



Python Data Access

tutorialspoint

SIMPLY EASY LEARNING



www.tutorialspoint.com

 <https://www.facebook.com/tutorialspointindia>

 <https://twitter.com/tutorialspoint>

About the Tutorial

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985-1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). This tutorial gives enough understanding on Python programming language.

It provides various modules to communicate with various databases. In this tutorial, we are going to discuss python modules to communicate with the databases MySQL, PostgreSQL, SQLite and, MongoDB.

Audience

This tutorial is designed for python programmers who would like to understand the mysql-connector-python, pycog2, Python sqlite3 module and, pymongo modules in detail.

Prerequisites

Before proceeding with this tutorial, you should have a good understanding of python programming language. It is also recommended to have basic understanding of the databases — MySQL, PostgreSQL, SQLite, MongoDB.

Copyright & Disclaimer

© Copyright 2019 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

About the Tutorial	i
Audience.....	i
Prerequisites.....	i
Copyright & Disclaimer	i
Table of Contents	ii
PYTHON MYSQL	1
1. Python MySQL — Introduction	2
What is mysql-connector-python?	2
Installing python from scratch.....	4
2. Python MySQL — Database Connection.....	7
Establishing connection with MySQL using python.....	8
3. Python MySQL — Create Database	10
Creating a database in MySQL using python	10
4. Python MySQL — Create Table	12
Creating a table in MySQL using python	13
5. Python MySQL — Insert Data.....	15
Inserting data in MySQL table using python.....	15
6. Python MySQL — Select Data	19
Reading data from a MYSQL table using Python	20
7. Python MySQL — Where Clause	23
WHERE clause using python	24
8. Python MySQL — Order By	26
ORDER BY clause using python.....	27
9. Python MySQL — Update Table	30
Updating the contents of a table using Python.....	31
10. Python MySQL - Delete Data.....	33

Removing records of a table using python..... 34

11. Python MySQL — Drop Table36

 Removing a table using python 37

12. Python MySQL — Limit40

 Limit clause using python 41

13. Python MySQL — Join43

 MYSQL JOIN using python 44

14. Python MySQL - Cursor Object46

PYTHON POSTGRESQL..... 48

15. Python PostgreSQL — Introduction.....50

16. Python PostgreSQL — Database Connection.....52

 Establishing connection using python 52

17. Python PostgreSQL — Create Database54

 Creating a database using python 55

18. Python PostgreSQL - Create Table.....56

 Creating a table using python..... 57

19. Python PostgreSQL — Insert Data59

 Inserting data using python..... 60

20. Python PostgreSQL — Select Data.....63

 Retrieving data using python..... 64

21. Python PostgreSQL — Where Clause.....67

 Where clause using python 68

22. Python PostgreSQL — Order By70

 ORDER BY clause using python..... 72

23. Python PostgreSQL — Update Table74

 Updating records using python 75

24. Python PostgreSQL — Delete Data.....78

Deleting data using python 79

25. Python PostgreSQL — Drop Table 82

 Removing an entire table using Python..... 83

26. Python PostgreSQL – Limit 85

 Limit clause using python 86

27. Python PostgreSQL — Join 88

 Joins using python 89

28. Python PostgreSQL — Cursor Object 91

PYTHON SQLITE..... 93

29. Python SQLite — Introduction 94

30. Python SQLite — Establishing Connection..... 97

 Establishing connection using python 97

31. Python SQLite — Create Table 98

 Creating a table using python..... 99

32. Python SQLite — Insert Data..... 101

 Inserting data using python..... 102

33. Python SQLite — Select Data 104

 Retrieving data using python..... 106

34. Python SQLite — Where Clause 108

 Where clause using python 109

35. Python SQLite — Order By 111

 ORDER BY clause using python..... 113

36. Python SQLite — Update Table 115

 Updating existing records using python 116

37. Python SQLite — Delete Data 119

 Deleting data using python 120

38. Python SQLite — Drop Table..... 122

Dropping a table using Python 123

39. Python SQLite — Limit 124

 LIMIT clause using Python 125

40. Python SQLite — Join 127

 Join clause using python..... 128

41. Python SQLite — Cursor Object..... 130

PYTHON MONGODB 132

42. Python MongoDB — Introduction 133

 Installation 133

43. Python MongoDB — Create Database..... 134

 Creating database using python 134

44. Python MongoDB — Create Collection..... 136

 Creating a collection using python 136

45. Python MongoDB — Insert Document 138

 Creating a collection using python 139

46. Python MongoDB — Find..... 142

 Retrieving data (find) using python 143

47. Python MongoDB — Query..... 146

48. Python MongoDB — Sort 149

 Sorting the documents using python 150

49. Python MongoDB — Delete Document 152

 Deleting documents using python..... 153

50. Python MongoDB — Drop Collection 157

 Dropping collection using python..... 157

51. Python MongoDB — Update 159

 Updating documents using python 160

52. Python MongoDB — Limit..... 164

Limiting the documents using python..... 165

Python MySQL

1. Python MySQL — Introduction

The Python standard for database interfaces is the Python DB-API. Most Python database interfaces adhere to this standard.

You can choose the right database for your application. Python Database API supports a wide range of database servers such as:

- GadFly
- mSQL
- MySQL
- PostgreSQL
- Microsoft SQL Server 2000
- Informix
- Interbase
- Oracle
- Sybase

Here is the list of available Python database interfaces: [Python Database Interfaces and APIs](#). You must download a separate DB API module for each database you need to access. For example, if you need to access an Oracle database as well as a MySQL database, you must download both the Oracle and the MySQL database modules.

What is mysql-connector-python?

MySQL Python/Connector is an interface for connecting to a MySQL database server from Python. It implements the Python Database API and is built on top of the MySQL.

How do I Install mysql-connector-python?

First of all, you need to make sure you have already installed python in your machine. To do so, open command prompt and type python in it and press *Enter*. If python is already installed in your system, this command will display its version as shown below:

```
C:\Users\Tutorialspoint>python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Now press *ctrl+z* and then *Enter* to get out of the python shell and create a folder (in which you intended to install Python-MySQL connector) named Python_MySQL as:

```
>>> ^Z
```

```
C:\Users\Tutorialspoint>d:
D:\>mkdir Python_MySQL
```

Verify PIP

PIP is a package manager in python using which you can install various modules/packages in Python. Therefore, to install Mysql-python mysql-connector-python you need to make sure that you have PIP installed in your computer and have its location added to path.

You can do so, by executing the pip command. If you didn't have PIP in your system or, if you haven't added its location in the **Path** environment variable, you will get an error message as:

```
D:\Python_MySQL>pip
'pip' is not recognized as an internal or external command,
operable program or batch file.
```

To install PIP, download the [get-pip.py](#) to the above created folder and, from command navigate it and install pip as follows:

```
D:\>cd Python_MySQL
D:\Python_MySQL>python get-pip.py
Collecting pip
  Downloading
https://files.pythonhosted.org/packages/8d/07/f7d7ced2f97ca3098c16565efbe6b15fa
fcb53e8d9bdb431e09140514b0/pip-19.2.2-py2.py3-none-any.whl (1.4MB)
|██████████████████████████████████████████████████████████████████████████████| 1.4MB 1.3MB/s
Collecting wheel
  Downloading
https://files.pythonhosted.org/packages/00/83/b4a77d044e78ad1a45610eb88f745be2f
d2c6d658f9798a15e384b7d57c9/wheel-0.33.6-py2.py3-none-any.whl
Installing collected packages: pip, wheel
  Consider adding this directory to PATH or, if you prefer to suppress this
warning, use --no-warn-script-location.
Successfully installed pip-19.2.2 wheel-0.33.6
```

Installing mysql-connector-python

Once you have Python and PIP installed, open command prompt and upgrade pip (optional) as shown below:

```
C:\Users\Tutorialspoint>python -m pip install --upgrade pip
Collecting pip
  Using cached
https://files.pythonhosted.org/packages/8d/07/f7d7ced2f97ca3098c16565efbe6b15fa
fcb53e8d9bdb431e09140514b0/pip-19.2.2-py2.py3-none-any.whl
```

```
Installing collected packages: pip
  Found existing installation: pip 19.0.3
    Uninstalling pip-19.0.3:
      Successfully uninstalled pip-19.0.3
Successfully installed pip-19.2.2
```

Then open command prompt in admin mode and install python MySQL connect as:

```
C:\WINDOWS\system32>pip install mysql-connector-python
Collecting mysql-connector-python
  Using cached
https://files.pythonhosted.org/packages/99/74/f41182e6b7aad62b038b6939dce784b7f9ec4f89e2ae14f9ba8190dc9ab/mysql_connector_python-8.0.17-py2.py3-none-any.whl
Collecting protobuf>=3.0.0 (from mysql-connector-python)
  Using cached
https://files.pythonhosted.org/packages/09/0e/614766ea191e649216b87d331a4179338c623e08c0cca291bcf8638730ce/protobuf-3.9.1-cp37-cp37m-win32.whl
Collecting six>=1.9 (from protobuf>=3.0.0->mysql-connector-python)
  Using cached
https://files.pythonhosted.org/packages/73/fb/00a976f728d0d1fecfe898238ce23f502a721c0ac0ecfedb80e0d88c64e9/six-1.12.0-py2.py3-none-any.whl
Requirement already satisfied: setuptools in c:\program files (x86)\python37-32\lib\site-packages (from protobuf>=3.0.0->mysql-connector-python) (40.8.0)
Installing collected packages: six, protobuf, mysql-connector-python
Successfully installed mysql-connector-python-8.0.17 protobuf-3.9.1 six-1.12.0
```

Verification

To verify the installation of the create a sample python script with the following line in it.

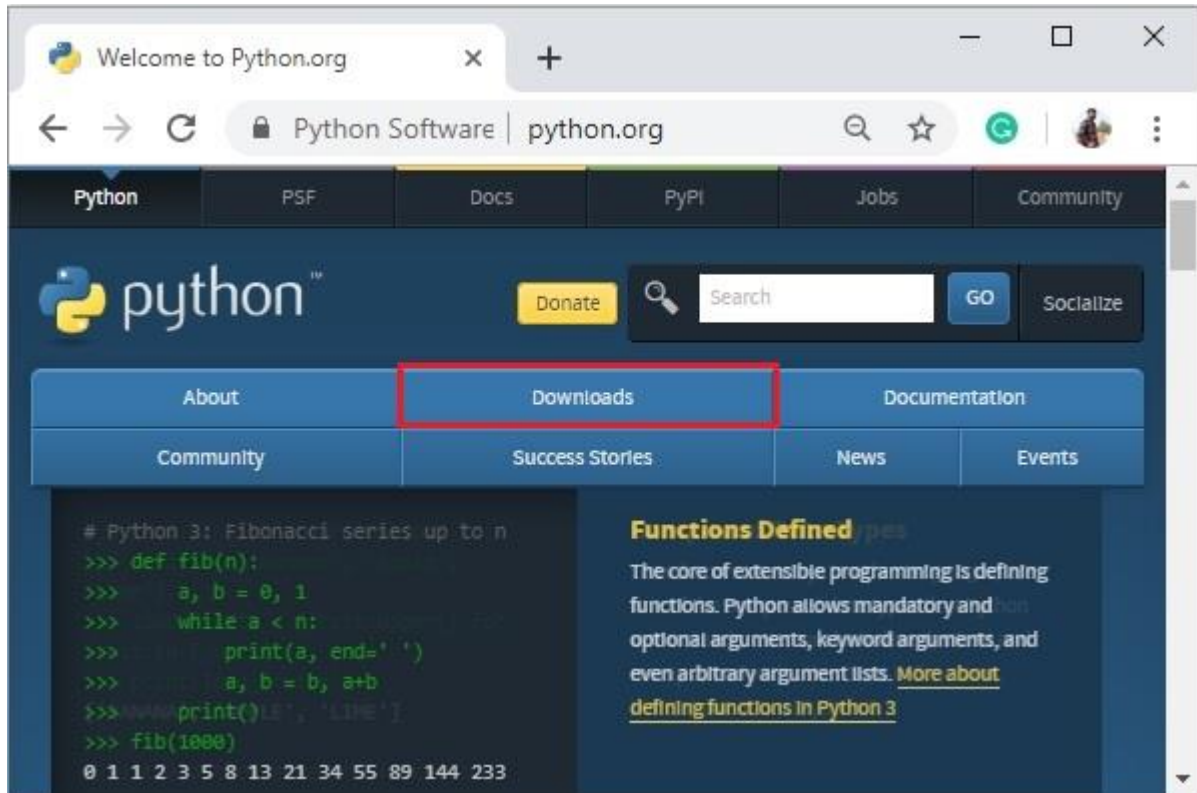
```
import mysql.connector
```

If the installation is successful, when you execute it, you should not get any errors:

```
D:\Python_MySQL>python test.py
D:\Python_MySQL>
```

Installing python from scratch

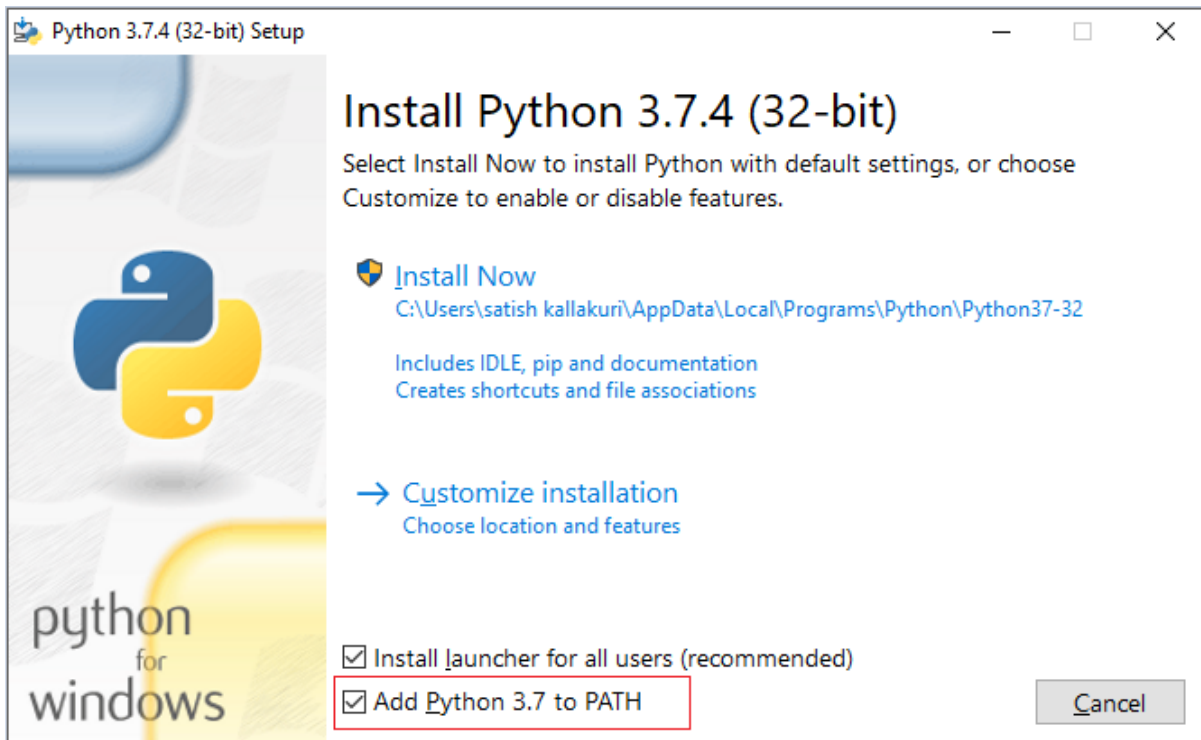
Simply, if you need to install Python from scratch. Visit the [Python Home Page](#).



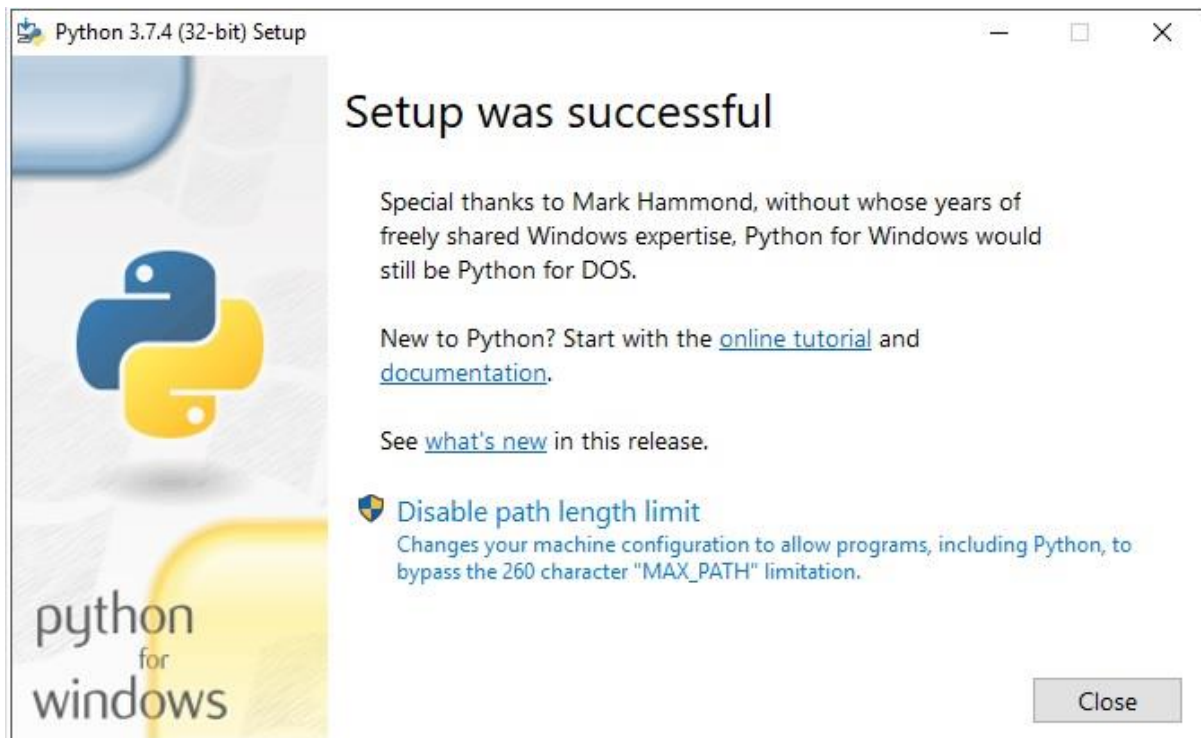
Click on the **Downloads** button, you will be redirected to the downloads page which provides links for latest version of python for various platforms choose one and download it.



For instance, we have downloaded python-3.7.4.exe (for windows). Start the installation process by double-clicking the downloaded .exe file.

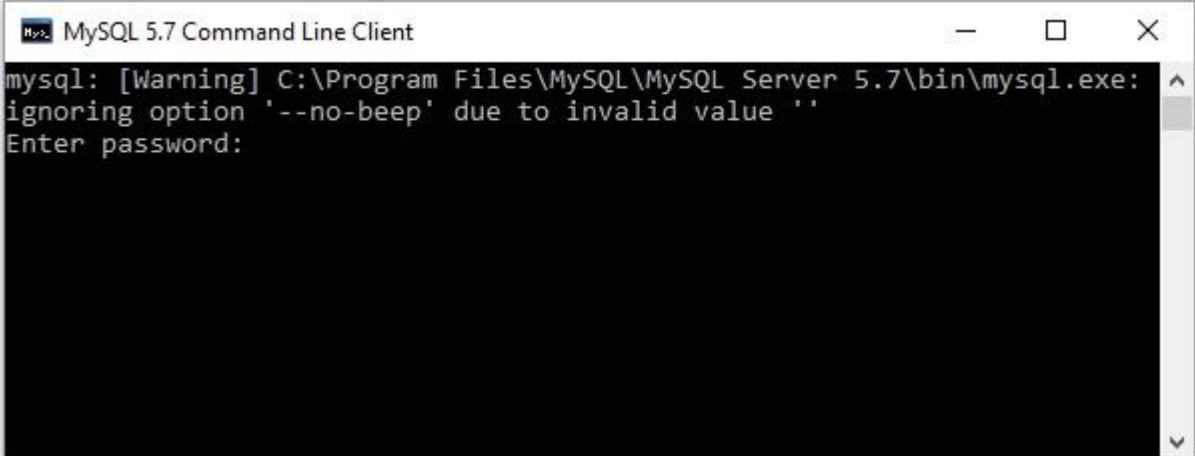


Check the Add Python 3.7 to Path option and proceed with the installation. After completion of this process, python will be installed in your system.



2. Python MySQL — Database Connection

To connect with MySQL, (one way is to) open the MySQL command prompt in your system as shown below:



```
MySQL 5.7 Command Line Client
mysql: [Warning] C:\Program Files\MySQL\MySQL Server 5.7\bin\mysql.exe:
ignoring option '--no-beep' due to invalid value ''
Enter password:
```

It asks for password here; you need to type the password you have set to the default user (root) at the time of installation.

Then a connection is established with MySQL displaying the following message:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.7.12-log MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

You can disconnect from the MySQL database any time using the **exit** command at *mysql>* prompt.

```
mysql> exit
Bye
```

Establishing connection with MySQL using python

Before establishing connection to MySQL database using python, assume:

- That we have created a database with name mydb.
- We have created a table EMPLOYEE with columns FIRST_NAME, LAST_NAME, AGE, SEX and INCOME.
- The credentials we are using to connect with MySQL are username: **root**, password: **password**.

You can establish a connection using the **connect()** constructor. This accepts username, password, host and, name of the database you need to connect with (optional) and, returns an object of the MySQLConnection class.

Example

Following is the example of connecting with MySQL database "mydb".

```
import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Executing an MYSQL function using the execute() method
cursor.execute("SELECT DATABASE()")

# Fetch a single row using fetchone() method.
data = cursor.fetchone()

print("Connection established to: ",data)

#Closing the connection
conn.close()
```

On executing, this script produces the following output:

```
D:\Python_MySQL>python EstablishCon.py
Connection established to: ('mydb',)
```


You can also establish connection to MySQL by passing credentials (user name, password, hostname, and database name) to ***connection.MySQLConnection()*** as shown below:

```
from mysql.connector import (connection)

#establishing the connection
conn = connection.MySQLConnection(user='root', password='password',
host='127.0.0.1', database='mydb')

#Closing the connection
conn.close()
```


3. Python MySQL — Create Database

You can create a database in MYSQL using the CREATE DATABASE query.

Syntax

Following is the syntax of the CREATE DATABASE query:

```
CREATE DATABASE name_of_the_database
```

Example

Following statement creates a database with name mydb in MySQL:

```
mysql> CREATE DATABASE mydb;  
Query OK, 1 row affected (0.04 sec)
```

If you observe the list of databases using the SHOW DATABASES statement, you can observe the newly created database in it as shown below:

```
mysql> SHOW DATABASES;  
+-----+  
| Database          |  
+-----+  
| information_schema |  
| logging           |  
| mydatabase        |  
| mydb             |  
| performance_schema |  
| students          |  
| sys               |  
+-----+  
26 rows in set (0.15 sec)
```

Creating a database in MySQL using python

After establishing connection with MySQL, to manipulate data in it you need to connect to a database. You can connect to an existing database or, create your own.

You would need special privileges to create or to delete a MySQL database. So if you have access to the root user, you can create any database.

Example

Following example establishes connection with MYSQL and creates a database in it.

