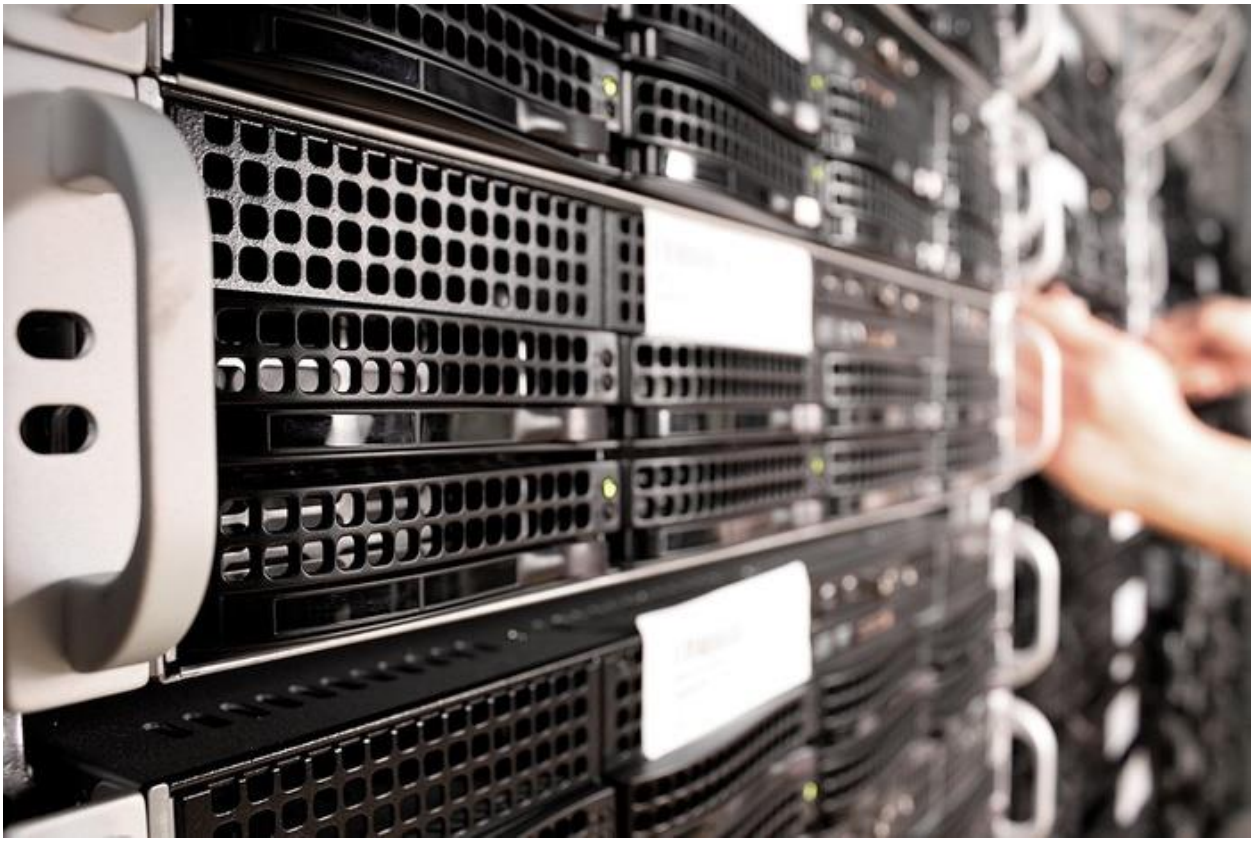
From: https://www.codementor.io/likegeeks/python-sqlite3-tutorial-database-programming-riqdhwx9z

Python SQLite3 tutorial (Database programming)

Published Jan 24, 2019



In this tutorial, we will work with SQLite3 database programmatically using Python.

SQLite in general, is a server-less database that can be used within almost all programming languages including Python. Server-less means there is no need to install a separate server to work with SQLite so you can connect directly with the database.

SQLite is a lightweight database that can provide a relational database management system with zero-configuration because there is no need to configure or setup anything to use it.

We will use SQLite version 3 or SQLite3, so let's get started.

Create Connection

To use SQLite3 in Python, first of all, you will have to import the *sqlite3* module and then create a connection object which will connect us to the database and will let us execute the SQL statements.

