https://pynative.com/python-sqlite/

# Python SQLite tutorial using sqlite3

This Python SQLite tutorial aims to demonstrate how to develop Python database applications with the SQLite database. You will learn how to perform SQLite database operations from Python.

As you all know, SQLite is a C-language library that implements a SQL database engine that is relatively quick, serverless, and self-contained, high-reliable. SQLite is the most commonly used database engine in the test environment (Refer to SQLite Home page).

SQLite comes built-in with most computers and mobile devices, and browsers. Python's official sqlite3 module helps us to work with the SQLite database.

Python sqlite3 module adheres to Python Database API Specification v2.0 (PEP 249). PEP 249 provides a SQL interface designed to encourage and maintain the similarity between the Python modules used to access databases.

## Python SQLite Database Connection

This section lets you know how to connect to SQLite database in Python using the sqlite3 module.

Use the following steps to connect to SQLite

How to Connect to SQLite Database in Python

1. **Import sqlite3 module**

   `import sqlite3` statement imports the sqlite3 module in the program. Using the classes and methods defined in the sqlite3 module we can communicate with the SQLite database.

2. **Use the connect() method**

   Use the `connect()` method of the `connector` class with the database name. To establish a connection to SQLite, you need to pass the database name you want to connect. If you specify the database file name that already presents on the disk, it will connect to it. But

if your specified SQLite database file doesn't exist, SQLite creates a new database for you.
This method returns the SQLite Connection Object if the connection is successful.

3. **Use the cursor() method**

Use the `cursor()` method of a connection class to create a cursor object to execute SQLite command/queries from Python.

4. **Use the execute() method**

The execute() methods run the SQL query and return the result.

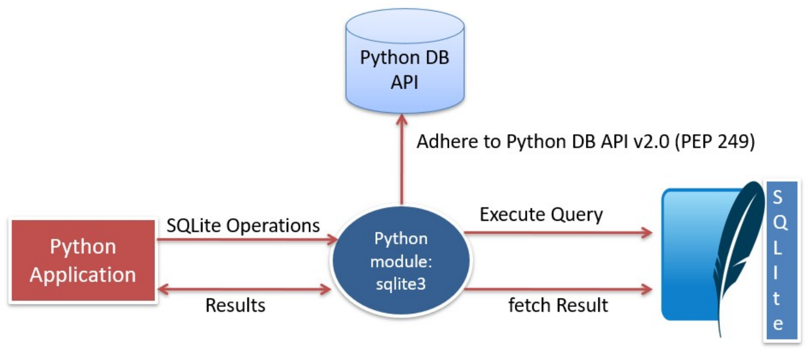
5. **Extract result using fetchall()**

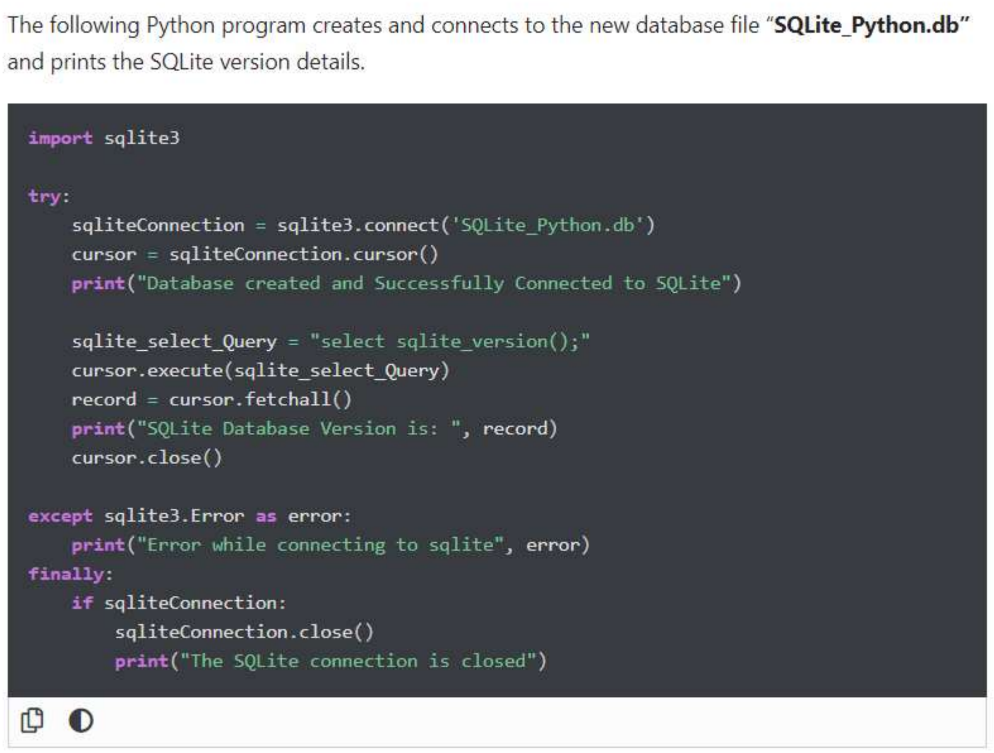
Use `cursor.fetchall()` or `fetchone()` or `fetchmany()` to read query result.

6. **Close cursor and connection objects**

use `cursor.clsoe()` and `connection.clsoe()` method to close the cursor and SQLite connections after your work completes.

7.**Catch database exception if any that may occur during this connection process.**




The following Python program creates and connects to the new database file “**SQLite_Python.db**” and prints the SQLite version details.

```python
import sqlite3

try:
    sqliteConnection = sqlite3.connect('SQLite_Python.db')
    cursor = sqliteConnection.cursor()
    print("Database created and Successfully Connected to SQLite")

    sqlite_select_Query = "select sqlite_version();"
    cursor.execute(sqlite_select_Query)
    record = cursor.fetchall()
    print("SQLite Database Version is: ", record)
    cursor.close()

except sqlite3.Error as error:
    print("Error while connecting to sqlite", error)
finally:
    if sqliteConnection:
        sqliteConnection.close()
        print("The SQLite connection is closed")
```

## Important points while connecting to SQLite

- The connection object is not thread-safe. The sqlite3 module doesn’t allow sharing connections between threads. If you still try to do so, you will get an exception at runtime.
- The `connect()` method accepts various arguments. In our example, we passed the database name argument to connect.
- Using a connection object, we can create a cursor object which allows us to execute SQLite command/queries through Python.
- We can create as many cursors as we want from a single connection object. Like a connection object, this cursor object is also not thread-safe. The sqlite3 module doesn’t allow sharing cursors between threads. If you still try to do so, you will get an exception at runtime.
- `try-except-finally block`: We placed all our code in this block to catch the SQLite database exceptions and errors during this process.

- Using the `Error` class, we can handle any database error and exception that may occur while working with SQLite from Python.
- The `Error` class helps us to understand the error in detail. It returns an error message and error code.
- It is always good practice to close the cursor and connection object once your work gets completed to avoid database issues.

## Create SQLite table from Python

This section will learn how to create a table in the SQLite database from Python. Create a table statement is a DDL query. Let see how to execute it from Python.

In this example, we are creating a `SqliteDb_developers` table inside the `SQLite_Python.db` database.

Steps for create aa table in SQLite from Python: –

- Connect to SQLite using a `sqlite3.connect()`.
- Prepare a create table query.
- Execute the query using a `cursor.execute(query)`

```python
import sqlite3

try:
    sqliteConnection = sqlite3.connect('SQLite_Python.db')
    sqlite_create_table_query = '''CREATE TABLE SqliteDb_developers (
                                id INTEGER PRIMARY KEY,
                                name TEXT NOT NULL,
                                email text NOT NULL UNIQUE,
                                joining_date datetime,
                                salary REAL NOT NULL);'''

    cursor = sqliteConnection.cursor()
    print("Successfully Connected to SQLite")
    cursor.execute(sqlite_create_table_query)
    sqliteConnection.commit()
    print("SQLite table created")

    cursor.close()

except sqlite3.Error as error:
    print("Error while creating a sqlite table", error)
finally:
    if sqliteConnection:
        sqliteConnection.close()
        print("sqlite connection is closed")
```

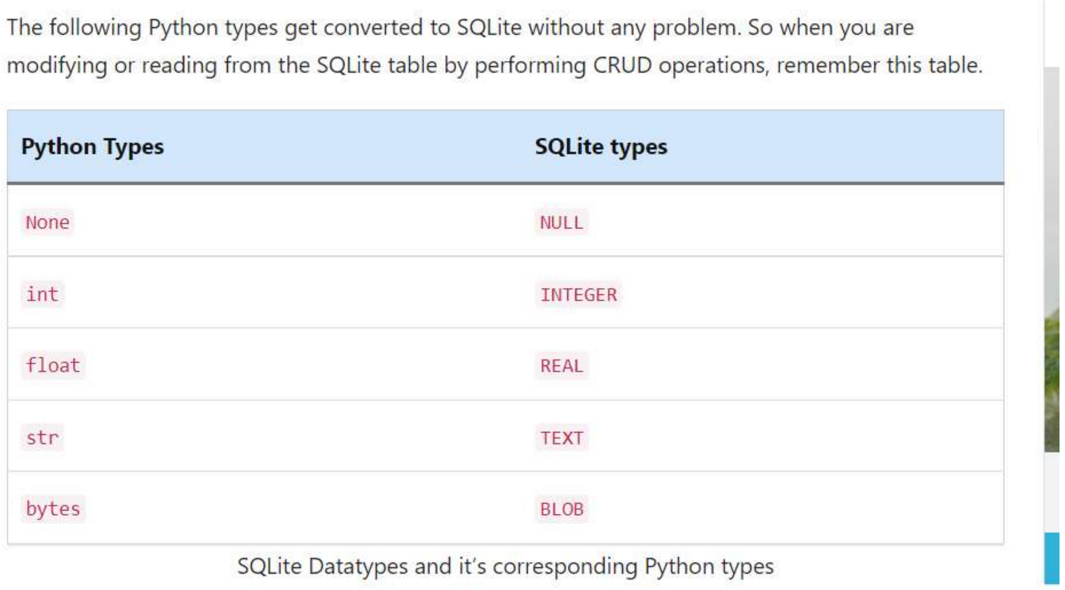
## SQLite Datatypes and it's corresponding Python types

Before executing SQLite CRUD operations from Python, first understand SQLite data type and their corresponding Python types, which will help us store and read data from the SQLite table.