```
import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Doping database MYDATABASE if already exists.
cursor.execute("DROP database IF EXISTS MyDatabase")

#Preparing query to create a database
sql = "CREATE database MYDATABASE";

#Creating a database
cursor.execute(sql)

#Retrieving the list of databases
print("List of databases: ")
cursor.execute("SHOW DATABASES")
print(cursor.fetchall())

#Closing the connection
conn.close()
```

Output

```
List of databases:
[('information_schema',), ('dbbug61332',), ('details',), ('exampledatabase',),
 ('mydatabase',), ('mydb',), ('mysql',), ('performance_schema',)]
```

4. Python MySQL — Create Table

The CREATE TABLE statement is used to create tables in MYSQL database. Here, you need to specify the name of the table and, definition (name and datatype) of each column.

Syntax

Following is the syntax to create a table in MySQL:

```
CREATE TABLE table_name(  
    column1 datatype,  
    column2 datatype,  
    column3 datatype,  
    .....  
    columnN datatype,  
);
```

Example

Following query creates a table named EMPLOYEE in MySQL with five columns namely, FIRST_NAME, LAST_NAME, AGE, SEX and, INCOME.

```
mysql> CREATE TABLE EMPLOYEE(  
    FIRST_NAME CHAR(20) NOT NULL,  
    LAST_NAME CHAR(20),  
    AGE INT,  
    SEX CHAR(1),  
    INCOME FLOAT);  
Query OK, 0 rows affected (0.42 sec)
```

The DESC statement gives you the description of the specified table. Using this you can verify if the table has been created or not as shown below:

```
mysql> Desc Employee;  
+-----+-----+-----+-----+-----+-----+  
| Field      | Type      | Null | Key | Default | Extra |  
+-----+-----+-----+-----+-----+-----+  
| FIRST_NAME | char(20)  | NO   |     | NULL    |       |  
| LAST_NAME  | char(20)  | YES  |     | NULL    |       |  
| AGE        | int(11)   | YES  |     | NULL    |       |  
| SEX        | char(1)   | YES  |     | NULL    |       |
```

12

```
| INCOME      | float  | YES |      | NULL  |      |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.07 sec)
```

Creating a table in MySQL using python

The method named **execute()** (invoked on the cursor object) accepts two variables:

- A String value representing the query to be executed.
- An optional args parameter which can be a tuple or, list or, dictionary, representing the parameters of the query (values of the place holders).

It returns an integer value representing the number of rows effected by the query.

Once a database connection is established, you can create tables by passing the CREATE TABLE query to the **execute()** method.

In short, to create a table using python:

- Import **mysql.connector** package.
- Create a connection object using the **mysql.connector.connect()** method, by passing the user name, password, host (optional default: localhost) and, database (optional) as parameters to it.
- Create a cursor object by invoking the **cursor()** method on the connection object created above.
- Then, execute the *CREATE TABLE* statement by passing it as a parameter to the **execute()** method.

Example

Following example creates a table named **Employee** in the database mydb.

```
import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Dropping EMPLOYEE table if already exists.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

#Creating table as per requirement
```

```
sql = '''CREATE TABLE EMPLOYEE(  
    FIRST_NAME CHAR(20) NOT NULL,  
    LAST_NAME CHAR(20),  
    AGE INT,  
    SEX CHAR(1),  
    INCOME FLOAT)'''  
cursor.execute(sql)  
  
#Closing the connection  
conn.close()
```

5. Python MySQL — Insert Data

You can add new rows to an existing table of MySQL using the **INSERT INTO** statement. In this, you need to specify the name of the table, column names, and values (in the same order as column names).

Syntax

Following is the syntax of the INSERT INTO statement of MySQL.

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)
VALUES (value1, value2, value3,...valueN);
```

Example

Following query inserts a record into the table named EMPLOYEE.

```
INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME) VALUES ('Mac',
'Mohan', 20, 'M', 2000);
```

You can verify the records of the table after insert operation using the SELECT statement as:

```
mysql> select * from Employee;
+-----+-----+-----+-----+-----+
| FIRST_NAME | LAST_NAME | AGE | SEX | INCOME |
+-----+-----+-----+-----+-----+
| Mac      | Mohan    | 20 | M   | 2000   |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

It is not mandatory to specify the names of the columns always, if you pass values of a record in the same order of the columns of the table you can execute the SELECT statement without the column names as follows:

```
INSERT INTO EMPLOYEE VALUES ('Mac', 'Mohan', 20, 'M', 2000);
```

Inserting data in MySQL table using python

The **execute()** method (invoked on the cursor object) accepts a query as parameter and executes the given query. To insert data, you need to pass the MySQL INSERT statement as a parameter to it.

```
cursor.execute("""INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)
VALUES ('Mac', 'Mohan', 20, 'M', 2000)""")
```

To insert data into a table in MySQL using python:

- import **mysql.connector** package.
- Create a connection object using the **mysql.connector.connect()** method, by passing the user name, password, host (optional default: localhost) and, database (optional) as parameters to it.
- Create a cursor object by invoking the **cursor()** method on the connection object created above.
- Then, execute the **INSERT** statement by passing it as a parameter to the **execute()** method.

Example

The following example executes SQL *INSERT* statement to insert a record into the EMPLOYEE table:

```
import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

# Preparing SQL query to INSERT a record into the database.
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
        LAST_NAME, AGE, SEX, INCOME)
        VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""

try:
    # Executing the SQL command
    cursor.execute(sql)
    # Commit your changes in the database
    conn.commit()
except:
    # Rolling back in case of error
    conn.rollback()
```

```
# Closing the connection
conn.close()
```

Inserting values dynamically

You can also use “%s” instead of values in the **INSERT** query of MySQL and pass values to them as lists as shown below:

```
cursor.execute("""INSERT INTO EMPLOYEE VALUES ('Mac', 'Mohan', 20, 'M',
2000)""", ('Ramya', 'Ramapriya', 25, 'F', 5000))
```

Example

Following example inserts a record into the Employee table dynamically.

```
import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

# Preparing SQL query to INSERT a record into the database.
insert_stmt = (
    "INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)"
    "VALUES (%s, %s, %s, %s, %s)"
)
data = ('Ramya', 'Ramapriya', 25, 'F', 5000)

try:
    # Executing the SQL command
    cursor.execute(insert_stmt, data)
    # Commit your changes in the database
    conn.commit()
except:
    # Rolling back in case of error
    conn.rollback()

print("Data inserted")
```



```
# Closing the connection  
conn.close()
```

Output

```
Data inserted
```

6. Python MySQL — Select Data

You can retrieve/fetch data from a table in MySQL using the SELECT query. This query/statement returns contents of the specified table in tabular form and it is called as result-set.

Syntax

Following is the syntax of the SELECT query:

```
SELECT column1, column2, columnN FROM table_name;
```

Example

Assume we have created a table in MySQL with name *cricketers_data* as:

```
CREATE TABLE cricketers_data(  
  First_Name VARCHAR(255),  
  Last_Name VARCHAR(255),  
  Date_Of_Birth date,  
  Place_Of_Birth VARCHAR(255),  
  Country VARCHAR(255)  
);
```

And if we have inserted 5 records in to it using INSERT statements as:

```
insert into cricketers_data values('Shikhar', 'Dhawan', DATE('1981-12-05'),  
'Delhi', 'India');  
insert into cricketers_data values('Jonathan', 'Trott', DATE('1981-04-22'),  
'CapeTown', 'SouthAfrica');  
insert into cricketers_data values('Kumara', 'Sangakkara', DATE('1977-10-27'),  
'Matale', 'Srilanka');  
insert into cricketers_data values('Virat', 'Kohli', DATE('1988-11-05'),  
'Delhi', 'India');  
insert into cricketers_data values('Rohit', 'Sharma', DATE('1987-04-30'),  
'Nagpur', 'India');
```

Following query retrieves the FIRST_NAME and Country values from the table.

```
mysql> select FIRST_NAME, Country from cricketers_data;  
+-----+-----+  
| FIRST_NAME | Country |
```

```

+-----+-----+
| Shikhar | India |
| Jonathan | SouthAfrica |
| Kumara | Srilanka |
| Virat | India |
| Rohit | India |
+-----+-----+
5 rows in set (0.00 sec)

```

You can also retrieve all the values of each record using * instated of the name of the columns as:

```

mysql> SELECT * from cricketers_data;
+-----+-----+-----+-----+-----+
| First_Name | Last_Name | Date_Of_Birth | Place_Of_Birth | Country |
+-----+-----+-----+-----+-----+
| Shikhar | Dhawan | 1981-12-05 | Delhi | India |
| Jonathan | Trott | 1981-04-22 | CapeTown | SouthAfrica |
| Kumara | Sangakkara | 1977-10-27 | Matale | Srilanka |
| Virat | Kohli | 1988-11-05 | Delhi | India |
| Rohit | Sharma | 1987-04-30 | Nagpur | India |
+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

```

Reading data from a MYSQL table using Python

READ Operation on any database means to fetch some useful information from the database. You can fetch data from MYSQL using the **fetch()** method provided by the mysql-connector-python.

The *cursor.MySQLCursor* class provides three methods namely **fetchall()**, **fetchmany()** and, **fetchone()** where,

- The **fetchall()** method retrieves all the rows in the result set of a query and returns them as list of tuples. (If we execute this after retrieving few rows it returns the remaining ones).
- The **fetchone()** method fetches the next row in the result of a query and returns it as a tuple.
- The **fetchmany()** method is similar to the fetchone() but, it retrieves the next set of rows in the result set of a query, instead of a single row.

Note: A result set is an object that is returned when a cursor object is used to query a table.

rowcount: This is a read-only attribute and returns the number of rows that were affected by an execute() method.

Example

Following example fetches all the rows of the EMPLOYEE table using the SELECT query and from the obtained result set initially, we are retrieving the first row using the fetchone() method and then fetching the remaining rows using the fetchall() method.

```
import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving single row
sql = '''SELECT * from EMPLOYEE'''

#Executing the query
cursor.execute(sql)

#Fetching 1st row from the table
result = cursor.fetchone();
print(result)

#Fetching 1st row from the table
result = cursor.fetchall();
print(result)

#Closing the connection
conn.close()
```

Output

```
('Krishna', 'Sharma', 19, 'M', 2000.0)
[('Raj', 'Kandukuri', 20, 'M', 7000.0), ('Ramya', 'Ramapriya', 25, 'M',
5000.0)]
```

Following example retrieves first two rows of the EMPLOYEE table using the fetchmany() method.

Example

```
import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving single row
sql = '''SELECT * from EMPLOYEE'''

#Executing the query
cursor.execute(sql)

#Fetching 1st row from the table
result = cursor.fetchmany(size =2);
print(result)

#Closing the connection
conn.close()
```

Output

```
[('Krishna', 'Sharma', 19, 'M', 2000.0), ('Raj', 'Kandukuri', 20, 'M', 7000.0)]
```

7. Python MySQL — Where Clause

If you want to fetch, delete or, update particular rows of a table in MySQL, you need to use the where clause to specify condition to filter the rows of the table for the operation.

For example, if you have a SELECT statement with where clause, only the rows which satisfies the specified condition will be retrieved.

Syntax

Following is the syntax of the WHERE clause:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

Example

Assume we have created a table in MySQL with name EMPLOYEES as:

```
mysql> CREATE TABLE EMPLOYEE(
    FIRST_NAME CHAR(20) NOT NULL,
    LAST_NAME CHAR(20),
    AGE INT,
    SEX CHAR(1),
    INCOME FLOAT);
Query OK, 0 rows affected (0.36 sec)
```

And if we have inserted 4 records in to it using INSERT statements as:

```
mysql> INSERT INTO EMPLOYEE VALUES
('Krishna', 'Sharma', 19, 'M', 2000),
('Raj', 'Kandukuri', 20, 'M', 7000),
('Ramya', 'Ramapriya', 25, 'F', 5000),
('Mac', 'Mohan', 26, 'M', 2000);
```

Following MySQL statement retrieves the records of the employees whose income is greater than 4000.

```
mysql> SELECT * FROM EMPLOYEE WHERE INCOME > 4000;
+-----+-----+-----+-----+-----+
| FIRST_NAME | LAST_NAME | AGE | SEX | INCOME |
```

```

+-----+-----+-----+-----+-----+
| Raj      | Kandukuri | 20 | M   | 7000 |
| Ramya    | Ramapriya | 25 | F   | 5000 |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

```

WHERE clause using python

To fetch specific records from a table using the python program:

- import **mysql.connector** package.
- Create a connection object using the **mysql.connector.connect()** method, by passing the user name, password, host (optional default: localhost) and, database (optional) as parameters to it.
- Create a cursor object by invoking the **cursor()** method on the connection object created above.
- Then, execute the *SELECT* statement with *WHERE* clause, by passing it as a parameter to the **execute()** method.

Example

Following example creates a table named Employee and populates it. Then using the where clause it retrieves the records with age value less than 23.

```

import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Doping EMPLOYEE table if already exists.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

sql = '''CREATE TABLE EMPLOYEE(
        FIRST_NAME CHAR(20) NOT NULL,
        LAST_NAME CHAR(20),
        AGE INT,
        SEX CHAR(1),

```

```
        INCOME FLOAT)'''
cursor.execute(sql)

#Populating the table
insert_stmt = "INSERT INTO EMPLOYEE (FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)
VALUES (%s, %s, %s, %s, %s)"
data = [('Krishna', 'Sharma', 19, 'M', 2000), ('Raj', 'Kandukuri', 20, 'M', 7000),
        ('Ramyra', 'Ramapriya', 25, 'F', 5000),('Mac', 'Mohan', 26, 'M', 2000)]
cursor.executemany(insert_stmt, data)
conn.commit()

#Retrieving specific records using the where clause
cursor.execute("SELECT * from EMPLOYEE WHERE AGE <23")
print(cursor.fetchall())

#Closing the connection
conn.close()
```

Output

```
[('Krishna', 'Sharma', 19, 'M', 2000.0), ('Raj', 'Kandukuri', 20, 'M',
7000.0)]
```


8. Python MySQL — Order By

While fetching data using SELECT query, you can sort the results in desired order (ascending or descending) using the OrderBy clause. By default, this clause sorts results in ascending order, if you need to arrange them in descending order you need to use "DESC" explicitly.

Syntax

Following is the syntax SELECT column-list

```
FROM table_name
[WHERE condition]
[ORDER BY column1, column2,.. columnN] [ASC | DESC]; of the ORDER BY clause:
```

Example

Assume we have created a table in MySQL with name EMPLOYEES as:

```
mysql> CREATE TABLE EMPLOYEE(
    FIRST_NAME CHAR(20) NOT NULL,
    LAST_NAME CHAR(20),
    AGE INT,
    SEX CHAR(1),
    INCOME FLOAT);
```

Query OK, 0 rows affected (0.36 sec)

And if we have inserted 4 records in to it using INSERT statements as:

```
mysql> INSERT INTO EMPLOYEE VALUES
('Krishna', 'Sharma', 19, 'M', 2000),
('Raj', 'Kandukuri', 20, 'M', 7000),
('Ramya', 'Ramapriya', 25, 'F', 5000),
('Mac', 'Mohan', 26, 'M', 2000);
```

Following statement retrieves the contents of the EMPLOYEE table in ascending order of the age.

```
mysql> SELECT * FROM EMPLOYEE ORDER BY AGE;
+-----+-----+-----+-----+-----+
| FIRST_NAME | LAST_NAME | AGE | SEX | INCOME |
```

```

+-----+-----+-----+-----+-----+
| Krishna | Sharma | 19 | M | 2000 |
| Raj     | Kandukuri | 20 | M | 7000 |
| Ramya   | Ramapriya | 25 | F | 5000 |
| Mac     | Mohan    | 26 | M | 2000 |
+-----+-----+-----+-----+
4 rows in set (0.04 sec)

```

You can also retrieve data in descending order using DESC as:

```

mysql> SELECT * FROM EMPLOYEE ORDER BY FIRST_NAME, INCOME DESC;
+-----+-----+-----+-----+-----+
| FIRST_NAME | LAST_NAME | AGE | SEX | INCOME |
+-----+-----+-----+-----+
| Krishna   | Sharma    | 19 | M   | 2000   |
| Mac       | Mohan     | 26 | M   | 2000   |
| Raj       | Kandukuri | 20 | M   | 7000   |
| Ramya     | Ramapriya | 25 | F   | 5000   |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

ORDER BY clause using python

To retrieve contents of a table in specific order, invoke the **execute()** method on the cursor object and, pass the SELECT statement along with ORDER BY clause, as a parameter to it.

Example

In the following example we are creating a table with name and Employee, populating it, and retrieving its records back in the (ascending) order of their age, using the ORDER BY clause.

```

import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

```

```

#Doping EMPLOYEE table if already exists.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")
sql = '''CREATE TABLE EMPLOYEE(
    FIRST_NAME CHAR(20) NOT NULL,
    LAST_NAME CHAR(20),
    AGE INT,
    SEX CHAR(1),
    INCOME FLOAT)'''
cursor.execute(sql)

#Populating the table
insert_stmt = "INSERT INTO EMPLOYEE (FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)
VALUES (%s, %s, %s, %s, %s)"
data = [('Krishna', 'Sharma', 26, 'M', 2000), ('Raj', 'Kandukuri', 20, 'M',
7000),
        ('Ramya', 'Ramapriya', 29, 'F', 5000),('Mac', 'Mohan', 26, 'M', 2000)]
cursor.executemany(insert_stmt, data)
conn.commit()

#Retrieving specific records using the ORDER BY clause
cursor.execute("SELECT * from EMPLOYEE ORDER BY AGE")

print(cursor.fetchall())

#Closing the connection
conn.close()

```

Output

```

[('Raj', 'Kandukuri', 20, 'M', 7000.0), ('Krishna', 'Sharma', 26, 'M', 2000.0),
('Mac', 'Mohan', 26, 'M', 2000.0), ('Ramya', 'Ramapriya', 29, 'F', 5000.0)]

```

In the same way you can retrieve data from a table in descending order using the ORDER BY clause.

Example

```

import mysql.connector

#establishing the connection

```

```
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving specific records using the ORDERBY clause
cursor.execute("SELECT * from EMPLOYEE ORDER BY INCOME DESC")

print(cursor.fetchall())

#Closing the connection
conn.close()
```

Output

```
[('Raj', 'Kandukuri', 20, 'M', 7000.0), ('Ramya', 'Ramapriya', 29, 'F',
5000.0), ('Krishna', 'Sharma', 26, 'M', 2000.0), ('Mac', 'Mohan', 26, 'M',
2000.0)]
```

9. Python MySQL — Update Table

UPDATE Operation on any database updates one or more records, which are already available in the database. You can update the values of existing records in MySQL using the UPDATE statement. To update specific rows, you need to use the WHERE clause along with it.

Syntax

Following is the syntax of the UPDATE statement in MySQL:

```
UPDATE table_name
SET column1 = value1, column2 = value2..., columnN = valueN
WHERE [condition];
```

You can combine N number of conditions using the AND or the OR operators.

Example

Assume we have created a table in MySQL with name EMPLOYEES as:

```
mysql> CREATE TABLE EMPLOYEE(
    FIRST_NAME CHAR(20) NOT NULL,
    LAST_NAME CHAR(20),
    AGE INT,
    SEX CHAR(1),
    INCOME FLOAT);
Query OK, 0 rows affected (0.36 sec)
```

And if we have inserted 4 records in to it using INSERT statements as:

```
mysql> INSERT INTO EMPLOYEE VALUES
('Krishna', 'Sharma', 19, 'M', 2000),
('Raj', 'Kandukuri', 20, 'M', 7000),
('Ramya', 'Ramapriya', 25, 'F', 5000),
('Mac', 'Mohan', 26, 'M', 2000);
```

Following MySQL statement increases the age of all male employees by one year:

```
mysql> UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = 'M';
Query OK, 3 rows affected (0.06 sec)
```

```
Rows matched: 3  Changed: 3  Warnings: 0
```

If you retrieve the contents of the table, you can see the updated values as:

```
mysql> select * from EMPLOYEE;
+-----+-----+-----+-----+-----+
| FIRST_NAME | LAST_NAME | AGE | SEX | INCOME |
+-----+-----+-----+-----+
| Krishna   | Sharma    | 20 | M   | 2000   |
| Raj       | Kandukuri | 21 | M   | 7000   |
| Ramya     | Ramapriya | 25 | F   | 5000   |
| Mac       | Mohan     | 27 | M   | 2000   |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

Updating the contents of a table using Python

To update the records in a table in MySQL using python:

- import **mysql.connector** package.
- Create a connection object using the **mysql.connector.connect()** method, by passing the user name, password, host (optional default: localhost) and, database (optional) as parameters to it.
- Create a cursor object by invoking the **cursor()** method on the connection object created above.
- Then, execute the *UPDATE* statement by passing it as a parameter to the **execute()** method.

Example

The following example increases age of all the males by one year.

```
import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()
```

```
#Preparing the query to update the records
sql = '''UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = 'M' '''

try:
    # Execute the SQL command
    cursor.execute(sql)
    # Commit your changes in the database
    conn.commit()
except:
    # Rollback in case there is any error
    conn.rollback()

#Retrieving data
sql = '''SELECT * from EMPLOYEE'''

#Executing the query
cursor.execute(sql)

#Displaying the result
print(cursor.fetchall())

#Closing the connection
conn.close()
```

Output

```
[('Krishna', 'Sharma', 22, 'M', 2000.0), ('Raj', 'Kandukuri', 23, 'M', 7000.0), ('Ramya', 'Ramapriya', 26, 'F', 5000.0)]
```

10. Python MySQL - Delete Data

To delete records from a MySQL table, you need to use the **DELETE FROM** statement. To remove specific records, you need to use WHERE clause along with it.

Syntax

Following is the syntax of the DELETE query in MYSQL:

```
DELETE FROM table_name [WHERE Clause]
```

Example

Assume we have created a table in MySQL with name EMPLOYEES as:

```
mysql> CREATE TABLE EMPLOYEE(  
    FIRST_NAME CHAR(20) NOT NULL,  
    LAST_NAME CHAR(20),  
    AGE INT,  
    SEX CHAR(1),  
    INCOME FLOAT);
```

```
Query OK, 0 rows affected (0.36 sec)
```

And if we have inserted 4 records in to it using INSERT statements as:

```
mysql> INSERT INTO EMPLOYEE VALUES  
    ('Krishna', 'Sharma', 19, 'M', 2000),  
    ('Raj', 'Kandukuri', 20, 'M', 7000),  
    ('Ramyra', 'Ramapriya', 25, 'F', 5000),  
    ('Mac', 'Mohan', 26, 'M', 2000);
```

Following MySQL statement deletes the record of the employee with *FIRST_NAME* "Mac".