A connection object is created using the *connect()* function:

```
import sqlite3

con = sqlite3.connect('mydatabase.db')
```

A new file called 'mydatabase.db' will be created where our database will be stored.

SQLite3 Cursor

To execute SQLite statements in Python, you need a cursor object. You can create it using the *cursor()* method.

The SQLite3 cursor is a method of the connection object. To execute the SQLite3 statements, a connection is established at first and then an object of the cursor is created using the connection object as follows:

```
con = sqlite3.connect('mydatabase.db')

cursorObj = con.cursor()
```

Now we can use the cursor object to call the *execute()* method to execute any SQL queries.

Create Database

When you create a connection with SQLite, a database file is automatically created if it doesn't already exist. This database file is created on disk, we can also create a database in RAM by using :memory: with the connect function. This database is called in-memory database.

Consider the code below in which we have created a database with a *try*, *except* and *finally* blocks to handle any exceptions:

```
import sqlite3

from sqlite3 import Error

def sql_connection():

    try:

        con = sqlite3.connect(':memory:')

        print("Connection is established: Database is created in memory")

    except Error:
```

```
        print(Error)
    finally:
        con.close()
sql_connection()
```

First, the *sqlite3* module is imported, then a function named sql_connection is defined. Inside the function, we have a *try* block where the *connect()* function is returning a connection object after establishing the connection.

Then we have *except* block which in case of any exceptions, prints the error message. If there are no errors, the connection will be established and a message will be displayed as follows.

After that, we have closed our connection in the *finally* block. Closing a connection is optional but it is a good programming practice, so you free the memory from any unused resources.

Create Table

To create a table in SQLite3, you can use the Create Table query in the *execute()* method. Consider the following steps:

1. The connection object is created
2. Cursor object is created using the connection object
3. Using cursor object, execute method is called with create table query as the parameter

Let's create employees with the following attributes:

employees (id, name, salary, department, position, hireDate)

The code will be like this:

```
import sqlite3
from sqlite3 import Error
def sql_connection():
    try:
        con = sqlite3.connect('mydatabase.db')
        return con
    except Error:
        print(Error)
def sql_table(con):
```

```
    cursorObj = con.cursor()

    cursorObj.execute("CREATE TABLE employees(id integer PRIMARY KEY, name
text, salary real, department text, position text, hireDate text)")

    con.commit()

con = sql_connection()

sql_table(con)
```

In the above code, we have defined two methods, the first one establishes a connection and the second method creates a cursor object to execute the create table statement.

The *commit()* method saves all the changes we make. In the end, both methods are called.

To check if our table is created, you can use the DB browser for sqlite ;to view your table. Open you mydatabase.db file with the program and you should see your table:

Insert in Table

To insert data in a table, we use the INSERT INTO statement. Consider the following line of code:

cursorObj.execute("INSERT INTO employees VALUES(1, 'John', 700, 'HR', 'Manager', '2017-01-04')")

To check if the data is inserted, click on Browse Data in the DB Browser:

We can also pass values/arguments to an INSERT statement in the *execute()* method. You can use the question mark (?) as a placeholder for each value. The syntax of the INSERT will be like the following:

cursorObj.execute('''INSERT INTO employees(id, name, salary, department, position, hireDate) VALUES(?, ?, ?, ?, ?, ?)''', entities)

Where entities contain the values for the placeholders as follows:

entities = (2, 'Andrew', 800, 'IT', 'Tech', '2018-02-06')

The entire code is as follows:

```
import sqlite3

con = sqlite3.connect('mydatabase.db')

def sql_insert(con, entities):
```

```
    cursorObj = con.cursor()

    cursorObj.execute('INSERT INTO employees(id, name, salary, department,
position, hireDate) VALUES(?, ?, ?, ?, ?, ?)', entities)

    con.commit()

entities = (2, 'Andrew', 800, 'IT', 'Tech', '2018-02-06')

sql_insert(con, entities)
```

Update Table

To update the table simply create a connection, then create a cursor object using the connection and finally use the UPDATE statement in the *execute()* method.

Suppose that we want to update the name of the employee whose id equals 2. For updating, we will use the UPDATE statement and for the employee whose id equals 2. We will use the WHERE clause as a condition to select this employee.