SQLite database engine has multiple storage classes to store values. Every value stored in an SQLite database has one of the following storage classes or data types.

**SQLite DataTypes**:

- **NULL: –** The value is a NULL value.
- **INTEGER**: – To store the numeric value. The integer stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the number.
- **REAL**: – The value is a floating-point value, for example, 3.14 value of PI
- **TEXT**: – The value is a text string, TEXT value stored using the UTF-8, UTF-16BE or UTF-16LE encoding.
- **BLOB**: – The value is a blob of data, i.e., binary data. It is used to store images and files.



The following Python types get converted to SQLite without any problem. So when you are modifying or reading from the SQLite table by performing CRUD operations, remember this table.

| Python Types | SQLite types |
|---|---|
| `None` | `NULL` |
| `int` | `INTEGER` |
| `float` | `REAL` |
| `str` | `TEXT` |
| `bytes` | `BLOB` |

SQLite Datatypes and it's corresponding Python types

# Python SQLite tutorial using sqlite3

Updated on: March 9, 2021 | 20 Comments

This Python SQLite tutorial aims to demonstrate how to develop Python database applications with the SQLite database. You will learn how to perform SQLite database operations from Python.

As you all know, SQLite is a C-language library that implements a SQL database engine that is relatively quick, serverless, and self-contained, high-reliable. SQLite is the most commonly used database engine in the test environment (Refer to SQLite Home page).

SQLite comes built-in with most computers and mobile devices, and browsers. Python's official sqlite3 module helps us to work with the SQLite database.

Python sqlite3 module adheres to Python Database API Specification v2.0 (PEP 249). PEP 249 provides a SQL interface designed to encourage and maintain the similarity between the Python modules used to access databases.

**Let see each section now.**

## Table of contents

# Python SQLite Database Connection

This section lets you know how to connect to SQLite database in Python using the sqlite3 module.

Use the following steps to connect to SQLite

How to Connect to SQLite Database in Python

1. **Import sqlite3 module**

   `import sqlite3` statement imports the sqlite3 module in the program. Using the classes and methods defined in the sqlite3 module we can communicate with the SQLite database.

2. **Use the connect() method**

   Use the `connect()` method of the `connector` class with the database name. To establish a connection to SQLite, you need to pass the database name you want to connect. If you specify the database file name that already presents on the disk, it will connect to it. But if your specified SQLite database file doesn't exist, SQLite creates a new database for you.
   This method returns the SQLite Connection Object if the connection is successful.

3. **Use the cursor() method**

   Use the `cursor()` method of a connection class to create a cursor object to execute SQLite command/queries from Python.

4. **Use the execute() method**

   The execute() methods run the SQL query and return the result.

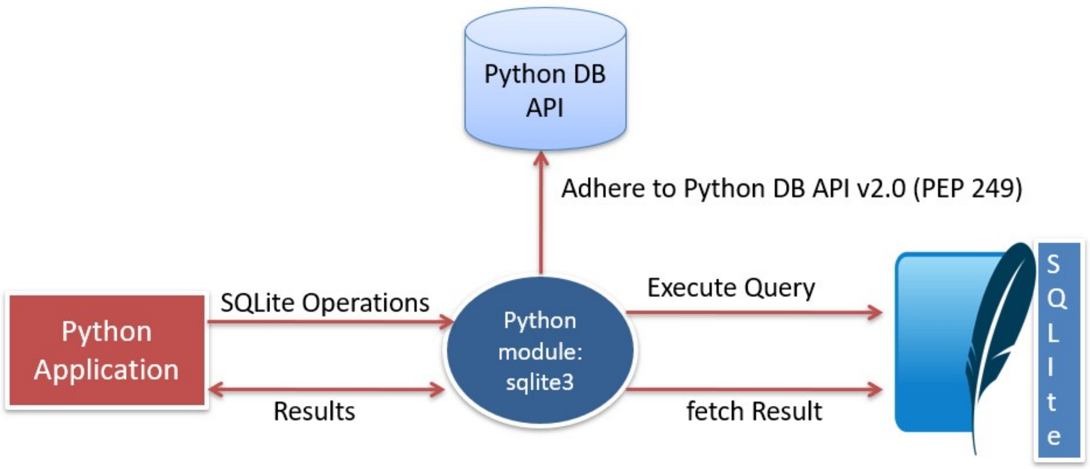
5. **Extract result using fetchall()**

   Use `cursor.fetchall()` or `fetchone()` or `fetchmany()` to read query result.

6. **Close cursor and connection objects**

   use `cursor.clsoe()` and `connection.clsoe()` method to close the cursor and SQLite connections after your work completes

7. **Catch database exception if any that may occur during this connection process.**



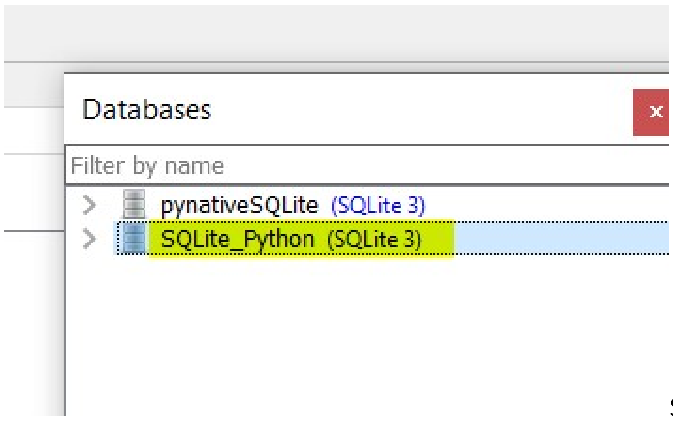
Python sqlite3 module working

The following Python program creates and connects to the new database file "**SQLite_Python.db**" and prints the SQLite version details.

```python
import sqlite3

try:
    sqliteConnection = sqlite3.connect('SQLite_Python.db')
    cursor = sqliteConnection.cursor()
    print("Database created and Successfully Connected to SQLite")

    sqlite_select_Query = "select sqlite_version();"
    cursor.execute(sqlite_select_Query)
    record = cursor.fetchall()
    print("SQLite Database Version is: ", record)
    cursor.close()

except sqlite3.Error as error:
    print("Error while connecting to sqlite", error)
finally:
    if sqliteConnection:
        sqliteConnection.close()
        print("The SQLite connection is closed")
```

**Output**

```
Database created and Successfully Connected to SQLite SQLite Database Version is:
[('3.28.0',)] The SQLite connection is closed
```



SQLite_Python_db

## Important points while connecting to SQLite

- The connection object is not thread-safe. The sqlite3 module doesn't allow sharing connections between threads. If you still try to do so, you will get an exception at runtime.
- The `connect()` method accepts various arguments. In our example, we passed the database name argument to connect.
- Using a connection object, we can create a cursor object which allows us to execute SQLite command/queries through Python.
- We can create as many cursors as we want from a single connection object. Like a connection object, this cursor object is also not thread-safe. The sqlite3 module doesn't allow sharing cursors between threads. If you still try to do so, you will get an exception at runtime.
- `try-except-finally block`: We placed all our code in this block to catch the SQLite database exceptions and errors during this process.
- Using the `Error` class, we can handle any database error and exception that may occur while working with SQLite from Python.
- The `Error` class helps us to understand the error in detail. It returns an error message and error code.
- It is always good practice to close the cursor and connection object once your work gets completed to avoid database issues.

## Create SQLite table from Python

This section will learn how to create a table in the SQLite database from Python. Create a table statement is a DDL query. Let see how to execute it from Python.

In this example, we are creating a `SqliteDb_developers` table inside the `SQLite_Python.db` database.

Steps for create aa table in SQLite from Python: –

- Connect to SQLite using a `sqlite3.connect()`.
- Prepare a create table query.
- Execute the query using a `cursor.execute(query)`

```
import sqlite3

try:
    sqliteConnection = sqlite3.connect('SQLite_Python.db')
    sqlite_create_table_query = '''CREATE TABLE SqliteDb_developers (
                                id INTEGER PRIMARY KEY,
                                name TEXT NOT NULL,
                                email text NOT NULL UNIQUE,
                                joining_date datetime,
                                salary REAL NOT NULL);'''

    cursor = sqliteConnection.cursor()
    print("Successfully Connected to SQLite")
    cursor.execute(sqlite_create_table_query)
    sqliteConnection.commit()
    print("SQLite table created")

    cursor.close()

except sqlite3.Error as error:
    print("Error while creating a sqlite table", error)
finally:
    if sqliteConnection:
        sqliteConnection.close()
        print("sqlite connection is closed")
```

**Output**

```
Successfully Connected to SQLite SQLite table created: the sqlite connection is
```

## SQLite Datatypes and it's corresponding Python types

Before executing SQLite CRUD operations from Python, first understand SQLite data type and their corresponding Python types, which will help us store and read data from the SQLite table.

SQLite database engine has multiple storage classes to store values. Every value stored in an SQLite database has one of the following storage classes or data types.

**SQLite DataTypes**:

- **NULL: –** The value is a NULL value.
- **INTEGER**: – To store the numeric value. The integer stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the number.
- **REAL**: – The value is a floating-point value, for example, 3.14 value of PI
- **TEXT**: – The value is a text string, TEXT value stored using the UTF-8, UTF-16BE or UTF-16LE encoding.
- **BLOB**: – The value is a blob of data, i.e., binary data. It is used to store images and files.

The following Python types get converted to SQLite without any problem. So when you are modifying or reading from the SQLite table by performing CRUD operations, remember this table.

| Python Types | SQLite types |
|---|---|
| `None` | `NULL` |
| `int` | `INTEGER` |
| `float` | `REAL` |
| `str` | `TEXT` |
| `bytes` | `BLOB` |

SQLite Datatypes and it’s corresponding Python types

## Perform SQLite  CRUD Operations from Python

Most of the time, we need to manipulate the SQLite table’s data from Python. To perform these data manipulations, we execute DML queries, i.e., SQLite Insert, Update, Delete operations from Python.

Now, we know the table and its column details, so let’s move to the crud operations.

