

Database CRUD Operations in Python

In this tutorial you will learn how to perform CRUD operations in Python with the SQLite database. Python has built-in support for SQLite in the form of the `sqlite3` module. This module contains functions for performing persistent CRUD operations on SQLite database.

Sqlite Database

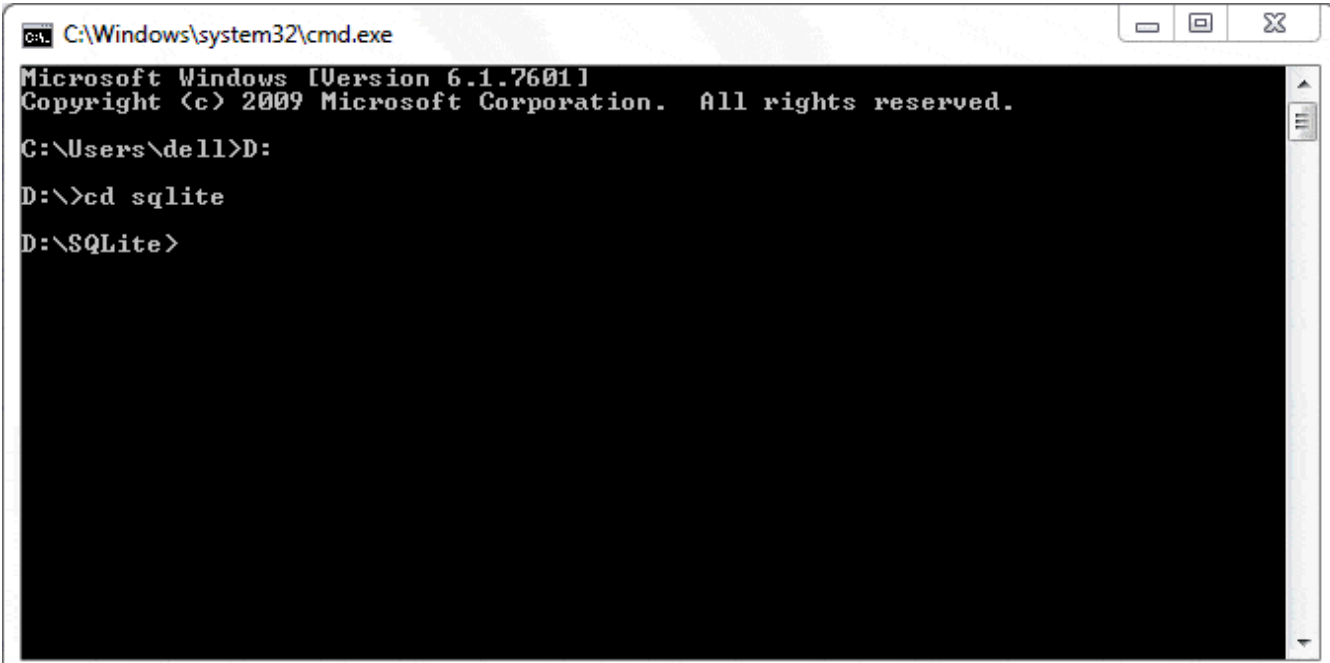
SQLite is a self-contained transactional relational database engine that doesn't require a server configuration, as in the case of Oracle, MySQL, etc. It is an open source and in-process library developed by D. Richard Hipp in August 2000. The entire SQLite database is contained in a single file, which can be put anywhere in the computer's file system.

SQLite is widely used as an embedded database in mobile devices, web browsers and other stand-alone applications. In spite of being small in size, it is a fully ACID compliant database conforming to ANSI SQL standards.

SQLite is freely downloadable from the official web site <https://www.sqlite.org/download.html>. This page contains pre-compiled binaries for all major operating systems. A bundle of command-line tools contain command-line shell and other utilities to manage SQLite database files.

We shall download the latest version of SQLite (version 3.25.1) along with command-line tools and extract the archive.

To create a new SQLite database, navigate from the command prompt to the folder where you have unzipped the archive and enter the following command:



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\dell>D:
D:\>cd sqlite
D:\SQLite>
```

Sqlite3 Command

It is now possible to execute any SQL query. The following statement creates a new table. (Ensure that the statement ends with a semicolon)

```
sqlite> create table student(name text, age int, marks real);
```

Add a record in the above table.