Consider the following code:

```
import sqlite3

con = sqlite3.connect('mydatabase.db')

def sql_update(con):

    cursorObj = con.cursor()

    cursorObj.execute('UPDATE employees SET name = "Rogers" where id = 2')

    con.commit()

sql_update(con)
```

This will change the name from Andrew to Rogers as follows:

Select statement

The select statement is used to select data from a particular table. If you want to select all the columns of the data from a table, you can use the asterisk (*). The syntax for this will be as follows:

select * from table_name

In SQLite3, the SELECT statement is executed in the execute method of the cursor object. For example, select all the columns of the employees' table, run the following code:

cursorObj.execute('SELECT * FROM employees ')

If you want to select a few columns from a table then specify the columns like the following:

select column1, column2 from tables_name

For example,

cursorObj.execute('SELECT id, name FROM employees')

The select statement selects the required data from the database table and if you want to fetch the selected data, the *fetchall()* method of the cursor object is used. This is demonstrated in the next section.

Fetch all data

To fetch the data from a database we will execute the SELECT statement and then will use the *fetchall()* method of the cursor object to store the values into a variable. After that, we will loop through the variable and print all values.

The code will be like this:

```
import sqlite3

con = sqlite3.connect('mydatabase.db')

def sql_fetch(con):

    cursorObj = con.cursor()

    cursorObj.execute('SELECT * FROM employees')

    rows = cursorObj.fetchall()

    for row in rows:

        print(row)

sql_fetch(con)
```

The above code will print out the records in our database as follows:

You can also use the *fetchall()* in one line as follows:

[print(row) for row in cursorObj.fetchall()]

If you want to fetch specific data from the database, you can use the WHERE clause. For example, we want to fetch the ids and names of those employees whose salary is greater than 800. For this, let's populate our table with more rows, then execute our query.

You can use the insert statement to populate the data or you can enter them manually in the DB browser program.

Now, to fetch id and names of those who have a salary greater than 800:

```
import sqlite3

con = sqlite3.connect('mydatabase.db')

def sql_fetch(con):

    cursorObj = con.cursor()

    cursorObj.execute('SELECT id, name FROM employees WHERE salary > 800.0')

    rows = cursorObj.fetchall()

    for row in rows:

        print(row)

sql_fetch(con)
```

In the above SELECT statement, instead of using the asterisk (*), we specified the id and name attributes. The result will be like the following:

SQLite3 rowcount

The SQLite3 rowcount is used to return the number of rows that are affected or selected by the latest executed SQL query.

When we use rowcount with the SELECT statement, -1 will be returned as how many rows are selected is unknown until they are all fetched. Consider the example below:

print(cursorObj.execute('SELECT * FROM employees').rowcount)

Therefore, to get the row count, you need to fetch all the data, and then get the length of the result:

rows = cursorObj.fetchall()
print len (rows)

When the DELETE statement is used without any condition (a where clause), all the rows in the table will be deleted and the total number of deleted rows will be returned by rowcount.

print(cursorObj.execute('DELETE FROM employees').rowcount)

If no row is deleted 0 will be returned.

List tables

To list all tables in a SQLite3 database, you should query sqlite_master table and then use the *fetchall()* to fetch the results from the SELECT statement.

The sqlite_master is the master table in SQLite3 which stores all tables.

```
import sqlite3

con = sqlite3.connect('mydatabase.db')

def sql_fetch(con):

    cursorObj = con.cursor()

    cursorObj.execute('SELECT name from sqlite_master where type= "table"')

    print(cursorObj.fetchall())

sql_fetch(con)
```

This will list all the tables as follows:

Check if table exists or not

When creating a table, we should make sure that the table is not already existed. Similarly, when removing/ deleting a table, the table should exist.

