



TASK

SQLite

[Visit our website](#)

Introduction

WELCOME TO THE SQLITE TASK!

In this task, you will learn how to write code to create and manipulate a database with SQLite.

WHAT IS SQLITE?

SQLite is built in Python to provide a simple relational database management system (RDBMS). It is very easy to set up, as well as exceptionally fast and lightweight (thus 'lite'). Important features to note about SQLite are that it is self-contained, serverless, and 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. The source code is found in files called `sqlite3.c` and the header file in `sqlite3.h`. When you want to use SQLite in an application, ensure that you have these files in your project directory when compiling your code.
- **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.

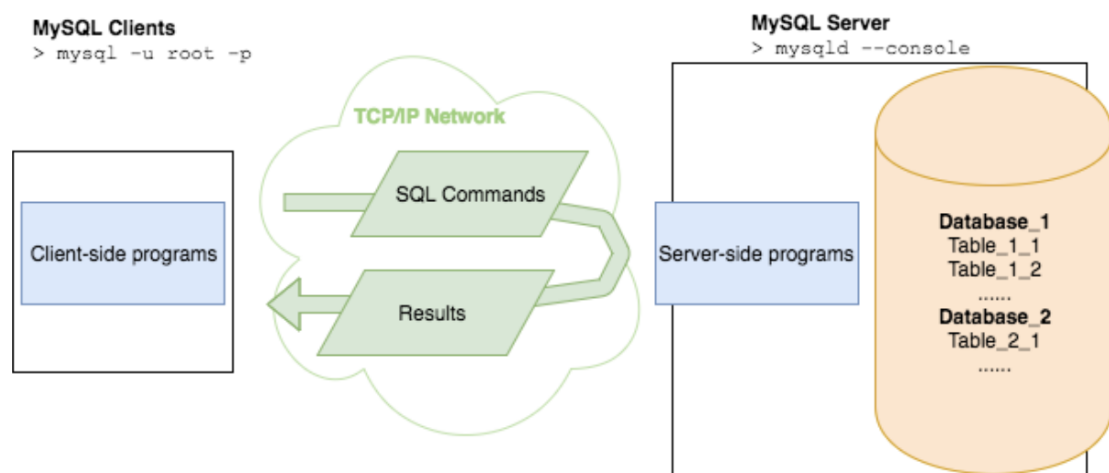


Image source: [SQLite Is Severless](#)

Such systems include MySQL and the Java Database Client (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. 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 serverless characteristic described previously.

PYTHON'S SQLITE MODULE

It is easy to create and manipulate databases with Python. To allow us to use SQLite with Python, the Python Standard Library includes a module called "`sqlite3`". To use the SQLite3 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 a SQLite3 DB  
  
db = sqlite3.connect('data/student_db.db')
```

Creating and Deleting Tables

To make any changes to the database, we need a [cursor object](#). A cursor object is an object that is used to execute SQL statements. Next, `.commit` is used 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

cursor.execute('''
    CREATE TABLE student(id INTEGER PRIMARY KEY, name TEXT,
                           grade INTEGER)
''')
db.commit()
```

In the above code snippet, a cursor object is obtained from the database connection (db). Subsequently, a 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. 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, we use the cursor again to execute a SQL statement. When using data stored in Python variables to insert data into a database table, use the "?" placeholder. It is not secure to use **string operations** or **concatenation** to make your queries.

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

cursor.execute('''INSERT INTO student(name, grade)
VALUES(:name,:grade)''' ,
{'name':name3, 'grade':grade3})

```

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 1 and ascending in increments of 1 for each new record. If you need to get the **ID** of the row you just inserted, use **lastrowid**:

```

id = cursor.lastrowid
print('Last row id: %d' % 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.

```
id = 3
cursor.execute(''SELECT name, grade FROM student WHERE id=?'', (id,))
student = cursor.fetchone()

print(student)
```

The cursor object works as an iterator, invoking **fetchall()** automatically:

```
cursor.execute(''SELECT name, grade FROM student'')
for row in cursor:
    # row[0] returns the first column in the query (name), row[1] returns
    the grade column.
    print('{0} : {1}'.format(row[0], row[1]))
```

Updating and Deleting Data

Updating or deleting data is similar to inserting data:

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

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

cursor.execute(''DROP TABLE student'')

