Python SQLite tutorial

pynative.com/python-sqlite/

June 24, 2019

The purpose of this Python SQLite tutorial is 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 of the computers and mobile devices and browsers. Python's official sqlite3 module helps us to work with the SQLite database.

Recommended Exercise and Quiz:

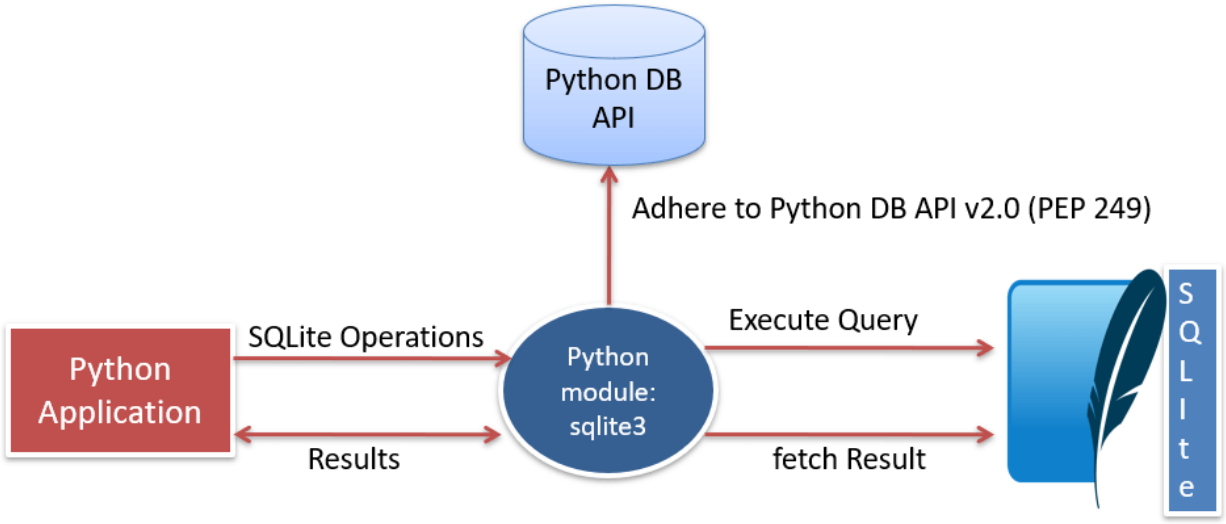
Python SQLite Exercise

Python Database Quiz

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

This tutorial mainly focuses on: –

- Connecting to the SQLite database from Python and Creating an SQLite database and tables.
- Next, we will cover SQLite Datatypes and it's corresponding Python types.
- Next, we will learn how to perform SQLite CRUD operation i.e., data insertion, data retrieval, data update, and data deletion from Python.
- We will also learn how to execute SQLite scripts from Python.
- Insert/Retrieve digital data in SQLite using Python.
- Next, it will cover SQLite transaction Handling, creating and calling SQLite functions, and error-handling techniques to develop robust python programs with SQLite database.
- It will also let you know how to create and manage an in-memory database and convert SQLite values to custom Python types.
- Take a backup of SQLite database from within Python



Python sqlite3 module working

Let see each section now.

Python SQLite Connection

This section lets you know how to **create an SQLite database and connect to it** through python using the sqlite3 module.

To establish a connection to SQLite, you need to specify the **database name** you want to connect. If you specify the database file name that already presents on disk, it will connect to it. But if your specified SQLite database file does not exist, SQLite creates a new database for you.

You need to **follow the following steps to connect to SQLite**

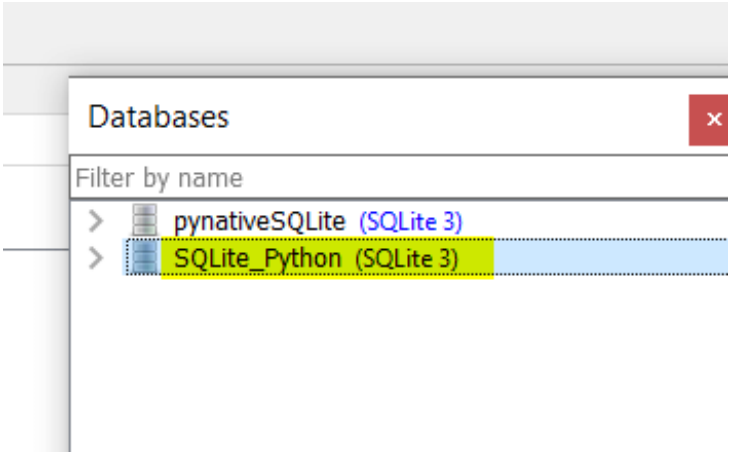
- Use the `connect()` method of a sqlite3 module and pass the database name as an argument.
- Create a cursor object using the connection object returned by the connect method to execute SQLite queries from Python.
- Close the Cursor object and SQLite database connection object when work is done.
- Catch database exception if any that may occur during this connection process.

