

**4****Interaction with Database****TOPICS COVERED :**

1. Configuring database
2. Defining model
3. Basic data access
4. Inserting and updating data
5. Selecting objects
6. Deleting objects

**Python MYSQL Connectivity****Configuring database:**

- To build the real world applications, connecting with the databases is the necessity for the programming languages.
- However, python allows us to connect our application to the databases like MySQL, SQLite, MongoDB, and many others.
- MySQL connectivity, and we will perform the database operations in python.
- We will also cover the Python connectivity with the databases like MongoDB and SQLite.

**Install mysql.connector:**

- To connect the python application with the MySQL database, we must import the mysql.connector module in the program.
- The mysql.connector is not a built-in module that comes with the python installation. We need to install it to get it working.
- Execute the following command to install it using pip installer.  
> **python -m pip install mysql-connector**

**Or follow the following steps:**

1. Click the link:  
<https://files.pythonhosted.org/packages/8f/6d/fb8ebcbbbaee68b172ce3dfd08c7b8660d09f91d8d5411298bcacbd309f96/mysql-connector-python-8.0.13.tar.gz> to download the source code.
2. Extract the archived file.
3. Open the terminal (CMD for windows) and change the present working directory to the source code directory.  
\$ **cd mysql-connector-python-8.0.13/**
4. Run the file named setup.py with python (python3 in case you have also installed python 2) with the parameter build.

**\$ python setup.py build**

5. Run the following command to install the mysql-connector.

**\$ python setup.py install**

- This will take a bit of time to install mysql-connector for python. We can verify the installation once the process gets over by importing mysql-connector on the python shell.

```
Python 3.4.9 (default, Aug 14 2018, 21:28:57)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import mysql.connector
>>>
```

- Hence, we have successfully installed mysql-connector for python on our system.

### **Database Connection:**

- We will discuss the steps to connect the python application to the database.
- There are the following steps to connect a python application to our database.
  1. Import **mysql.connector** module
  2. Create the connection object.
  3. Create the cursor object
  4. Execute the query

#### **➤ Creating the connection:**

- To create a connection between the MySQL database and the python application, the `connect()` method of `mysql.connector` module is used.
- Pass the database details like `HostName`, `username`, and the database password in the method call.
- The method returns the connection object.

#### **Syntax:**

```
ConnectionObject= mysql.connector.connect(host = <hostname> , user = <username> ,
passwd =<password> )
```

#### **Example:**

```
import mysql.connector
#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "")

#printing the connection object
print(myconn)
```

#### **Output:**

<mysql.connector.connection.MySQLConnection object at 0x7fb142edd780>

- Here, we must notice that we can specify the database name in the connect() method if we want to connect to a specific database.

**Example:**

```
import mysql.connector
```

```
#Create the connection object
```

```
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "", database = "mydb")
```

```
#printing the connection object
```

```
print(myconn)
```

**Output:**

<mysql.connector.connection.MySQLConnection object at 0x7ff64aa3d7b8>

**Basic data access:**

**1. Creating a cursor object:**

- The cursor object can be defined as an abstraction specified in the Python DB-API 2.0.
- It facilitates us to have multiple separate working environments through the same connection to the database.
- We can create the cursor object by calling the 'cursor' function of the connection object.
- The cursor object is an **important aspect of executing queries to the databases.**
- The syntax to create the cursor object is given below.

```
<my_cur> = conn.cursor()
```

**Example:**

```
import mysql.connector
```

```
#Create the connection object
```

```
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = " ", database = "mydb")
```

```
#printing the connection object
```

```
print(myconn)
```

```
#creating the cursor object
```

```
cur = myconn.cursor()
```

```
print(cur)
```

**Output:**

```
<mysql.connector.connection.MySQLConnection object at 0x7faa17a15748>  
MySQLCursor: (Nothing executed yet)
```

**2. Creating new databases:**

- We will create the new database PythonDB.
- **Getting the list of existing databases**
- We can get the list of all the databases by using the following MySQL query.  
    **> show databases;**

**Example:**

```
import mysql.connector
```

```
#Create the connection object
```

```
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "")
```

```
#creating the cursor object
```

```
cur = myconn.cursor()
```

```
try:
```

```
    dbs = cur.execute("show databases")
```

```
except:
```

```
    myconn.rollback()
```

```
for x in cur:
```

```
    print(x)
```

```
myconn.close()
```

**Output:**

```
('EmployeeDB',)
```

```
('Test',)
```

```
('TestDB',)
```

```
('information_schema',)
```

```
('jvatpoint',)
```

```
('jvatpoint1',)
```

```
('mydb',)
```

```
('mysql',)
```

```
('performance_schema',)
```

```
('testDB',)
```

➤ **Creating the new database:**

- The new database can be created by using the following SQL query.  
    > **create database <database-name>**

**Example:**

```
import mysql.connector
```

```
#Create the connection object
```

```
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "")
```

```
#creating the cursor object
```

```
cur = myconn.cursor()
```

```
try:
```

```
    #creating a new database
```

```
    cur.execute("create database PythonDB2")
```

```
    #getting the list of all the databases which will now include the new database PythonDB
```

```
    dbs = cur.execute("show databases")
```

```
except:
```

```
    myconn.rollback()
```

```
for x in cur:
```

```
    print(x)
```

```
myconn.close()
```

**Output:**

```
('EmployeeDB',)
```

```
('PythonDB',)
```

```
('Test',)
```

```
('TestDB',)
```

```
('anshika',)
```

```
('information_schema',)
```

```
('jvatpoint',)
```

```
('jvatpoint1',)
```

```
('mydb',)
```

```
('mydb1',)
```

```
('mysql',)
```

```
('performance_schema',)  
( 'testDB',)
```

### 3. Creating the table:

- We will create the new table Employee.
  - We have to mention the database name while establishing the connection object.
  - We can create the new table by using the CREATE TABLE statement of SQL.
  - In our database PythonDB, the table Employee will have the four columns, i.e., name, id, salary, and dept\_id initially.
  - The following query is used to create the new table Employee.
- ```
> create table Employee (name varchar(20) not null, id int primary key, salary float not null,  
Dept_Id int not null)
```