```
mysql> DELETE FROM EMPLOYEE WHERE FIRST_NAME = 'Mac';  
Query OK, 1 row affected (0.12 sec)
```

If you retrieve the contents of the table, you can see only 3 records since we have deleted one.

```
mysql> select * from EMPLOYEE;  
  
+-----+-----+-----+-----+-----+  
  
| FIRST_NAME | LAST_NAME | AGE | SEX | INCOME |
```



```

+-----+-----+-----+-----+-----+
| Krishna | Sharma | 20 | M | 2000 |
| Raj     | Kandukuri | 21 | M | 7000 |
| Ramya   | Ramapriya | 25 | F | 5000 |
+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

```

If you execute the DELETE statement without the WHERE clause all the records from the specified table will be deleted.

```

mysql> DELETE FROM EMPLOYEE;
Query OK, 3 rows affected (0.09 sec)

```

If you retrieve the contents of the table, you will get an empty set as shown below:

```

mysql> select * from EMPLOYEE;
Empty set (0.00 sec)

```

Removing records of a table using python

DELETE operation is required when you want to delete some records from your database.

To delete the records in a table:

- import **mysql.connector** package.
- Create a connection object using the **mysql.connector.connect()** method, by passing the user name, password, host (optional default: localhost) and, database (optional) as parameters to it.
- Create a cursor object by invoking the **cursor()** method on the connection object created above.
- Then, execute the **DELETE** statement by passing it as a parameter to the **execute()** method.

Example

Following program deletes all the records from EMPLOYEE whose AGE is more than 20:

```

import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method

```

```

cursor = conn.cursor()

#Retrieving single row
print("Contents of the table: ")
cursor.execute("SELECT * from EMPLOYEE")
print(cursor.fetchall())

#Preparing the query to delete records
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (25)
try:
    # Execute the SQL command
    cursor.execute(sql)
    # Commit your changes in the database
    conn.commit()
except:
    # Roll back in case there is any error
    conn.rollback()

#Retrieving data
print("Contents of the table after delete operation ")
cursor.execute("SELECT * from EMPLOYEE")
print(cursor.fetchall())

#Closing the connection
conn.close()

```

Output

```

Contents of the table:
[('Krishna', 'Sharma', 22, 'M', 2000.0), ('Raj', 'Kandukuri', 23, 'M', 7000.0),
 ('Ramyra', 'Ramapriya', 26, 'F', 5000.0), ('Mac', 'Mohan', 20, 'M', 2000.0),
 ('Ramyra', 'Rama priya', 27, 'F', 9000.0)]
Contents of the table after delete operation:
[('Krishna', 'Sharma', 22, 'M', 2000.0), ('Raj', 'Kandukuri', 23, 'M', 7000.0),
 ('Mac', 'Mohan', 20, 'M', 2000.0)]

```

11. Python MySQL — Drop Table

You can remove an entire table using the **DROP TABLE** statement. You just need to specify the name of the table you need to delete.

Syntax

Following is the syntax of the DROP TABLE statement in MySQL:

```
DROP TABLE table_name;
```

Example

Before deleting a table get the list of tables using the SHOW TABLES statement as follows:

```
mysql> SHOW TABLES;
+-----+
| Tables_in_mydb |
+-----+
| contact          |
| cricketers_data |
| employee         |
| sample          |
| tutorials        |
+-----+
5 rows in set (0.00 sec)
```

Following statement removes the table named sample from the database completely:

```
mysql> DROP TABLE sample;
Query OK, 0 rows affected (0.29 sec)
```

Since we have deleted the table named sample from MySQL, if you get the list of tables again you will not find the table name sample in it.

```
mysql> SHOW TABLES;
+-----+
| Tables_in_mydb |
+-----+
| contact          |
| cricketers_data |
```

```
| employee      |
| tutorials     |
+-----+
4 rows in set (0.00 sec)
```

Removing a table using python

You can drop a table whenever you need to, using the DROP statement of MYSQL, but you need to be very careful while deleting any existing table because the data lost will not be recovered after deleting a table.

To drop a table from a MYSQL database using python invoke the **execute()** method on the cursor object and pass the drop statement as a parameter to it.

Example

Following table drops a table named EMPLOYEE from the database.

```
import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving the list of tables
print("List of tables in the database: ")
cursor.execute("SHOW Tables")
print(cursor.fetchall())

#Doping EMPLOYEE table if already exists
cursor.execute("DROP TABLE EMPLOYEE")
print("Table dropped... ")

#Retrieving the list of tables
print("List of tables after dropping the EMPLOYEE table: ")
cursor.execute("SHOW Tables")
print(cursor.fetchall())
```

```
#Closing the connection
conn.close()
```

Output

```
List of tables in the database:
[('employee',), ('employeeedata',), ('sample',), ('tutorials',)]
Table dropped...
List of tables after dropping the EMPLOYEE table:
[('employeeedata',), ('sample',), ('tutorials',)]
```

Drop table only if exists

If you try to drop a table which does not exist in the database, an error occurs as:

```
mysql.connector.errors.ProgrammingError: 1051 (42S02): Unknown table
'mydb.employee'
```

You can prevent this error by verifying whether the table exists before deleting, by adding the IF EXISTS to the DELETE statement.

```
import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving the list of tables
print("List of tables in the database: ")
cursor.execute("SHOW Tables")
print(cursor.fetchall())

#Doping EMPLOYEE table if already exists
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")
print("Table dropped... ")

#Retrieving the list of tables
print("List of tables after dropping the EMPLOYEE table: ")
```

```
cursor.execute("SHOW Tables")
print(cursor.fetchall())

#Closing the connection
conn.close()
```

Output

```
List of tables in the database:
[('employeedata',), ('sample',), ('tutorials',)]
Table dropped...
List of tables after dropping the EMPLOYEE table:
[('employeedata',), ('sample',),
 ('tutorials',)]
```

12. Python MySQL — Limit

While fetching records if you want to limit them by a particular number, you can do so, using the LIMIT clause of MYSQL.

Example

Assume we have created a table in MySQL with name EMPLOYEES as:

```
mysql> CREATE TABLE EMPLOYEE(  
    FIRST_NAME CHAR(20) NOT NULL,  
    LAST_NAME CHAR(20),  
    AGE INT,  
    SEX CHAR(1),  
    INCOME FLOAT);  
Query OK, 0 rows affected (0.36 sec)
```

And if we have inserted 4 records in to it using INSERT statements as:

```
mysql> INSERT INTO EMPLOYEE VALUES  
    ('Krishna', 'Sharma', 19, 'M', 2000),  
    ('Raj', 'Kandukuri', 20, 'M', 7000),  
    ('Ramya', 'Ramapriya', 25, 'F', 5000),  
    ('Mac', 'Mohan', 26, 'M', 2000);
```

Following SQL statement retrieves first two records of the Employee table using the LIMIT clause.

```
SELECT * FROM EMPLOYEE LIMIT 2;  
+-----+-----+-----+-----+-----+  
| FIRST_NAME | LAST_NAME | AGE | SEX | INCOME |  
+-----+-----+-----+-----+-----+  
| Krishna   | Sharma    | 19 | M   | 2000   |  
| Raj       | Kandukuri | 20 | M   | 7000   |  
+-----+-----+-----+-----+-----+  
2 rows in set (0.00 sec)
```

Limit clause using python

If you invoke the **execute()** method on the cursor object by passing the SELECT query along with the LIMIT clause, you can retrieve required number of records.

Example

Following python example creates and populates a table with name EMPLOYEE and, using the LIMIT clause it fetches the first two records of it.

```
import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving single row
sql = '''SELECT * from EMPLOYEE LIMIT 2'''

#Executing the query
cursor.execute(sql)

#Fetching the data
result = cursor.fetchall();
print(result)

#Closing the connection
conn.close()
```

Output

```
[('Krishna', 'Sharma', 26, 'M', 2000.0), ('Raj', 'Kandukuri', 20, 'M', 7000.0)]
```

LIMIT with OFFSET

If you need to limit the records starting from nth record (not 1st), you can do so, using OFFSET along with LIMIT.

```
import mysql.connector
```



```
#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving single row
sql = '''SELECT * from EMPLOYEE LIMIT 2 OFFSET 2'''

#Executing the query
cursor.execute(sql)

#Fetching the data
result = cursor.fetchall();
print(result)

#Closing the connection
conn.close()
```

Output

```
[('Ramya', 'Ramapriya', 29, 'F', 5000.0), ('Mac', 'Mohan', 26, 'M', 2000.0)]
```

13. Python MySQL — Join

When you have divided the data in two tables you can fetch combined records from these two tables using Joins.

Example

Suppose we have created a table with name EMPLOYEE and populated data into it as shown below:

```
mysql> CREATE TABLE EMPLOYEE(  
    FIRST_NAME CHAR(20) NOT NULL,  
    LAST_NAME CHAR(20),  
    AGE INT,  
    SEX CHAR(1),  
    INCOME FLOAT,  
    CONTACT INT  
);  
Query OK, 0 rows affected (0.36 sec)  
  
INSERT INTO Employee VALUES ('Ramyra', 'Rama Priya', 27, 'F', 9000, 101),  
('Vinay', 'Bhattacharya', 20, 'M', 6000, 102), ('Sharukh', 'Sheik', 25, 'M',  
8300, 103), ('Sarmista', 'Sharma', 26, 'F', 10000, 104), ('Trupthi', 'Mishra',  
24, 'F', 6000, 105);  
Query OK, 5 rows affected (0.08 sec)  
Records: 5 Duplicates: 0 Warnings: 0
```

Then, if we have created another table and populated it as:

```
CREATE TABLE CONTACT(  
    ID INT NOT NULL,  
    EMAIL CHAR(20) NOT NULL,  
    PHONE LONG,  
    CITY CHAR(20));  
Query OK, 0 rows affected (0.49 sec)
```

```
INSERT INTO CONTACT (ID, EMAIL, CITY) VALUES (101, 'Krishna@mymail.com',  
'Hyderabad'), (102, 'Raja@mymail.com', 'Vishakhapatnam'), (103,  
'Krishna@mymail.com', 'Pune'), (104, 'Raja@mymail.com', 'Mumbai');
```

```
Query OK, 4 rows affected (0.10 sec)
Records: 4 Duplicates: 0 Warnings: 0
```

Following statement retrieves data combining the values in these two tables:

```
mysql> SELECT * from EMPLOYEE INNER JOIN CONTACT ON EMPLOYEE.CONTACT =
CONTACT.ID;

+-----+-----+-----+-----+-----+-----+-----+-----+
| FIRST_NAME | LAST_NAME | AGE | SEX | INCOME | CONTACT | ID | EMAIL |
| PHONE | CITY |
+-----+-----+-----+-----+-----+-----+-----+-----+
| Ramya | Rama Priya | 27 | F | 9000 | 101 | 101 |
Krishna@mymail.com | NULL | Hyderabad |
| Vinay | Bhattacharya | 20 | M | 6000 | 102 | 102 |
Raja@mymail.com | NULL | Vishakhapatnam |
| Sharukh | Sheik | 25 | M | 8300 | 103 | 103 |
Krishna@mymail.com | NULL | Pune |
| Sarmista | Sharma | 26 | F | 10000 | 104 | 104 |
Raja@mymail.com | NULL | Mumbai |
+-----+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

MYSQL JOIN using python

Following example retrieves data from the above two tables combined by contact column of the EMPLOYEE table and ID column of the CONTACT table.

```
import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving single row
sql = '''SELECT * from EMPLOYEE INNER JOIN CONTACT ON EMPLOYEE.CONTACT =
CONTACT.ID'''
```

```
#Executing the query
cursor.execute(sql)

#Fetching 1st row from the table
result = cursor.fetchall();

print(result)

#Closing the connection
conn.close()
```

Output

```
[('Krishna', 'Sharma', 26, 'M', 2000, 101, 101, 'Krishna@mymail.com',
9848022338, 'Hyderabad'), ('Raj', 'Kandukuri', 20, 'M', 7000, 102, 102,
'Raja@mymail.com', 9848022339, 'Vishakhapatnam'), ('Ramya', 'Ramapriya', 29,
'F', 5000, 103, 103, 'Krishna@mymail.com', 9848022337, 'Pune'), ('Mac',
'Mohan', 26, 'M', 2000, 104, 104, 'Raja@mymail.com', 9848022330, 'Mumbai')]
```

14. Python MySQL - Cursor Object

The MySQLCursor of mysql-connector-python (and similar libraries) is used to execute statements to communicate with the MySQL database.

Using the methods of it you can execute SQL statements, fetch data from the result sets, call procedures.

You can create **Cursor** object using the cursor() method of the Connection object/class.

Example

```
import mysql.connector

#establishing the connection
conn = mysql.connector.connect(user='root', password='password',
host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()
```

Methods

Following are the various methods provided by the Cursor class/object.

Method	Description
callproc()	This method is used to call existing procedures MySQL database.
close()	This method is used to close the current cursor object.
Info()	This method gives information about the last query.
executemany()	This method accepts a list series of parameters list. Prepares an MySQL query and executes it with all the parameters.
execute()	This method accepts a MySQL query as a parameter and executes the given query.
fetchall()	This method retrieves all the rows in the result set of a query and returns them as list of tuples. (If we execute this after retrieving few rows it returns the remaining ones)

fetchone()	This method fetches the next row in the result of a query and returns it as a tuple.
fetchmany()	This method is similar to the fetchone() but, it retrieves the next set of rows in the result set of a query, instead of a single row.
fetchwarnings()	This method returns the warnings generated by the last executed query.

Properties

Following are the properties of the Cursor class:

Property	Description
column_names	This is a read only property which returns the list containing the column names of a result-set.
description	This is a read only property which returns the list containing the description of columns in a result-set.
lastrowid	This is a read only property, if there are any auto-incremented columns in the table, this returns the value generated for that column in the last INSERT or, UPDATE operation.
rowcount	This returns the number of rows returned/updated in case of SELECT and UPDATE operations.
statement	This property returns the last executed statement.

Python PostgreSQL

15. Python PostgreSQL — Introduction

Installation

PostgreSQL is a powerful, open source object-relational database system. It has more than 15 years of active development phase and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness.

To communicate with PostgreSQL using Python you need to install `psycopg`, an adapter provided for python programming, the current version of this is **psycopg2**.

`psycopg2` was written with the aim of being very small and fast, and stable as a rock. It is available under PIP (package manager of python)

Installing Psycog2 using PIP

First of all, make sure python and PIP is installed in your system properly and, PIP is up-to-date.

To upgrade PIP, open command prompt and execute the following command:

```
C:\Users\Tutorialspoint>python -m pip install --upgrade pip
Collecting pip
  Using cached
  https://files.pythonhosted.org/packages/8d/07/f7d7ced2f97ca3098c16565efbe6b15fa
  fcba53e8d9bdb431e09140514b0/pip-19.2.2-py2.py3-none-any.whl
Installing collected packages: pip
  Found existing installation: pip 19.0.3
  Uninstalling pip-19.0.3:
    Successfully uninstalled pip-19.0.3
Successfully installed pip-19.2.2
```

Then, open command prompt in admin mode and execute the **`pip install psycopg2-binary`** command as shown below:

```
C:\WINDOWS\system32>pip install psycopg2-binary
Collecting psycopg2-binary
  Using cached
  https://files.pythonhosted.org/packages/80/79/d0d13ce4c2f1addf4786f4a2ded802c2d
  f66ddf3c1b1a982ed8d4cb9fc6d/psycopg2_binary-2.8.3-cp37-cp37m-win32.whl
Installing collected packages: psycopg2-binary
Successfully installed psycopg2-binary-2.8.3
```

Verification

To verify the installation, create a sample python script with the following line in it.

```
import mysql.connector
```

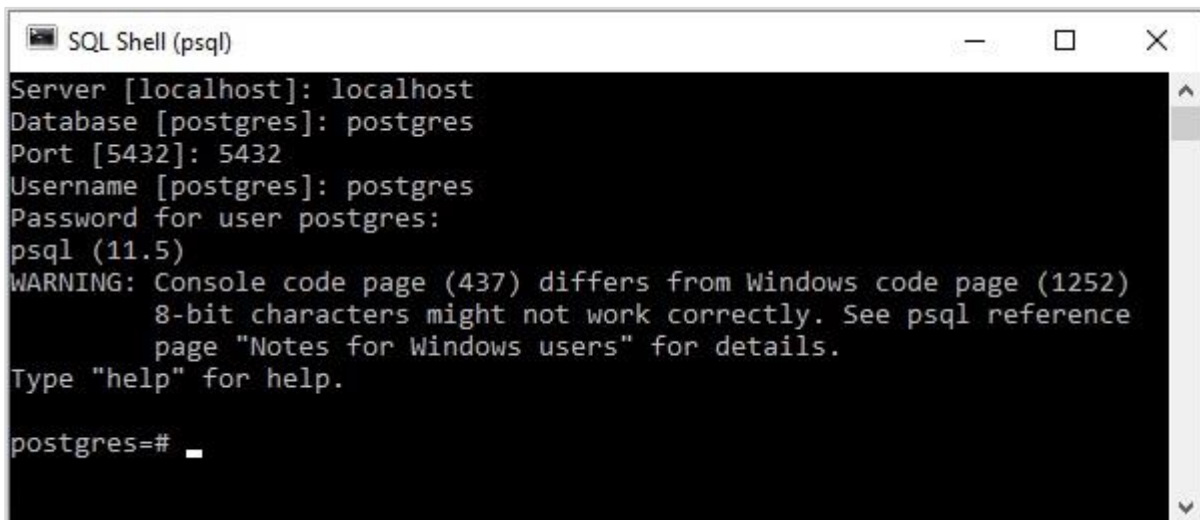
If the installation is successful, when you execute it, you should not get any errors:

```
D:\Python_PostgreSQL>import psycopg2  
D:\Python_PostgreSQL>
```

16. Python PostgreSQL — Database Connection

PostgreSQL provides its own shell to execute queries. To establish connection with the PostgreSQL database, make sure that you have installed it properly in your system. Open the PostgreSQL shell prompt and pass details like Server, Database, username, and password. If all the details you have given are appropriate, a connection is established with PostgreSQL database.

While passing the details you can go with the default server, database, port and, user name suggested by the shell.



```
SQL Shell (psql)
Server [localhost]: localhost
Database [postgres]: postgres
Port [5432]: 5432
Username [postgres]: postgres
Password for user postgres:
psql (11.5)
WARNING: Console code page (437) differs from Windows code page (1252)
8-bit characters might not work correctly. See psql reference
page "Notes for Windows users" for details.
Type "help" for help.
postgres=# _
```

Establishing connection using python

The connection class of the **psycopg2** represents/handles an instance of a connection. You can create new connections using the **connect()** function. This accepts the basic connection parameters such as dbname, user, password, host, port and returns a connection object. Using this function, you can establish a connection with the PostgreSQL.

Example

The following Python code shows how to connect to an existing database. If the database does not exist, then it will be created and finally a database object will be returned. The name of the default database of PostgreSQL is *postgres*. Therefore, we are supplying it as the database name.

```
import psycopg2

#establishing the connection

conn = psycopg2.connect(database="postgres", user='postgres',
password='password', host='127.0.0.1', port= '5432')
```

```
#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Executing an MYSQL function using the execute() method
cursor.execute("select version()")

# Fetch a single row using fetchone() method.
data = cursor.fetchone()

print("Connection established to: ",data)

#Closing the connection
conn.close()

Connection established to: ('PostgreSQL 11.5, compiled by Visual C++ build
1914, 64-bit',)
```

Output

```
Connection established to: ('PostgreSQL 11.5, compiled by Visual C++ build
1914, 64-bit',)
```

17. Python PostgreSQL — Create Database

You can create a database in PostgreSQL using the CREATE DATABASE statement. You can execute this statement in PostgreSQL shell prompt by specifying the name of the database to be created after the command.

Syntax

Following is the syntax of the CREATE DATABASE statement.

```
CREATE DATABASE dbname;
```

Example

Following statement creates a database named *testdb* in PostgreSQL.

```
postgres=# CREATE DATABASE testdb;  
CREATE DATABASE
```

You can list out the database in PostgreSQL using the \l command. If you verify the list of databases, you can find the newly created database as follows:

```
postgres=# \l  
  
                List of databases  
  Name          | Owner   | Encoding | Collate          | Ctype          |  
-----+-----+-----+-----+-----+  
 mydb          | postgres | UTF8     | English_United States.1252 | ..... |  
 postgres     | postgres | UTF8     | English_United States.1252 | ..... |  
 template0    | postgres | UTF8     | English_United States.1252 | ..... |  
 template1    | postgres | UTF8     | English_United States.1252 | ..... |  
 testdb      | postgres | UTF8     | English_United States.1252 | ..... |  
(5 rows)
```

You can also create a database in PostgreSQL from command prompt using the command *createdb*, a wrapper around the SQL statement CREATE DATABASE.

```
C:\Program Files\PostgreSQL\11\bin> createdb -h localhost -p 5432 -U postgres sampledb  
Password:
```

Creating a database using python

The cursor class of psycopg2 provides various methods execute various PostgreSQL commands, fetch records and copy data. You can create a cursor object using the cursor() method of the Connection class.

The execute() method of this class accepts a PostgreSQL query as a parameter and executes it.

Therefore, to create a database in PostgreSQL, execute the CREATE DATABASE query using this method.

Example

Following python example creates a database named mydb in PostgreSQL database.

```
import psycopg2

#establishing the connection
conn = psycopg2.connect(database="postgres", user='postgres',
password='password', host='127.0.0.1', port= '5432')
conn.autocommit = True

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Preparing query to create a database
sql = '''CREATE database mydb''';

#Creating a database
cursor.execute(sql)
print("Database created successfully.....")

#Closing the connection
conn.close()
```

Output

```
Database created successfully.....
```

18. Python PostgreSQL - Create Table

You can create a new table in a database in PostgreSQL using the CREATE TABLE statement. While executing this you need to specify the name of the table, column names and their data types.

Syntax

Following is the syntax of the CREATE TABLE statement in PostgreSQL.

```
CREATE TABLE table_name(  
    column1 datatype,  
    column2 datatype,  
    column3 datatype,  
    .....  
    columnN datatype,  
);
```

Example

Following example creates a table with name CRICKETERS in PostgreSQL.

```
postgres=# CREATE TABLE CRICKETERS (  
    First_Name VARCHAR(255),  
    Last_Name VARCHAR(255),  
    Age INT,  
    Place_Of_Birth VARCHAR(255),  
    Country VARCHAR(255));  
CREATE TABLE  
postgres=#
```

You can get the list of tables in a database in PostgreSQL using the \dt command. After creating a table, if you can verify the list of tables you can observe the newly created table in it as follows:

```
postgres=# \dt  
          List of relations  
Schema | Name      | Type | Owner  
-----+-----+-----+-----  
public | cricketers | table | postgres
```

```
(1 row)
postgres=#
```

In the same way, you can get the description of the created table using \d as shown below:

```
postgres=# \d cricketers
                Table "public.cricketers"
   Column      |          Type          | Collation | Nullable | Default
-----+-----+-----+-----+-----
 first_name    | character varying(255) |           |          |
 last_name     | character varying(255) |           |          |
 age           | integer                |           |          |
 place_of_birth | character varying(255) |           |          |
 country       | character varying(255) |           |          |

postgres=#
```

Creating a table using python

To create a table using python you need to execute the CREATE TABLE statement using the execute() method of the Cursor of *pyscopg2*.