To check if the table doesn't already exist we use *"if not exists"* with the CREATE TABLE statement as follows:

create table if not exists table_name (column1, column2, …, columnN)

For example:

```
import sqlite3

con = sqlite3.connect('mydatabase.db')

def sql_fetch(con):

    cursorObj = con.cursor()

    cursorObj.execute('create table if not exists projects(id integer, name
text)')

    con.commit()

sql_fetch(con)
```

Similarly, to check if the table exists when deleting we use *"if exists"* with the DROP TABLE statement as follows:

drop table if exists table_name

For example,

cursorObj.execute('drop table if exists projects')

We can also check if the table we want to access exists or not by executing the following query:

```
cursorObj.execute('SELECT name from sqlite\_master WHERE type = "table" AND name = "employees"')

print(cursorObj.fetchall())
```

If the employees' table exists, its name will be returned as follows:

If the table name we specified doesn't exist, an empty array will be returned:

Drop Table

You can drop/delete a table using the DROP statement. The syntax of DROP statement is as follows:

drop table table_name

To drop a table, the table should exist in the database. Therefore, it is recommended to use *"if exists"* with the drop statement as follows:

drop table if exists table_name

For example,

```
import sqlite3

con = sqlite3.connect('mydatabase.db')

def sql_fetch(con):

    cursorObj = con.cursor()

    cursorObj.execute('DROP table if exists employees')

    con.commit()

sql_fetch(con)
```

SQLite3 Exceptions

Exceptions are the run time errors. In Python programming, all exceptions are the instances of the class derived from the BaseException.

In SQLite3, we have the following main Python exceptions:

DatabaseError

Any error related to database raises the DatabaseError.

IntegrityError

IntegrityError is a subclass of DatabaseError and is raised when there is a data integrity issue, for example,e foreign data isn't updated in all tables resulting in the inconsistency of the data.

ProgrammingError

The exception ProgrammingError raises when there are syntax errors or table is not found or function is called with the wrong number of parameters/ arguments.

OperationalError

This exception is raised when the database operations are failed, for example, unusual disconnection. This is not the fault of the programmers.

NotSupportedError

When you use some methods that aren't defined or supported by database NotSupportedError exception is raised.

SQLite3 Executemany (Bulk insert)

You can use the executemany statement to insert multiple rows at once.

Consider the following code:

```
import sqlite3

con = sqlite3.connect('mydatabase.db')

cursorObj = con.cursor()

cursorObj.execute('create table if not exists projects(id integer, name
text)')

data = [(1, "Ridesharing"), (2, "Water Purifying"), (3, "Forensics"), (4,
"Botany")]

cursorObj.executemany("INSERT INTO projects VALUES(?, ?)", data)
```

```
con.commit()
```

Here we have created a table with two columns, then “data” has four values for each column. This variable is passed to the *executemany()* method along with the query.

Note that we have used the placeholder to pass the values.

The above code will generate the following result:

Close Connection

Once you are done with your database, it is a good practice to close the connection. The connection can be closed by using the *close()* method.

To close a connection, use the connection object and call the *close()* method as follows:

```
con = sqlite3.connect('mydatabase.db')

#program statements

con.close()
```

SQLite3 datetime

In Python SQLite3 database, we can easily store date or time by importing the *datatime* module. The following formats are the most commonly used formats for datetime:

```
YYYY-MM-DD

YYYY-MM-DD HH:MM

YYYY-MM-DD HH:MM:SS

YYYY-MM-DD HH:MM:SS.SSS

HH:MM

HH:MM:SS

HH:MM:SS.SSS

now
```

Consider the following code:

```
import sqlite3

import datetime

con = sqlite3.connect('mydatabase.db')
```

```
cursorObj = con.cursor()

cursorObj.execute('create table if not exists assignments(id integer, name
text, date date)')

data = [(1, "Ridesharing", datetime.date(2017, 1, 2)), (2, "Water Purifying",
datetime.date(2018, 3, 4))]

cursorObj.executemany("INSERT INTO assignments VALUES(?, ?, ?)", data)

con.commit()
```

In this code, the datetime module is imported first and we have created a table named assignments with three columns.

The data type of the third column is a date. To insert the date in the column, we have used *datetime.date*. Similarly, we can use *datetime.time* to handle time.

The above code will generate the following output:

The great flexibility and mobility of the SQLite3 database make it the first choice for any developer to use it and ship it with any product he works with.

SQLite3 databases are used in Windows, Linux, Mac OS, Android, and iOS projects due to its awesome portability. So one file is shipped with your project and it’s done.

I hope you find the tutorial useful. Keep coming back.

Thank you.