# Python Insert into SQLite Table

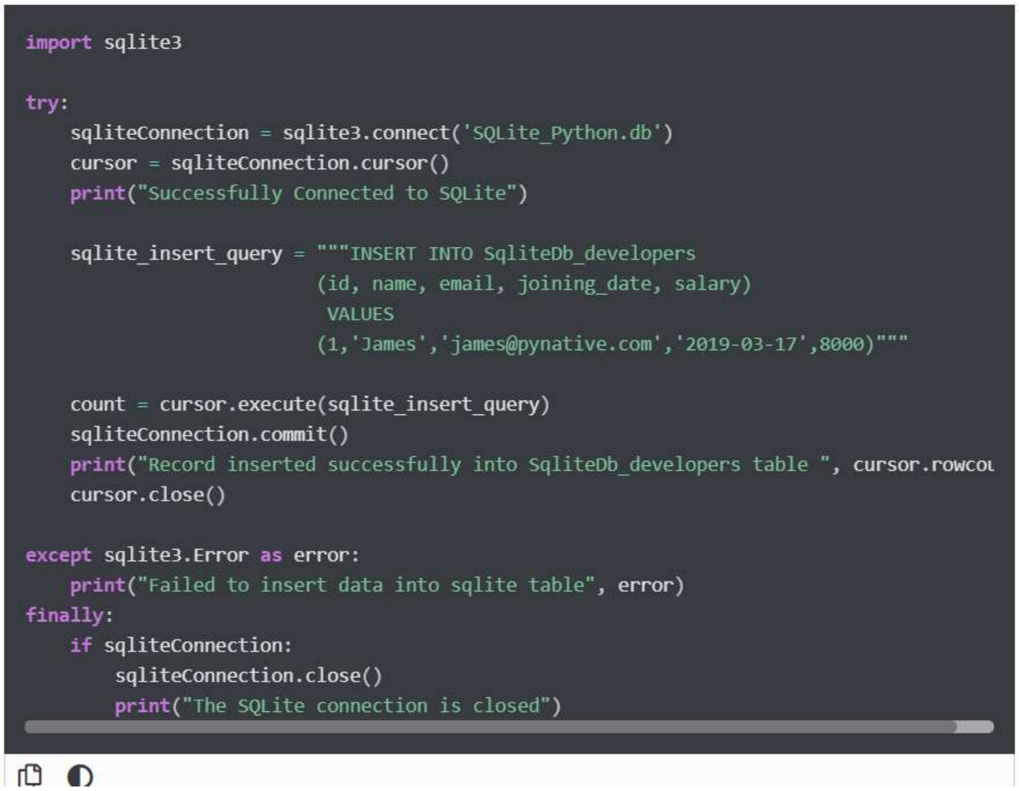
Updated on: March 9, 2021 | 8 Comments

Learn to execute the SQLite INSERT Query from Python to add new rows to the SQLite table using a Python sqlite3 module.

**Goals of this lesson**: –

- Insert single and multiple rows into the SQLite table
- Insert Integer, string, float, double, and `datetime` values into SQLite table
- Use a parameterized query to insert Python variables as dynamic data into a table

**Example**

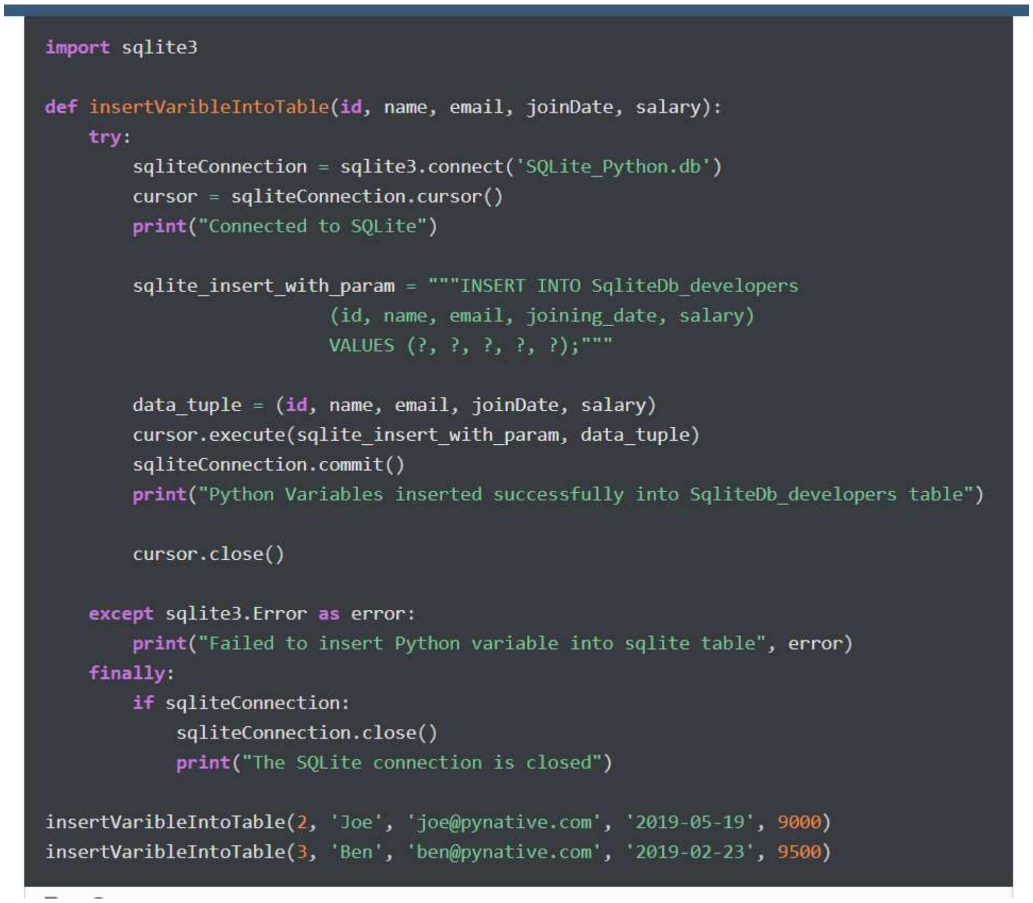


```python
import sqlite3

try:
    sqliteConnection = sqlite3.connect('SQLite_Python.db')
    cursor = sqliteConnection.cursor()
    print("Successfully Connected to SQLite")

    sqlite_insert_query = """INSERT INTO SqliteDb_developers
                          (id, name, email, joining_date, salary)
                           VALUES
                          (1,'James','james@pynative.com','2019-03-17',8000)"""

    count = cursor.execute(sqlite_insert_query)
    sqliteConnection.commit()
    print("Record inserted successfully into SqliteDb_developers table ", cursor.rowcou
    cursor.close()

except sqlite3.Error as error:
    print("Failed to insert data into sqlite table", error)
finally:
    if sqliteConnection:
        sqliteConnection.close()
        print("The SQLite connection is closed")
```

# Using Python variables in SQLite INSERT query

Sometimes we need to insert a Python variable value into a table's column. This value can be anything, including integer, string, float, and DateTime. For example, in the registration form person enter his/her details. You can take those values in Python variables and insert them into the SQLite table.

We use a parameterized query to insert Python variables into the table. Using a parameterized query, we can pass python variables as a query parameter in which placeholders (`?`)

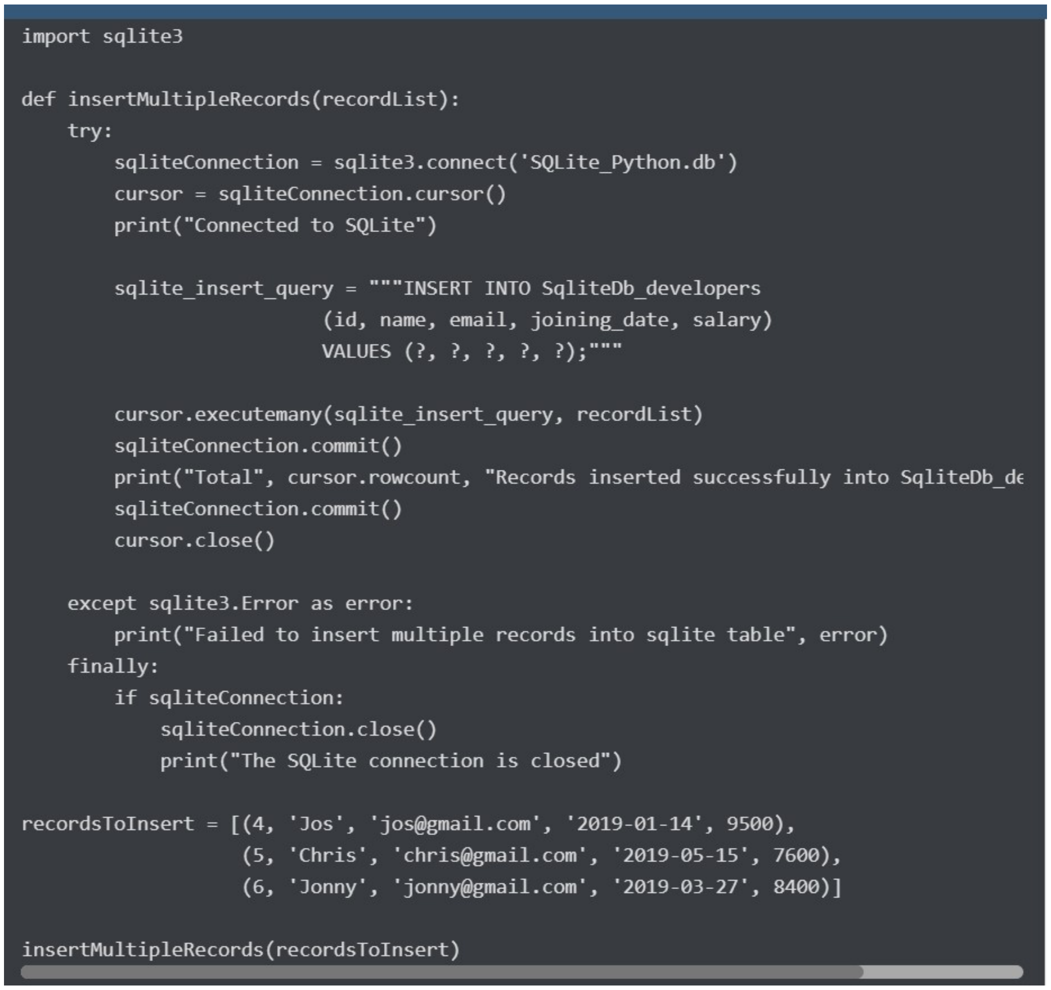


```python
import sqlite3

def insertVaribleIntoTable(id, name, email, joinDate, salary):
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db')
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")

        sqlite_insert_with_param = """INSERT INTO SqliteDb_developers
                          (id, name, email, joining_date, salary)
                          VALUES (?, ?, ?, ?, ?);"""

        data_tuple = (id, name, email, joinDate, salary)
        cursor.execute(sqlite_insert_with_param, data_tuple)
        sqliteConnection.commit()
        print("Python Variables inserted successfully into SqliteDb_developers table")

        cursor.close()

    except sqlite3.Error as error:
        print("Failed to insert Python variable into sqlite table", error)
    finally:
        if sqliteConnection:
            sqliteConnection.close()
            print("The SQLite connection is closed")

insertVaribleIntoTable(2, 'Joe', 'joe@pynative.com', '2019-05-19', 9000)
insertVaribleIntoTable(3, 'Ben', 'ben@pynative.com', '2019-02-23', 9500)
```

## Python Insert multiple rows into SQLite table using the cursor’s `executemany()`

For example, You wanted to add all records from the CSV file into the SQLite table. Instead of executing the INSERT query every time to add each record, you can perform a bulk insert operation in a single query using a cursor’s `executemany()` function.