#### Example:

```
import mysql.connector
```

```
#Create the connection object
```

```
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "",  
database = "PythonDB")
```

```
#creating the cursor object
```

```
cur = myconn.cursor()
```

```
try:
```

```
    #Creating a table with name Employee having four columns i.e., name, id, salary, and de  
partment id
```

```
    dbs = cur.execute("create table Employee(name varchar(20) not null, id int(20) not null  
primary key, salary float not null, Dept_id int not null)")
```

```
except:
```

```
    myconn.rollback()
```

```
myconn.close()
```

```
javatpoint@localhost:~  
File Edit View Search Terminal Help  
Reading table information for completion of table and column names  
You can turn off this feature to get a quicker startup with -A  
  
Database changed  
MariaDB [PythonDB]> show tables;  
+-----+  
| Tables_in_PythonDB |  
+-----+  
| Employee            |  
+-----+  
1 row in set (0.00 sec)  
  
MariaDB [PythonDB]> desc Employee;  
+-----+-----+-----+-----+-----+-----+  
| Field | Type          | Null | Key | Default | Extra |  
+-----+-----+-----+-----+-----+-----+  
name	varchar(20)	NO		NULL	
id	int(20)	NO	PRI	NULL	
salary	float	NO		NULL	
Dept_id	int(11)	NO		NULL	
+-----+-----+-----+-----+-----+-----+  
4 rows in set (0.01 sec)  
  
MariaDB [PythonDB]> 
```

- Now, we may check that the table Employee is present in the database.

#### 4. Alter Table:

- Sometimes, we may forget to create some columns, or we may need to update the table schema.
- The alter statement used to alter the table schema if required.
- Here, we will add the column branch\_name to the table Employee.
- The following SQL query is used for this purpose.  
>alter table Employee add branch\_name varchar(20) **not** null

#### Example:

```
import mysql.connector
```

```
#Create the connection object
```

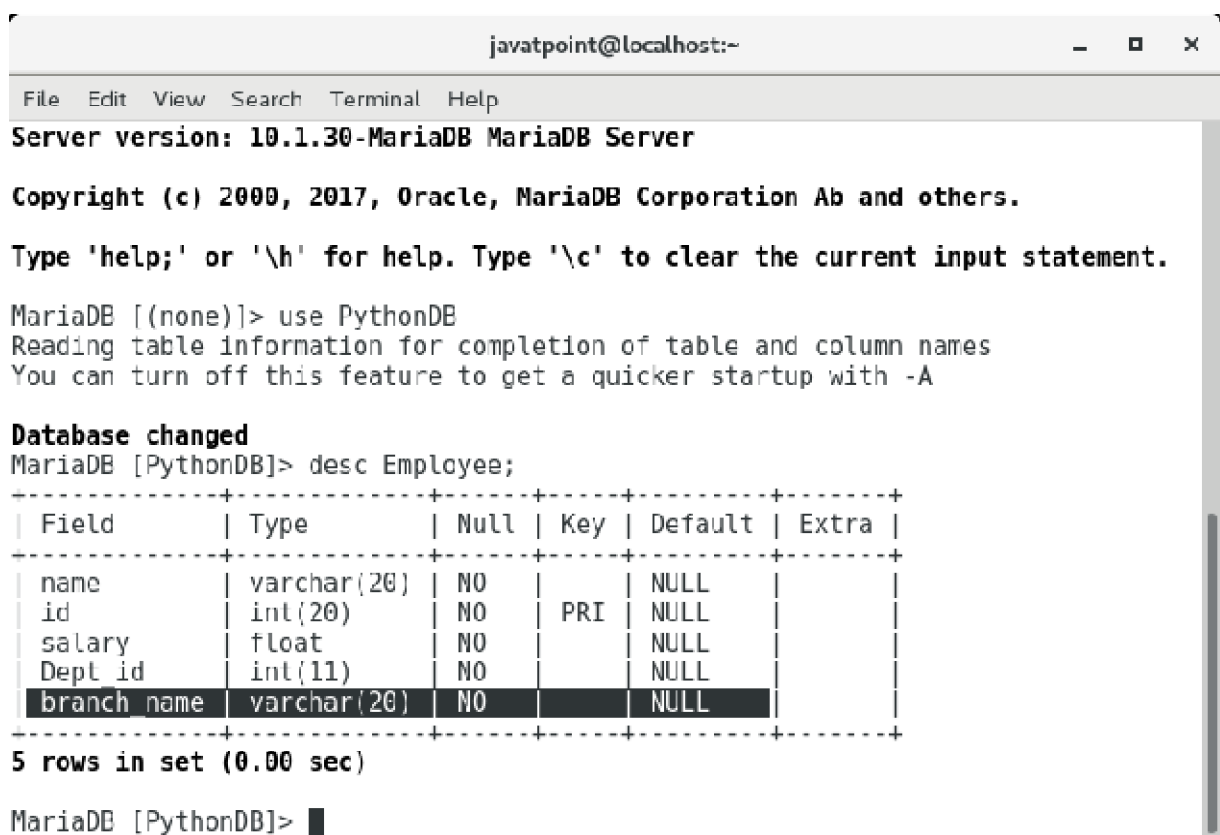
```
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "",  
database = "PythonDB")
```

```
#creating the cursor object
```

```
cur = myconn.cursor()

try:
    #adding a column branch name to the table Employee
    cur.execute("alter table Employee add branch_name varchar(20) not null")
except:
    myconn.rollback()

myconn.close()
```



The screenshot shows a terminal window titled 'jvatpoint@localhost:~'. The terminal output displays the MariaDB server version (10.1.30), copyright information, and a prompt to use PythonDB. The user enters 'desc Employee;', and the terminal shows the table structure for the 'Employee' table. The table has 6 columns: name, id, salary, Dept\_id, branch\_name, and an extra column. The 'branch\_name' column is highlighted in the output.

```
Server version: 10.1.30-MariaDB MariaDB Server

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

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

MariaDB [(none)]> use PythonDB
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MariaDB [PythonDB]> desc Employee;
+-----+-----+-----+-----+-----+-----+
| Field      | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
name	varchar(20)	NO		NULL	
id	int(20)	NO	PRI	NULL	
salary	float	NO		NULL	
Dept_id	int(11)	NO		NULL	
branch_name	varchar(20)	NO		NULL	
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

MariaDB [PythonDB]> █
```

### Inserting and updating data:

#### 1. Insert Operation:

##### ➤ Adding a record to the table:

- The **INSERT INTO** statement is used to add a record to the table.
- In python, we can mention the format specifier (%s) in place of values.
- We provide the actual values in the form of tuple in the execute() method of the cursor.