The following Python example creates a table with name employee.

```
import psycopg2

#Establishing the connection
conn = psycopg2.connect(database="mydb", user='postgres', password='password',
host='127.0.0.1', port= '5432')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Doping EMPLOYEE table if already exists.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

#Creating table as per requirement
sql = '''CREATE TABLE EMPLOYEE(
        FIRST_NAME  CHAR(20) NOT NULL,
        LAST_NAME   CHAR(20),
        AGE INT,
```



```
        SEX CHAR(1),
        INCOME FLOAT)'''
cursor.execute(sql)
print("Table created successfully.....")

#Closing the connection
conn.close()
```

Output

```
Table created successfully.....
```

19. Python PostgreSQL — Insert Data

You can insert record into an existing table in PostgreSQL using the **INSERT INTO** statement. While executing this, you need to specify the name of the table, and values for the columns in it.

Syntax

Following is the recommended syntax of the INSERT statement:

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)
VALUES (value1, value2, value3,...valueN);
```

Where, column1, column2, column3,.. are the names of the columns of a table, and value1, value2, value3,... are the values you need to insert into the table.

Example

Assume we have created a table with name CRICKETERS using the CREATE TABLE statement as shown below:

```
postgres=# CREATE TABLE CRICKETERS (
First_Name VARCHAR(255),
Last_Name VARCHAR(255),
Age INT,
Place_Of_Birth VARCHAR(255),
Country VARCHAR(255));
CREATE TABLE
postgres=#
```

Following PostgreSQL statement inserts a row in the above created table:

```
postgres=# insert into CRICKETERS (First_Name, Last_Name, Age, Place_Of_Birth,
Country) values('Shikhar', 'Dhawan', 33, 'Delhi', 'India');
INSERT 0 1
postgres=#
```

While inserting records using the *INSERT INTO* statement, if you skip any columns names Record will be inserted leaving empty spaces at columns which you have skipped.

```
postgres=# insert into CRICKETERS (First_Name, Last_Name, Country)
values('Jonathan', 'Trott', 'SouthAfrica');
INSERT 0 1
```

You can also insert records into a table without specifying the column names, if the order of values you pass is same as their respective column names in the table.

```
postgres=# insert into CRICKETERS values('Kumara', 'Sangakkara', 41, 'Matale',
'Srilanka');
INSERT 0 1
postgres=# insert into CRICKETERS values('Virat', 'Kohli', 30, 'Delhi',
'India');
INSERT 0 1
postgres=# insert into CRICKETERS values('Rohit', 'Sharma', 32, 'Nagpur',
'India');
INSERT 0 1
postgres=#
```

After inserting the records into a table you can verify its contents using the SELECT statement as shown below:

```
postgres=# SELECT * from CRICKETERS;
 first_name | last_name | age | place_of_birth | country
-----+-----+-----+-----+-----
 Shikhar   | Dhawan   | 33 | Delhi           | India
 Jonathan  | Trott    |    |                 | SouthAfrica
 Kumara    | Sangakkara | 41 | Matale         | Srilanka
 Virat     | Kohli    | 30 | Delhi           | India
 Rohit     | Sharma   | 32 | Nagpur          | India
(5 rows)
```

Inserting data using python

The cursor class of psycopg2 provides a method with name execute() method. This method accepts the query as a parameter and executes it.

Therefore, to insert data into a table in PostgreSQL using python:

- Import **psycopg2** package.
- Create a connection object using the **connect()** method, by passing the user name, password, host (optional default: localhost) and, database (optional) as parameters to it.
- Turn off the auto-commit mode by setting false as value to the attribute **autocommit**.
- The **cursor()** method of the **Connection** class of the psycopg2 library returns a cursor object. Create a cursor object using this method.

- Then, execute the INSERT statement(s) by passing it/them as a parameter to the execute() method.

Example

Following Python program creates a table with name EMPLOYEE in PostgreSQL database and inserts records into it using the execute() method:

```
import psycopg2

#Establishing the connection
conn = psycopg2.connect(database="mydb", user='postgres', password='password',
host='127.0.0.1', port= '5432')

#Setting auto commit false
conn.autocommit = True

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

# Preparing SQL queries to INSERT a record into the database.
cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
      INCOME) VALUES ('Ramya', 'Rama priya', 27, 'F', 9000)''')

cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
      INCOME) VALUES ('Vinay', 'Battacharya', 20, 'M', 6000)''')

cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
      INCOME) VALUES ('Sharukh', 'Sheik', 25, 'M', 8300)''')

cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
      INCOME) VALUES ('Sarmista', 'Sharma', 26, 'F', 10000)''')

cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
      INCOME) VALUES ('Tripthi', 'Mishra', 24, 'F', 6000)''')

# Commit your changes in the database
conn.commit()

print("Records inserted.....")
```

```
# Closing the connection  
conn.close()
```

Output

```
Records inserted.....
```

20. Python PostgreSQL — Select Data

You can retrieve the contents of an existing table in PostgreSQL using the SELECT statement. At this statement, you need to specify the name of the table and, it returns its contents in tabular format which is known as result set.

Syntax

Following is the syntax of the SELECT statement in PostgreSQL:

```
SELECT column1, column2, columnN FROM table_name;
```

Example

Assume we have created a table with name CRICKETERS using the following query:

```
postgres=# CREATE TABLE CRICKETERS ( First_Name VARCHAR(255), Last_Name  
VARCHAR(255), Age int, Place_Of_Birth VARCHAR(255), Country VARCHAR(255));  
CREATE TABLE  
postgres=#
```

And if we have inserted 5 records in to it using INSERT statements as:

```
postgres=# insert into CRICKETERS values('Shikhar', 'Dhawan', 33, 'Delhi',  
'India');  
INSERT 0 1  
postgres=# insert into CRICKETERS values('Jonathan', 'Trott', 38, 'CapeTown',  
'SouthAfrica');  
INSERT 0 1  
postgres=# insert into CRICKETERS values('Kumara', 'Sangakkara', 41, 'Matale',  
'Srilanka');  
INSERT 0 1  
postgres=# insert into CRICKETERS values('Virat', 'Kohli', 30, 'Delhi',  
'India');  
INSERT 0 1  
postgres=# insert into CRICKETERS values('Rohit', 'Sharma', 32, 'Nagpur',  
'India');  
INSERT 0 1
```

Following SELECT query retrieves the values of the columns FIRST_NAME, LAST_NAME and, COUNTRY from the CRICKETERS table.

```
postgres=# SELECT FIRST_NAME, LAST_NAME, COUNTRY FROM CRICKETERS;  
first_name | last_name | country
```

```

-----+-----+-----
Shikhar   | Dhawan   | India
Jonathan  | Trott    | SouthAfrica
Kumara    | Sangakkara | Srilanka
Virat     | Kohli    | India
Rohit     | Sharma   | India
(5 rows)

```

If you want to retrieve all the columns of each record you need to replace the names of the columns with "*" as shown below:

```

postgres=# SELECT * FROM CRICKETERS;
 first_name | last_name | age | place_of_birth | country
-----+-----+-----+-----+-----
Shikhar    | Dhawan   | 33 | Delhi          | India
Jonathan   | Trott    | 38 | CapeTown       | SouthAfrica
Kumara     | Sangakkara | 41 | Matale        | Srilanka
Virat      | Kohli    | 30 | Delhi          | India
Rohit      | Sharma   | 32 | Nagpur         | India
(5 rows)

postgres=#

```

Retrieving data using python

READ Operation on any database means to fetch some useful information from the database. You can fetch data from PostgreSQL using the fetch() method provided by the psycopg2.

The Cursor class provides three methods namely fetchall(), fetchmany() and fetchone() where,

- The fetchall() method retrieves all the rows in the result set of a query and returns them as list of tuples. (If we execute this after retrieving few rows, it returns the remaining ones).
- The fetchone() method fetches the next row in the result of a query and returns it as a tuple.
- The fetchmany() method is similar to the fetchone() but, it retrieves the next set of rows in the result set of a query, instead of a single row.

Note: A result set is an object that is returned when a cursor object is used to query a table.

Example

The following Python program connects to a database named mydb of PostgreSQL and retrieves all the records from a table named EMPLOYEE.

```
import psycopg2

#establishing the connection
conn = psycopg2.connect(database="mydb", user='postgres', password='password',
host='127.0.0.1', port= '5432')

#Setting auto commit false
conn.autocommit = True

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving data
cursor.execute('''SELECT * from EMPLOYEE''')

#Fetching 1st row from the table
result = cursor.fetchone();
print(result)

#Fetching 1st row from the table
result = cursor.fetchall();
print(result)

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

```
('Ramya', 'Rama priya', 27, 'F', 9000.0)
[('Vinay', 'Battacharya', 20, 'M', 6000.0),
('Sharukh', 'Sheik', 25, 'M', 8300.0),
('Sarmista', 'Sharma', 26, 'F', 10000.0),
```



```
('Tripthi', 'Mishra', 24, 'F', 6000.0)]
```

21. Python PostgreSQL — Where Clause

While performing SELECT, UPDATE or, DELETE operations, you can specify condition to filter the records using the WHERE clause. The operation will be performed on the records which satisfies the given condition.

Syntax

Following is the syntax of the WHERE clause in PostgreSQL:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [search_condition]
```

You can specify a search_condition using comparison or logical operators. like >, <, =, LIKE, NOT, etc. The following examples would make this concept clear.

Example

Assume we have created a table with name CRICKETERS using the following query:

```
postgres=# CREATE TABLE CRICKETERS ( First_Name VARCHAR(255), Last_Name
VARCHAR(255), Age int, Place_Of_Birth VARCHAR(255), Country VARCHAR(255));
CREATE TABLE
postgres=#
```

And if we have inserted 5 records in to it using INSERT statements as:

```
postgres=# insert into CRICKETERS values('Shikhar', 'Dhawan', 33, 'Delhi',
'India');
INSERT 0 1
postgres=# insert into CRICKETERS values('Jonathan', 'Trott', 38, 'CapeTown',
'SouthAfrica');
INSERT 0 1
postgres=# insert into CRICKETERS values('Kumara', 'Sangakkara', 41, 'Matale',
'Srilanka');
INSERT 0 1
postgres=# insert into CRICKETERS values('Virat', 'Kohli', 30, 'Delhi',
'India');
INSERT 0 1
postgres=# insert into CRICKETERS values('Rohit', 'Sharma', 32, 'Nagpur',
'India');
INSERT 0 1
```

Following SELECT statement retrieves the records whose age is greater than 35:

```
postgres=# SELECT * FROM CRICKETERS WHERE AGE > 35;
 first_name | last_name | age | place_of_birth | country
-----+-----+-----+-----+-----
 Jonathan   | Trott     | 38 | CapeTown       | SouthAfrica
 Kumara     | Sangakkara | 41 | Matale         | Srilanka
(2 rows)

postgres=#
```

Where clause using python

To fetch specific records from a table using the python program execute the *SELECT* statement with *WHERE* clause, by passing it as a parameter to the **execute()** method.

Example

Following python example demonstrates the usage of WHERE command using python.

```
import psycopg2

#establishing the connection
conn = psycopg2.connect(database="mydb", user='postgres', password='password',
host='127.0.0.1', port= '5432')

#Setting auto commit false
conn.autocommit = True

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Doping EMPLOYEE table if already exists.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

sql = '''CREATE TABLE EMPLOYEE(
        FIRST_NAME  CHAR(20) NOT NULL,
        LAST_NAME   CHAR(20),
        AGE INT,
        SEX CHAR(1),

        INCOME FLOAT)'''
```

```
cursor.execute(sql)

#Populating the table
insert_stmt = "INSERT INTO EMPLOYEE (FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)
VALUES (%s, %s, %s, %s, %s)"
data = [('Krishna', 'Sharma', 19, 'M', 2000), ('Raj', 'Kandukuri', 20, 'M',
7000),
        ('Ramy', 'Ramapriya', 25, 'M', 5000), ('Mac', 'Mohan', 26, 'M', 2000)]
cursor.executemany(insert_stmt, data)

#Retrieving specific records using the where clause
cursor.execute("SELECT * from EMPLOYEE WHERE AGE <23")

print(cursor.fetchall())

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

```
[('Krishna', 'Sharma', 19, 'M', 2000.0), ('Raj', 'Kandukuri', 20, 'M', 7000.0)]
```

22. Python PostgreSQL — Order By

Usually if you try to retrieve data from a table, you will get the records in the same order in which you have inserted them.

Using the **ORDER BY** clause, while retrieving the records of a table you can sort the resultant records in ascending or descending order based on the desired column.

Syntax

Following is the syntax of the ORDER BY clause in PostgreSQL.

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

Example

Assume we have created a table with name CRICKETERS using the following query:

```
postgres=# CREATE TABLE CRICKETERS ( First_Name VARCHAR(255), Last_Name
VARCHAR(255), Age int, Place_Of_Birth VARCHAR(255), Country VARCHAR(255));
CREATE TABLE
postgres=#
```

And if we have inserted 5 records in to it using INSERT statements as:

```
postgres=# insert into CRICKETERS values('Shikhar', 'Dhawan', 33, 'Delhi',
'India');
INSERT 0 1
postgres=# insert into CRICKETERS values('Jonathan', 'Trott', 38, 'CapeTown',
'SouthAfrica');
INSERT 0 1
postgres=# insert into CRICKETERS values('Kumara', 'Sangakkara', 41, 'Matale',
'Srilanka');
INSERT 0 1
postgres=# insert into CRICKETERS values('Virat', 'Kohli', 30, 'Delhi',
'India');
INSERT 0 1
postgres=# insert into CRICKETERS values('Rohit', 'Sharma', 32, 'Nagpur',
'India');
INSERT 0 1
```

Following SELECT statement retrieves the rows of the CRICKETERS table in the ascending order of their age:

```
postgres=# SELECT * FROM CRICKETERS ORDER BY AGE;
 first_name | last_name | age | place_of_birth | country
-----+-----+-----+-----+-----
Virat      | Kohli     | 30  | Delhi          | India
Rohit      | Sharma    | 32  | Nagpur         | India
Shikhar    | Dhawan    | 33  | Delhi          | India
Jonathan   | Trott     | 38  | CapeTown       | SouthAfrica
Kumara     | Sangakkara | 41  | Matale         | Srilanka
(5 rows)
```

You can use more than one column to sort the records of a table. Following SELECT statements sort the records of the CRICKETERS table based on the columns age and FIRST_NAME.

```
postgres=# SELECT * FROM CRICKETERS ORDER BY AGE, FIRST_NAME;
 first_name | last_name | age | place_of_birth | country
-----+-----+-----+-----+-----
Virat      | Kohli     | 30  | Delhi          | India
Rohit      | Sharma    | 32  | Nagpur         | India
Shikhar    | Dhawan    | 33  | Delhi          | India
Jonathan   | Trott     | 38  | CapeTown       | SouthAfrica
Kumara     | Sangakkara | 41  | Matale         | Srilanka
(5 rows)
```

By default, the **ORDER BY** clause sorts the records of a table in ascending order. You can arrange the results in descending order using DESC as:

```
postgres=# SELECT * FROM CRICKETERS ORDER BY AGE DESC;
 first_name | last_name | age | place_of_birth | country
-----+-----+-----+-----+-----
Kumara     | Sangakkara | 41  | Matale         | Srilanka
Jonathan   | Trott     | 38  | CapeTown       | SouthAfrica
Shikhar    | Dhawan    | 33  | Delhi          | India
Rohit      | Sharma    | 32  | Nagpur         | India
Virat      | Kohli     | 30  | Delhi          | India
(5 rows)
```

ORDER BY clause using python

To retrieve contents of a table in specific order, invoke the execute() method on the cursor object and, pass the SELECT statement along with ORDER BY clause, as a parameter to it.

Example

In the following example, we are creating a table with name and Employee, populating it, and retrieving its records back in the (ascending) order of their age, using the ORDER BY clause.

```
import psycopg2

#establishing the connection
conn = psycopg2.connect(database="mydb", user='postgres', password='password',
host='127.0.0.1', port= '5432')

#Setting auto commit false
conn.autocommit = True

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Doping EMPLOYEE table if already exists.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")
#Creating a table
sql = '''CREATE TABLE EMPLOYEE(
            FIRST_NAME  CHAR(20) NOT NULL,
            LAST_NAME   CHAR(20),
            AGE INT, SEX CHAR(1),
            INCOME INT,
            CONTACT INT)'''
cursor.execute(sql)

#Populating the table
insert_stmt = "INSERT INTO EMPLOYEE (FIRST_NAME, LAST_NAME, AGE, SEX, INCOME,
CONTACT) VALUES (%s, %s, %s, %s, %s, %s)"
data = [('Krishna', 'Sharma', 26, 'M', 2000, 101), ('Raj', 'Kandukuri', 20,
'M', 7000, 102),
        ('Ramya', 'Ramapriya', 29, 'F', 5000, 103),('Mac', 'Mohan', 26, 'M',
2000, 104)]
```

```
cursor.executemany(insert_stmt, data)
conn.commit()

#Retrieving specific records using the ORDER BY clause
cursor.execute("SELECT * from EMPLOYEE ORDER BY AGE")

print(cursor.fetchall())

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

```
[('Sharukh', 'Sheik', 25, 'M', 8300.0), ('Sarmista', 'Sharma', 26, 'F', 10000.0)]
```


23. Python PostgreSQL — Update Table

You can modify the contents of existing records of a table in PostgreSQL using the UPDATE statement. To update specific rows, you need to use the WHERE clause along with it.

Syntax

Following is the syntax of the UPDATE statement in PostgreSQL:

```
UPDATE table_name
SET column1 = value1, column2 = value2..., columnN = valueN
WHERE [condition];
```

Example

Assume we have created a table with name CRICKETERS using the following query:

```
postgres=# CREATE TABLE CRICKETERS ( First_Name VARCHAR(255), Last_Name
VARCHAR(255), Age int, Place_Of_Birth VARCHAR(255), Country VARCHAR(255));
CREATE TABLE
postgres=#
```

And if we have inserted 5 records in to it using INSERT statements as:

```
postgres=# insert into CRICKETERS values('Shikhar', 'Dhawan', 33, 'Delhi',
'India');
INSERT 0 1
postgres=# insert into CRICKETERS values('Jonathan', 'Trott', 38, 'CapeTown',
'SouthAfrica');
INSERT 0 1
postgres=# insert into CRICKETERS values('Kumara', 'Sangakkara', 41, 'Matale',
'Srilanka');
INSERT 0 1
postgres=# insert into CRICKETERS values('Virat', 'Kohli', 30, 'Delhi',
'India');
INSERT 0 1
postgres=# insert into CRICKETERS values('Rohit', 'Sharma', 32, 'Nagpur',
'India');
INSERT 0 1
```

Following statement modifies the age of the cricketer, whose first name is **Shikhar**:

```
postgres=# UPDATE CRICKETERS SET AGE = 45 WHERE FIRST_NAME = 'Shikhar' ;
```

```
UPDATE 1
postgres=#
```

If you retrieve the record whose FIRST_NAME is Shikhar you observe that the age value has been changed to 45:

```
postgres=# SELECT * FROM CRICKETERS WHERE FIRST_NAME = 'Shikhar';
 first_name | last_name | age | place_of_birth | country
-----+-----+-----+-----+-----
 Shikhar   | Dhawan   | 45 | Delhi          | India
(1 row)

postgres=#
```

If you haven't used the WHERE clause, values of all the records will be updated. Following UPDATE statement increases the age of all the records in the CRICKETERS table by 1:

```
postgres=# UPDATE CRICKETERS SET AGE = AGE+1;
UPDATE 5
```

If you retrieve the contents of the table using SELECT command, you can see the updated values as:

```
postgres=# SELECT * FROM CRICKETERS;
 first_name | last_name | age | place_of_birth | country
-----+-----+-----+-----+-----
 Jonathan   | Trott     | 39 | CapeTown       | SouthAfrica
 Kumara     | Sangakkara | 42 | Matale         | Srilanka
 Virat      | Kohli     | 31 | Delhi          | India
 Rohit      | Sharma    | 33 | Nagpur         | India
 Shikhar    | Dhawan    | 46 | Delhi          | India
(5 rows)
```

Updating records using python

The cursor class of psycopg2 provides a method with name execute() method. This method accepts the query as a parameter and executes it.

Therefore, to insert data into a table in PostgreSQL using python:

- Import **psycopg2** package.

- Create a connection object using the **connect()** method, by passing the user name, password, host (optional default: localhost) and, database (optional) as parameters to it.
- Turn off the auto-commit mode by setting false as value to the attribute **autocommit**.
- The **cursor()** method of the **Connection** class of the psycopg2 library returns a cursor object. Create a cursor object using this method.
- Then, execute the UPDATE statement by passing it as a parameter to the execute() method.

Example

Following Python code updates the contents of the *Employee* table and retrieves the results:

```
import psycopg2

#establishing the connection
conn = psycopg2.connect(database="mydb", user='postgres', password='password',
host='127.0.0.1', port= '5432')

#Setting auto commit false
conn.autocommit = True

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Fetching all the rows before the update
print("Contents of the Employee table: ")
sql = '''SELECT * from EMPLOYEE'''
cursor.execute(sql)
print(cursor.fetchall())

#Updating the records
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = 'M'"
cursor.execute(sql)
print("Table updated..... ")

#Fetching all the rows after the update
print("Contents of the Employee table after the update operation: ")
```

```
sql = '''SELECT * from EMPLOYEE'''
cursor.execute(sql)
print(cursor.fetchall())

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

```
Contents of the Employee table:
[('Ramya', 'Rama priya', 27, 'F', 9000.0), ('Vinay', 'Battacharya', 20, 'M',
6000.0), ('Sharukh', 'Sheik', 25, 'M', 8300.0), ('Sarmista', 'Sharma', 26, 'F',
10000.0), ('Tripthi', 'Mishra', 24, 'F', 6000.0)]
Table updated.....
Contents of the Employee table after the update operation:
[('Ramya', 'Rama priya', 27, 'F', 9000.0), ('Sarmista', 'Sharma', 26, 'F',
10000.0), ('Tripthi', 'Mishra', 24, 'F', 6000.0), ('Vinay', 'Battacharya', 21,
'M', 6000.0), ('Sharukh', 'Sheik', 26, 'M', 8300.0)]
```

24. Python PostgreSQL — Delete Data

You can delete the records in an existing table using the **DELETE FROM** statement of PostgreSQL database. To remove specific records, you need to use WHERE clause along with it.