The `executemany()` method takes two arguments `SQL query` and records to update.



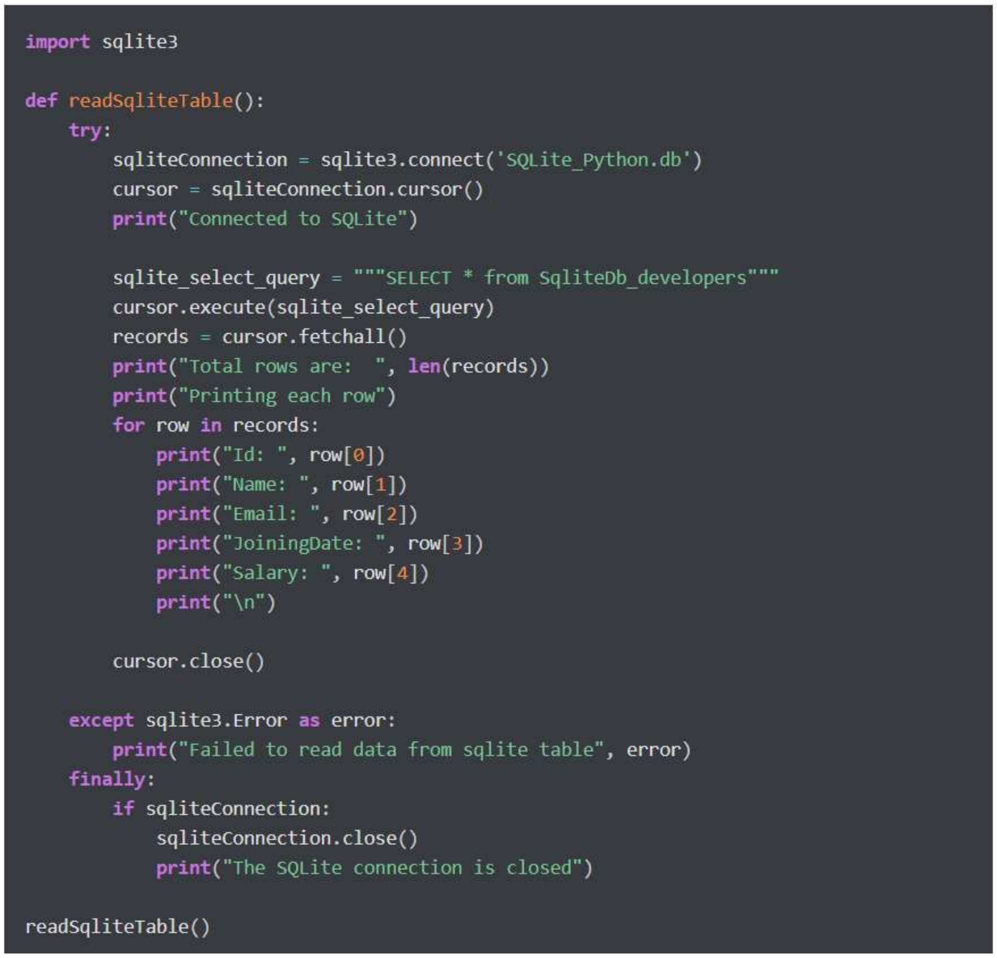
```
import sqlite3

def insertMultipleRecords(recordList):
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db')
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")

        sqlite_insert_query = """INSERT INTO SqliteDb_developers
                          (id, name, email, joining_date, salary)
                          VALUES (?, ?, ?, ?, ?);"""

        cursor.executemany(sqlite_insert_query, recordList)
        sqliteConnection.commit()
        print("Total", cursor.rowcount, "Records inserted successfully into SqliteDb_de
        sqliteConnection.commit()
        cursor.close()

    except sqlite3.Error as error:
        print("Failed to insert multiple records into sqlite table", error)
    finally:
        if sqliteConnection:
            sqliteConnection.close()
            print("The SQLite connection is closed")

recordsToInsert = [(4, 'Jos', 'jos@gmail.com', '2019-01-14', 9500),
                   (5, 'Chris', 'chris@gmail.com', '2019-05-15', 7600),
                   (6, 'Jonny', 'jonny@gmail.com', '2019-03-27', 8400)]

insertMultipleRecords(recordsToInsert)
```

# Python Select from SQLite Table

This lesson demonstrates how to execute SQLite SELECT Query from Python to retrieve rows from the SQLite table using the built-in module sqlite3.

### Goals of this lesson

- Fetch all rows using a `cursor.fetchall()`
- Use `cursor.fetchmany(size)` to fetch limited rows, and fetch only a single row using `cursor.fetchone()`
- Use the Python variables in the SQLite Select query to pass dynamic values.



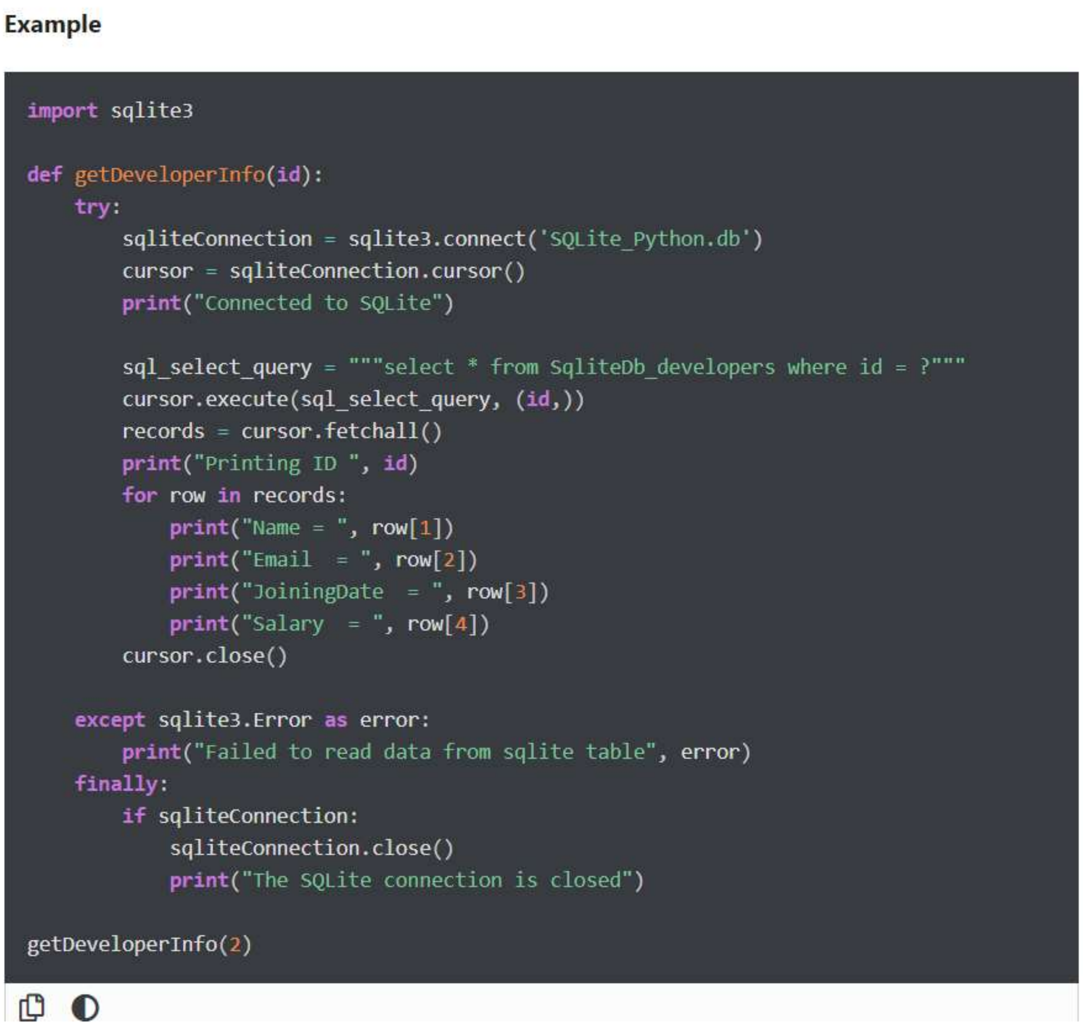
```python
import sqlite3

def readSqliteTable():
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db')
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")

        sqlite_select_query = """SELECT * from SqliteDb_developers"""
        cursor.execute(sqlite_select_query)
        records = cursor.fetchall()
        print("Total rows are:  ", len(records))
        print("Printing each row")
        for row in records:
            print("Id: ", row[0])
            print("Name: ", row[1])
            print("Email: ", row[2])
            print("JoiningDate: ", row[3])
            print("Salary: ", row[4])
            print("\n")

        cursor.close()

    except sqlite3.Error as error:
        print("Failed to read data from sqlite table", error)
    finally:
        if sqliteConnection:
            sqliteConnection.close()
            print("The SQLite connection is closed")

readSqliteTable()
```

## Use Python variables as parameters in SQLite Select Query

We often need to pass a variable to SQLite select query in where clause to check some condition.