**Example:**

```
import mysql.connector
#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "",
database = "PythonDB")
#creating the cursor object
cur = myconn.cursor()
sql = "insert into Employee(name, id, salary, dept_id, branch_name) values (%s, %s, %s, %s, %s)"

#The row values are provided in the form of tuple
val = ("John", 110, 25000.00, 201, "Newyork")

try:
    #inserting the values into the table
    cur.execute(sql,val)

    #commit the transaction
    myconn.commit()

except:
    myconn.rollback()

print(cur.rowcount,"record inserted!")
myconn.close()
```

**Output:**

1 record inserted!

```
javatpoint@localhost:~  
File Edit View Search Terminal Help  
[javatpoint@localhost ~]$ mysql -u root -p  
Enter password:  
Welcome to the MariaDB monitor.  Commands end with ; or \g.  
Your MariaDB connection id is 56  
Server version: 10.1.30-MariaDB MariaDB Server  
  
Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.  
  
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
  
MariaDB [(none)]> use PythonDB;  
Reading table information for completion of table and column names  
You can turn off this feature to get a quicker startup with -A  
  
Database changed  
MariaDB [PythonDB]> select * from Employee;  
+-----+-----+-----+-----+-----+  
| name | id  | salary | Dept_id | branch_name |  
+-----+-----+-----+-----+-----+  
| John | 101 | 25000  | 201     | Newyork      |  
+-----+-----+-----+-----+-----+  
1 row in set (0.00 sec)  
  
MariaDB [PythonDB]> 
```

### ➤ Insert multiple rows:

- We can also insert multiple rows at once using the python script.
- The multiple rows are mentioned as the list of various tuples.
- Each element of the list is treated as one particular row, whereas each element of the tuple is treated as one particular column value (attribute).

#### Example:

```
import mysql.connector
```

```
#Create the connection object
```

```
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "",database  
= "PythonDB")
```

```
#creating the cursor object
```

```
cur = myconn.cursor()
```

```
sql = "insert into Employee(name, id, salary, dept_id, branch_name) values (%s, %s, %s,  
%s, %s)"
```

```
val = [("John", 102, 25000.00, 201, "Newyork"),("David",103,25000.00,202,"Port of spain")  
,"(Nick",104,90000.00,201,"Newyork")]
```

```
try:
    #inserting the values into the table
    cur.executemany(sql,val)

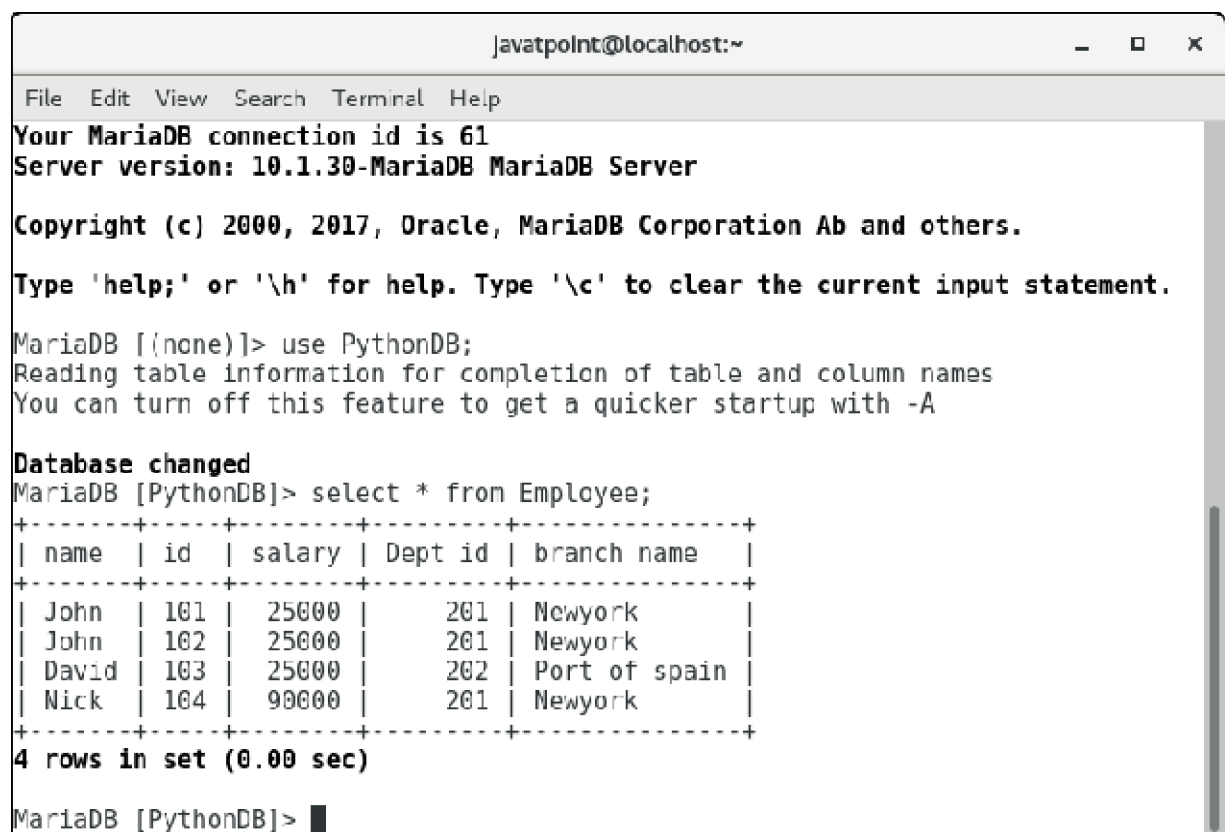
    #commit the transaction
    myconn.commit()
    print(cur.rowcount,"records inserted!")

except:
    myconn.rollback()

myconn.close()
```

**Output:**

3 records inserted!



The screenshot shows a terminal window titled 'javatpoint@localhost:~'. The terminal displays the output of a MariaDB connection and a query. The connection is successful, showing the server version as 10.1.30-MariaDB. The user then runs a query to select all records from the 'Employee' table. The results are displayed in a table format with 4 rows and 5 columns: name, id, salary, Dept id, and branch name.

```
javatpoint@localhost:~
File Edit View Search Terminal Help
Your MariaDB connection id is 61
Server version: 10.1.30-MariaDB MariaDB Server

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

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

MariaDB [(none)]> use PythonDB;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MariaDB [PythonDB]> select * from Employee;
+-----+-----+-----+-----+-----+
| name  | id   | salary | Dept id | branch name |
+-----+-----+-----+-----+-----+
John	101	25000	201	Newyork
John	102	25000	201	Newyork
David	103	25000	202	Port of spain
Nick	104	90000	201	Newyork
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

MariaDB [PythonDB]> 
```

**➤ Row ID:**

- In SQL, a particular row is represented by an insertion id which is known as **row id**.
- We can get the last inserted row id by using the attribute lastrowid of the cursor object.