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

```
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")
```

You should get the following output after connecting to SQLite from Python.

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



Understand the SQLite connection Code in detail

import sqlite3

> This line imports the sqlite3 module in our program. Using the classes and methods defined in the sqlite3 module we can communicate with the SQLite database.

sqlite3.connect()

- Using the `connect()` method we can create a connection to the SQLite database. This method returns the SQLite Connection Object.

- 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.

cursor = sqliteConnection.cursor()

- 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 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.

After this, we created a SQLite query to get the database version.

cursor.execute()

- Using the cursor's execute method we can execute a database operation or query from Python. The `cursor.execute()` method takes an SQLite query as a parameter and returns the resultSet i.e. nothing but a database rows.
- We can retrieve query result from resultSet using cursor methods such as
- In our example, we are executing a `SELECT version();` query to fetch the SQLite version.

try-except-finally block: We placed all our code in the try-except block to catch the SQLite database exceptions and errors that may occur during this process.

- Using the **`sqlite3.Error`** class of sqlite3 module, we can handle any database error and exception that may occur while working with SQLite from Python.
- Using this approach we can make our application robust. The `sqlite3.Error` class helps us to understand the error in detail. It returns an error message and error code.

cursor.close() and connection.close()

> 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

In this section, we will learn how to create a table in the SQLite database from Python using the sqlite3 module. 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.

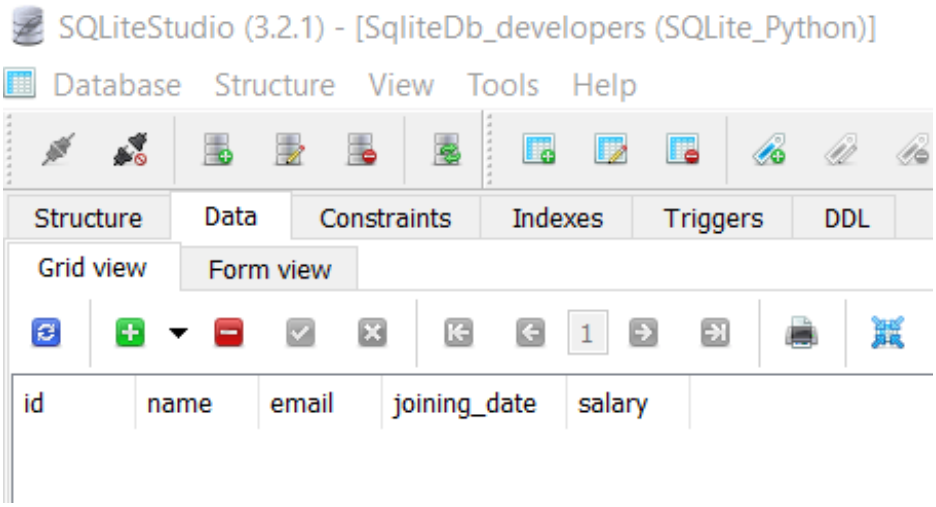
Steps for creating a table in SQLite from Python: –

- Connect to SQLite using a `sqlite3.connect()` . I have explained the SQLite connection code at the start of this article.
- Prepare a create table query.
- Execute the query using a `cursor.execute(query)`
- In the end, Close the SQLite database connection and cursor object.

```
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 closed
```



SQLite Datatypes and it's corresponding Python types

Before proceeding further on executing SQLite CRUD operations from Python first understand SQLite data type and their corresponding Python types, which will help us to 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 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 type	SQLite type
None	NULL
int	INTEGER
float	REAL
str	TEXT

bytes	BLOB

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 can execute DML queries i.e., SQLite Insert, Update, Delete operations from Python.

Now, we know the table, and it's column details so let's move to the crud operations. I have created a separate tutorial on each operation to cover it in detail. Let see each section now.