Syntax

Following is the syntax of the DELETE query in PostgreSQL:

```
DELETE FROM table_name [WHERE Clause]
```

Example

Assume we have created a table with name CRICKETERS using the following query:

```
postgres=# CREATE TABLE CRICKETERS ( First_Name VARCHAR(255), Last_Name
VARCHAR(255), Age int, Place_Of_Birth VARCHAR(255), Country VARCHAR(255));
CREATE TABLE
postgres=#
```

And if we have inserted 5 records in to it using INSERT statements as:

```
postgres=# insert into CRICKETERS values ('Shikhar', 'Dhawan', 33, 'Delhi',
'India');
INSERT 0 1
postgres=# insert into CRICKETERS values ('Jonathan', 'Trott', 38, 'CapeTown',
'SouthAfrica');
INSERT 0 1
postgres=# insert into CRICKETERS values ('Kumara', 'Sangakkara', 41, 'Matale',
'Srilanka');
INSERT 0 1
postgres=# insert into CRICKETERS values ('Virat', 'Kohli', 30, 'Delhi',
'India');
INSERT 0 1
postgres=# insert into CRICKETERS values ('Rohit', 'Sharma', 32, 'Nagpur',
'India');
INSERT 0 1
```

Following statement deletes the record of the cricketer whose last name is 'Sangakkara'.

```
postgres=# DELETE FROM CRICKETERS WHERE LAST_NAME = 'Sangakkara';
DELETE 1
```

If you retrieve the contents of the table using the SELECT statement, you can see only 4 records since we have deleted one.

```
postgres=# SELECT * FROM CRICKETERS;
 first_name | last_name | age | place_of_birth | country
-----+-----+-----+-----+-----
 Jonathan   | Trott     | 39  | CapeTown       | SouthAfrica
 Virat      | Kohli     | 31  | Delhi          | India
 Rohit      | Sharma    | 33  | Nagpur         | India
 Shikhar    | Dhawan    | 46  | Delhi          | India
(4 rows)
```

If you execute the DELETE FROM statement without the WHERE clause all the records from the specified table will be deleted.

```
postgres=# DELETE FROM CRICKETERS;
DELETE 4
```

Since you have deleted all the records, if you try to retrieve the contents of the CRICKETERS table, using SELECT statement you will get an empty result set as shown below:

```
postgres=# SELECT * FROM CRICKETERS;
 first_name | last_name | age | place_of_birth | country
-----+-----+-----+-----+-----
(0 rows)
```

Deleting data using python

The cursor class of psycopg2 provides a method with name execute() method. This method accepts the query as a parameter and executes it.

Therefore, to insert data into a table in PostgreSQL using python:

- Import **psycopg2** package.
- Create a connection object using the **connect()** method, by passing the user name, password, host (optional default: localhost) and, database (optional) as parameters to it.
- Turn off the auto-commit mode by setting false as value to the attribute **autocommit**.
- The **cursor()** method of the **Connection** class of the psycopg2 library returns a cursor object. Create a cursor object using this method.

- Then, execute the DELETE statement by passing it as a parameter to the execute() method.

Example

Following Python code deletes records of the EMPLOYEE table with age values greater than 25:

```
import psycopg2

#establishing the connection
conn = psycopg2.connect(database="mydb", user='postgres', password='password',
host='127.0.0.1', port= '5432')

#Setting auto commit false
conn.autocommit = True

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving contents of the table
print("Contents of the table: ")
cursor.execute('''SELECT * from EMPLOYEE''')
print(cursor.fetchall())

#Deleting records
cursor.execute('''DELETE FROM EMPLOYEE WHERE AGE > 25''')

#Retrieving data after delete
print("Contents of the table after delete operation ")
cursor.execute("SELECT * from EMPLOYEE")
print(cursor.fetchall())

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

Contents of the table:

```
[('Ramya', 'Rama priya', 27, 'F', 9000.0), ('Sarmista', 'Sharma', 26, 'F', 10000.0), ('Tripthi', 'Mishra', 24, 'F', 6000.0), ('Vinay', 'Battacharya', 21, 'M', 6000.0), ('Sharukh', 'Sheik', 26, 'M', 8300.0)]
```

Contents of the table after delete operation:

```
[('Tripthi', 'Mishra', 24, 'F', 6000.0), ('Vinay', 'Battacharya', 21, 'M', 6000.0)]
```


25. Python PostgreSQL — Drop Table

You can drop a table from PostgreSQL database using the DROP TABLE statement.

Syntax

Following is the syntax of the DROP TABLE statement in PostgreSQL:

```
DROP TABLE table_name;
```

Example

Assume we have created two tables with name CRICKETERS and EMPLOYEES using the following queries:

```
postgres=# CREATE TABLE CRICKETERS (First_Name VARCHAR(255), Last_Name
VARCHAR(255), Age int, Place_Of_Birth VARCHAR(255), Country VARCHAR(255));
CREATE TABLE
postgres=#
postgres=# CREATE TABLE EMPLOYEE(FIRST_NAME CHAR(20) NOT NULL, LAST_NAME
CHAR(20), AGE INT, SEX CHAR(1), INCOME FLOAT);
CREATE TABLE
postgres=#
```

Now if you verify the list of tables using the “\dt” command, you can see the above created tables as:

```
postgres=# \dt;
                List of relations
 Schema |   Name   | Type  | Owner
-----+-----+-----+-----
 public | cricketers | table | postgres
 public | employee  | table | postgres
(2 rows)
postgres=#
```

Following statement deletes the table named Employee from the database:

```
postgres=# DROP table employee;
DROP TABLE
```

Since you have deleted the Employee table, if you retrieve the list of tables again, you can observe only one table in it.

```
postgres=# \dt;
          List of relations
 Schema |   Name   | Type  | Owner
-----+-----+-----+-----
 public | cricketers | table | postgres
(1 row)

postgres=#
```

If you try to delete the Employee table again, since you have already deleted it, you will get an error saying "table does not exist" as shown below:

```
postgres=# DROP table employee;
ERROR:  table "employee" does not exist
postgres=#
```

To resolve this, you can use the IF EXISTS clause along with the DELETE statement. This removes the table if it exists else skips the DELETE operation.

```
postgres=# DROP table IF EXISTS employee;
NOTICE:  table "employee" does not exist, skipping
DROP TABLE
postgres=#
```

Removing an entire table using Python

You can drop a table whenever you need to, using the DROP statement. But you need to be very careful while deleting any existing table because the data lost will not be recovered after deleting a table.

```
import psycopg2

#establishing the connection
conn = psycopg2.connect(database="mydb", user='postgres', password='password',
host='127.0.0.1', port= '5432')

#Setting auto commit false
conn.autocommit = True

#Creating a cursor object using the cursor() method
```

```
cursor = conn.cursor()

#Doping EMPLOYEE table if already exists
cursor.execute("DROP TABLE emp")
print("Table dropped... ")

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

```
#Table dropped...
```

26. Python PostgreSQL – Limit

While executing a PostgreSQL SELECT statement you can limit the number of records in its result using the LIMIT clause.

Syntax

Following is the syntax of the LIMIT clause in PostgreSQL:

```
SELECT column1, column2, columnN
FROM table_name
LIMIT [no of rows]
```

Example

Assume we have created a table with name CRICKETERS using the following query:

```
postgres=# CREATE TABLE CRICKETERS ( First_Name VARCHAR(255), Last_Name
VARCHAR(255), Age int, Place_Of_Birth VARCHAR(255), Country VARCHAR(255));
CREATE TABLE
postgres=#
```

And if we have inserted 5 records in to it using INSERT statements as:

```
postgres=# insert into CRICKETERS values ('Shikhar', 'Dhawan', 33, 'Delhi',
'India');
INSERT 0 1
postgres=# insert into CRICKETERS values ('Jonathan', 'Trott', 38, 'CapeTown',
'SouthAfrica');
INSERT 0 1
postgres=# insert into CRICKETERS values ('Kumara', 'Sangakkara', 41, 'Matale',
'Srilanka');
INSERT 0 1
postgres=# insert into CRICKETERS values ('Virat', 'Kohli', 30, 'Delhi',
'India');
INSERT 0 1
postgres=# insert into CRICKETERS values ('Rohit', 'Sharma', 32, 'Nagpur',
'India');
INSERT 0 1
```

Following statement retrieves the first 3 records of the Cricketers table using the LIMIT clause:

```
postgres=# SELECT * FROM CRICKETERS LIMIT 3;
 first_name | last_name | age | place_of_birth | country
-----+-----+-----+-----+-----
 Shikhar   | Dhawan   | 33 | Delhi          | India
 Jonathan  | Trott    | 38 | CapeTown       | SouthAfrica
 Kumara    | Sangakkara | 41 | Matale        | Srilanka
(3 rows)
```

If you want to get records starting from a particular record (offset) you can do so, using the OFFSET clause along with LIMIT.

```
postgres=# SELECT * FROM CRICKETERS LIMIT 3 OFFSET 2;
 first_name | last_name | age | place_of_birth | country
-----+-----+-----+-----+-----
 Kumara    | Sangakkara | 41 | Matale        | Srilanka
 Virat     | Kohli     | 30 | Delhi         | India
 Rohit     | Sharma    | 32 | Nagpur        | India
(3 rows)

postgres=#
```

Limit clause using python

Following python example retrieves the contents of a table named EMPLOYEE, limiting the number of records in the result to 2:

```
import psycopg2

#establishing the connection
conn = psycopg2.connect(database="mydb", user='postgres', password='password',
host='127.0.0.1', port= '5432')

#Setting auto commit false
conn.autocommit = True

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving single row
sql = '''SELECT * from EMPLOYEE LIMIT 2 OFFSET 2'''
```

```
#Executing the query
cursor.execute(sql)

#Fetching the data
result = cursor.fetchall();
print(result)

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

```
[('Sharukh', 'Sheik', 25, 'M', 8300.0), ('Sarmista', 'Sharma', 26, 'F', 10000.0)]
```

27. Python PostgreSQL — Join

When you have divided the data in two tables you can fetch combined records from these two tables using Joins.

Example

Assume we have created a table with name CRICKETERS and inserted 5 records into it as shown below:

```
postgres=# CREATE TABLE CRICKETERS (First_Name VARCHAR(255), Last_Name
VARCHAR(255), Age int, Place_Of_Birth VARCHAR(255), Country VARCHAR(255));
postgres=# insert into CRICKETERS values ('Shikhar', 'Dhawan', 33, 'Delhi',
'India');
postgres=# insert into CRICKETERS values ('Jonathan', 'Trott', 38, 'CapeTown',
'SouthAfrica');
postgres=# insert into CRICKETERS values ('Kumara', 'Sangakkara', 41, 'Matale',
'Srilanka');
postgres=# insert into CRICKETERS values ('Virat', 'Kohli', 30, 'Delhi',
'India');
postgres=# insert into CRICKETERS values ('Rohit', 'Sharma', 32, 'Nagpur',
'India');
```

And, if we have created another table with name OdiStats and inserted 5 records into it as:

```
postgres=# CREATE TABLE ODISTats (First_Name VARCHAR(255), Matches INT, Runs
INT, AVG FLOAT, Centuries INT, HalfCenturies INT);
postgres=# insert into OdiStats values ('Shikhar', 133, 5518, 44.5, 17, 27);
postgres=# insert into OdiStats values ('Jonathan', 68, 2819, 51.25, 4, 22);
postgres=# insert into OdiStats values ('Kumara', 404, 14234, 41.99, 25, 93);
postgres=# insert into OdiStats values ('Virat', 239, 11520, 60.31, 43, 54);
postgres=# insert into OdiStats values ('Rohit', 218, 8686, 48.53, 24, 42);
```

Following statement retrieves data combining the values in these two tables:

```
postgres=# SELECT
    Cricketers.First_Name, Cricketers.Last_Name, Cricketers.Country,
    OdiStats.matches, OdiStats.runs, OdiStats.centuries, OdiStats.halfcenturies
  from Cricketers INNER JOIN OdiStats ON Cricketers.First_Name = OdiStats.First_Name;
first_name | last_name | country | matches | runs | centuries | halfcenturies
-----+-----+-----+-----+-----+-----+-----
```

Shikhar	Dhawan	India	133	5518	17	27
Jonathan	Trott	SouthAfrica	68	2819	4	22
Kumara	Sangakkara	Srilanka	404	14234	25	93
Virat	Kohli	India	239	11520	43	54
Rohit	Sharma	India	218	8686	24	42

(5 rows)

postgres=#

Joins using python

When you have divided the data in two tables you can fetch combined records from these two tables using Joins.

Example

Following python program demonstrates the usage of the JOIN clause:

```
import psycopg2

#establishing the connection
conn = psycopg2.connect(database="mydb", user='postgres', password='password',
host='127.0.0.1', port= '5432')

#Setting auto commit false
conn.autocommit = True

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving single row
sql = '''SELECT * from EMP INNER JOIN CONTACT ON EMP.CONTACT = CONTACT.ID'''

#Executing the query
cursor.execute(sql)

#Fetching 1st row from the table
result = cursor.fetchall();

print(result)
```



```
#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

```
[('Ramya', 'Rama priya', 27, 'F', 9000.0, 101, 101, 'Krishna@mymail.com',
'Hyderabad'), ('Vinay', 'Battacharya', 20, 'M', 6000.0, 102, 102,
'Raja@mymail.com', 'Vishakhapatnam'), ('Sharukh', 'Sheik', 25, 'M', 8300.0,
103, 103, 'Krishna@mymail.com ', 'Pune'), ('Sarmista', 'Sharma', 26, 'F',
10000.0, 104, 104, 'Raja@mymail.com', 'Mumbai')]
```

28. Python PostgreSQL — Cursor Object

The `Cursor` class of the `psycopg` library provide methods to execute the PostgreSQL commands in the database using python code.

Using the methods of it you can execute SQL statements, fetch data from the result sets, call procedures.

You can create **Cursor** object using the `cursor()` method of the `Connection` object/class.

Example

```
import psycopg2

#establishing the connection
conn = psycopg2.connect(database="mydb", user='postgres', password='password',
host='127.0.0.1', port= '5432')

#Setting auto commit false
conn.autocommit = True

#Creating a cursor object using the cursor() method
cursor = conn.cursor()
```

Methods

Following are the various methods provided by the `Cursor` class/object.

Method	Description
<code>callproc()</code>	This method is used to call existing procedures PostgreSQL database.
<code>close()</code>	This method is used to close the current cursor object.
<code>executemany()</code>	This method accepts a list series of parameters list. Prepares an MySQL query and executes it with all the parameters.
<code>execute()</code>	This method accepts a MySQL query as a parameter and executes the given query.

fetchall()	This method retrieves all the rows in the result set of a query and returns them as list of tuples. (If we execute this after retrieving few rows it returns the remaining ones)
fetchone()	This method fetches the next row in the result of a query and returns it as a tuple.
fetchmany()	This method is similar to the fetchone() but, it retrieves the next set of rows in the result set of a query, instead of a single row.

Properties

Following are the properties of the Cursor class:

Property	Description
description	This is a read only property which returns the list containing the description of columns in a result-set.
lastrowid	This is a read only property, if there are any auto-incremented columns in the table, this returns the value generated for that column in the last INSERT or, UPDATE operation.
rowcount	This returns the number of rows returned/updated in case of SELECT and UPDATE operations.
closed	This property specifies whether a cursor is closed or not, if so it returns true, else false.
connection	This returns a reference to the connection object using which this cursor was created.
name	This property returns the name of the cursor.
scrollable	This property specifies whether a particular cursor is scrollable.

Python SQLite

29. Python SQLite — Introduction

Installation

SQLite3 can be integrated with Python using `sqlite3` module, which was written by Gerhard Haring. It provides an SQL interface compliant with the DB-API 2.0 specification described by PEP 249. You do not need to install this module separately because it is shipped by default along with Python version 2.5.x onwards.

To use `sqlite3` module, you must first create a connection object that represents the database and then optionally you can create a cursor object, which will help you in executing all the SQL statements.

Python `sqlite3` module APIs

Following are important `sqlite3` module routines, which can suffice your requirement to work with SQLite database from your Python program. If you are looking for a more sophisticated application, then you can look into Python `sqlite3` module's official documentation.

S.No.	API & Description
1	<code>sqlite3.connect(database [,timeout ,other optional arguments])</code> This API opens a connection to the SQLite database file. You can use <code>":memory:"</code> to open a database connection to a database that resides in RAM instead of on disk. If database is opened successfully, it returns a connection object.
2	<code>connection.cursor([cursorClass])</code> This routine creates a cursor which will be used throughout your database programming with Python. This method accepts a single optional parameter <code>cursorClass</code> . If supplied, this must be a custom cursor class that extends <code>sqlite3.Cursor</code> .
3	<code>cursor.execute(sql [, optional parameters])</code> This routine executes an SQL statement. The SQL statement may be parameterized (i. e. placeholders instead of SQL literals). The <code>sqlite3</code> module supports two kinds of placeholders: question marks and named placeholders (named style). For example – <code>cursor.execute("insert into people values (?, ?)", (who, age))</code>
4	<code>connection.execute(sql [, optional parameters])</code>

	This routine is a shortcut of the above execute method provided by the cursor object and it creates an intermediate cursor object by calling the cursor method, then calls the cursor's execute method with the parameters given.
5	cursor.executemany(sql, seq_of_parameters) This routine executes an SQL command against all parameter sequences or mappings found in the sequence sql.
6	connection.executemany(sql[, parameters]) This routine is a shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's executemany method with the parameters given.
7	cursor.executescript(sql_script) This routine executes multiple SQL statements at once provided in the form of script. It issues a COMMIT statement first, then executes the SQL script it gets as a parameter. All the SQL statements should be separated by a semi colon (;).
8	connection.executescript(sql_script) This routine is a shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's executescript method with the parameters given.
9	connection.total_changes() This routine returns the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.
10	connection.commit() This method commits the current transaction. If you don't call this method, anything you did since the last call to commit() is not visible from other database connections.
11	connection.rollback() This method rolls back any changes to the database since the last call to commit().
12	connection.close() This method closes the database connection. Note that this does not automatically call commit(). If you just close your database connection without calling commit() first, your changes will be lost!

13	cursor.fetchone() This method fetches the next row of a query result set, returning a single sequence, or None when no more data is available.
14	cursor.fetchmany([size = cursor.arraysize]) This routine fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available. The method tries to fetch as many rows as indicated by the size parameter.
15	cursor.fetchall() This routine fetches all (remaining) rows of a query result, returning a list. An empty list is returned when no rows are available.

30. Python SQLite — Establishing Connection

To establish connection with SQLite Open command prompt, browse through the location of where you have installed SQLite and just execute the command **sqlite3** as shown below:

```
C:\Users\Tutorialspoint>cd C:\sqlite
C:\sqlite>sqlite3
SQLite version 3.22.0 2018-01-22 18:45:57
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

Establishing connection using python

You can communicate with SQLite2 database using the SQLite3 python module. To do so, first of all you need to establish a connection (create a connection object).

To establish a connection with SQLite3 database using python you need to:

- Import the sqlite3 module using the import statement.
- The connect() method accepts the name of the database you need to connect with as a parameter and, returns a Connection object.

Example

```
import sqlite3
conn = sqlite3.connect('example.db')
```

Output

```
print("Connection established .....")
```


31. Python SQLite — Create Table

Using the SQLite CREATE TABLE statement you can create a table in a database.

Syntax

Following is the syntax to create a table in SQLite database:

```
CREATE TABLE database_name.table_name(  
    column1 datatype PRIMARY KEY(one or more columns),  
    column2 datatype,  
    column3 datatype,  
    .....  
    columnN datatype  
);
```

Example

Following SQLite query/statement creates a table with name **CRICKETERS** in SQLite database:

```
sqlite> CREATE TABLE CRICKETERS (  
    First_Name VARCHAR(255),  
    Last_Name VARCHAR(255),  
    Age int,  
    Place_Of_Birth VARCHAR(255),  
    Country VARCHAR(255)  
);  
sqlite>
```

Let us create one more table OdiStats describing the One-day cricket statistics of each player in CRICKETERS table.

```
sqlite> CREATE TABLE ODISTats (  
    First_Name VARCHAR(255),  
    Matches INT,  
    Runs INT,  
    AVG FLOAT,  
    Centuries INT,  
    HalfCenturies INT
```

```
);
sqlite>
```

You can get the list of tables in a database in SQLite database using the **.tables** command. After creating a table, if you can verify the list of tables you can observe the newly created table in it as:

```
sqlite> . tables
CRICKETERS    ODISStats
sqlite>
```

Creating a table using python

The Cursor object contains all the methods to execute queries and fetch data etc. The cursor method of the connection class returns a cursor object.

Therefore, to create a table in SQLite database using python:

- Establish connection with a database using the connect() method.
- Create a cursor object by invoking the cursor() method on the above created connection object.
- Now execute the CREATE TABLE statement using the execute() method of the Cursor class.

Example

Following Python program creates a table named Employee in SQLite3:

```
import sqlite3

#Connecting to sqlite
conn = sqlite3.connect('example.db')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Doping EMPLOYEE table if already exists.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

#Creating table as per requirement
sql = '''CREATE TABLE EMPLOYEE(
        FIRST_NAME  CHAR(20) NOT NULL,
        LAST_NAME   CHAR(20),
        AGE INT,
```

```
        SEX CHAR(1),
        INCOME FLOAT)'''
cursor.execute(sql)
print("Table created successfully.....")

# Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

```
Table created successfully.....
```

32. Python SQLite — Insert Data

You can add new rows to an existing table of SQLite using the `INSERT INTO` statement. In this, you need to specify the name of the table, column names, and values (in the same order as column names).

Syntax

Following is the recommended syntax of the `INSERT` statement:

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)
VALUES (value1, value2, value3,...valueN);
```

Where, `column1`, `column2`, `column3,..` are the names of the columns of a table and `value1`, `value2`, `value3,..` are the values you need to insert into the table.

Example

Assume we have created a table with name `CRICKETERS` using the `CREATE TABLE` statement as shown below:

```
sqlite> CREATE TABLE CRICKETERS (
    First_Name VARCHAR(255),
    Last_Name VARCHAR(255),
    Age int,
    Place_Of_Birth VARCHAR(255),
    Country VARCHAR(255)
);
sqlite>
```

Following PostgreSQL statement inserts a row in the above created table.

```
sqlite> insert into CRICKETERS (First_Name, Last_Name, Age, Place_Of_Birth,
Country) values('Shikhar', 'Dhawan', 33, 'Delhi', 'India');
sqlite>
```

While inserting records using the `INSERT INTO` statement, if you skip any columns names, this record will be inserted leaving empty spaces at columns which you have skipped.

```
sqlite> insert into CRICKETERS (First_Name, Last_Name, Country) values
('Jonathan', 'Trott', 'SouthAfrica');
sqlite>
```

You can also insert records into a table without specifying the column names, if the order of values you pass is same as their respective column names in the table.