**Example:**

```
import mysql.connector
#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
#creating the cursor object
cur = myconn.cursor()

sql = "insert into Employee(name, id, salary, dept_id, branch_name) values (%s, %s, %s, %s, %s)"

val = ("Mike",105,28000,202,"Guyana")

try:
    #inserting the values into the table
    cur.execute(sql,val)

    #commit the transaction
    myconn.commit()

    #getting rowid
    print(cur.rowcount,"record inserted! id:",cur.lastrowid)

except:
    myconn.rollback()

myconn.close()
```

**Output:**

1 record inserted! Id: 0

**2. Update Operation:**

- The UPDATE-SET statement is used to update any column inside the table.
- The following SQL query is used to update a column.  
> update Employee set name = 'alex' where id = 110

**Example:**

```
import mysql.connector
```

```
#Create the connection object
```

```
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "",  
database = "PythonDB")
```

```
#creating the cursor object
```

```
cur = myconn.cursor()
```

```
try:
```

```
    #updating the name of the employee whose id is 110
```

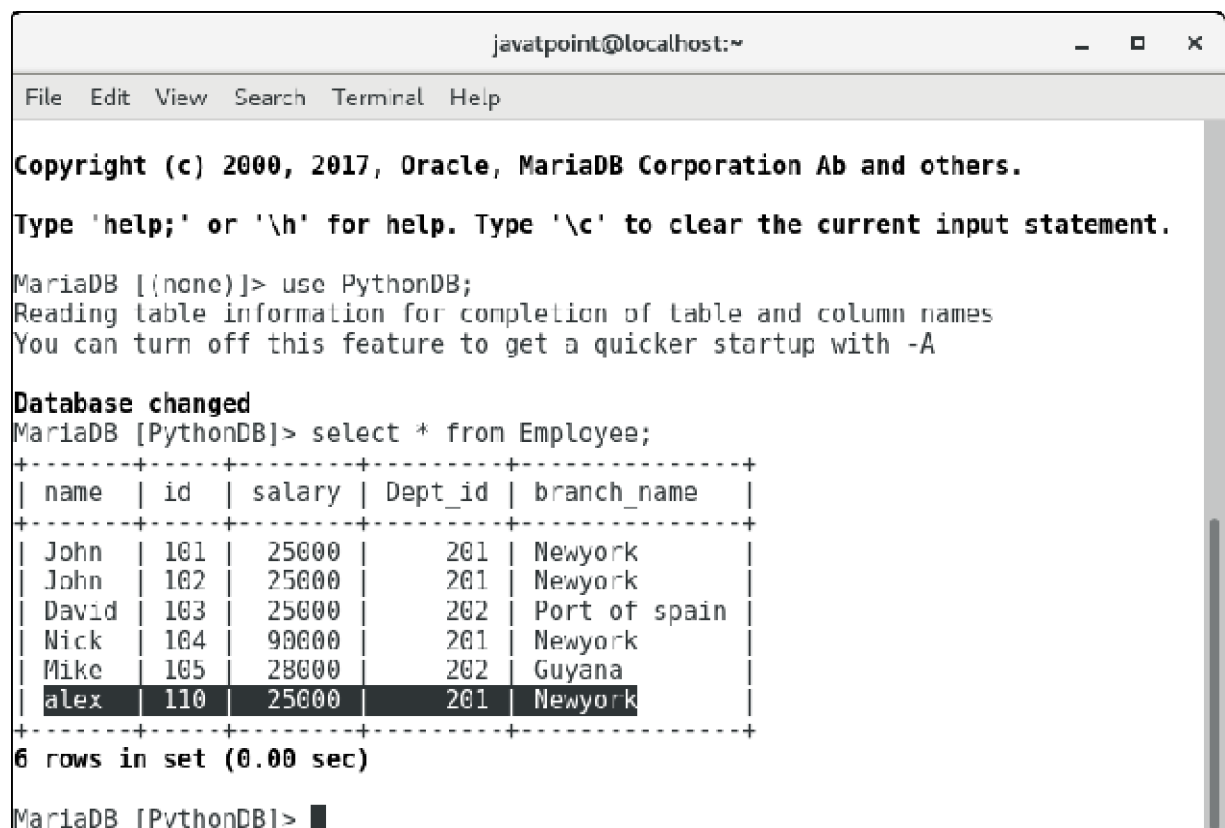
```
    cur.execute("update Employee set name = 'alex' where id = 110")
```

```
    myconn.commit()
```

```
except:
```

```
    myconn.rollback()
```

```
myconn.close()
```



```
javatpoint@localhost:~  
File Edit View Search Terminal Help  
Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.  
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
MariaDB [(none)]> use PythonDB;  
Reading table information for completion of table and column names  
You can turn off this feature to get a quicker startup with -A  
  
Database changed  
MariaDB [PythonDB]> select * from Employee;  
+-----+-----+-----+-----+-----+  
| name | id | salary | Dept_id | branch_name |  
+-----+-----+-----+-----+-----+  
John	101	25000	201	Newyork
John	102	25000	201	Newyork
David	103	25000	202	Port of spain
Nick	104	90000	201	Newyork
Mike	105	28000	202	Guyana
alex	110	25000	201	Newyork
+-----+-----+-----+-----+-----+  
6 rows in set (0.00 sec)  
  
MariaDB [PythonDB]> 
```

**Selecting objects:**

- The SELECT statement is used to read the values from the databases.
- We can restrict the output of a select query by using various clause in SQL like where, limit, etc.
- Python provides the fetchall() method returns the data stored inside the table in the form of rows.
- We can iterate the result to get the individual rows.
- We will extract the data from the database by using the python script.
- We will also format the output to print it on the console.

**Example:**

```
import mysql.connector
```

```
#Create the connection object
```

```
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "",  
database = "PythonDB")
```

```
#creating the cursor object
```

```
cur = myconn.cursor()
```

```
try:
```

```
    #Reading the Employee data
```

```
    cur.execute("select * from Employee")
```

```
    #fetching the rows from the cursor object
```

```
    result = cur.fetchall()
```

```
    #printing the result
```

```
    for x in result:
```

```
        print(x);
```

```
except:
```

```
    myconn.rollback()
```

```
myconn.close()
```

**Output:**

```
('John', 101, 25000.0, 201, 'Newyork')
```

```
('John', 102, 25000.0, 201, 'Newyork')
```

```
('David', 103, 25000.0, 202, 'Port of spain')
```

```
('Nick', 104, 90000.0, 201, 'Newyork')
```

```
('Mike', 105, 28000.0, 202, 'Guyana')
```

➤ **Reading specific columns:**

- We can read the specific columns by mentioning their names instead of using star (\*).
- In the following example, we will read the name, id, and salary from the Employee table and print it on the console.

