due to some being updated and others not).

- **Zero configuration required:** You don't need to install SQLite prior to using it in an application or system. This is because of the previously described serverless characteristic.

# Python's SQLite module

It is easy to create and manipulate databases with Python. To enable the use of SQLite with Python, the Python standard library includes a module called `sqlite3`. To use this module, we need to add an `import` statement to our Python script:

```
import sqlite3
```

We can then use the function `sqlite3.connect()` to connect to the database. We pass the name of the database file to this function to open or create the database.

```
# Creates or opens a file called student_db with an SQLite3 DB
db = sqlite3.connect("student_db.db")
```

## Creating and deleting tables

To make any changes to the database, we need a **cursor object**, which is an object that is used to execute SQL statements. Next, we use `.commit()` to save changes to the database. It is important to remember to commit changes since this ensures the atomicity of the database. If you close the connection using `close()` or the connection to the file is lost, changes that have not been committed will be lost.

Below we create a student table with `id`, `name`, and `grade` columns:

```
cursor = db.cursor()  # Get a cursor object
```

```
# Execute a SQL command to create the student table
cursor.execute('''
    CREATE TABLE student(
        id INTEGER PRIMARY KEY,
        name TEXT,
        grade INTEGER
    )
''')

# Commit the changes to the database to ensure they are saved
db.commit()
```

In the above code snippet, a cursor object is obtained from the database connection (`db`). Subsequently, an SQL query is executed using the cursor to create a new table named `student` with columns named `id` (as the primary key), `name`, and `grade`. The `db.commit()` statement is used to commit the transaction, finalising the table creation in the database.

## Using IF NOT EXISTS

When working with databases and creating tables, it is often necessary to ensure that a table does not already exist before attempting to create the table. The `IF NOT EXISTS` clause can be used to create a table only if it does not already exist, which helps prevent errors that might occur if the table is already present in the database.

Let's have a look at an example demonstrating how the `CREATE TABLE` statement can be modified to include the `IF NOT EXISTS` clause, ensuring the table is created only if it does not already exist.

```
# Get the cursor object
cursor = db.cursor()

# Create the student table if it does not exist
cursor.execute('''
    CREATE TABLE IF NOT EXISTS student (
        id INTEGER PRIMARY KEY,
        name TEXT,
        grade INTEGER
    )
''')

# Commit the changes to the database
db.commit()
```

Always remember that the `commit()` function is invoked on the `db` object, not the `cursor` object. If we type `cursor.commit()`, we will get the following error message:

```
AttributeError: 'sqlite3.Cursor' object has no attribute 'commit'
```

## Inserting into the database

To insert data into a database we use prepared statements. This is a secure method for executing SQL queries with Python. Prepared statements involve using placeholders, such as "?", in your SQL query instead of directly including data. This approach ensures that your data is handled safely and efficiently. Avoid using string operations or concatenation to construct SQL queries, as these methods can be less secure.

In this example, we are going to insert two students into the database, whose information is stored in Python variables.

```
name1 = 'Andres'
grade1 = 60

name2 = 'John'
grade2 = 90

# Insert student 1
cursor.execute('''
    INSERT INTO student(name, grade)
    VALUES (?, ?)
''', (name1, grade1))

print('First user inserted')

# Insert student 2
cursor.execute('''
    INSERT INTO student(name, grade)
    VALUES (?, ?)
''', (name2, grade2))

print('Second user inserted')

db.commit()
```

In the example above, the values of the Python variables are passed inside a **tuple**. You could also use a dictionary with the named style placeholder:

```
name3 = 'Sheila'
grade3 = 40

# Insert student 3 using named parameters
cursor.execute('''
    INSERT INTO student (name, grade)
    VALUES (:name, :grade)
''', {'name': name3, 'grade': grade3})

print('Third user inserted')
```

If you need to insert several users, use `executemany` and a list with the tuples:

```
students_grades = [(name1, grade1), (name2, grade2), (name3, grade3)]

cursor.executemany(
    '''INSERT INTO student(name, grade) VALUES(?, ?)''', students_grades
)

db.commit()
```

Each record inserted into the table gets a unique `id` value starting at one and ascending in increments of one for each new record. If you need to get the `id` of the row you just inserted, use `lastrowid`:

```
# Get the ID of the last inserted row
last_row_id = cursor.lastrowid
print(f'Last row ID: {last_row_id}')
```

Use `rollback()` to roll back any change to the database since the last call to commit:

```
cursor.execute('''UPDATE student SET grade = ? WHERE id = ?''', (65, 2))

db.rollback()
```

## Retrieving data

To retrieve data, execute a `SELECT` SQL statement against the `cursor` object, and then use `fetchone()` to retrieve a single row or `fetchall()` to retrieve all the rows.