```
sqlite> insert into CRICKETERS values('Kumara', 'Sangakkara', 41, 'Matale', 'Srilanka');
sqlite> insert into CRICKETERS values('Virat', 'Kohli', 30, 'Delhi', 'India');
sqlite> insert into CRICKETERS values('Rohit', 'Sharma', 32, 'Nagpur', 'India');
sqlite>
```

After inserting the records into a table you can verify its contents using the SELECT statement as shown below:

```
sqlite> select * from cricketers;
Shikhar | Dhawan | 33 | Delhi | India
Jonathan | Trott | | | SouthAfrica
Kumara | Sangakkara | 41 | Matale | Srilanka
Virat | Kohli | 30 | Delhi | India
Rohit | Sharma | 32 | Nagpur | India
sqlite>
```

Inserting data using python

To add records to an existing table in SQLite database:

- Import sqlite3 package.
- Create a connection object using the *connect()* method by passing the name of the database as a parameter to it.
- The **cursor()** method returns a cursor object using which you can communicate with SQLite3. Create a cursor object by invoking the cursor() object on the (above created) Connection object.
- Then, invoke the execute() method on the cursor object, by passing an INSERT statement as a parameter to it.

Example

Following python example inserts records into to a table named EMPLOYEE:

```
import sqlite3

#Connecting to sqlite
conn = sqlite3.connect('example.db')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

# Preparing SQL queries to INSERT a record into the database.
```

```
cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
      INCOME) VALUES ('Ramya', 'Rama Priya', 27, 'F', 9000)''')

cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
      INCOME) VALUES ('Vinay', 'Battacharya', 20, 'M', 6000)''')

cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
      INCOME) VALUES ('Sharukh', 'Sheik', 25, 'M', 8300)''')

cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
      INCOME) VALUES ('Sarmista', 'Sharma', 26, 'F', 10000)''')

cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
      INCOME) VALUES ('Tripthi', 'Mishra', 24, 'F', 6000)''')

# Commit your changes in the database
conn.commit()

print("Records inserted.....")

# Closing the connection
conn.close()
```

Output

```
Records inserted.....
```

33. Python SQLite — Select Data

You can retrieve data from an SQLite table using the SELECT query. This query/statement returns contents of the specified relation (table) in tabular form and it is called as result-set.

Syntax

Following is the syntax of the SELECT statement in SQLite:

```
SELECT column1, column2, columnN FROM table_name;
```

Example

Assume we have created a table with name CRICKETERS using the following query:

```
sqlite> CREATE TABLE CRICKETERS (  
    First_Name VARCHAR(255),  
    Last_Name VARCHAR(255),  
    Age int,  
    Place_Of_Birth VARCHAR(255),  
    Country VARCHAR(255)  
);  
sqlite>
```

And if we have inserted 5 records in to it using INSERT statements as:

```
sqlite> insert into CRICKETERS values('Shikhar', 'Dhawan', 33, 'Delhi', 'India');  
sqlite> insert into CRICKETERS values('Jonathan', 'Trott', 38, 'CapeTown',  
'SouthAfrica');  
sqlite> insert into CRICKETERS values('Kumara', 'Sangakkara', 41, 'Matale', 'Srilanka');  
sqlite> insert into CRICKETERS values('Virat', 'Kohli', 30, 'Delhi', 'India');  
sqlite> insert into CRICKETERS values('Rohit', 'Sharma', 32, 'Nagpur', 'India');  
sqlite>
```

Following SELECT query retrieves the values of the columns FIRST_NAME, LAST_NAME and, COUNTRY from the CRICKETERS table.

```
sqlite> SELECT FIRST_NAME, LAST_NAME, COUNTRY FROM CRICKETERS;  
Shikhar|Dhawan|India  
Jonathan|Trott|SouthAfrica  
Kumara|Sangakkara|Srilanka
```

```
Virat|Kohli|India
Rohit|Sharma|India
sqlite>
```

As you observe, the SELECT statement of the SQLite database just returns the records of the specified tables. To get a formatted output you need to set the **header**, and **mode** using the respective commands before the SELECT statement as shown below:

```
sqlite> .header on
sqlite> .mode column
sqlite> SELECT FIRST_NAME, LAST_NAME, COUNTRY FROM CRICKETERS;
First_Name  Last_Name      Country
-----
Shikhar     Dhawan         India
Jonathan    Trott          SouthAfric
Kumara      Sangakkara     Srilanka
Virat       Kohli           India
Rohit       Sharma          India
sqlite>
```

If you want to retrieve all the columns of each record, you need to replace the names of the columns with "*" as shown below:

```
sqlite> .header on
sqlite> .mode column
sqlite> SELECT * FROM CRICKETERS;
First_Name  Last_Name  Age      Place_Of_Birth  Country
-----
Shikhar     Dhawan     33       Delhi           India
Jonathan    Trott      38       CapeTown        SouthAfric
Kumara      Sangakkara 41       Matale          Srilanka
Virat       Kohli      30       Delhi           India
Rohit       Sharma     32       Nagpur          India
sqlite>
```

In *SQLite* by default the width of the columns is 10 values beyond this width are chopped (observe the country column of 2nd row in above table). You can set the width of each column to required value using the **.width** command, before retrieving the contents of a table as shown below:

```
sqlite> .width 10, 10, 4, 10, 13
sqlite> SELECT * FROM CRICKETERS;
```


First_Name	Last_Name	Age	Place_Of_B	Country
Shikhar	Dhawan	33	Delhi	India
Jonathan	Trott	38	CapeTown	SouthAfrica
Kumara	Sangakkara	41	Matale	Srilanka
Virat	Kohli	30	Delhi	India
Rohit	Sharma	32	Nagpur	India

sqlite>

Retrieving data using python

READ Operation on any database means to fetch some useful information from the database. You can fetch data from MYSQL using the fetch() method provided by the sqlite python module.

The sqlite3.Cursor class provides three methods namely fetchall(), fetchmany() and, fetchone() where,

- The fetchall() method retrieves all the rows in the result set of a query and returns them as list of tuples. (If we execute this after retrieving few rows it returns the remaining ones).
- The fetchone() method fetches the next row in the result of a query and returns it as a tuple.
- The fetchmany() method is similar to the fetchone() but, it retrieves the next set of rows in the result set of a query, instead of a single row.

Note: A result set is an object that is returned when a cursor object is used to query a table.

Example

Following example fetches all the rows of the EMPLOYEE table using the SELECT query and from the obtained result set initially, we are retrieving the first row using the fetchone() method and then fetching the remaining rows using the fetchall() method.

Following Python program shows how to fetch and display records from the COMPANY table created in the above example.

```
import sqlite3

#Connecting to sqlite
conn = sqlite3.connect('example.db')

#Creating a cursor object using the cursor() method
```

```
cursor = conn.cursor()

#Retrieving data
cursor.execute('''SELECT * from EMPLOYEE''')

#Fetching 1st row from the table
result = cursor.fetchone();
print(result)

#Fetching 1st row from the table
result = cursor.fetchall();
print(result)

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

```
('Ramy', 'Rama priya', 27, 'F', 9000.0)
[('Vinay', 'Battacharya', 20, 'M', 6000.0),
 ('Sharukh', 'Sheik', 25, 'M', 8300.0),
 ('Sarmista', 'Sharma', 26, 'F', 10000.0),
 ('Tripthi', 'Mishra', 24, 'F', 6000.0)]
```

34. Python SQLite — Where Clause

If you want to fetch, delete or, update particular rows of a table in SQLite, you need to use the where clause to specify condition to filter the rows of the table for the operation.

For example, if you have a SELECT statement with where clause, only the rows which satisfies the specified condition will be retrieved.

Syntax

Following is the syntax of the WHERE clause in SQLite:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [search_condition]
```

You can specify a search_condition using comparison or logical operators. like >, <, =, LIKE, NOT, etc. The following examples would make this concept clear.

Example

Assume we have created a table with name CRICKETERS using the following query:

```
sqlite> CREATE TABLE CRICKETERS (
    First_Name VARCHAR(255),
    Last_Name VARCHAR(255),
    Age int,
    Place_Of_Birth VARCHAR(255),
    Country VARCHAR(255)
);
sqlite>
```

And if we have inserted 5 records in to it using INSERT statements as:

```
sqlite> insert into CRICKETERS values('Shikhar', 'Dhawan', 33, 'Delhi',
'India');
sqlite> insert into CRICKETERS values('Jonathan', 'Trott', 38, 'CapeTown',
'SouthAfrica');
sqlite> insert into CRICKETERS values('Kumara', 'Sangakkara', 41, 'Matale',
'Srilanka');
sqlite> insert into CRICKETERS values('Virat', 'Kohli', 30, 'Delhi', 'India');
sqlite> insert into CRICKETERS values('Rohit', 'Sharma', 32, 'Nagpur',
'India');
```

```
sqlite>
```

Following SELECT statement retrieves the records whose age is greater than 35:

```
sqlite> SELECT * FROM CRICKETERS WHERE AGE > 35;
First_Name  Last_Name   Age  Place_Of_B  Country
-----
Jonathan    Trott       38   CapeTown    SouthAfrica
Kumara      Sangakkara  41   Matale      Srilanka
sqlite>
```

Where clause using python

The Cursor object/class contains all the methods to execute queries and fetch data, etc. The cursor method of the connection class returns a cursor object.

Therefore, to create a table in SQLite database using python:

- Establish connection with a database using the connect() method.
- Create a cursor object by invoking the cursor() method on the above created connection object.
- Now execute the CREATE TABLE statement using the execute() method of the Cursor class.

Example

Following example creates a table named Employee and populates it. Then using the where clause it retrieves the records with age value less than 23.

```
import sqlite3

#Connecting to sqlite
conn = sqlite3.connect('example.db')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Doping EMPLOYEE table if already exists.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

sql = '''CREATE TABLE EMPLOYEE(
        FIRST_NAME  CHAR(20) NOT NULL,
        LAST_NAME   CHAR(20),
```

```

        AGE INT,
        SEX CHAR(1),
        INCOME FLOAT)'''
cursor.execute(sql)

#Populating the table
cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
        INCOME) VALUES ('Ramya', 'Rama priya', 27, 'F', 9000)''')
cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
        INCOME) VALUES ('Vinay', 'Battacharya', 20, 'M', 6000)''')
cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
        INCOME) VALUES ('Sharukh', 'Sheik', 25, 'M', 8300)''')
cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
        INCOME) VALUES ('Sarmista', 'Sharma', 26, 'F', 10000)''')
cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,
        INCOME) VALUES ('Tripthi', 'Mishra', 24, 'F', 6000)''')

#Retrieving specific records using the where clause
cursor.execute("SELECT * from EMPLOYEE WHERE AGE <23")

print(cursor.fetchall())

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()

```

Output

```
[('Vinay', 'Battacharya', 20, 'M', 6000.0)]
```

35. Python SQLite — Order By

While fetching data using SELECT query, you will get the records in the same order in which you have inserted them.

You can sort the results in desired order (ascending or descending) using the **Order By** clause. By default, this clause sorts results in ascending order, if you need to arrange them in descending order you need to use "DESC" explicitly.

Syntax

Following is the syntax of the ORDER BY clause in SQLite.

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

Example

Assume we have created a table with name CRICKETERS using the following query:

```
sqlite> CREATE TABLE CRICKETERS (
    First_Name VARCHAR(255),
    Last_Name VARCHAR(255),
    Age int,
    Place_Of_Birth VARCHAR(255),
    Country VARCHAR(255)
);
sqlite>
```

And if we have inserted 5 records in to it using INSERT statements as:

```
sqlite> insert into CRICKETERS values('Shikhar', 'Dhawan', 33, 'Delhi',
'India');
sqlite> insert into CRICKETERS values('Jonathan', 'Trott', 38, 'CapeTown',
'SouthAfrica');
sqlite> insert into CRICKETERS values('Kumara', 'Sangakkara', 41, 'Matale',
'Srilanka');
sqlite> insert into CRICKETERS values('Virat', 'Kohli', 30, 'Delhi', 'India');
sqlite> insert into CRICKETERS values('Rohit', 'Sharma', 32, 'Nagpur',
'India');
```

```
sqlite>
```

Following SELECT statement retrieves the rows of the CRICKETERS table in the ascending order of their age:

```
sqlite> SELECT * FROM CRICKETERS ORDER BY AGE;
First_Name Last_Name Age Place_Of_B Country
-----
Virat Kohli 30 Delhi India
Rohit Sharma 32 Nagpur India
Shikhar Dhawan 33 Delhi India
Jonathan Trott 38 CapeTown SouthAfrica
Kumara Sangakkara 41 Matale Srilanka
sqlite>
```

You can use more than one column to sort the records of a table. Following SELECT statements sorts the records of the CRICKETERS table based on the columns *AGE* and *FIRST_NAME*.

```
sqlite> SELECT * FROM CRICKETERS ORDER BY AGE, FIRST_NAME;
First_Name Last_Name Age Place_Of_B Country
-----
Virat Kohli 30 Delhi India
Rohit Sharma 32 Nagpur India
Shikhar Dhawan 33 Delhi India
Jonathan Trott 38 CapeTown SouthAfrica
Kumara Sangakkara 41 Matale Srilanka
sqlite>
```

By default, the **ORDER BY** clause sorts the records of a table in ascending order you can arrange the results in descending order using DESC as:

```
sqlite> SELECT * FROM CRICKETERS ORDER BY AGE DESC;
First_Name Last_Name Age Place_Of_B Country
-----
Kumara Sangakkara 41 Matale Srilanka
Jonathan Trott 38 CapeTown SouthAfrica
Shikhar Dhawan 33 Delhi India
Rohit Sharma 32 Nagpur India
Virat Kohli 30 Delhi India
sqlite>
```

ORDER BY clause using python

To retrieve contents of a table in specific order, invoke the execute() method on the cursor object and, pass the SELECT statement along with ORDER BY clause, as a parameter to it.

Example

In the following example we are creating a table with name and Employee, populating it, and retrieving its records back in the (ascending) order of their age, using the ORDER BY clause.

```
import psycopg2

#establishing the connection
conn = psycopg2.connect(database="mydb", user='postgres', password='password',
host='127.0.0.1', port= '5432')

#Setting auto commit false
conn.autocommit = True

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Doping EMPLOYEE table if already exists.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")
#Creating a table
sql = '''CREATE TABLE EMPLOYEE(
            FIRST_NAME  CHAR(20) NOT NULL,
            LAST_NAME   CHAR(20),
            AGE INT, SEX CHAR(1),
            INCOME INT,
            CONTACT INT)'''
cursor.execute(sql)

#Populating the table
#Populating the table
cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)
VALUES ('Ramy', 'Rama priya', 27, 'F', 9000),('Vinay', 'Battacharya', 20, 'M',
6000), ('Sharukh', 'Sheik', 25, 'M', 8300), ('Sarmista', 'Sharma', 26, 'F',
10000),('Tripthi', 'Mishra', 24, 'F', 6000)''')
```



```
conn.commit()

#Retrieving specific records using the ORDER BY clause
cursor.execute("SELECT * from EMPLOYEE ORDER BY AGE")

print(cursor.fetchall())

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

```
[('Vinay', 'Battacharya', 20, 'M', 6000, None),
 ('Tripthi', 'Mishra', 24, 'F', 6000, None),
 ('Sharukh', 'Sheik', 25, 'M', 8300, None),
 ('Sarmista', 'Sharma', 26, 'F', 10000, None),
 ('Ramya', 'Rama priya', 27, 'F', 9000, None)]
```

36. Python SQLite — Update Table

UPDATE Operation on any database implies modifying the values of one or more records of a table, which are already available in the database. You can update the values of existing records in SQLite using the UPDATE statement.

To update specific rows, you need to use the WHERE clause along with it.

Syntax

Following is the syntax of the UPDATE statement in SQLite:

```
UPDATE table_name
SET column1 = value1, column2 = value2..., columnN = valueN
WHERE [condition];
```

Example

Assume we have created a table with name CRICKETERS using the following query:

```
sqlite> CREATE TABLE CRICKETERS (
    First_Name VARCHAR(255),
    Last_Name VARCHAR(255),
    Age int,
    Place_Of_Birth VARCHAR(255),
    Country VARCHAR(255)
);
sqlite>
```

And if we have inserted 5 records in to it using INSERT statements as:

```
sqlite> insert into CRICKETERS values('Shikhar', 'Dhawan', 33, 'Delhi',
'India');
sqlite> insert into CRICKETERS values('Jonathan', 'Trott', 38, 'CapeTown',
'SouthAfrica');
sqlite> insert into CRICKETERS values('Kumara', 'Sangakkara', 41, 'Matale',
'Srilanka');
sqlite> insert into CRICKETERS values('Virat', 'Kohli', 30, 'Delhi', 'India');
sqlite> insert into CRICKETERS values('Rohit', 'Sharma', 32, 'Nagpur',
'India');
sqlite>
```

Following Statement modifies the age of the cricketer, whose first name is **Shikhar**:

```
sqlite> UPDATE CRICKETERS SET AGE = 45 WHERE FIRST_NAME = 'Shikhar' ;
sqlite>
```

If you retrieve the record whose FIRST_NAME is Shikhar you observe that the age value has been changed to 45:

```
sqlite> SELECT * FROM CRICKETERS WHERE FIRST_NAME = 'Shikhar';
First_Name  Last_Name  Age  Place_Of_B  Country
-----
Shikhar     Dhawan    45   Delhi       India
sqlite>
```

If you haven't used the WHERE clause values of all the records will be updated. Following UPDATE statement increases the age of all the records in the CRICKETERS table by 1:

```
sqlite> UPDATE CRICKETERS SET AGE = AGE+1;
sqlite>
```

If you retrieve the contents of the table using SELECT command, you can see the updated values as:

```
sqlite> SELECT * FROM CRICKETERS;
First_Name  Last_Name  Age  Place_Of_B  Country
-----
Shikhar     Dhawan    46   Delhi       India
Jonathan    Trott     39   CapeTown    SouthAfrica
Kumara      Sangakkara 42   Matale     Srilanka
Virat       Kohli     31   Delhi       India
Rohit       Sharma    33   Nagpur     India
sqlite>
```

Updating existing records using python

To add records to an existing table in SQLite database:

- Import sqlite3 package.
- Create a connection object using the *connect()* method by passing the name of the database as a parameter to it.
- The *cursor()* method returns a cursor object using which you can communicate with SQLite3 . Create a cursor object by invoking the cursor() object on the (above created) Connection object.

- Then, invoke the `execute()` method on the cursor object, by passing an UPDATE statement as a parameter to it.

Example

Following Python example, creates a table with name EMPLOYEE, inserts 5 records into it and, increases the age of all the male employees by 1:

```
import sqlite3

#Connecting to sqlite
conn = sqlite3.connect('example.db')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Doping EMPLOYEE table if already exists.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

#Creating table as per requirement
sql = '''CREATE TABLE EMPLOYEE(
    FIRST_NAME CHAR(20) NOT NULL,
    LAST_NAME CHAR(20),
    AGE INT,
    SEX CHAR(1),
    INCOME FLOAT)'''
cursor.execute(sql)

#Inserting data
cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)
VALUES ('Ramya', 'Rama priya', 27, 'F', 9000),('Vinay', 'Battacharya', 20, 'M',
6000), ('Sharukh', 'Sheik', 25, 'M', 8300), ('Sarmista', 'Sharma', 26, 'F',
10000),('Tripthi', 'Mishra', 24, 'F', 6000)''')
conn.commit()

#Fetching all the rows before the update
print("Contents of the Employee table: ")
cursor.execute('''SELECT * from EMPLOYEE''')
print(cursor.fetchall())

#Updating the records
```

```

sql = '''UPDATE EMPLOYEE SET AGE=AGE+1 WHERE SEX = 'M' '''
cursor.execute(sql)
print("Table updated..... ")

#Fetching all the rows after the update
print("Contents of the Employee table after the update operation: ")
cursor.execute('''SELECT * from EMPLOYEE''')
print(cursor.fetchall())

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()

```

Output

```

Contents of the Employee table:
[('Ramya', 'Rama priya', 27, 'F', 9000.0), ('Vinay', 'Battacharya', 20, 'M',
6000.0), ('Sharukh', 'Sheik', 25, 'M', 8300.0), ('Sarmista', 'Sharma', 26, 'F',
10000.0), ('Tripthi', 'Mishra', 24, 'F', 6000.0)]
Table updated.....
Contents of the Employee table after the update operation:
[('Ramya', 'Rama priya', 27, 'F', 9000.0), ('Vinay', 'Battacharya', 21, 'M',
6000.0), ('Sharukh', 'Sheik', 26, 'M', 8300.0), ('Sarmista', 'Sharma', 26, 'F',
10000.0), ('Tripthi', 'Mishra', 24, 'F', 6000.0)]

```

37. Python SQLite — Delete Data

To delete records from a SQLite table, you need to use the DELETE FROM statement. To remove specific records, you need to use WHERE clause along with it.

Syntax

Following is the syntax of the DELETE query in SQLite:

```
DELETE FROM table_name [WHERE Clause]
```

Example

Assume we have created a table with name CRICKETERS using the following query:

```
sqlite> CREATE TABLE CRICKETERS (  
    First_Name VARCHAR(255),  
    Last_Name VARCHAR(255),  
    Age int,  
    Place_Of_Birth VARCHAR(255),  
    Country VARCHAR(255)  
);  
sqlite>
```

And if we have inserted 5 records in to it using INSERT statements as:

```
sqlite> insert into CRICKETERS values('Shikhar', 'Dhawan', 33, 'Delhi',  
'India');  
sqlite> insert into CRICKETERS values('Jonathan', 'Trott', 38, 'CapeTown',  
'SouthAfrica');  
sqlite> insert into CRICKETERS values('Kumara', 'Sangakkara', 41, 'Matale',  
'Srilanka');  
sqlite> insert into CRICKETERS values('Virat', 'Kohli', 30, 'Delhi', 'India');  
sqlite> insert into CRICKETERS values('Rohit', 'Sharma', 32, 'Nagpur',  
'India');  
sqlite>
```

Following statement deletes the record of the cricketer whose last name is 'Sangakkara'.

```
sqlite> DELETE FROM CRICKETERS WHERE LAST_NAME = 'Sangakkara';  
sqlite>
```

If you retrieve the contents of the table using the SELECT statement, you can see only 4 records since we have deleted one.

```
sqlite> SELECT * FROM CRICKETERS;
First_Name  Last_Name  Age  Place_Of_B  Country
-----
Shikhar     Dhawan     46   Delhi       India
Jonathan    Trott      39   CapeTown    SouthAfrica
Virat       Kohli      31   Delhi       India
Rohit       Sharma     33   Nagpur      India
sqlite>
```

If you execute the DELETE FROM statement without the WHERE clause, all the records from the specified table will be deleted.

```
sqlite> DELETE FROM CRICKETERS;
sqlite>
```

Since you have deleted all the records, if you try to retrieve the contents of the CRICKETERS table, using SELECT statement you will get an empty result set as shown below:

```
sqlite> SELECT * FROM CRICKETERS;
sqlite>
```

Deleting data using python

To add records to an existing table in SQLite database:

- Import sqlite3 package.
- Create a connection object using the *connect()* method by passing the name of the database as a parameter to it.
- The *cursor()* method returns a cursor object using which you can communicate with SQLite3 . Create a cursor object by invoking the cursor() object on the (above created) Connection object.
- Then, invoke the execute() method on the cursor object, by passing an DELETE statement as a parameter to it.