- **Insert data into SQLite Table from Python** – In this section, we will learn how to execute INSERT command from python to insert records to the SQLite table.
- **Read SQLite Table's data from Python** – In this article, we will learn how to execute SQLite SELECT query from a Python application to fetch the table's rows. Also, I will let you know how to use `fetchall()` , `fetchmany()` , and `fetchone()` methods of a cursor class to fetch limited rows from the table to enhance the performance.
- **Update data of SQLite table from Python** – In this section, we will learn how to execute the UPDATE query from python to modify records of the SQLite table.
- **Delete data from SQLite table from Python** – In this section, we will learn how to execute the DELETE query from python to remove records from SQLite table.

Execute SQL File (scripts) using cursor's executescript function

SQLite scripts are handy for most of the daily job. SQLite script is a **set of SQL commands saved as a file (in .sql format)**.
An SQLite script contains one or more SQL operations which you'll then execute from your command line prompt whenever required.

Below are the **few common scenarios where we can use SQLite scripts**

- Back-Up Multiple Databases at Once.
- Compare Row Counts in Tables From Two Different Databases With the Same Schema.
- keep all your CREATE TABLE SQL commands in a database script. So you can create database schema on any server.

You can execute your script from the SQLite command line using the .read command, like this:

```
sqlite> .read mySQLiteScript.sql
```

For this example, I have created a sample SQLite script which will create two tables. This

is the script that I have created.

```
CREATE TABLE hardware (
 id INTEGER PRIMARY KEY,
 name TEXT NOT NULL,
 price REAL NOT NULL
);

CREATE TABLE software (
 id INTEGER PRIMARY KEY,
 name TEXT NOT NULL,
 price REAL NOT NULL
);
```

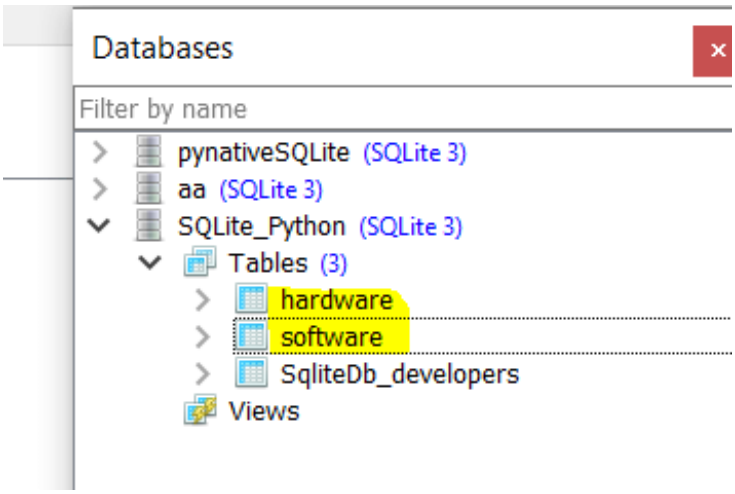
Now **let's see how to execute an SQLite script from within Python.**

```
import sqlite3

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

    with open('E:\pynative\Python\photos\sqlite_create_tables.sql', 'r') as sqlite_file:
        sql_script = sqlite_file.read()

    cursor.executescript(sql_script)
    print("SQLite script executed successfully")
    cursor.close()

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

Output: SQLite tables created by executing a SQL script from Python.