**Example:**

```
import mysql.connector
#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "",
database = "PythonDB")
#creating the cursor object
cur = myconn.cursor()
try:
    #Reading the Employee data
    cur.execute("select name, id, salary from Employee")

    #fetching the rows from the cursor object
    result = cur.fetchall()
    #printing the result
    for x in result:
        print(x);
except:
    myconn.rollback()
myconn.close()
```

**Output:**

```
('John', 101, 25000.0)
('John', 102, 25000.0)
('David', 103, 25000.0)
('Nick', 104, 90000.0)
('Mike', 105, 28000.0)
```

➤ **The fetchone() method:**

- The fetchone() method is used to fetch only one row from the table.
- The fetchone() method returns the next row of the result-set.

**Example:**

```
import mysql.connector

#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")

#creating the cursor object
cur = myconn.cursor()

try:
    #Reading the Employee data
    cur.execute("select name, id, salary from Employee")

    #fetching the first row from the cursor object
    result = cur.fetchone()

    #printing the result
    print(result)

except:
    myconn.rollback()

myconn.close()
```

**Output:**

('John', 101, 25000.0)

**➤ Formatting the result:**

- We can format the result by iterating over the result produced by the fetchall() or fetchone() method of cursor object since the result exists as the tuple object which is not readable.

**Example:**

```
import mysql.connector

#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "",
database = "PythonDB")

#creating the cursor object
```



```
cur = myconn.cursor()

try:

    #Reading the Employee data
    cur.execute("select name, id, salary from Employee")

    #fetching the rows from the cursor object
    result = cur.fetchall()

    print("Name  id  Salary");
    for row in result:
        print("%s  %d  %d"%(row[0],row[1],row[2]))
except:
    myconn.rollback()

myconn.close()
```

Output:

```
Name  id  Salary
John  101  25000
John  102  25000
David 103  25000
Nick  104  90000
Mike  105  28000
```

### ➤ Using where clause:

- We can restrict the result produced by the select statement by using the where clause.
- This will extract only those columns which satisfy the where condition.

### Example: printing the names that start with j

```
import mysql.connector
```

```
#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "",
database = "PythonDB")

#creating the cursor object
cur = myconn.cursor()

try:
    #Reading the Employee data
```

```
cur.execute("select name, id, salary from Employee where name like 'J%")

#fetching the rows from the cursor object
result = cur.fetchall()

print("Name   id   Salary");

for row in result:
    print("%s   %d   %d"%(row[0],row[1],row[2]))
except:
    myconn.rollback()

myconn.close()
```

**Output:**

```
Name   id   Salary
John   101   25000
John   102   25000
```

**Example: printing the names with id = 101, 102, and 103**

```
import mysql.connector
```

```
#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "",
database = "PythonDB")

#creating the cursor object
cur = myconn.cursor()

try:
    #Reading the Employee data
    cur.execute("select name, id, salary from Employee where id in (101,102,103)")

    #fetching the rows from the cursor object
    result = cur.fetchall()

    print("Name   id   Salary");

    for row in result:
        print("%s   %d   %d"%(row[0],row[1],row[2]))
except:
```

```
myconn.rollback()
```

```
myconn.close()
```

**Output:**

```
Name id Salary
John 101 25000
John 102 25000
David 103 2500
```

**➤ Ordering the result:**

- The ORDER BY clause is used to order the result.

**Example:**

```
import mysql.connector
```

```
#Create the connection object
```

```
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "",
database = "PythonDB")
```

```
#creating the cursor object
```

```
cur = myconn.cursor()
```

```
try:
```

```
    #Reading the Employee data
```

```
    cur.execute("select name, id, salary from Employee order by name")
```

```
    #fetching the rows from the cursor object
```

```
    result = cur.fetchall()
```

```
    print("Name id Salary");
```

```
    for row in result:
```

```
        print("%s %d %d"%(row[0],row[1],row[2]))
```

```
except:
```

```
    myconn.rollback()
```

```
myconn.close()
```

**Output:**

```
Name id Salary
```

```
David 103 25000
John 101 25000
John 102 25000
Mike 105 28000
Nick 104 90000
```

➤ **Order by DESC:**

- This orders the result in the decreasing order of a particular column.

**Example:**

```
import mysql.connector
```

```
#Create the connection object
```

```
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "",
database = "PythonDB")
```

```
#creating the cursor object
```

```
cur = myconn.cursor()
```

```
try:
```

```
    #Reading the Employee data
```

```
    cur.execute("select name, id, salary from Employee order by name desc")
```

```
    #fetching the rows from the cursor object
```

```
    result = cur.fetchall()
```

```
    #printing the result
```

```
    print("Name  id  Salary");
```

```
    for row in result:
```

```
        print("%s  %d  %d"%(row[0],row[1],row[2]))
```

```
except:
```

```
    myconn.rollback()
```

```
myconn.close()
```

**Output:**

| Name  | id  | Salary |
|-------|-----|--------|
| Nick  | 104 | 90000  |
| Mike  | 105 | 28000  |
| John  | 101 | 25000  |
| John  | 102 | 25000  |
| David | 103 | 25000  |

**Deleting objects:**

- The DELETE FROM statement is used to delete a specific record from the table.
- Here, we must impose a condition using WHERE clause otherwise all the records from the table will be removed.
- The following SQL query is used to delete the employee detail whose id is 110 from the table.  
> delete from Employee where id = 110

**Example**

```
import mysql.connector
```

```
#Create the connection object
```

```
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "",  
database = "PythonDB")
```

```
#creating the cursor object
```

```
cur = myconn.cursor()
```

```
try:
```

```
    #Deleting the employee details whose id is 110
```

```
    cur.execute("delete from Employee where id = 110")
```

```
    myconn.commit()
```

```
except:
```

```
    myconn.rollback()
```

```
myconn.close()
```

**Performing Transactions:**

- Transactions ensure the data consistency of the database.
- We have to make sure that more than one applications must not modify the records while performing the database operations.
- The transactions have the following properties.

**1. Atomicity:**

- Either the transaction completes, or nothing happens. If a transaction contains 4 queries then all these queries must be executed, or none of them must be executed.

**2. Consistency:**

- The database must be consistent before the transaction starts and the database must also be consistent after the transaction is completed.

**3. Isolation:**

- Intermediate results of a transaction are not visible outside the current transaction.

**4. Durability:**

- Once a transaction was committed, the effects are persistent, even after a system failure.

**Python commit() method:**

- Python provides the commit() method which ensures the changes made to the database consistently take place.
- The syntax to use the commit() method is given below.  
**conn.commit() #conn is the connection object**
- All the operations that modify the records of the database do not take place until the commit() is called.