Example

Following python example deletes the records from EMPLOYEE table with age value greater than 25.

```
import sqlite3
#Connecting to sqlite
conn = sqlite3.connect('example.db')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving contents of the table
print("Contents of the table: ")
cursor.execute('''SELECT * from EMPLOYEE''')
print(cursor.fetchall())

#Deleting records
cursor.execute('''DELETE FROM EMPLOYEE WHERE AGE > 25''')

#Retrieving data after delete
print("Contents of the table after delete operation ")
cursor.execute("SELECT * from EMPLOYEE")
print(cursor.fetchall())

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

```
Contents of the table:
[('Ramya', 'Rama priya', 27, 'F', 9000.0), ('Vinay', 'Battacharya', 21, 'M',
6000.0), ('Sharukh', 'Sheik', 26, 'M', 8300.0), ('Sarmista', 'Sharma', 26, 'F',
10000.0), ('Tripthi', 'Mishra', 24, 'F', 6000.0)]
Contents of the table after delete operation
[('Vinay', 'Battacharya', 21, 'M', 6000.0), ('Tripthi', 'Mishra', 24, 'F',
6000.0)]
```


38. Python SQLite — Drop Table

You can remove an entire table using the DROP TABLE statement. You just need to specify the name of the table you need to delete.

Syntax

Following is the syntax of the DROP TABLE statement in PostgreSQL:

```
DROP TABLE table_name;
```

Example

Assume we have created two tables with name CRICKETERS and EMPLOYEES using the following queries:

```
sqlite> CREATE TABLE CRICKETERS (First_Name VARCHAR(255), Last_Name  
VARCHAR(255), Age int, Place_Of_Birth VARCHAR(255), Country VARCHAR(255));  
sqlite> CREATE TABLE EMPLOYEE(FIRST_NAME CHAR(20) NOT NULL, LAST_NAME  
CHAR(20), AGE INT, SEX CHAR(1), INCOME FLOAT);  
sqlite>
```

Now if you verify the list of tables using the **.tables** command, you can see the above created tables in it (list) as:

```
sqlite> .tables  
CRICKETERS  EMPLOYEE  
sqlite>
```

Following statement deletes the table named Employee from the database:

```
sqlite> DROP table employee;  
sqlite>
```

Since you have deleted the Employee table, if you retrieve the list of tables again, you can observe only one table in it.

```
sqlite> .tables  
CRICKETERS  
sqlite>
```

If you try to delete the Employee table again, since you have already deleted it you will get an error saying "no such table" as shown below:

```
sqlite> DROP table employee;
```

```
Error: no such table: employee
sqlite>
```

To resolve this, you can use the IF EXISTS clause along with the DELTE statement. This removes the table if it exists else skips the DLETE operation.

```
sqlite> DROP table IF EXISTS employee;
sqlite>
```

Dropping a table using Python

You can drop a table whenever you need to, using the DROP statement of MYSQL, but you need to be very careful while deleting any existing table because the data lost will not be recovered after deleting a table.

Example

To drop a table from a SQLite3 database using python invoke the **execute()** method on the cursor object and pass the drop statement as a parameter to it.

```
import sqlite3
#Connecting to sqlite
conn = sqlite3.connect('example.db')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Doping EMPLOYEE table if already exists
cursor.execute("DROP TABLE emp")
print("Table dropped... ")

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

```
Table dropped...
```

39. Python SQLite — Limit

While fetching records if you want to limit them by a particular number, you can do so, using the LIMIT clause of SQLite.

Syntax

Following is the syntax of the LIMIT clause in SQLite:

```
SELECT column1, column2, columnN
FROM table_name
LIMIT [no of rows]
```

Example

Assume we have created a table with name CRICKETERS using the following query:

```
sqlite> CREATE TABLE CRICKETERS (
    First_Name VARCHAR(255),
    Last_Name VARCHAR(255),
    Age int,
    Place_Of_Birth VARCHAR(255),
    Country VARCHAR(255)
);
sqlite>
```

And if we have inserted 5 records in to it using INSERT statements as:

```
sqlite> insert into CRICKETERS values('Shikhar', 'Dhawan', 33, 'Delhi',
'India');
sqlite> insert into CRICKETERS values('Jonathan', 'Trott', 38, 'CapeTown',
'SouthAfrica');
sqlite> insert into CRICKETERS values('Kumara', 'Sangakkara', 41, 'Matale',
'Srilanka');
sqlite> insert into CRICKETERS values('Virat', 'Kohli', 30, 'Delhi', 'India');
sqlite> insert into CRICKETERS values('Rohit', 'Sharma', 32, 'Nagpur',
'India');
sqlite>
```

Following statement retrieves the first 3 records of the Cricketers table using the LIMIT clause:

```

sqlite> SELECT * FROM CRICKETERS LIMIT 3;
First_Name  Last_Name  Age  Place_Of_B  Country
-----
Shikhar     Dhawan    33   Delhi       India
Jonathan    Trott     38   CapeTown    SouthAfrica
Kumara      Sangakkara 41   Matale      Srilanka
sqlite>

```

If you need to limit the records starting from nth record (not 1st), you can do so, using OFFSET along with LIMIT.

```

sqlite> SELECT * FROM CRICKETERS LIMIT 3 OFFSET 2;
First_Name  Last_Name  Age  Place_Of_B  Country
-----
Kumara      Sangakkara 41   Matale      Srilanka
Virat       Kohli     30   Delhi       India
Rohit       Sharma    32   Nagpur      India
sqlite>

```

LIMIT clause using Python

If you Invoke the execute() method on the cursor object by passing the SELECT query along with the LIMIT clause, you can retrieve required number of records.

Example

Following python example retrieves the first two records of the EMPLOYEE table using the LIMIT clause.

```

import sqlite3

#Connecting to sqlite
conn = sqlite3.connect('example.db')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving single row
sql = '''SELECT * from EMPLOYEE LIMIT 3'''

```

```
#Executing the query
cursor.execute(sql)

#Fetching the data
result = cursor.fetchall();
print(result)

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

```
[('Ramya', 'Rama priya', 27, 'F', 9000.0), ('Vinay', 'Battacharya', 20, 'M', 6000.0), ('Sharukh', 'Sheik', 25, 'M', 8300.0)]
```

40. Python SQLite — Join

When you have divided the data in two tables you can fetch combined records from these two tables using Joins.

Example

Assume we have created a table with name CRICKETERS using the following query:

```
sqlite> CREATE TABLE CRICKETERS (  
    First_Name VARCHAR(255),  
    Last_Name VARCHAR(255),  
    Age int,  
    Place_Of_Birth VARCHAR(255),  
    Country VARCHAR(255)  
);  
sqlite>
```

Let us create one more table OdiStats describing the One-day cricket statistics of each player in CRICKETERS table.

```
sqlite> CREATE TABLE ODiStats (  
    First_Name VARCHAR(255),  
    Matches INT,  
    Runs INT,  
    AVG FLOAT,  
    Centuries INT,  
    HalfCenturies INT  
);  
sqlite>
```

Following statement retrieves data combining the values in these two tables:

```
sqlite> SELECT  
    Cricketers.First_Name, Cricketers.Last_Name, Cricketers.Country,  
    OdiStats.matches, OdiStats.runs, OdiStats.centuries, OdiStats.halfcenturies  
    from Cricketers INNER JOIN OdiStats ON Cricketers.First_Name = OdiStats.First_Name;  
First_Name  Last_Name  Country  Matches  Runs  Centuries  HalfCenturies  
-----  
Shikhar    Dhawan    Indi    133    5518    17    27
```

Jonathan	Trott	Sout	68	2819	4	22
Kumara	Sangakkara	Sril	404	14234	25	93
Virat	Kohli	Indi	239	11520	43	54
Rohit	Sharma	Indi	218	8686	24	42

sqlite>

Join clause using python

Following SQLite example, demonstrates the JOIN clause using python:

```
import sqlite3

#Connecting to sqlite
conn = sqlite3.connect('example.db')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()

#Retrieving data
sql = '''SELECT * from EMP INNER JOIN CONTACT ON EMP.CONTACT = CONTACT.ID'''

#Executing the query
cursor.execute(sql)

#Fetching 1st row from the table
result = cursor.fetchall();

print(result)

#Commit your changes in the database
conn.commit()

#Closing the connection
conn.close()
```

Output

```
[('Ramya', 'Rama priya', 27, 'F', 9000.0, 101, 101, 'Krishna@mymail.com', 'Hyderabad'), ('Vinay', 'Battacharya', 20, 'M', 6000.0, 102, 102,
```

```
'Raja@mymail.com', 'Vishakhapatnam'), ('Sharukh', 'Sheik', 25, 'M', 8300.0,  
103, 103, 'Krishna@mymail.com', 'Pune'), ('Sarmista', 'Sharma', 26, 'F',  
10000.0, 104, 104, 'Raja@mymail.com', 'Mumbai']
```


41. Python SQLite — Cursor Object

The `sqlite3.Cursor` class is an instance using which you can invoke methods that execute SQLite statements, fetch data from the result sets of the queries. You can create **Cursor** object using the `cursor()` method of the Connection object/class.

Example

```
import sqlite3

#Connecting to sqlite
conn = sqlite3.connect('example.db')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()
```

Methods

Following are the various methods provided by the Cursor class/object.

Method	Description
<code>execute()</code>	This routine executes an SQL statement. The SQL statement may be parameterized (i.e., placeholders instead of SQL literals). The <code>psycopg2</code> module supports placeholder using <code>%s</code> sign For example: <code>cursor.execute("insert into people values (%s, %s)", (who, age))</code>
<code>executemany()</code>	This routine executes an SQL command against all parameter sequences or mappings found in the sequence <code>sql</code> .
<code>fetchone()</code>	This method fetches the next row of a query result set, returning a single sequence, or <code>None</code> when no more data is available.
<code>fetchmany()</code>	This routine fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available. The method tries to fetch as many rows as indicated by the size parameter.
<code>fetchall()</code>	This routine fetches all (remaining) rows of a query result, returning a list. An empty list is returned when no rows are available.

Properties

Following are the properties of the Cursor class:

Method	Description
arraySize	This is a read/write property you can set the number of rows returned by the fetchmany() method.
description	This is a read only property which returns the list containing the description of columns in a result-set.
lastrowid	This is a read only property, if there are any auto-incremented columns in the table, this returns the value generated for that column in the last INSERT or, UPDATE operation.
rowcount	This returns the number of rows returned/updated in case of SELECT and UPDATE operations.
connection	This read-only attribute provides the SQLite database Connection used by the Cursor object.

Python MongoDB

42. Python MongoDB — Introduction

Pymongo is a python distribution which provides tools to work with MongoDB, it is the most preferred way to communicate with MongoDB database from python.

Installation

To install pymongo first of all make sure you have installed python3 (along with PIP) and MongoDB properly. Then execute the following command.

```
C:\WINDOWS\system32>pip install pymongo
Collecting pymongo
Using cached
https://files.pythonhosted.org/packages/cb/a6/b0ae3781b0ad75825e00e29dc5489b53512625e02328d73556e1ecdf12f8/pymongo-3.9.0-cp37-cp37m-win32.whl
Installing collected packages: pymongo
Successfully installed pymongo-3.9.0
```

Verification

Once you have installed pymongo, open a new text document, paste the following line in it and, save it as test.py.

```
import pymongo
```

If you have installed pymongo properly, if you execute the test.py as shown below, you should not get any issues.

```
D:\Python_MongoDB>test.py
D:\Python_MongoDB>
```

43. Python MongoDB — Create Database

Unlike other databases, MongoDB does not provide separate command to create a database.

In general, the use command is used to select/switch to the specific database. This command initially verifies whether the database we specify exists, if so, it connects to it. If the database, we specify with the use command doesn't exist a new database will be created.

Therefore, you can create a database in MongoDB using the **Use** command.

Syntax

Basic syntax of **use DATABASE** statement is as follows:

```
use DATABASE_NAME
```

Example

Following command creates a database named in mydb.

```
>use mydb
switched to db mydb
```

You can verify your creation by using the *db* command, this displays the current database.

```
>db
mydb
```

Creating database using python

To connect to MongoDB using pymongo, you need to import and create a MongoClient, then you can directly access the database you need to create in attribute passion.

Example

Following example creates a database in MangoDB.

```
from pymongo import MongoClient

#Creating a pymongo client
client = MongoClient('localhost', 27017)
```

```
#Getting the database instance
db = client['mydb']

print("Database created.....")

#Verification
print("List of databases after creating new one")
print(client.list_database_names())
```

Output

```
Database created.....
List of databases after creating new one:
['admin', 'config', 'local', 'mydb']
```

You can also specify the port and host names while creating a MongoClient and can access the databases in dictionary style.

Example

```
from pymongo import MongoClient

#Creating a pymongo client
client = MongoClient('localhost', 27017)

#Getting the database instance
db = client['mydb']

print("Database created.....")
```

Output

```
Database created.....
```

44. Python MongoDB — Create Collection

A collection in MongoDB holds a set of documents, it is analogous to a table in relational databases.

You can create a collection using the ***createCollection()*** method. This method accepts a String value representing the name of the collection to be created and an options (optional) parameter.

Using this you can specify the following:

- The *size* of the collection.
- The *max* number of documents allowed in the capped collection.
- Whether the collection we create should be capped collection (fixed size collection).
- Whether the collection we create should be auto-indexed.

Syntax

Following is the syntax to create a collection in MongoDB.

```
db.createCollection("CollectionName")
```

Example

Following method creates a collection named ExampleCollection.

```
> use mydb
switched to db mydb
> db.createCollection("ExampleCollection")
{ "ok" : 1 }
>
```

Similarly, following is a query that creates a collection using the options of the createCollection() method.

```
>db.createCollection("mycol", { capped : true, autoIndexId : true, size :
    6142800, max : 10000 } )
{ "ok" : 1 }
>
```

Creating a collection using python

Following python example connects to a database in MongoDB (mydb) and, creates a collection in it.

Example

```
from pymongo import MongoClient

#Creating a pymongo client
client = MongoClient('localhost', 27017)

#Getting the database instance
db = client['mydb']

#Creating a collection
collection = db['example']

print("Collection created.....")
```

Output

```
Collection created.....
```


45. Python MongoDB — Insert Document

You can store documents into MongoDB using the *insert()* method. This method accepts a JSON document as a parameter.

Syntax

Following is the syntax of the insert method.

```
>db.COLLECTION_NAME.insert(DOCUMENT_NAME)
```

Example

```
> use mydb
switched to db mydb
> db.createCollection("sample")
{ "ok" : 1 }
> doc1 = {"name": "Ram", "age": "26", "city": "Hyderabad"}
{ "name" : "Ram", "age" : "26", "city" : "Hyderabad" }
> db.sample.insert(doc1)
WriteResult({ "nInserted" : 1 })
>
```

Similarly, you can also insert multiple documents using the *insert()* method.

```
> use testDB
switched to db testDB
> db.createCollection("sample")
{ "ok" : 1 }
> data = [{"_id": "1001", "name": "Ram", "age": "26", "city": "Hyderabad"},
{"_id": "1002", "name": "Rahim", "age": 27, "city": "Bangalore" }, {"_id":
"1003", "name": "Robert", "age": 28, "city": "Mumbai" }]
[
  {
    "_id" : "1001",
    "name" : "Ram",
    "age" : "26",
    "city" : "Hyderabad"
  },
  {
```

```

        "_id" : "1002",
        "name" : "Rahim",
        "age" : 27,
        "city" : "Bangalore"
    },
    {
        "_id" : "1003",
        "name" : "Robert",
        "age" : 28,
        "city" : "Mumbai"
    }
]
> db.sample.insert(data)
BulkWriteResult({
  "writeErrors" : [ ],
  "writeConcernErrors" : [ ],
  "nInserted" : 3,
  "nUpserted" : 0,
  "nMatched" : 0,
  "nModified" : 0,
  "nRemoved" : 0,
  "upserted" : [ ]
})
>

```

Creating a collection using python

Pymongo provides a method named `insert_one()` to insert a document in MongoDB. To this method, we need to pass the document in dictionary format.

Example

Following example inserts a document in the collection named example.

```

from pymongo import MongoClient

#Creating a pymongo client
client = MongoClient('localhost', 27017)

```

```
#Getting the database instance
db = client['mydb']

#Creating a collection
coll = db['example']

#Inserting document into a collection
doc1 = {"name": "Ram", "age": "26", "city": "Hyderabad"}
coll.insert_one(doc1)

print(coll.find_one())
```

Output

```
{'_id': ObjectId('5d63ad6ce043e2a93885858b'), 'name': 'Ram', 'age': '26',
'city': 'Hyderabad'}
```

To insert multiple documents into MongoDB using pymongo, you need to invoke the `insert_many()` method.

```
from pymongo import MongoClient

#Creating a pymongo client
client = MongoClient('localhost', 27017)

#Getting the database instance
db = client['mydb']

#Creating a collection
coll = db['example']

#Inserting document into a collection
data = [{"_id": "101", "name": "Ram", "age": "26", "city": "Hyderabad"},
{"_id": "102", "name": "Rahim", "age": "27", "city": "Bangalore"},
{"_id": "103", "name": "Robert", "age": "28", "city": "Mumbai"}]

res = coll.insert_many(data)
print("Data inserted .....")
print(res.inserted_ids)
```

Output

```
Data inserted .....  
['101', '102', '103']
```

46. Python MongoDB — Find

You can read/retrieve stored documents from MongoDB using the ***find()*** method. This method retrieves and displays all the documents in MongoDB in a non-structured way.

Syntax

Following is the syntax of the ***find()*** method.

```
>db.CollectionName.find()
```

Example

Assume we have inserted 3 documents into a database named testDB in a collection named sample using the following queries:

```
> use testDB
> db.createCollection("sample")
> data = [
{"_id": "1001", "name": "Ram", "age": "26", "city": "Hyderabad"},
{"_id": "1002", "name" : "Rahim", "age" : 27, "city" : "Bangalore" },
{"_id": "1003", "name" : "Robert", "age" : 28, "city" : "Mumbai" }]
> db.sample.insert(data)
```

You can retrieve the inserted documents using the `find()` method as:

```
> use testDB
switched to db testDB
> db.sample.find()
{ "_id" : "1001", "name" : "Ram", "age" : "26", "city" : "Hyderabad" }
{ "_id" : "1002", "name" : "Rahim", "age" : 27, "city" : "Bangalore" }
{ "_id" : "1003", "name" : "Robert", "age" : 28, "city" : "Mumbai" }
>
```

You can also retrieve first document in the collection using the `findOne()` method as:

```
> db.sample.findOne()
{ "_id" : "1001", "name" : "Ram", "age" : "26", "city" : "Hyderabad" }
```

Retrieving data (find) using python

The ***find_One()*** method of pymongo is used to retrieve a single document based on your query, in case of no matches this method returns nothing and if you doesn't use any query it returns the first document of the collection.

This method comes handy whenever you need to retrieve only one document of a result or, if you are sure that your query returns only one document.

Example

Following python example retrieve first document of a collection:

```
from pymongo import MongoClient

#Creating a pymongo client
client = MongoClient('localhost', 27017)

#Getting the database instance
db = client['mydatabase']

#Creating a collection
coll = db['example']

#Inserting document into a collection
data = [{"_id": "101", "name": "Ram", "age": "26", "city": "Hyderabad"},
        {"_id": "102", "name": "Rahim", "age": "27", "city": "Bangalore"},
        {"_id": "103", "name": "Robert", "age": "28", "city": "Mumbai"}]

res = coll.insert_many(data)
print("Data inserted .....")
print(res.inserted_ids)

#Retrieving the first record using the find_one() method
print("First record of the collection: ")
print(coll.find_one())

#Retrieving a record with id 103 using the find_one() method
print("Record whose id is 103: ")
print(coll.find_one({"_id": "103"}))
```

Output

```
Data inserted .....
['101', '102', '103']
First record of the collection:
{'_id': '101', 'name': 'Ram', 'age': '26', 'city': 'Hyderabad'}
Record whose id is 103:
{'_id': '103', 'name': 'Robert', 'age': '28', 'city': 'Mumbai'}
```

To get multiple documents in a single query (single call of find method), you can use the **find()** method of the pymongo. If haven't passed any query, this returns all the documents of a collection and, if you have passed a query to this method, it returns all the matched documents.

Example

```
#Getting the database instance
db = client['myDB']

#Creating a collection
coll = db['example']

#Inserting document into a collection
data = [{"_id": "101", "name": "Ram", "age": "26", "city": "Hyderabad"},
        {"_id": "102", "name": "Rahim", "age": "27", "city": "Bangalore"},
        {"_id": "103", "name": "Robert", "age": "28", "city": "Mumbai"}]

res = coll.insert_many(data)
print("Data inserted .....")

#Retrieving all the records using the find() method
print("Records of the collection: ")
for doc1 in coll.find():
    print(doc1)

#Retrieving records with age greater than 26 using the find() method
print("Record whose age is more than 26: ")
for doc2 in coll.find({"age":{"$gt":"26"}}):
    print(doc2)
```

Output

```
Data inserted .....  
Records of the collection:  
{'_id': '101', 'name': 'Ram', 'age': '26', 'city': 'Hyderabad'}  
{'_id': '102', 'name': 'Rahim', 'age': '27', 'city': 'Bangalore'}  
{'_id': '103', 'name': 'Robert', 'age': '28', 'city': 'Mumbai'}  
Record whose age is more than 26:  
{'_id': '102', 'name': 'Rahim', 'age': '27', 'city': 'Bangalore'}  
{'_id': '103', 'name': 'Robert', 'age': '28', 'city': 'Mumbai'}
```


47. Python MongoDB — Query

While retrieving using **find()** method, you can filter the documents using the query object. You can pass the query specifying the condition for the required documents as a parameter to this method.

Operators

Following is the list of operators used in the queries in MongoDB.

Operation	Syntax	Example
Equality	<code>{"key" : "value"}</code>	<code>db.mycol.find({"by":"tutorials point"})</code>
Less Than	<code>{"key" : {\$lt:"value"}}</code>	<code>db.mycol.find({"likes":{\$lt:50}})</code>
Less Than Equals	<code>{"key" : {\$lte:"value"}}</code>	<code>db.mycol.find({"likes":{\$lte:50}})</code>
Greater Than	<code>{"key" : {\$gt:"value"}}</code>	<code>db.mycol.find({"likes":{\$gt:50}})</code>
Greater Than Equals	<code>{"key" {\$gte:"value"}}</code>	<code>db.mycol.find({"likes":{\$gte:50}})</code>
Not Equals	<code>{"key":{\$ne: "value"}}</code>	<code>db.mycol.find({"likes":{\$ne:50}})</code>

Example1

Following example retrieves the document in a collection whose name is sarmista.