Here is an example using `fetchone()` to retrieve the first student record that matches the specified ID:

```
# Define the ID of the student we want to retrieve
id = 3

# Execute a query to select the name and grade of the student with the
specified ID
cursor.execute('''SELECT name, grade FROM student WHERE id = ?''', (id,))

# Fetch the first row that matches the query
student = cursor.fetchone()

# Print the retrieved student's name and grade
print(student)
```

**Note:** When using a single variable in a query with placeholders, ensure you include a comma to create a single-element tuple, for example "`(id,)`". This is necessary for the query to work correctly.

If you use `fetchone()` and there are multiple rows that match the criteria, only the first one will be retrieved. If you expect multiple rows, rather use `fetchall()`.

To retrieve all student records where the grade is below a specified threshold, we can use `fetchall()` as shown below:

```
# Define the grade threshold
grade_threshold = 80

# Execute a query to select all students with grades less than the specified
threshold
cursor.execute('SELECT name, grade FROM student WHERE grade < ?',
(grade_threshold,))

# Fetch all rows that match the query
students = cursor.fetchall()

# Print each student's name and grade
print(f'Students with a grade less than {grade_threshold}:')
for student in students:
    print(f'{student[0]} : {student[1]}')
```

In the above example, the code selects all student names and grades from the table where each student's grade is less than a specified threshold. This query returns a result set, which is a list of tuples, with each tuple containing a pair of student names and their corresponding grades. By looping through this list, you can use indexing

(`student[0]` to access the name and `student[1]` to access the grade) to retrieve and display the name and grade for each student.

Alternatively, you can use the `cursor` object, which works as an iterator, invoking `fetchall()` automatically:

```
cursor.execute('''SELECT name, grade FROM student''')

# Iterate over the result set returned by the query
for row in cursor:
    # Each 'row' is a tuple where row[0] is the student's name and row[1]
    # is their grade.
    # Print the student's name and grade in a formatted string
    print(f'{row[0]} : {row[1]}')
```

## Updating and deleting data

Updating or deleting data is similar to inserting data:

```
# Update user with id 1
grade = 100
user_id = 1
cursor.execute('''UPDATE student SET grade = ? WHERE id = ?''', (grade,
user_id))

# Delete user with id 2
user_id = 2
cursor.execute('''DELETE FROM student WHERE id = ?''', (user_id,))

# Drop the student table
cursor.execute('''DROP TABLE student''')

# Commit the changes
db.commit()
```

When we are done working with the `db`, we need to close the connection. Failing to close the connection could result in issues such as incomplete transactions, data corruption, and resource leaks.

```
db.close()
```

# SQLite database exceptions

It is very common for exceptions to occur when working with databases, so it is important to handle these exceptions in your code.

In the example below, we use a `try/except/finally` clause to catch any exception in the code. We put in the code that we would like to execute, but that may throw an exception (or cause an error) in the `try` block. Within the `except` block, we write the code that will be executed if an exception does occur. If no exception is thrown, the `except` block will be ignored. The `finally` clause will always be executed, whether an exception was thrown or not. When working with databases, the `finally` clause is very important, because it always closes the database connection correctly. View this resource to find out more **about exceptions**.

```
import sqlite3

try:
    # Creates or opens a file called student_db with an SQLite3 DB
    db = sqlite3.connect('student_db.db')
    # Get a cursor object
    cursor = db.cursor()
    # Checks if the table "users" exists and if not creates it
    cursor.execute(
        '''
        CREATE TABLE IF NOT EXISTS
        users(id INTEGER PRIMARY KEY, name TEXT, grade INTEGER)
        '''
    )
    # Commit the change
    db.commit()
# Catch the exception
except Exception as e:
    # Roll back any change if something goes wrong
    db.rollback()
    raise e
finally:
    # Close the db connection
    db.close()
```

Notice that the `except` block of our `try/except/finally` clause in the example above will be executed if any type of error occurs:

```
 # Catch the exception
except Exception as e:
    raise e
```

This is called a catch-all clause. In a real application, you should catch a specific exception. To see what type of exceptions could occur, see here for **database API v2.0 exceptions**.