```
sqlite> insert into student values('Ramesh', 21, 55.50);
```

To retrieve the record, use the SELECT query as below:

```
sqlite> select * from student;
Ramesh|21|55.5
```

Python DB-API

[Python Database API](#) is a set of standards recommended by a Special Interest Group for database module standardization. Python modules that provide database interfacing functionality with all major database products are required to adhere to this standard. DB-API standards were further modified to DB-API 2.0 by another [Python Enhancement proposal](#) (PEP-249).

Standard Python distribution has in-built support for SQLite database connectivity. It contains sqlite3 module which adheres to DB-API 2.0 and is written by Gerhard Haring. Other RDBMS products also have DB-API compliant modules:

- > MySQL: [PyMySQL module](#)
- > Oracle: [Cx-Oracle module](#)
- > SQL Server: [PyMsSql module](#)
- > PostgreSQL: [psycopg2 module](#)
- > ODBC: [pyodbc module](#)

As per the prescribed standards, the first step in the process is to obtain the connection to the object representing the database. In order to establish a connection with a SQLite database, sqlite3 module needs to be imported and the `connect()` function needs to be executed.

```
>>> import sqlite3
>>> db=sqlite3.connect('test.db')
```

The `connect()` function returns a connection object referring to the existing database or a new database if it doesn't exist.

The following methods are defined in the connection class:

Method	Description
<code>cursor()</code>	Returns a Cursor object which uses this Connection.
<code>commit()</code>	Explicitly commits any pending transactions to the database. The method should be a no-op if the underlying db does not support transactions.
<code>rollback()</code>	This optional method causes a transaction to be rolled back to the starting point. It may not be implemented everywhere.
<code>close()</code>	Closes the connection to the database permanently. Attempts to use the connection after calling this method will raise a DB-API Error.

A cursor is a Python object that enables you to work with the database. It acts as a handle for a given SQL query; it allows the retrieval of one or more rows of the result. Hence, a cursor object is obtained from the connection to execute SQL queries using the following statement:

```
>>> cur=db.cursor()
```

The following methods of the cursor object are useful.

Method	Description
execute()	Executes the SQL query in a string parameter
executemany()	Executes the SQL query using a set of parameters in the list of tuples
fetchone()	Fetches the next row from the query result set.
fetchall()	Fetches all remaining rows from the query result set.
callproc()	Calls a stored procedure.
close()	Closes the cursor object.

The `commit()` and `rollback()` methods of the connection class ensure transaction control. The `execute()` method of the cursor receives a string containing the SQL query. A string having an incorrect SQL query raises an exception, which should be properly handled. That's why the `execute()` method is placed within the try block and the effect of the SQL query is persistently saved using the `commit()` method. If however, the SQL query fails, the resulting exception is processed by the except block and the pending transaction is undone using the `rollback()` method.

Typical use of the `execute()` method is as follows:

Example:

Copy

```
try:
    cur=db.cursor()
    cur.execute("Query")
    db.commit()
    print ("success message")
except:
    print ("error")
    db.rollback()
db.close()
```

A Look Ahead at Xbox's Plans for 2022

1.2M50

[]

number of...[Watch Video](#)

How John Madden...
How John Madden Became the, Biggest Name in...[Watch Video](#)

Riot Games To Pay...
Riot Games To Pay \$100 Million , To Settle Gender...[Watch Video](#)

'Final Fantasy...
Final Fantasy XVI Development Delayed by Nearly 6...[Watch Video](#)

00:16/01:27

10 Sec

Create a New Table

A string enclosing the CREATE TABLE query is passed as parameter to the `execute()` method of the cursor object. The following code creates the student table in the test.db database.

Example: Create a New Table in Sqlite

Copy

```
import sqlite3
db=sqlite3.connect('test.db')
try:
```

```

cur =db.cursor()
cur.execute('''CREATE TABLE student (
StudentID INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT (20) NOT NULL,
age INTEGER,
marks REAL);''')
print ('table created successfully')
except:
    print ('error in operation')
    db.rollback()
db.close()

```

This can be verified using the `.tables` command in sqlite shell.