**Python rollback() method:**

- The rollback() method is used to revert the changes that are done to the database.
- This method is useful in the sense that, if some error occurs during the database operations, we can rollback that transaction to maintain the database consistency.
- The syntax to use the rollback() is given below.  
**Conn.rollback()**

**Closing the connection:**

- We need to close the database connection once we have done all the operations regarding the database.
- Python provides the close() method.
- The syntax to use the close() method is given below.  
**conn.close()**
- In the following example, we are deleting all the employees who are working for the CS department.

**Example:**

```
import mysql.connector
```

```
#Create the connection object
```

```
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "",  
database = "PythonDB")
```

```
#creating the cursor object
```

```
cur = myconn.cursor()
```

```
try:
```

```
    cur.execute("delete from Employee where Dept_id = 201")
```

```
    myconn.commit()
```

```
    print("Deleted !")
```

```
except:
```

```
    print("Can't delete !")
```

```
    myconn.rollback()
```

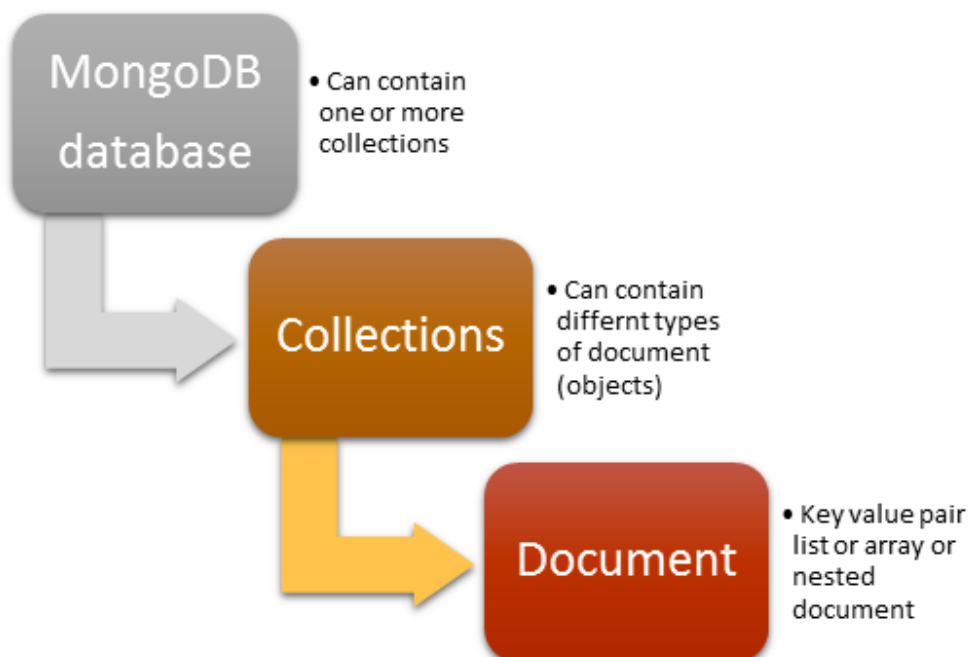
```
myconn.close()
```

**Output:**

Deleted !

### Python MongoDB Connectivity

- Data in MongoDB is made up of three types of components: **databases**, **collections**, and **documents**.
- The database sits at the top of the hierarchy, collections at the next level, and documents at the bottom.



- A **database** provides a container for storing and organizing data.
- Each database contains one or more collections, and each **collection** contains zero or more documents.

- A database can contain multiple collections, but a collection cannot span multiple databases.
- Likewise, a collection can contain multiple documents, but a document cannot span multiple collections.
- MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability.
- MongoDB works on concept of collection and document.

### 1. Database:

- Database is a physical container for collections.
- Each database gets its own set of files on the file system.
- A single MongoDB server typically has multiple databases.

### 2. Collection:

- Collection is a group of MongoDB documents.
- It is the equivalent of an RDBMS table.
- A collection exists within a single database.
- Collections do not enforce a schema.
- Documents within a collection can have different fields.
- Typically, all documents in a collection are of similar or related purpose.

### 3. Document:

- A document is a set of key-value pairs.
- Documents have dynamic schema.
- Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.
- The following table shows the relationship of RDBMS terminology with MongoDB.

| RDBMS                             | MongoDB                                                  |
|-----------------------------------|----------------------------------------------------------|
| Database                          | Database                                                 |
| Table                             | Collection                                               |
| Tuple/Row                         | Document                                                 |
| column                            | Field                                                    |
| Table Join                        | Embedded Documents                                       |
| Primary Key                       | Primary Key (Default key _id provided by MongoDB itself) |
| <b>Database Server and Client</b> |                                                          |
| mysql/Oracle                      | mongod                                                   |
| mysql/sqlplus                     | mongo                                                    |



- Any relational database has a typical schema design that shows number of tables and the relationship between these tables.
- While in MongoDB, there is no concept of relationship.

### **Advantages of MongoDB over RDBMS:**

- Schema less – MongoDB is a document database in which one collection holds different documents. Number of fields, content and size of the document can differ from one document to another.
- Structure of a single object is clear.
- No complex joins.
- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- Tuning.
- Ease of scale-out – MongoDB is easy to scale.
- Conversion/mapping of application objects to database objects not needed.
- Uses internal memory for storing the (windowed) working set, enabling faster access of data.

### **Why Use MongoDB?**

- Document Oriented Storage – Data is stored in the form of JSON style documents.
- Index on any attribute
- Replication and high availability
- Auto-Sharding
- Rich queries
- Fast in-place updates
- Professional support by MongoDB

### **Where to Use MongoDB?**

- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub

### **PyMongo Configuring database:**

- Python needs a MongoDB driver to access the MongoDB database.
- We will use the MongoDB driver "**PyMongo**".
- We recommend that you use PIP to install "PyMongo".
- PIP is most likely already installed in your Python environment.
- Navigate your command line to the location of PIP, and type the following:
- Download and install "PyMongo":

**C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>python -m pip install pymongo**

- Now you have downloaded and installed a mongoDB driver.

### **Test PyMongo:**

- To test if the installation was successful, or if you already have "pymongo" installed, create a Python page with the following content:

**demo\_mongodb\_test.py:**

```
import pymongo
```

- If the above code was executed with no errors, "pymongo" is installed and ready to be used.

### **Defining model:**

- Data in MongoDB has a flexible schema.documents in the same collection.
- They do not need to have the same set of fields or structure Common fields in a collection's documents may hold different types of data.

### **Data Model Design**

- MongoDB provides two types of data models: — Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.