```
Successfully Connected to SQLite
SQLite script executed successfully
sqlite connection is closed
```

Note: After connecting to SQLite, We read all content of an SQLite script file stored on disk and copied it into a python string variable. Then we called `cursor.executscript(script)` method to execute all SQL statements in one call.



Insert/Retrieve digital data in SQLite using Python

- In this section, I will let you know how to insert or save any digital information such as a **file**, **image**, **video**, or a **song** as a **BLOB** data into the SQLite table from python.
- Also, learn how we can read a file, image, video, song or any digital data stored in SQLite using Python.

Refer our complete guide on **Python SQLite BLOB to Insert and Retrieve file and images.**

Create Or Redefine SQLite Functions using Python

Python sqlite3 module provides us the ability to create and redefine SQL functions from within Python. I have created a separate tutorial to cover it in detail. Please refer to How to create and redefine SQL functions from within Python.

Refer our complete guide on **Create Or Redefine SQLite Functions from within Python.**

Working with SQLite date and timestamp types in Python and vice-versa

Sometimes we need to insert or read date or DateTime value from an SQLite table. So if you are working with a date or timestamp values, then please refer our separate tutorial on **working with SQLite DateTime values in Python**.

SQLite Database Exceptions

Exception sqlite3.**Warning**

> A subclass of Exception. And you can ignore it if you want it doesn't stop the execution.

Exception sqlite3.**Error**

> The base class of the other exceptions in the sqlite3 module. It is a subclass of Exception.

Exception sqlite3.**DatabaseError**

> Exception raised for errors that are related to the database.
> Examples: If you try and open a file as a sqlite3 database that is NOT a database file, you will get sqlite3.DatabaseError: file is encrypted or is not a database

Exception sqlite3.**IntegrityError**

> Subclass of a DatabaseError. You will get this Exception when the relational integrity of the database is affected, e.g., a foreign key check fails.

Exception sqlite3.**ProgrammingError**

> It is also a subclass of DatabaseError. This exception raised because of programming errors, e.g., creating a table with the same which already exists, syntax error in the SQL queries.

Exception sqlite3.**OperationalError**

- It is also a subclass of DatabaseError. This error is not in our control. This exception raised for errors that are related to the database's operation.
- Examples: an accidental disconnect, server down, a timeout occurs, the data source issues. server down

Exception sqlite3.**NotSupportedError**

- You will get an exception raised when database API was used which is not supported by the database.
- Example: calling the rollback() method on a connection that does not support the transaction. Calling commit after creating table command.

So **it will always be advisable to write all your database operation code in the try block** so you can catch exceptions in except block if any and take the corrective actions against it.

For example, let's try to insert data into a table that doesn't exist in the SQLite database and Print the full exception stack.

```
import sqlite3
import traceback
import sys

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

    sqlite_insert_query = """INSERT INTO unknown_table_1
                          (id, text)  VALUES  (1, 'Demo Text')"""

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

except sqlite3.Error as error:
    print("Failed to insert data into sqlite table")
    print("Exception class is: ", error.__class__)
    print("Exception is", error.args)
    print('Printing detailed SQLite exception traceback: ')
    exc_type, exc_value, exc_tb = sys.exc_info()
    print(traceback.format_exception(exc_type, exc_value, exc_tb))
finally:
    if (sqliteConnection):
        sqliteConnection.close()
        print("The SQLite connection is closed")
```

Output:

```
Successfully Connected to SQLite
Failed to insert data into sqlite table
Exception class is:  <class 'sqlite3.OperationalError'>
Exception is ('no such table: unknown_table_1',)
Printing detailed SQLite exception traceback:
['Traceback (most recent call last):\n', '  File "E:/demos/sqlite_demos/sqlite_errors.py", line 13, in
<module>\n    count = cursor.execute(sqlite_insert_query)\n', 'sqlite3.OperationalError: no such table:
unknown_table_1\n']
The SQLite connection is closed
```

Change SQLite connection timeout when connecting from within Python

It can be the scenario when multiple connections access an SQLite database, and one of the processes performing some data modification operation on the database, To perform data modification connection needs to take lock i.e., the SQLite database is locked until that transaction is committed. The **timeout parameter** we specify while connecting to the database determines **how long the connection should wait for the lock** to go away until raising an exception.

The **default value for the timeout parameter is 5.0 (five seconds**). You don't need to specify it while connecting because it is a default value. i.e., whenever you are connecting to SQLite from Python, and you didn't get a response within 5 seconds, your program will raise an exception. But if you are facing connection timeout issue and wanted to increase it, you can do this using a timeout argument of a `sqlite3.connect` function.

Let see how to change the timeout value while connecting SQLite from within Python.

```
import sqlite3

def readSqliteTable():
    try:
        sqliteConnection = sqlite3.connect('SQLite_Python.db', timeout=20)
        cursor = sqliteConnection.cursor()
        print("Connected to SQLite")

        sqlite_select_query = """SELECT count(*) from SqliteDb_developers"""
        cursor.execute(sqlite_select_query)
        totalRows = cursor.fetchone()
        print("Total rows are:  ", totalRows)
        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()
```

Output:

```
Connected to SQLite
Total rows are:   (2,)
The Sqlite connection is closed
```

Identify total changes since the SQLite database connection was opened

For audit or statistics purpose if you want to find the numbers of database rows that have been modified, inserted, or deleted since the database connection was opened you can use the `connection.total_changes` method of a Python sqlite3 module.

The `connection.total_changes` method returns the total number of database rows that have been affected. Let see the Python example to find total changes that have been done since the database connection was opened.