Let's say the application wants to fetch person details by giving any id at runtime. To handle such a requirement, we need to use a parameterized query.

A parameterized query is a query in which placeholders (`?`) are used for parameters and the parameter values supplied at execution time.



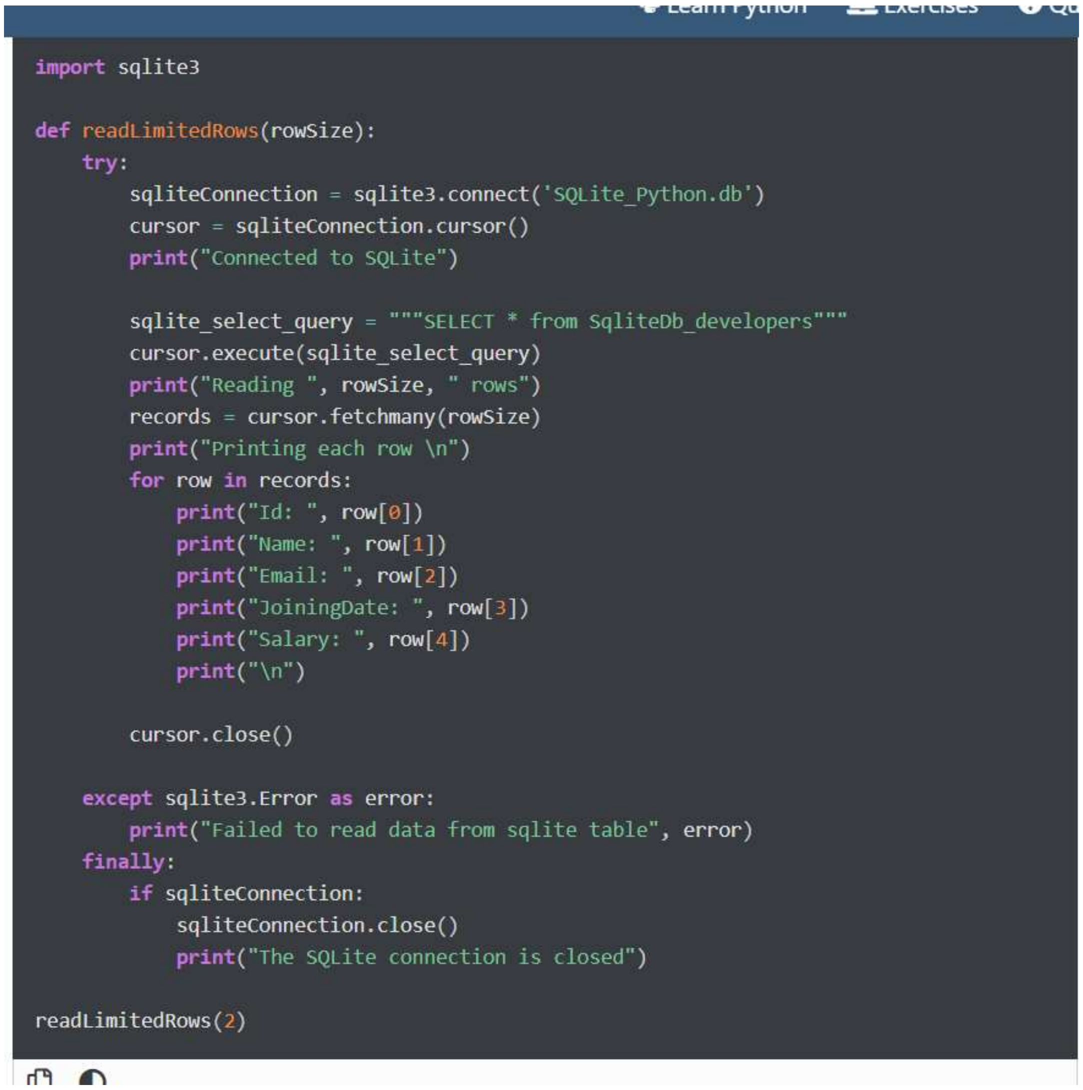
**Example**

```python
import sqlite3

def getDeveloperInfo(id):
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db')
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")

        sql_select_query = """select * from SqliteDb_developers where id = ?"""
        cursor.execute(sql_select_query, (id,))
        records = cursor.fetchall()
        print("Printing ID ", id)
        for row in records:
            print("Name = ", row[1])
            print("Email  = ", row[2])
            print("JoiningDate  = ", row[3])
            print("Salary  = ", row[4])
        cursor.close()

    except sqlite3.Error as error:
        print("Failed to read data from sqlite table", error)
    finally:
        if sqliteConnection:
            sqliteConnection.close()
            print("The SQLite connection is closed")

getDeveloperInfo(2)
```

## Select limited rows from SQLite table using cursor.fetchmany()

In some circumstances, fetching all the data rows from a table is a time-consuming task if a table contains thousands of rows.

To fetch all rows, we have to use more resources, so we need more space and processing time. To enhance performance, use the `fetchmany(SIZE)` method of a cursor class to fetch fewer rows.

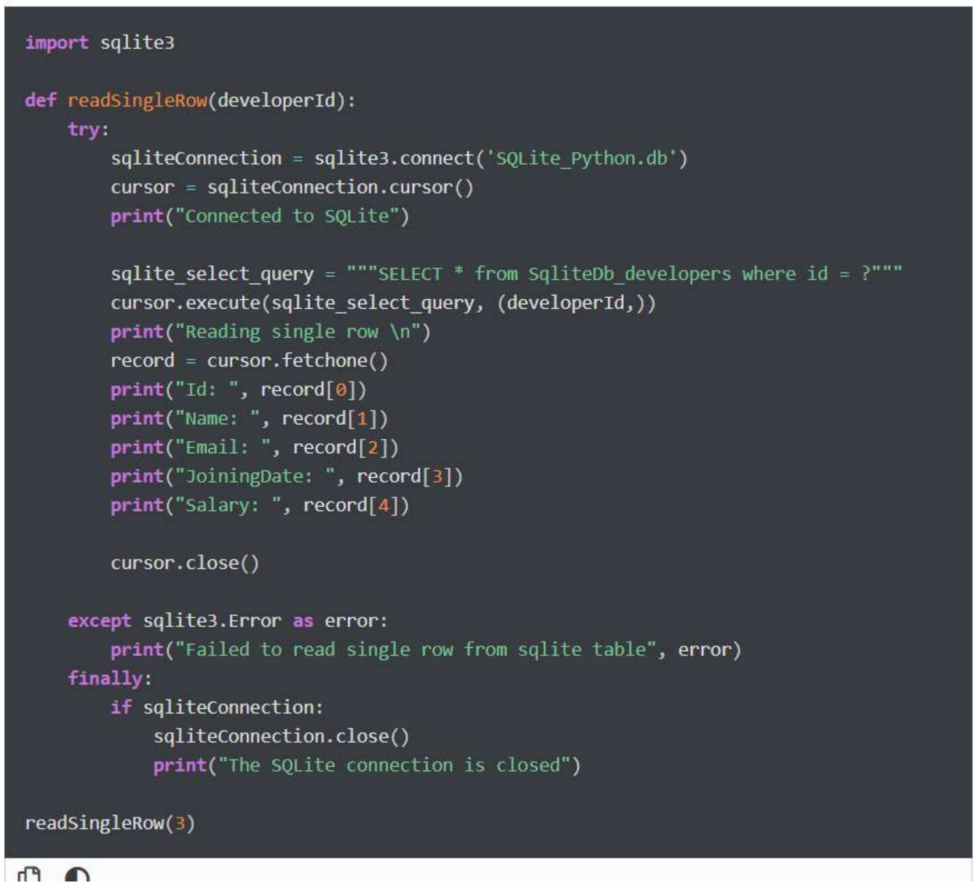


```
import sqlite3

def readLimitedRows(rowSize):
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db')
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")

        sqlite_select_query = """SELECT * from SqliteDb_developers"""
        cursor.execute(sqlite_select_query)
        print("Reading ", rowSize, " rows")
        records = cursor.fetchmany(rowSize)
        print("Printing each row \n")
        for row in records:
            print("Id: ", row[0])
            print("Name: ", row[1])
            print("Email: ", row[2])
            print("JoiningDate: ", row[3])
            print("Salary: ", row[4])
            print("\n")

        cursor.close()

    except sqlite3.Error as error:
        print("Failed to read data from sqlite table", error)
    finally:
        if sqliteConnection:
            sqliteConnection.close()
            print("The SQLite connection is closed")

readLimitedRows(2)
```

## Select a single row from SQLite table

When you want to read only one row from the SQLite table, then you should use fetchone() method of a cursor class. You can also use this method in situations when you know the query is going to return only one row.