```

E:\SQLite>sqlite3 test.db
SQLite version 3.25.1 2018-09-18 20:20:44
Enter ".help" for usage hints.
sqlite> .tables
student

```

Insert a Record

Once again, the `execute()` method of the cursor object should be called with a string argument representing the INSERT query syntax. We have created a student table having three fields: name, age and marks. The string holding the INSERT query is defined as:

```
qry="INSERT INTO student (name, age, marks) VALUES ('Rajeev',20,50);"
```

We have to use it as a parameter to the `execute()` method. To account for possible exceptions, the `execute()` statement is placed in the try block as explained earlier. The complete code for the inset operation is as follows:

Example: Insert a Record in Sqlite

 Copy

```

import sqlite3
db=sqlite3.connect('test.db')
qry="insert into student (name, age, marks) values('Rajeev', 20, 50);"
try:
    cur=db.cursor()
    cur.execute(qry)
    db.commit()
    print ("one record added successfully")
except:
    print ("error in operation")
    db.rollback()
db.close()

```

You can check the result by using the SELECT query in Sqlite shell.

```

sqlite> select * from student;
1|Rajeev|20|50.0

```

Using Parameters in a Query

Often, the values of Python variables need to be used in SQL operations. One way is to use Python's string `format()` function to put Python data in a string. However, this may lead to SQL injection attacks to your program. Instead, use parameter substitution as recommended in Python DB-API. The `?` character is used as a placeholder in the query string and provides the values in the form of a tuple in the `execute()` method. The following example inserts a record using the parameter substitution method:

Example:	 Copy
<pre>import sqlite3 db=sqlite3.connect('test.db') qry="insert into student (name, age, marks) values(?,?,?);" try: cur=db.cursor() cur.execute(qry, ('Vijaya', 16,75)) db.commit() print ("one record added successfully") except: print("error in operation") db.rollback() db.close()</pre>	

The `executemany()` method is used to add multiple records at once. Data to be added should be given in a list of tuples, with each tuple containing one record. The list object (containing tuples) is the parameter of the `executemany()` method, along with the query string.


Example:	 Copy
<pre>import sqlite3 db=sqlite3.connect('test.db') qry="insert into student (name, age, marks) values(?,?,?);" students=[('Amar', 18, 70), ('Deepak', 25, 87)] try: cur=db.cursor() cur.executemany(qry, students) db.commit() print ("records added successfully") except: print ("error in operation") db.rollback() db.close()</pre>	

Retrieve Records

When the query string holds a `SELECT` query, the `execute()` method forms a result set object containing the records returned. Python DB-API defines two methods to fetch the records:

- 1. `fetchone()`: Fetches the next available record from the result set. It is a tuple consisting of values of each column of the fetched record.
- 2. `fetchall()`: Fetches all remaining records in the form of a list of tuples. Each tuple corresponds to one record and contains values of each column in the table.

When using the `fetchone()` method, use a loop to iterate through the result set, as below:

Example: Fetch Records	 Copy
<pre>import sqlite3 db=sqlite3.connect('test.db')</pre>	

```
sql="SELECT * from student;"
cur=db.cursor()
cur.execute(sql)
while True:
    record=cur.fetchone()
    if record==None:
        break
    print (record)
db.close()
```

When executed, the following output is displayed in the Python shell:

Result:

```
(1, 'Rajeev', 20, 50.0)
(2, 'Vijaya', 16, 75.0)
(3, 'Amar', 18, 70.0)
(4, 'Deepak', 25, 87.0)
```

The `fetchall()` method returns a list of tuples, each being one record.

Example:

 Copy

```
students=cur.fetchall()
for rec in students:
    print (rec)
```

Update a Record

The query string in the `execute()` method should contain an UPDATE query syntax. To update the value of 'age' to 17 for 'Amar', define the string as below:

```
qry="update student set age=17 where name='Amar';"
```

You can also use the substitution technique to pass the parameter to the UPDATE query.

Example: Update Record

 Copy

```
import sqlite3
db=sqlite3.connect('test.db')
qry="update student set age=? where name=?"
try:
    cur=db.cursor()
    cur.execute(qry, (19, 'Deepak'))
    db.commit()
    print("record updated successfully")
except:
    print("error in operation")
    db.rollback()
db.close()
```

Delete a Record

The query string should contain the DELETE query syntax. For example, the below code is used to delete 'Bill' from the student table.