#### **1. Embedded Data Model:**

- In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.
- For example, assume we are getting the details of employees in three different documents namely, Personal\_details, Contact and, Address, you can embed all the three documents in a single one as shown below –

```
{
  _id: ,
  Emp_ID: "10025AE336"
  Personal_details:{
    First_Name: "Radhika",
    Last_Name: "Sharma",
    Date_Of_Birth: "1995-09-26"
  },
  Contact: {
    e-mail: "radhika_sharma.123@gmail.com",
    phone: "9848022338"
  },
  Address: {
    city: "Hyderabad",
    Area: "Madapur",
    State: "Telangana"
  }
}
```

```
}
```

## 2. Normalized Data Model

- In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:

### Employee:

```
{
    _id: <ObjectId101>,
    Emp_ID: "10025AE336"
}
```

### Personal\_details:

```
{
    _id: <ObjectId102>,
    empDocID: " ObjectId101",
    First_Name: "Radhika",
    Last_Name: "Sharma",
    Date_Of_Birth: "1995-09-26"
}
```

### Contact:

```
{
    _id: <ObjectId103>,
    empDocID: " ObjectId101",
    e-mail: "radhika_sharma.123@gmail.com",
    phone: "9848022338"
}
```

### Address:

```
{
    _id: <ObjectId104>,
    empDocID: " ObjectId101",
    city: "Hyderabad",
    Area: "Madapur",
    State: "Telangana"
}
```

**Basic data access:****➤ Creating a Database**

- To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.
- MongoDB will create the database if it does not exist, and make a connection to it.

**Example:**

**Create a database called "mydatabase":**

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
```

```
mydb = myclient["mydatabase"]
```

- **Important:** In MongoDB, a database is not created until it gets content!
- MongoDB waits until you have created a collection (table), with at least one document (record) before it actually creates the database (and collection).

**➤ Check if Database Exists:**

- **Remember:** In MongoDB, a database is not created until it gets content, so if this is your first time creating a database, you should complete the next two chapters (create collection and create document) before you check if the database exists!
- You can check if a database exist by listing all databases in you system:

**Example:**

**Return a list of your system's databases:**

```
print(myclient.list_database_names())
```

- Or you can check a specific database by name:

**Example:**

**Check if "mydatabase" exists:**

```
dblist = myclient.list_database_names()
```

```
if "mydatabase" in dblist:
```

```
    print("The database exists.")
```

➤ **Creating a Collection:**

- A **collection** in MongoDB is the same as a **table** in SQL databases.
- To create a collection in MongoDB, use database object and specify the name of the collection you want to create.
- MongoDB will create the collection if it does not exist.

**Example:**

**Create a collection called "customers":**

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")  
mydb = myclient["mydatabase"]
```

```
mycol = mydb["customers"]
```

- **Important:** In MongoDB, a collection is not created until it gets content!
- MongoDB waits until you have inserted a document before it actually creates the collection.

➤ **Check if Collection Exists:**

- **Remember:** In MongoDB, a collection is not created until it gets content, so if this is your first time creating a collection, you should complete the next chapter (create document) before you check if the collection exists!
- You can check if a collection exist in a database by listing all collections:

**Example:**

**Return a list of all collections in your database:**

```
print(mydb.list_collection_names())
```

- Or you can check a specific collection by name:

**Example:**

**Check if the "customers" collection exists:**

```
collist = mydb.list_collection_names()  
if "customers" in collist:  
    print("The collection exists.")
```

- A **document** in MongoDB is the same as a **record** in SQL databases.

**Inserting and updating data:****➤ Insert Into Collection:**

- To insert a record, or document as it is called in MongoDB, into a collection, we use the `insert_one()` method.
- The first parameter of the `insert_one()` method is a dictionary containing the name(s) and value(s) of each field in the document you want to insert.

**Example:****Insert a record in the "customers" collection:**

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
```

```
mydict = { "name": "John", "address": "Highway 37" }
```

```
x = mycol.insert_one(mydict)
```

**➤ Return the \_id Field:**

- The `insert_one()` method returns a `InsertOneResult` object, which has a property, `inserted_id`, that holds the id of the inserted document.

**Example:****Insert another record in the "customers" collection, and return the value of the `_id` field:**

```
mydict = { "name": "Peter", "address": "Lowstreet 27" }
```

```
x = mycol.insert_one(mydict)
```

```
print(x.inserted_id)
```

- If you do not specify an `_id` field, then MongoDB will add one for you and assign a unique id for each document.
- In the example above no `_id` field was specified, so MongoDB assigned a unique `_id` for the record (document).

**➤ Insert Multiple Documents:**

- To insert multiple documents into a collection in MongoDB, we use the `insert_many()` method.
- The first parameter of the `insert_many()` method is a list containing dictionaries with the data you want to insert:

**Example:**

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mylist = [
    { "name": "Amy", "address": "Apple st 652"},
    { "name": "Hannah", "address": "Mountain 21"},
    { "name": "Michael", "address": "Valley 345"},
    { "name": "Sandy", "address": "Ocean blvd 2"},
    { "name": "Betty", "address": "Green Grass 1"},
    { "name": "Richard", "address": "Sky st 331"},
    { "name": "Susan", "address": "One way 98"},
    { "name": "Vicky", "address": "Yellow Garden 2"},
    { "name": "Ben", "address": "Park Lane 38"},
    { "name": "William", "address": "Central st 954"},
    { "name": "Chuck", "address": "Main Road 989"},
    { "name": "Viola", "address": "Sideway 1633"}
]

x = mycol.insert_many(mylist)
```

#print list of the \_id values of the inserted documents:  
print(x.inserted\_ids)

- The **insert\_many()** method returns a InsertManyResult object, which has a property, **inserted\_ids**, that holds the ids of the inserted documents.

➤ **Insert Multiple Documents, with Specified IDs:**

- If you do not want MongoDB to assign unique ids for you document, you can specify the \_id field when you insert the document(s).
- Remember that the values has to be unique. Two documents cannot have the same \_id.