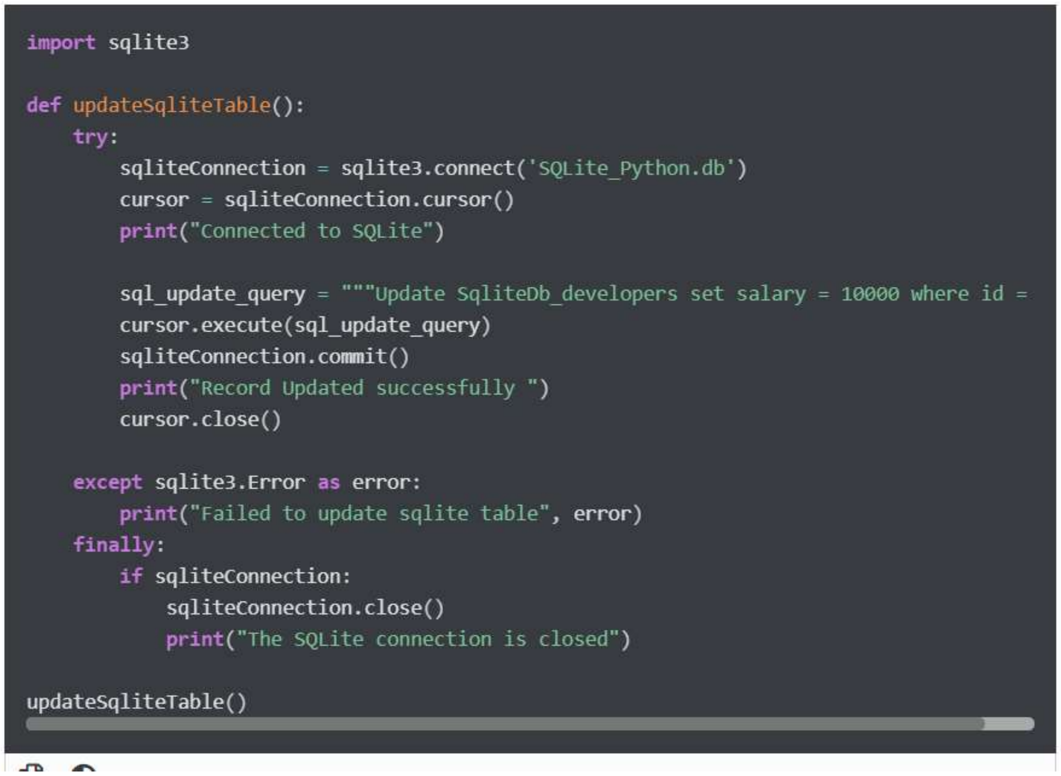
The `cursor.fetchone()` method retrieves the next row from the result set.



```python
import sqlite3

def readSingleRow(developerId):
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db')
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")

        sqlite_select_query = """SELECT * from SqliteDb_developers where id = ?"""
        cursor.execute(sqlite_select_query, (developerId,))
        print("Reading single row \n")
        record = cursor.fetchone()
        print("Id: ", record[0])
        print("Name: ", record[1])
        print("Email: ", record[2])
        print("JoiningDate: ", record[3])
        print("Salary: ", record[4])

        cursor.close()

    except sqlite3.Error as error:
        print("Failed to read single row from sqlite table", error)
    finally:
        if sqliteConnection:
            sqliteConnection.close()
            print("The SQLite connection is closed")

readSingleRow(3)
```

# Python Update SQLite Table

In this lesson, learn to execute an UPDATE Query from a Python application to update the SQLite table's data. You'll learn how to use Python's sqlite3 module to update the SQLite table.



```
import sqlite3

def updateSqliteTable():
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db')
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")

        sql_update_query = """Update SqliteDb_developers set salary = 10000 where id =
        cursor.execute(sql_update_query)
        sqliteConnection.commit()
        print("Record Updated successfully ")
        cursor.close()

    except sqlite3.Error as error:
        print("Failed to update sqlite table", error)
    finally:
        if sqliteConnection:
            sqliteConnection.close()
            print("The SQLite connection is closed")

updateSqliteTable()
```

Home » Python » Databases » Python Update SQLite Table

# Python Update SQLite Table

Updated on: March 9, 2021 | 8 Comments

In this lesson, learn to execute an UPDATE Query from a Python application to update the SQLite table's data. You'll learn how to use Python's sqlite3 module to update the SQLite table.

**Also Read**:

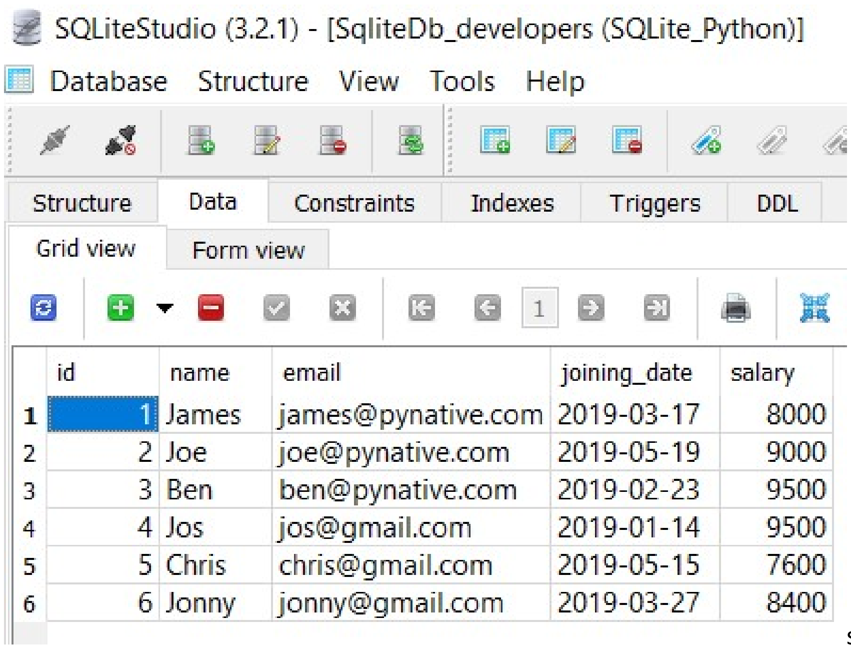
- Solve Python SQLite Exercise
- Read Python SQLite Tutorial (Complete Guide)

## Table of contents

## Prerequisites

Before executing the following program, please make sure you know the SQLite table name and its column details.

For this lesson, I am using the 'SqliteDb_developers' table present in my SQLite database.


SQLiteStudio (3.2.1) - [SqliteDb_developers (SQLite_Python)]
Database Structure View Tools Help
Structure Data Constraints Indexes Triggers DDL
Grid view Form view

| | id | name | email | joining_date | salary |
|---|---|---|---|---|---|
| 1 | 1 | James | james@pynative.com | 2019-03-17 | 8000 |
| 2 | 2 | Joe | joe@pynative.com | 2019-05-19 | 9000 |
| 3 | 3 | Ben | ben@pynative.com | 2019-02-23 | 9500 |
| 4 | 4 | Jos | jos@gmail.com | 2019-01-14 | 9500 |
| 5 | 5 | Chris | chris@gmail.com | 2019-05-15 | 7600 |
| 6 | 6 | Jonny | jonny@gmail.com | 2019-03-27 | 8400 |

sqlitedb_developers table with data

If a table is not present in your SQLite database, then please refer to the following articles: –

- Create SQLite table from Python.
- Insert data into SQLite Table from Python

## Steps to update a single row of SQLite table

As of now, the 'SqliteDb_developers' table contains six rows, so let's update the salary of a developer whose id is 4. To perform SQLite UPDATE query from Python, you need to follow these simple steps:

How to Update SQLite Table in Python

1. **Connect to MySQL from Python**

   Refer to Python SQLite database connection to connect to SQLite database from Python using sqlite3 module.

2. **Prepare a SQL Update Query**

   Prepare an update statement query with data to update. Mention the column name we want to update and its new value. For example, `UPDATE table_name SET column1 = value1, column2 = value2...., columnN = valueN WHERE [condition];`

3. **Execute the UPDATE query, using cursor.execute()**

This method executes the operation stored in the UPDATE query.

4. **Commit your changes**

   After the successful execution of the SQLite update query, Don't forget to commit your changes to the database using `connection.comit()`.

5. **Extract the number of rows affected**

   After a successful update operation, use a `cursor.rowcount` method to get the number of rows affected. The count depends on how many rows you are updating.

6. **Verify result using the SQL SELECT query**

   Execute a SQLite select query from Python to see the new changes

7. **Close the cursor object and database connection object**

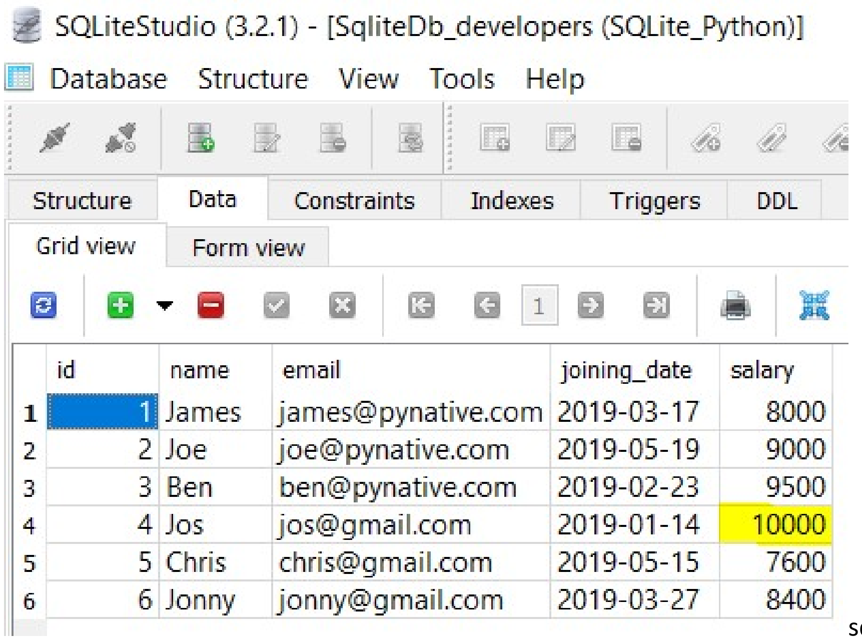
   use `cursor.clsoe()` and `connection.clsoe()` method to close SQLite connections once the update operation completes.