```
import sqlite3

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 (4, 'Jos', 'jos@gmail.com', '2019-01-14', 9500);"""
    cursor.execute(sqlite_insert_query)

    sql_update_query = """Update SqliteDb_developers set salary = 10000 where id = 4"""
    cursor.execute(sql_update_query)

    sql_delete_query = """DELETE from SqliteDb_developers where id = 4"""
    cursor.execute(sql_delete_query)

    sqliteConnection.commit()
    cursor.close()

except sqlite3.Error as error:
    print("Error while working with SQLite", error)
finally:
    if (sqliteConnection):
        print("Total Rows affected since the database connection was opened: ",
sqliteConnection.total_changes)
        sqliteConnection.close()
        print("sqlite connection is closed")
```

Output:

```
Connected to SQLite
Total Rows affected since the database connection was opened:  3
sqlite connection is closed
```

Take a backup of SQLite database from within Python

Python sqlite3 module provides a function to take a backup of the SQLite database. Using a `connection.backup()` method you can take the backup of SQLite database.

```
connection.backup(target, *, pages=0, progress=None, name="main", sleep=0.250)
```

This function takes a backup of the SQLite database, and a copy will be written into the argument target, that must be another Connection instance. By default, or when pages are either 0 or a negative integer, the entire database is copied in a single step; otherwise, the method performs a loop copying up to pages at a time.

The name argument specifies the database that you want to copy. The sleep argument defines the number of seconds to sleep by between successive attempts to backup the remaining pages of a database. sleep argument can be specified either as an integer or a floating-point value.

Let see the example to copy an existing database into another.

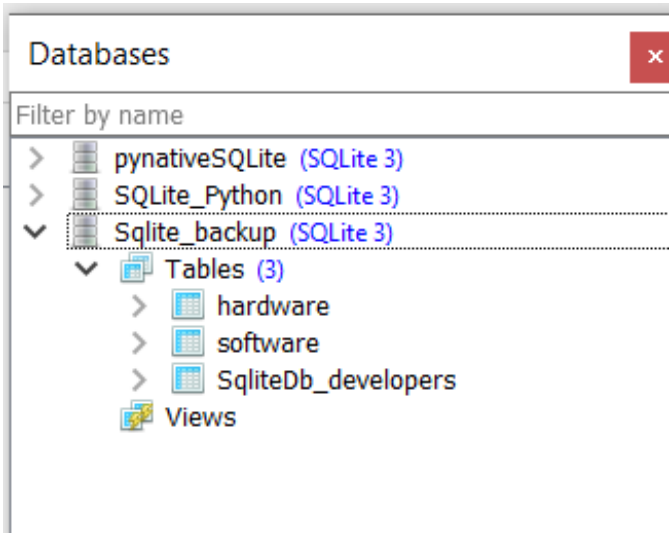
```
import sqlite3

def progress(status, remaining, total):
    print(f'Copied {total-remaining} of {total} pages...')

try:
    #existing DB
    sqliteCon = sqlite3.connect('SQLite_Python.db')
    #copy into this DB
    backupCon = sqlite3.connect('Sqlite_backup.db')
    with backupCon:
        sqliteCon.backup(backupCon, pages=3, progress=progress)
    print("backup successful")
except sqlite3.Error as error:
    print("Error while taking backup: ", error)
finally:
    if(backupCon):
        backupCon.close()
        sqliteCon.close()
```

Output:

```
Copied 3 of 26 pages...
Copied 6 of 26 pages...
Copied 9 of 26 pages...
Copied 12 of 26 pages...
Copied 15 of 26 pages...
Copied 18 of 26 pages...
Copied 21 of 26 pages...
Copied 24 of 26 pages...
Copied 26 of 26 pages...
backup successful
```

Note:

- After connecting to SQLite, We opened both the databases using two different connections.
- Next, we executed a `connection.backup()` method using a first connection instance. Also, we specified the number of database pages to copy in each iteration.



Python SQLite Programming Quiz and Exercise Project

I have created a Python Database Programming Quiz and Exercise to practice and master database programming in Python.

which provides multiple-choice questions to get familiar with Python database concepts and APIs.

- Solve our Python Database Programming Quiz which provides multiple-choice questions to get familiar with Python database APIs.
- Solve our Python Database Programming Exercise to master database programming in Python.

That’s it. Folks Let me know your comments in the section below.