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. 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 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. Find out more about exceptions [here](#).

```
try:
    # Creates or opens a file called student_db with a 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 [DB-API 2.0 Exceptions](#).

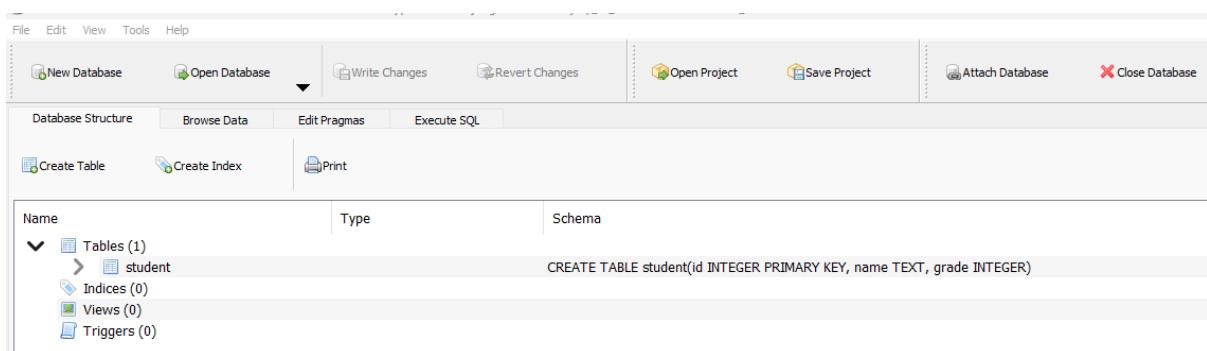
DB Browser for SQLite

DB Browser for SQLite is a free, open-source tool that allows you to interactively browse and manage databases created with SQLite, a popular database management system. 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 [here](#).

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** database as follows:

1. Open the DB Browser for SQLite.
2. To import a database, click on "File" in the menu bar and 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.

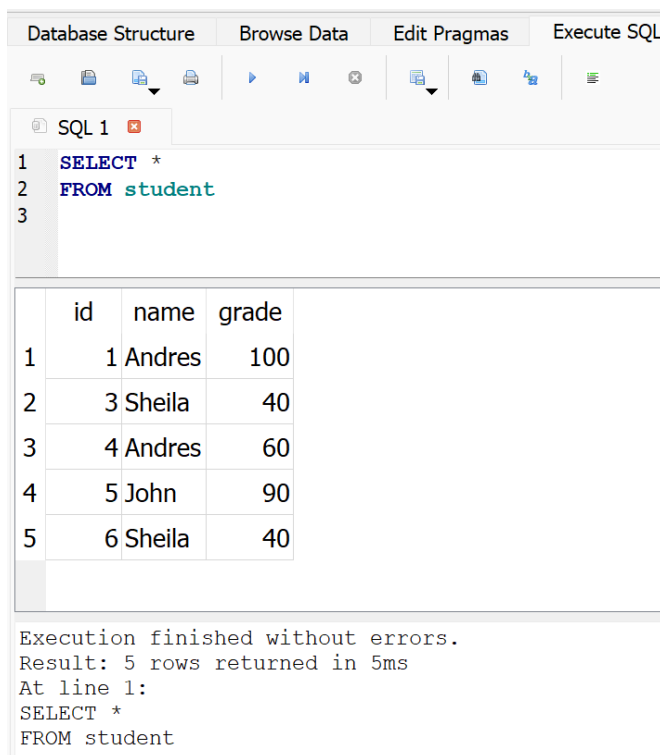
Running SQL Queries

If you navigate to the “Execute SQL” tab you will be presented with a three-paneled 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.



The screenshot shows the 'Execute SQL' tab in the DB Browser for SQLite. The 'SQL 1' pane contains the query: `SELECT *`
`FROM student`. Below the query editor, the results are displayed in a table with 5 rows and 3 columns: `id`, `name`, and `grade`. The results are as follows:

	id	name	grade
1	1	Andres	100
2	3	Sheila	40
3	4	Andres	60
4	5	John	90
5	6	Sheila	40

At the bottom of the window, the execution status is shown: 'Execution finished without errors. Result: 5 rows returned in 5ms. At line 1: SELECT * FROM student'.

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.

Instructions

To become more comfortable with the concepts covered in this section of the learning material, read and run the accompanying example files provided before doing the compulsory task.

Compulsory Task 1

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 a grade of 80.
- Include a screenshot in your submission showing the **python_programming** table contents in the DB Browser for SQLite.



Rate us

Share your thoughts

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

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

[Click here](#) to share your thoughts anonymously.



REFERENCES:

Python Central. (2013, April 11). Introduction to SQLite in Python. Retrieved April 15, 2019, from Python Central:

<https://www.pythoncentral.io/introduction-to-sqlite-in-python/>