**Example**

```
import sqlite3

def updateSqliteTable():
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db')
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")

        sql_update_query = """Update SqliteDb_developers set salary = 10000 where
id = 4"""
        cursor.execute(sql_update_query)
        sqliteConnection.commit()
        print("Record Updated successfully ")
        cursor.close()

    except sqlite3.Error as error:
        print("Failed to update sqlite table", error)
    finally:
        if sqliteConnection:
            sqliteConnection.close()
            print("The SQLite connection is closed")

updateSqliteTable()
```

**Output**

```
Connected to SQLite Record Updated successfully  The SQLite connection is closed
```


SQLiteStudio (3.2.1) - [SqliteDb_developers (SQLite_Python)]
Database Structure View Tools Help
Structure Data Constraints Indexes Triggers DDL
Grid view Form view

| | id | name | email | joining_date | salary |
|---|---|---|---|---|---|
| 1 | 1 | James | james@pynative.com | 2019-03-17 | 8000 |
| 2 | 2 | Joe | joe@pynative.com | 2019-05-19 | 9000 |
| 3 | 3 | Ben | ben@pynative.com | 2019-02-23 | 9500 |
| 4 | 4 | Jos | jos@gmail.com | 2019-01-14 | 10000 |
| 5 | 5 | Chris | chris@gmail.com | 2019-05-15 | 7600 |
| 6 | 6 | Jonny | jonny@gmail.com | 2019-03-27 | 8400 |

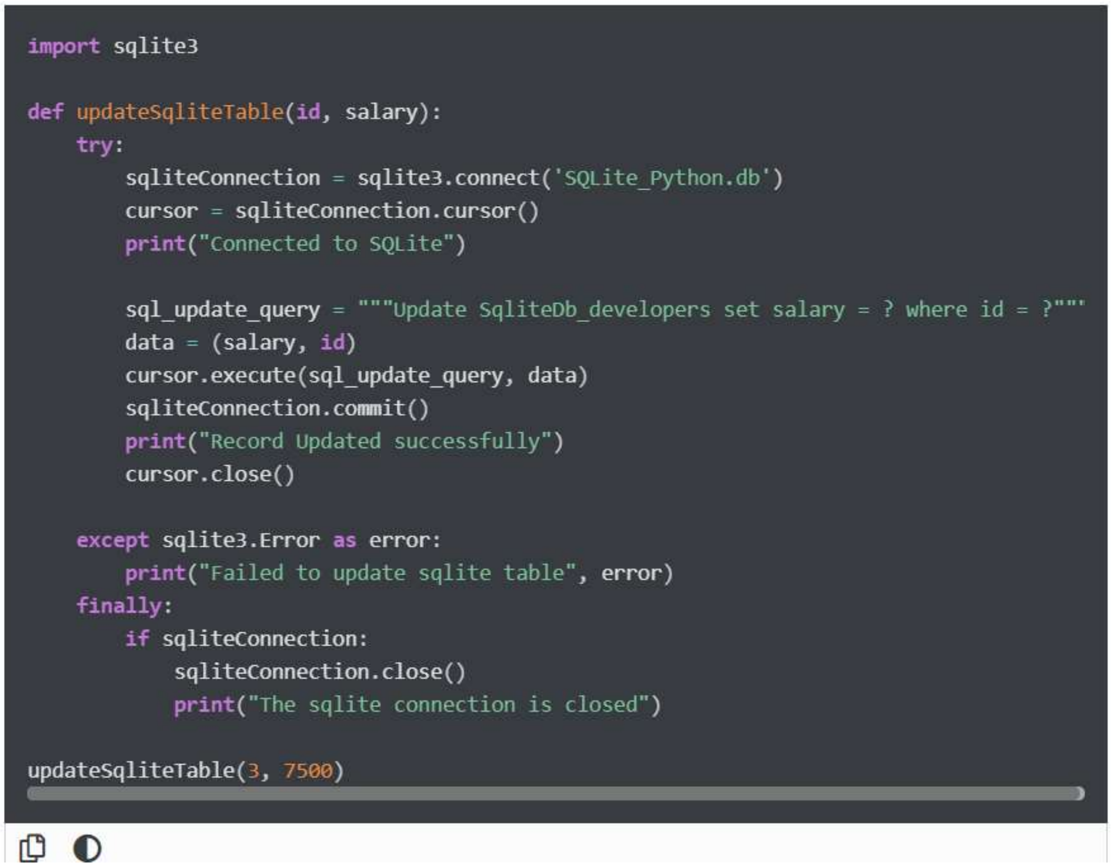
sqlitedb_developers table after updating the row from Python

**Note**: Note: If you are doing multiple update operations and wanted to revert your change in case of failure of any operations, use the `rollback()` method of a connection class to revert the changes. Use the rollback() method of a connection class. in except block.

## Using Python variables in SQLite UPDATE query

Most of the time, we need to update a table with some runtime values. For example, when users update their profile or any other details through a user interface, we need to update a table with those new values. In such cases, It is always best practice to use a parameterized query.

The parameterized query uses placeholders (`?`) inside SQL statements that contain input from users. It helps us to update runtime values and prevent SQL injection concerns.



```python
import sqlite3

def updateSqliteTable(id, salary):
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db')
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")

        sql_update_query = """Update SqliteDb_developers set salary = ? where id = ?"""
        data = (salary, id)
        cursor.execute(sql_update_query, data)
        sqliteConnection.commit()
        print("Record Updated successfully")
        cursor.close()

    except sqlite3.Error as error:
        print("Failed to update sqlite table", error)
    finally:
        if sqliteConnection:
            sqliteConnection.close()
            print("The sqlite connection is closed")

updateSqliteTable(3, 7500)
```

## Update multiple rows of SQLite table using cursor’s `executemany()`

In the above example, we have used execute() method of cursor object to update a single record. But sometimes, we need to update multiple rows of the SQLite table. For example, you want to increase the salary of developers by 20%.

Instead of executing the UPDATE query every time to update each record, you can perform bulk update operations in a single query using the `cursor.executemany()` method.

The `executemany(query, seq_param)` method accepts the following two parameters

- SQL query
- list of records to be updated.

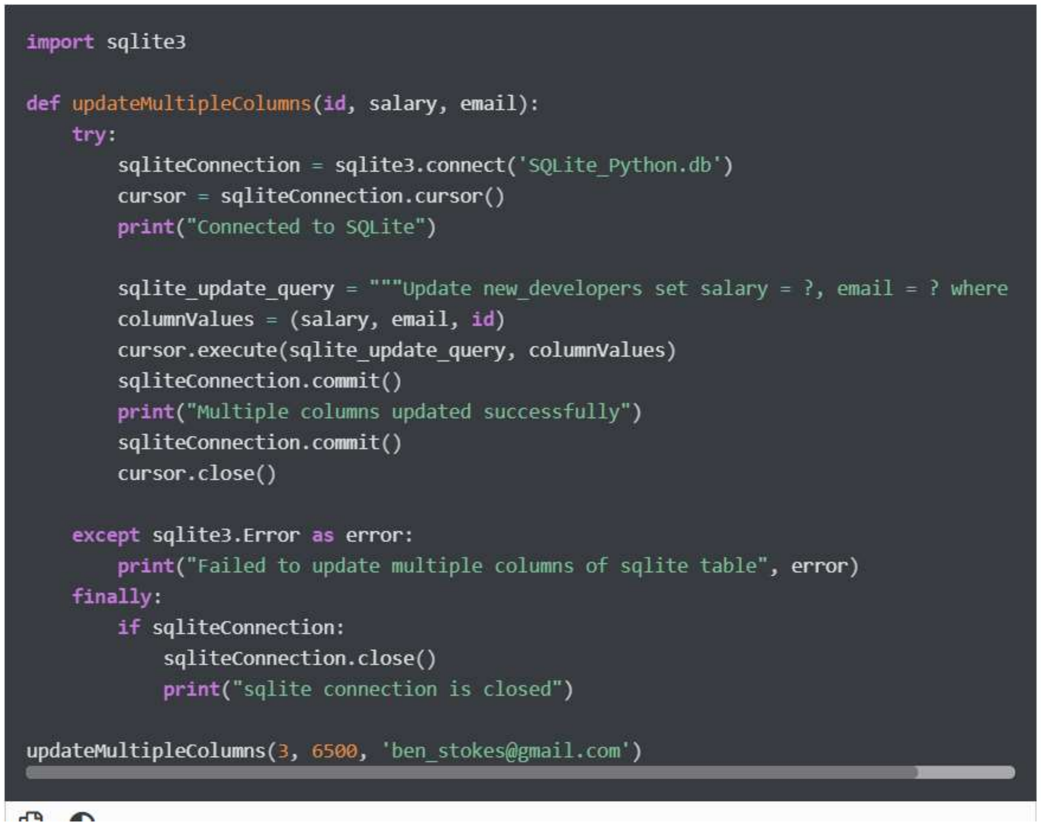
Now, let see the example. In this example, we are updating three rows.

Now, let see the example. In this example, we are updating three rows.

```
import sqlite3

def updateMultipleRecords(recordList):
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db')
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")

        sqlite_update_query = """Update SqliteDb_developers set salary = ? where id = 
        cursor.executemany(sqlite_update_query, recordList)
        sqliteConnection.commit()
        print("Total", cursor.rowcount, "Records updated successfully")
        sqliteConnection.commit()
        cursor.close()

    except sqlite3.Error as error:
        print("Failed to update multiple records of sqlite table", error)
    finally:
        if sqliteConnection:
            sqliteConnection.close()
            print("The SQLite connection is closed")

records_to_update = [(9700, 4), (7800, 5), (8400, 6)]
updateMultipleRecords(records_to_update)
```

## Update multiple Columns of SQLite table

We can also update multiple columns of an SQLite table in a single query. Just prepare a parameterized query using a placeholder to update multiple columns. Let see this with an example program.