```
from pymongo import MongoClient

#Creating a pymongo client
client = MongoClient('localhost', 27017)

#Getting the database instance
db = client['sdsegf']

#Creating a collection
coll = db['example']
```

```

#Inserting document into a collection
data = [{"_id": "1001", "name": "Ram", "age": "26", "city": "Hyderabad"},
{"_id": "1002", "name": "Rahim", "age": "27", "city": "Bangalore"},
{"_id": "1003", "name": "Robert", "age": "28", "city": "Mumbai"},
{"_id": "1004", "name": "Romeo", "age": "25", "city": "Pune"},
{"_id": "1005", "name": "Sarmista", "age": "23", "city": "Delhi"},
{"_id": "1006", "name": "Rasajna", "age": "26", "city": "Chennai"}]

res = coll.insert_many(data)
print("Data inserted .....")

#Retrieving data
print("Documents in the collection: ")
for doc1 in coll.find({"name":"Sarmista"}):
    print(doc1)

```

Output

```

Data inserted .....
Documents in the collection:
{'_id': '1005', 'name': 'Sarmista', 'age': '23', 'city': 'Delhi'}

```

Example2

Following example retrieves the document in a collection whose age value is greater than 26.

```

from pymongo import MongoClient

#Creating a pymongo client
client = MongoClient('localhost', 27017)

#Getting the database instance
db = client['ghhj']

#Creating a collection
coll = db['example']

```

```
#Inserting document into a collection
data = [{"_id": "1001", "name": "Ram", "age": "26", "city": "Hyderabad"},
{"_id": "1002", "name": "Rahim", "age": "27", "city": "Bangalore"},
{"_id": "1003", "name": "Robert", "age": "28", "city": "Mumbai"},
{"_id": "1004", "name": "Romeo", "age": "25", "city": "Pune"},
{"_id": "1005", "name": "Sarmista", "age": "23", "city": "Delhi"},
{"_id": "1006", "name": "Rasajna", "age": "26", "city": "Chennai"}]

res = coll.insert_many(data)
print("Data inserted .....")

#Retrieving data
print("Documents in the collection: ")
for doc in coll.find({"age":{"$gt":"26"}}):
    print(doc)
```

Output

```
Data inserted .....
Documents in the collection:
{'_id': '1002', 'name': 'Rahim', 'age': '27', 'city': 'Bangalore'}
{'_id': '1003', 'name': 'Robert', 'age': '28', 'city': 'Mumbai'}
```

48. Python MongoDB — Sort

While retrieving the contents of a collection, you can sort and arrange them in ascending or descending orders using the **sort()** method.

To this method, you can pass the field(s) and the sorting order which is 1 or -1. Where, 1 is for ascending order and -1 is descending order.

Syntax

Following is the syntax of the `sort()` method.

```
>db.COLLECTION_NAME.find().sort({KEY:1})
```

Example

Assume we have created a collection and inserted 5 documents into it as shown below:

```
> use testDB
switched to db testDB
> db.createCollection("myColl")
{ "ok" : 1 }
> data = [
... {"_id": "1001", "name": "Ram", "age": "26", "city": "Hyderabad"},
... {"_id": "1002", "name": "Rahim", "age": 27, "city": "Bangalore"},
... {"_id": "1003", "name": "Robert", "age": 28, "city": "Mumbai"},
... {"_id": "1004", "name": "Romeo", "age": 25, "city": "Pune"},
... {"_id": "1005", "name": "Sarmista", "age": 23, "city": "Delhi"},
... {"_id": "1006", "name": "Rasajna", "age": 26, "city": "Chennai"}]

> db.sample.insert(data)
BulkWriteResult({
  "writeErrors" : [ ],
  "writeConcernErrors" : [ ],
  "nInserted" : 6,
  "nUpserted" : 0,
  "nMatched" : 0,
  "nModified" : 0,
  "nRemoved" : 0,
  "upserted" : [ ]
})
```

```
})
```

Following line retrieves all the documents of the collection which are sorted in ascending order based on age.

```
> db.sample.find().sort({age:1})
{ "_id" : "1005", "name" : "Sarmista", "age" : 23, "city" : "Delhi" }
{ "_id" : "1004", "name" : "Romeo", "age" : 25, "city" : "Pune" }
{ "_id" : "1006", "name" : "Rasajna", "age" : 26, "city" : "Chennai" }
{ "_id" : "1002", "name" : "Rahim", "age" : 27, "city" : "Bangalore" }
{ "_id" : "1003", "name" : "Robert", "age" : 28, "city" : "Mumbai" }
{ "_id" : "1001", "name" : "Ram", "age" : "26", "city" : "Hyderabad" }
```

Sorting the documents using python

To sort the results of a query in ascending or, descending order pymongo provides the **sort()** method. To this method, pass a number value representing the number of documents you need in the result.

By default, this method sorts the documents in ascending order based on the specified field. If you need to sort in descending order pass -1 along with the field name:

```
coll.find().sort("age",-1)
```

Example

Following example retrieves all the documents of a collection arranged according to the age values in ascending order:

```
from pymongo import MongoClient

#Creating a pymongo client
client = MongoClient('localhost', 27017)

#Getting the database instance
db = client['b_mydb']

#Creating a collection
coll = db['myColl']

#Inserting document into a collection
data = [{"_id": "1001", "name": "Ram", "age": "26", "city": "Hyderabad"},
{"_id": "1002", "name": "Rahim", "age": "27", "city": "Bangalore"},
```

```

{"_id": "1003", "name": "Robert", "age": "28", "city": "Mumbai"},
{"_id": "1004", "name": "Romeo", "age": 25, "city": "Pune"},
{"_id": "1005", "name": "Sarmista", "age": 23, "city": "Delhi"},
{"_id": "1006", "name": "Rasajna", "age": 26, "city": "Chennai"]}

res = coll.insert_many(data)
print("Data inserted .....")

#Retrieving first 3 documents using the find() and limit() methods
print("List of documents (sorted in ascending order based on age): ")
for doc1 in coll.find().sort("age"):
    print(doc1)

```

Output

```

Data inserted .....
List of documents (sorted in ascending order based on age):
{'_id': '1005', 'name': 'Sarmista', 'age': 23, 'city': 'Delhi'}
{'_id': '1004', 'name': 'Romeo', 'age': 25, 'city': 'Pune'}
{'_id': '1006', 'name': 'Rasajna', 'age': 26, 'city': 'Chennai'}
{'_id': '1001', 'name': 'Ram', 'age': '26', 'city': 'Hyderabad'}
{'_id': '1002', 'name': 'Rahim', 'age': '27', 'city': 'Bangalore'}
{'_id': '1003', 'name': 'Robert', 'age': '28', 'city': 'Mumbai'}

```

49. Python MongoDB — Delete Document

You can delete documents in a collection using the ***remove()*** method of MongoDB. This method accepts two optional parameters:

- deletion criteria specifying the condition to delete documents.
- just one, if you pass true or 1 as second parameter, then only one document will be deleted.

Syntax

Following is the syntax of the `remove()` method:

```
>db.COLLECTION_NAME.remove(DELETION_CRITTERIA)
```

Example

Assume we have created a collection and inserted 5 documents into it as shown below:

```
> use testDB
switched to db testDB
> db.createCollection("myColl")
{ "ok" : 1 }
> data = [
... {"_id": "1001", "name": "Ram", "age": "26", "city": "Hyderabad"},
... {"_id": "1002", "name": "Rahim", "age": 27, "city": "Bangalore"},
... {"_id": "1003", "name": "Robert", "age": 28, "city": "Mumbai"},
... {"_id": "1004", "name": "Romeo", "age": 25, "city": "Pune"},
... {"_id": "1005", "name": "Sarmista", "age": 23, "city": "Delhi"},
... {"_id": "1006", "name": "Rasajna", "age": 26, "city": "Chennai"}]
> db.sample.insert(data)
BulkWriteResult({
  "writeErrors" : [ ],
  "writeConcernErrors" : [ ],
  "nInserted" : 6,
  "nUpserted" : 0,
  "nMatched" : 0,
  "nModified" : 0,
  "nRemoved" : 0,
  "upserted" : [ ]
```

152

```
})
```

Following query deletes the document(s) of the collection which have name value as Sarmista.

```
> db.sample.remove({"name": "Sarmista"})
WriteResult({ "nRemoved" : 1 })

> db.sample.find()
{ "_id" : "1001", "name" : "Ram", "age" : "26", "city" : "Hyderabad" }
{ "_id" : "1002", "name" : "Rahim", "age" : 27, "city" : "Bangalore" }
{ "_id" : "1003", "name" : "Robert", "age" : 28, "city" : "Mumbai" }
{ "_id" : "1004", "name" : "Romeo", "age" : 25, "city" : "Pune" }
{ "_id" : "1006", "name" : "Rasajna", "age" : 26, "city" : "Chennai" }
```

If you invoke **remove()** method without passing deletion criteria, all the documents in the collection will be deleted.

```
> db.sample.remove({})
WriteResult({ "nRemoved" : 5 })
> db.sample.find()
```

Deleting documents using python

To delete documents from a collection of MongoDB, you can delete documents from a collections using the methods **delete_one()** and **delete_many()** methods.

These methods accept a query object specifying the condition for deleting documents.

The `delete_one()` method deletes a single document, in case of a match. If no query is specified this method deletes the first document in the collection.

Example

Following python example deletes the document in the collection which has id value as 1006.

```
from pymongo import MongoClient

#Creating a pymongo client
client = MongoClient('localhost', 27017)

#Getting the database instance
db = client['lpaksgf']

#Creating a collection
```



```

coll = db['example']

#Inserting document into a collection
data = [{"_id": "1001", "name": "Ram", "age": "26", "city": "Hyderabad"},
{"_id": "1002", "name": "Rahim", "age": "27", "city": "Bangalore"},
{"_id": "1003", "name": "Robert", "age": "28", "city": "Mumbai"},
{"_id": "1004", "name": "Romeo", "age": 25, "city": "Pune"},
{"_id": "1005", "name": "Sarmista", "age": 23, "city": "Delhi"},
{"_id": "1006", "name": "Rasajna", "age": 26, "city": "Chennai"}]

res = coll.insert_many(data)
print("Data inserted .....")
#Deleting one document
coll.delete_one({"_id" : "1006"})

#Retrieving all the records using the find() method
print("Documents in the collection after update operation: ")
for doc2 in coll.find():
    print(doc2)

```

Output

```

Data inserted .....
Documents in the collection after update operation:
{'_id': '1001', 'name': 'Ram', 'age': '26', 'city': 'Hyderabad'}
{'_id': '1002', 'name': 'Rahim', 'age': '27', 'city': 'Bangalore'}
{'_id': '1003', 'name': 'Robert', 'age': '28', 'city': 'Mumbai'}
{'_id': '1004', 'name': 'Romeo', 'age': 25, 'city': 'Pune'}
{'_id': '1005', 'name': 'Sarmista', 'age': 23, 'city': 'Delhi'}

```

Similarly, the ***delete_many()*** method of pymongo deletes all the documents that satisfies the specified condition.

Example

Following example deletes all the documents in the collection whose age value is greater than 26:

```

from pymongo import MongoClient

#Creating a pymongo client

```

```

client = MongoClient('localhost', 27017)

#Getting the database instance
db = client['sampleDB']

#Creating a collection
coll = db['example']

#Inserting document into a collection
data = [{"_id": "1001", "name": "Ram", "age": "26", "city": "Hyderabad"},
{"_id": "1002", "name": "Rahim", "age": "27", "city": "Bangalore"},
{"_id": "1003", "name": "Robert", "age": "28", "city": "Mumbai"},
{"_id": "1004", "name": "Romeo", "age": "25", "city": "Pune"},
{"_id": "1005", "name": "Sarmista", "age": "23", "city": "Delhi"},
{"_id": "1006", "name": "Rasajna", "age": "26", "city": "Chennai"}]

res = coll.insert_many(data)
print("Data inserted .....")

#Deleting multiple documents
coll.delete_many({"age":{"$gt":"26"}})

#Retrieving all the records using the find() method
print("Documents in the collection after update operation: ")
for doc2 in coll.find():
    print(doc2)

```

Output

```

Data inserted .....
Documents in the collection after update operation:
{'_id': '1001', 'name': 'Ram', 'age': '26', 'city': 'Hyderabad'}
{'_id': '1004', 'name': 'Romeo', 'age': '25', 'city': 'Pune'}
{'_id': '1005', 'name': 'Sarmista', 'age': '23', 'city': 'Delhi'}
{'_id': '1006', 'name': 'Rasajna', 'age': '26', 'city': 'Chennai'}

```

If you invoke the `delete_many()` method without passing any query, this method deletes all the documents in the collection.

```
coll.delete_many({})
```

50. Python MongoDB — Drop Collection

You can delete collections using **drop()** method of MongoDB.

Syntax

Following is the syntax of drop() method:

```
db.COLLECTION_NAME.drop()
```

Example

Following example drops collection with name sample:

```
> show collections
myColl
sample
> db.sample.drop()
true
> show collections
myColl
```

Dropping collection using python

You can drop/delete a collection from the current database by invoking drop() method.

Example

```
from pymongo import MongoClient

#Creating a pymongo client
client = MongoClient('localhost', 27017)

#Getting the database instance
db = client['example2']

#Creating a collection
coll = db['collection']
coll.insert_one({"name": "Ram", "age": "26", "city": "Hyderabad"})
coll2 = db['coll']
```

```

col2.insert_one({"name": "Rahim", "age": "27", "city": "Bangalore"})
col3 = db['myColl']
col3.insert_one({"name": "Robert", "age": "28", "city": "Mumbai"})
col4 = db['data']
col4.insert_one({"name": "Romeo", "age": "25", "city": "Pune"})

#List of collections
print("List of collections:")
collections = db.list_collection_names()
for coll in collections:
    print(coll)

#Dropping a collection
col1.drop()
col4.drop()

print("List of collections after dropping two of them: ")

#List of collections
collections = db.list_collection_names()
for coll in collections:
    print(coll)

```

Output

```

List of collections:
coll
data
collection
myColl
List of collections after dropping two of them:
coll
myColl

```

51. Python MongoDB — Update

You can update the contents of an existing documents using the **update()** method or **save()** method.

The update method modifies the existing document whereas the save method replaces the existing document with the new one.

Syntax

Following is the syntax of the update() and save() methods of MangoDB:

```
>db.COLLECTION_NAME.update(SELECTION_CRITERIA, UPDATED_DATA)
Or,
db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})
```

Example

Assume we have created a collection in a database and inserted 3 records in it as shown below:

```
> use testdatabase
switched to db testdatabase
> data = [
... {"_id": "1001", "name": "Ram", "age": "26", "city": "Hyderabad"},
... {"_id": "1002", "name" : "Rahim", "age" : 27, "city" : "Bangalore" },
... {"_id": "1003", "name" : "Robert", "age" : 28, "city" : "Mumbai" }]
[
  {
    "_id" : "1001",
    "name" : "Ram",
    "age" : "26",
    "city" : "Hyderabad"
  },
  {
    "_id" : "1002",
    "name" : "Rahim",
    "age" : 27,
    "city" : "Bangalore"
  },
]
```

```

    {
        "_id" : "1003",
        "name" : "Robert",
        "age" : 28,
        "city" : "Mumbai"
    }
]
> db.createCollection("sample")
{ "ok" : 1 }
> db.sample.insert(data)

```

Following method updates the city value of the document with id 1002.

```

> db.sample.update({"_id":"1002"},{"$set":{"city":"Visakhapatnam"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.sample.find()
{ "_id" : "1001", "name" : "Ram", "age" : "26", "city" : "Hyderabad" }
{ "_id" : "1002", "name" : "Rahim", "age" : 27, "city" : "Visakhapatnam" }
{ "_id" : "1003", "name" : "Robert", "age" : 28, "city" : "Mumbai" }

```

Similarly you can replace the document with new data by saving it with same id using the save() method.

```

> db.sample.save({ "_id" : "1001", "name" : "Ram", "age" : "26", "city" :
"Vijayawada" })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.sample.find()
{ "_id" : "1001", "name" : "Ram", "age" : "26", "city" : "Vijayawada" }
{ "_id" : "1002", "name" : "Rahim", "age" : 27, "city" : "Visakhapatnam" }
{ "_id" : "1003", "name" : "Robert", "age" : 28, "city" : "Mumbai" }

```

Updating documents using python

Similar to find_one() method which retrieves single document, the update_one() method of pymongo updates a single document.

This method accepts a query specifying which document to update and the update operation.

Example

Following python example updates the location value of a document in a collection.

```
from pymongo import MongoClient
```

```

#Creating a pymongo client
client = MongoClient('localhost', 27017)

#Getting the database instance
db = client['myDB']

#Creating a collection
coll = db['example']

#Inserting document into a collection
data = [{"_id": "101", "name": "Ram", "age": "26", "city": "Hyderabad"},
{"_id": "102", "name": "Rahim", "age": "27", "city": "Bangalore"},
{"_id": "103", "name": "Robert", "age": "28", "city": "Mumbai"}]

res = coll.insert_many(data)
print("Data inserted .....")

#Retrieving all the records using the find() method
print("Documents in the collection: ")
for doc1 in coll.find():
    print(doc1)

coll.update_one({"_id": "102"}, {"$set": {"city": "Visakhapatnam"}})

#Retrieving all the records using the find() method
print("Documents in the collection after update operation: ")
for doc2 in coll.find():
    print(doc2)

```

Output

```

Data inserted .....
Documents in the collection:
{'_id': '101', 'name': 'Ram', 'age': '26', 'city': 'Hyderabad'}
{'_id': '102', 'name': 'Rahim', 'age': '27', 'city': 'Bangalore'}
{'_id': '103', 'name': 'Robert', 'age': '28', 'city': 'Mumbai'}

```


Documents in the collection after update operation:

```
{'_id': '101', 'name': 'Ram', 'age': '26', 'city': 'Hyderabad'}
{'_id': '102', 'name': 'Rahim', 'age': '27', 'city': 'Visakhapatnam'}
{'_id': '103', 'name': 'Robert', 'age': '28', 'city': 'Mumbai'}
```

Similarly, the ***update_many()*** method of pymongo updates all the documents that satisfies the specified condition.

Example

Following example updates the location value in all the documents in a collection (empty condition):

```
from pymongo import MongoClient

#Creating a pymongo client
client = MongoClient('localhost', 27017)

#Getting the database instance
db = client['myDB']

#Creating a collection
coll = db['example']

#Inserting document into a collection
data = [{"_id": "101", "name": "Ram", "age": "26", "city": "Hyderabad"},
        {"_id": "102", "name": "Rahim", "age": "27", "city": "Bangalore"},
        {"_id": "103", "name": "Robert", "age": "28", "city": "Mumbai"}]

res = coll.insert_many(data)
print("Data inserted .....")

#Retrieving all the records using the find() method
print("Documents in the collection: ")
for doc1 in coll.find():
    print(doc1)

coll.update_many({}, {"$set":{"city":"Visakhapatnam"}})

#Retrieving all the records using the find() method
```

```
print("Documents in the collection after update operation: ")
for doc2 in coll.find():
    print(doc2)
```

Output

```
Data inserted .....
Documents in the collection:
{'_id': '101', 'name': 'Ram', 'age': '26', 'city': 'Hyderabad'}
{'_id': '102', 'name': 'Rahim', 'age': '27', 'city': 'Bangalore'}
{'_id': '103', 'name': 'Robert', 'age': '28', 'city': 'Mumbai'}
Documents in the collection after update operation:
{'_id': '101', 'name': 'Ram', 'age': '26', 'city': 'Visakhapatnam'}
{'_id': '102', 'name': 'Rahim', 'age': '27', 'city': 'Visakhapatnam'}
{'_id': '103', 'name': 'Robert', 'age': '28', 'city': 'Visakhapatnam'}
```

52. Python MongoDB — Limit

While retrieving the contents of a collection you can limit the number of documents in the result using the `limit()` method. This method accepts a number value representing the number of documents you want in the result.

Syntax

Following is the syntax of the `limit()` method:

```
>db.COLLECTION_NAME.find().limit(NUMBER)
```

Example

Assume we have created a collection and inserted 5 documents into it as shown below:

```
> use testDB
switched to db testDB
> db.createCollection("sample")
{ "ok" : 1 }
> data = [
... {"_id": "1001", "name": "Ram", "age": "26", "city": "Hyderabad"},
... {"_id": "1002", "name": "Rahim", "age": 27, "city": "Bangalore"},
... {"_id": "1003", "name": "Robert", "age": 28, "city": "Mumbai"},
... {"_id": "1004", "name": "Romeo", "age": 25, "city": "Pune"},
... {"_id": "1005", "name": "Sarmista", "age": 23, "city": "Delhi"},
... {"_id": "1006", "name": "Rasajna", "age": 26, "city": "Chennai"}]

> db.sample.insert(data)
BulkWriteResult({
  "writeErrors" : [ ],
  "writeConcernErrors" : [ ],
  "nInserted" : 6,
  "nUpserted" : 0,
  "nMatched" : 0,
  "nModified" : 0,
  "nRemoved" : 0,
  "upserted" : [ ]
})
```

Following line retrieves the first 3 documents of the collection

```
> db.sample.find().limit(3)
{ "_id" : "1001", "name" : "Ram", "age" : "26", "city" : "Hyderabad" }
{ "_id" : "1002", "name" : "Rahim", "age" : 27, "city" : "Bangalore" }
{ "_id" : "1003", "name" : "Robert", "age" : 28, "city" : "Mumbai" }
```

Limiting the documents using python

To restrict the results of a query to a particular number of documents pymongo provides the **limit()** method. To this method pass a number value representing the number of documents you need in the result.

Example

Following example retrieves first three documents in a collection.

```
from pymongo import MongoClient

#Creating a pymongo client
client = MongoClient('localhost', 27017)

#Getting the database instance
db = client['l']

#Creating a collection
coll = db['myColl']

#Inserting document into a collection
data = [{"_id": "1001", "name": "Ram", "age": "26", "city": "Hyderabad"},
{"_id": "1002", "name": "Rahim", "age": "27", "city": "Bangalore"},
{"_id": "1003", "name": "Robert", "age": "28", "city": "Mumbai"},
{"_id": "1004", "name": "Romeo", "age": 25, "city": "Pune"},
{"_id": "1005", "name": "Sarmista", "age": 23, "city": "Delhi"},
{"_id": "1006", "name": "Rasajna", "age": 26, "city": "Chennai"}]
res = coll.insert_many(data)
print("Data inserted .....")

#Retrieving first 3 documents using the find() and limit() methods
print("First 3 documents in the collection: ")
for doc1 in coll.find().limit(3):
```

```
print(doc1)
```

Output

```
Data inserted .....
```

```
First 3 documents in the collection:
```

```
{'_id': '1001', 'name': 'Ram', 'age': '26', 'city': 'Hyderabad'}
```

```
{'_id': '1002', 'name': 'Rahim', 'age': '27', 'city': 'Bangalore'}
```

```
{'_id': '1003', 'name': 'Robert', 'age': '28', 'city': 'Mumbai'}
```