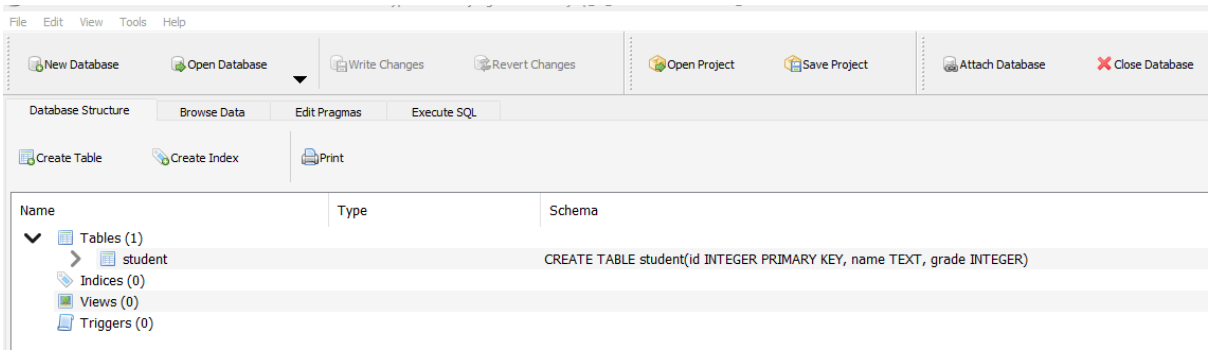
# DB Browser for SQLite

DB Browser for SQLite is a free, open-source tool that allows you to browse and manage databases created with SQLite interactively. With DB Browser, you can easily view, edit, and manipulate the data stored in your SQLite databases, as well as create new tables, indices, and relationships between them. It's a great way to explore and understand the structure and content of your databases, and it's available for Windows, macOS, and Linux platforms from **this website**.

## Visualising our `student_db` database

Now, we want to visualise the `student_db.db` database we created in the above SQLite operations. To do this, we are going to import the `student_db.db` database as follows:

1. Open the DB Browser for SQLite.
2. To import a database, click on “File” in the menu bar, and then select “Open Database...” from the drop-down menu.
3. In the “Open File” dialogue box, navigate to the location where your database file is saved (it should have a .db extension) and select it.
4. Click “Open” to import the database into DB Browser for SQLite.
5. The “Database Structure” tab should then become visible, as seen below:




Once the database is imported, you can start exploring it by clicking on the different tabs in the main window.

- The “Database Structure” tab displays all the tables in the database, along with their columns and data types.

- The “Browse Data” tab shows the actual data stored in each table.
- The “Execute SQL” tab allows you to execute SQL queries against the database.

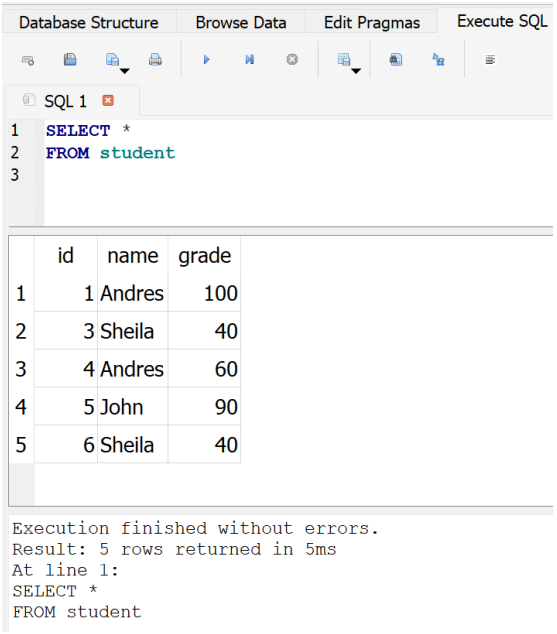
## Running SQL queries

If you navigate to the “Execute SQL” tab, you will be presented with a three-paned window and a small toolbar. The initial pane is labelled “SQL 1”. This is where we will type our queries as follows:

1. In the “SQL 1” pane, input the following:

```
SELECT * FROM student
```

2. In the small toolbar, click the “Run/Play” button.
3. The query should execute and present the student table from the `student_db` database.




As you can see, DB Browser for SQLite stands out as a valuable tool for efficiently managing SQLite databases. Whether you're exploring existing databases or creating new structures, this free and open-source application provides a user-friendly interface across various operating systems. With its versatility and accessibility, DB Browser for SQLite proves to be an indispensable companion for database exploration and manipulation.

**Take note**

Read and run the accompanying example files provided before doing the task to become more comfortable with the concepts covered in this task.

# Practical task

Follow these steps:

- Create a Python file called **database_manip.py**. Write the code to do the following tasks:
    - Create a table called `python_programming`.
    - Insert the following new rows into the `python_programming` table:

| id | name | grade |
|---|---|---|
| 55 | Carl Davis | 61 |
| 66 | Dennis Fredrickson | 88 |
| 77 | Jane Richards | 78 |
| 12 | Peyton Sawyer | 45 |
| 2 | Lucas Brooke | 99 |

    - Select all records with a `grade` between 60 and 80.
    - Change Carl Davis's `grade` to 65.
    - Delete Dennis Fredrickson's row.
    - Change the `grade` of all students with an `id` greater than 55 to 80.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

## Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Do you think we've done a good job or do you think the content of this task, or this course as a whole, can be improved?

Share your thoughts anonymously using this **form**.

# Reference list

SQLite. (n.d.). *SQLite is serverless*. **https://www.sqlite.org/serverless.html**