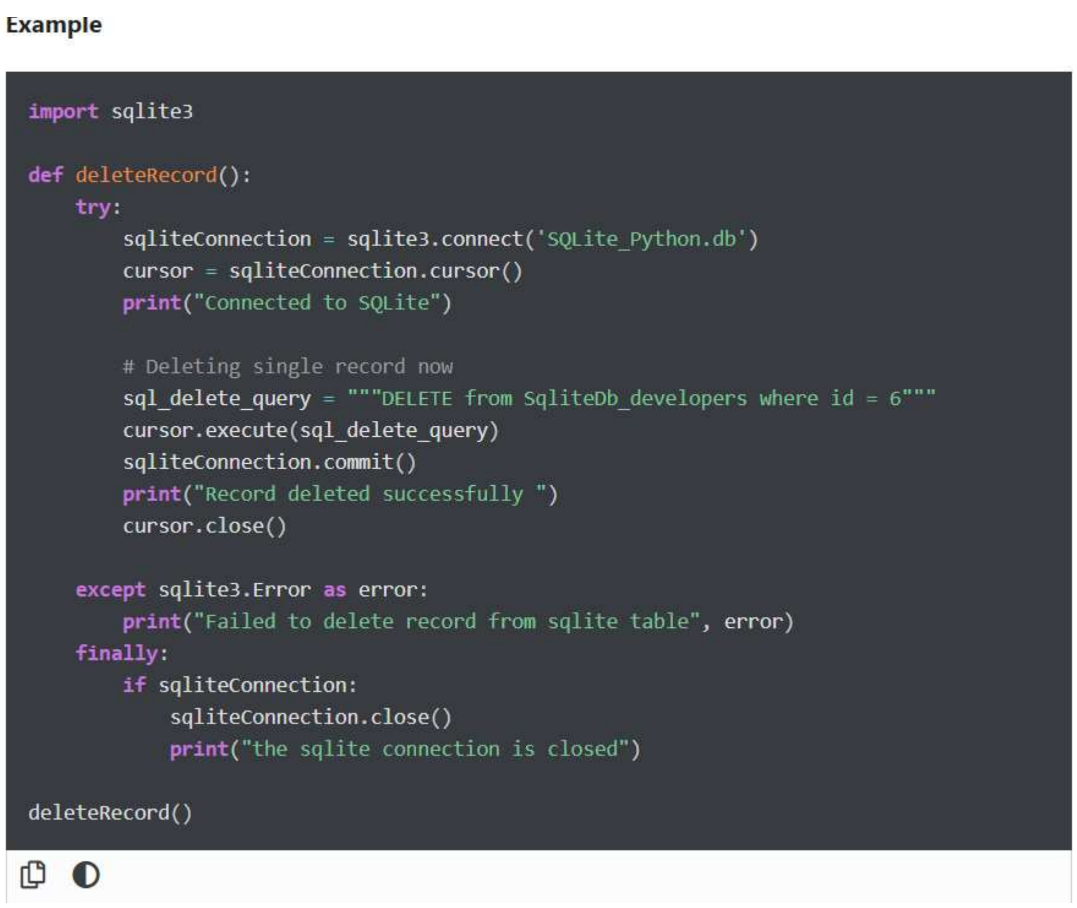
```python
import sqlite3

def updateMultipleColumns(id, salary, email):
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db')
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")

        sqlite_update_query = """Update new_developers set salary = ?, email = ? where
        columnValues = (salary, email, id)
        cursor.execute(sqlite_update_query, columnValues)
        sqliteConnection.commit()
        print("Multiple columns updated successfully")
        sqliteConnection.commit()
        cursor.close()

    except sqlite3.Error as error:
        print("Failed to update multiple columns of sqlite table", error)
    finally:
        if sqliteConnection:
            sqliteConnection.close()
            print("sqlite connection is closed")

updateMultipleColumns(3, 6500, 'ben_stokes@gmail.com')
```

# Python Delete from SQLite Table

Learn to delete data from an SQLite table using Python. You'll learn how to use Python's built-in module sqlite3 to delete data from the SQLite table.

**Goals of this lesson**

- Delete a single and multiple rows, all rows, a single column, and multiple columns from the SQLite table using Python
- Use a Python parameterized query to provide value at runtime to the SQLite delete query
- Execute a bulk delete using a single query



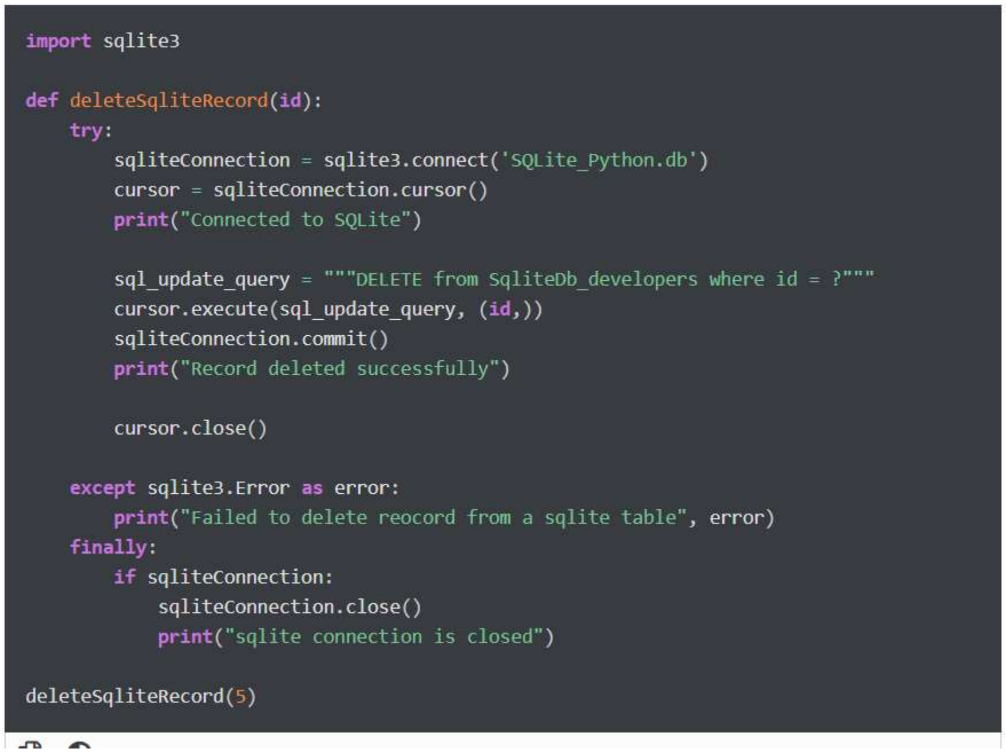
**Example**

```
import sqlite3

def deleteRecord():
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db')
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")

        # Deleting single record now
        sql_delete_query = """DELETE from SqliteDb_developers where id = 6"""
        cursor.execute(sql_delete_query)
        sqliteConnection.commit()
        print("Record deleted successfully ")
        cursor.close()

    except sqlite3.Error as error:
        print("Failed to delete record from sqlite table", error)
    finally:
        if sqliteConnection:
            sqliteConnection.close()
            print("the sqlite connection is closed")

deleteRecord()
```

## Use Python Variable in a query to Delete Row from SQLite table

Most of the time, we need to delete a row from an SQLite table where the id passed at runtime. For example, when a user cancels his/her subscription, we need to delete the entry from a table as per the user id. In such cases, It is always best practice to use a parameterized query.

The parameterized query uses placeholders (`?`) inside SQL statements that contain input from users. It helps us to delete runtime values and prevent SQL injection concerns.



```python
import sqlite3

def deleteSqliteRecord(id):
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db')
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")

        sql_update_query = """DELETE from SqliteDb_developers where id = ?"""
        cursor.execute(sql_update_query, (id,))
        sqliteConnection.commit()
        print("Record deleted successfully")

        cursor.close()

    except sqlite3.Error as error:
        print("Failed to delete reocord from a sqlite table", error)
    finally:
        if sqliteConnection:
            sqliteConnection.close()
            print("sqlite connection is closed")

deleteSqliteRecord(5)
```

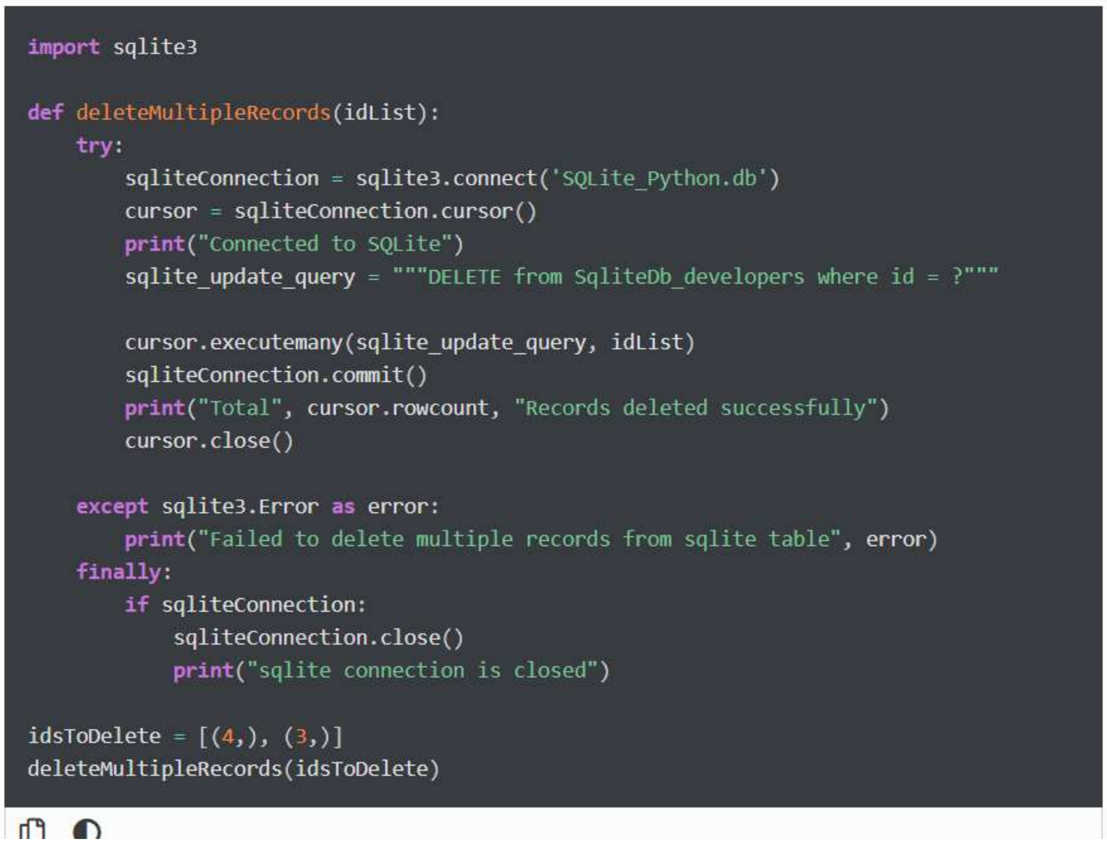
## Delete multiple rows from SQLite table

In the above example, we have used execute() method of cursor object to update a single record, but sometimes, we need to delete an N-number of rows. For example, You want to delete employee data from the developer's table who left the organization.

Instead of executing a delete query repeatedly to delete each record, you can perform the bulk delete operation in a single query using the `cursor.executemany()` method.

The `executemany(query, seq_param)` method accepts two parameters a SQL query and a list of records to delete.

In this example, we are removing three rows.



```
import sqlite3

def deleteMultipleRecords(idList):
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db')
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")
        sqlite_update_query = """DELETE from SqliteDb_developers where id = ?"""

        cursor.executemany(sqlite_update_query, idList)
        sqliteConnection.commit()
        print("Total", cursor.rowcount, "Records deleted successfully")
        cursor.close()

    except sqlite3.Error as error:
        print("Failed to delete multiple records from sqlite table", error)
    finally:
        if sqliteConnection:
            sqliteConnection.close()
            print("sqlite connection is closed")

idsToDelete = [(4,), (3,)]
deleteMultipleRecords(idsToDelete)
```