**Example:**

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mylist = [
```

```
{ "_id": 1, "name": "John", "address": "Highway 37"},
{ "_id": 2, "name": "Peter", "address": "Lowstreet 27"},
{ "_id": 3, "name": "Amy", "address": "Apple st 652"},
{ "_id": 4, "name": "Hannah", "address": "Mountain 21"},
{ "_id": 5, "name": "Michael", "address": "Valley 345"},
{ "_id": 6, "name": "Sandy", "address": "Ocean blvd 2"},
{ "_id": 7, "name": "Betty", "address": "Green Grass 1"},
{ "_id": 8, "name": "Richard", "address": "Sky st 331"},
{ "_id": 9, "name": "Susan", "address": "One way 98"},
{ "_id": 10, "name": "Vicky", "address": "Yellow Garden 2"},
{ "_id": 11, "name": "Ben", "address": "Park Lane 38"},
{ "_id": 12, "name": "William", "address": "Central st 954"},
{ "_id": 13, "name": "Chuck", "address": "Main Road 989"},
{ "_id": 14, "name": "Viola", "address": "Sideway 1633"}
]
```

```
x = mycol.insert_many(mylist)
```

```
#print list of the _id values of the inserted documents:
print(x.inserted_ids)
```

### ➤ **Python MongoDB Update:**

#### ➤ **Update Collection:**

- You can update a record, or document as it is called in MongoDB, by using the **update\_one()** method.
- The first parameter of the **update\_one()** method is a query object defining which document to update.
- **Note:** If the query finds more than one record, only the first occurrence is updated.
- The second parameter is an object defining the new values of the document.



**Example:****Change the address from "Valley 345" to "Canyon 123":**

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Valley 345" }
newvalues = { "$set": { "address": "Canyon 123" } }

mycol.update_one(myquery, newvalues)

#print "customers" after the update:
for x in mycol.find():
    print(x)
```

**➤ Update Many:**

- To update *all* documents that meets the criteria of the query, use the update\_many() method.

**Example:****Update all documents where the address starts with the letter "S":**

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$regex": "^S" } }
newvalues = { "$set": { "name": "Minnie" } }

x = mycol.update_many(myquery, newvalues)

print(x.modified_count, "documents updated.")
```

## **Selecting objects:**

### **Python MongoDB Find:**

- In MongoDB we use the **find** and **findOne** methods to find data in a collection.
- Just like the **SELECT** statement is used to find data in a table in a MySQL database.

#### ➤ **Find One:**

- To select data from a collection in MongoDB, we can use the `find_one()` method.
- The `find_one()` method returns the first occurrence in the selection.

#### **Example:**

##### **Find the first document in the customers collection:**

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")  
mydb = myclient["mydatabase"]  
mycol = mydb["customers"]
```

```
x = mycol.find_one()
```

```
print(x)
```

#### ➤ **Find All:**

- To select data from a table in MongoDB, we can also use the `find()` method.
- The `find()` method returns all occurrences in the selection.
- The first parameter of the `find()` method is a query object.
- In this example we use an empty query object, which selects all documents in the collection.
- No parameters in the `find()` method gives you the same result as **SELECT \*** in MySQL.

#### **Example:**

##### **Return all documents in the "customers" collection, and print each document:**

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")  
mydb = myclient["mydatabase"]  
mycol = mydb["customers"]
```

```
for x in mycol.find():  
    print(x)
```

➤ **Return Only Some Fields:**

- The second parameter of the find() method is an object describing which fields to include in the result.
- This parameter is optional, and if omitted, all fields will be included in the result.

**Example:**

**Return only the names and addresses, not the \_ids:**

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
```

```
for x in mycol.find({}, { "_id": 0, "name": 1, "address": 1 }):
    print(x)
```

- You are not allowed to specify both 0 and 1 values in the same object (except if one of the fields is the \_id field).
- If you specify a field with the value 0, all other fields get the value 1, and vice versa:

**Example:**

**This example will exclude "address" from the result:**

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
```

```
for x in mycol.find({}, { "address": 0 }):
    print(x)
```

**Example:**

You get an error if you specify both 0 and 1 values in the same object (except if one of the fields is the \_id field):

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
```

```
for x in mycol.find({}, { "name": 1, "address": 0 }):  
    print(x)
```

➤ **Python MongoDB Query:**

➤ **Filter the Result:**

- When finding documents in a collection, you can filter the result by using a query object.
- The first argument of the find() method is a query object, and is used to limit the search.

**Example:**

**Find document(s) with the address "Park Lane 38":**

```
import pymongo  
  
myclient = pymongo.MongoClient("mongodb://localhost:27017/")  
mydb = myclient["mydatabase"]  
mycol = mydb["customers"]  
  
myquery = { "address": "Park Lane 38" }  
  
mydoc = mycol.find(myquery)  
  
for x in mydoc:  
    print(x)
```

➤ **Advanced Query:**

- To make advanced queries you can use modifiers as values in the query object.
- E.g. to find the documents where the "address" field starts with the letter "S" or higher (alphabetically), use the greater than modifier: { "\$gt": "S" }:

**Example:**

**Find documents where the address starts with the letter "S" or higher:**

```
import pymongo  
  
myclient = pymongo.MongoClient("mongodb://localhost:27017/")  
mydb = myclient["mydatabase"]  
mycol = mydb["customers"]  
  
myquery = { "address": { "$gt": "S" } }
```

```
mydoc = mycol.find(myquery)
```

```
for x in mydoc:  
    print(x)
```

➤ **Filter With Regular Expressions:**

- You can also use regular expressions as a modifier.
- **Regular expressions can only be used to query strings.**
- To find only the documents where the "address" field starts with the letter "S", use the regular expression { "\$regex": "^S" }:

**Example:**

**Find documents where the address starts with the letter "S":**

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")  
mydb = myclient["mydatabase"]  
mycol = mydb["customers"]
```

```
myquery = { "address": { "$regex": "^S" } }
```

```
mydoc = mycol.find(myquery)
```

```
for x in mydoc:  
    print(x)
```

➤ **Python MongoDB Sort:**

➤ **Sort the Result:**

- Use the sort() method to sort the result in ascending or descending order.
- The sort() method takes one parameter for "fieldname" and one parameter for "direction" (ascending is the default direction).

**Example:**

**Sort the result alphabetically by name:**

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")  
mydb = myclient["mydatabase"]  
mycol = mydb["customers"]
```

```
mydoc = mycol.find().sort("name")
```

```
for x in mydoc:  
    print(x)
```

➤ **Sort Descending:**

- Use the value -1 as the second parameter to sort descending.  
sort("name", 1) #ascending  
sort("name", -1) #descending

**Example:**

**Sort the result reverse alphabetically by name:**

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")  
mydb = myclient["mydatabase"]  
mycol = mydb["customers"]
```

```
mydoc = mycol.find().sort("name", -1)
```

```
for x in mydoc:  
    print(x)
```

**Deleting objects:**

**Python MongoDB Delete Document:**

➤ **Delete Document:**

- To delete one document, we use the delete\_one() method.
- The first parameter of the delete\_one() method is a query object defining which document to delete.
- **Note:** If the query finds more than one document, only the first occurrence is deleted.

**Example:**

**Delete the document with the address "Mountain 21":**

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
```

```
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Mountain 21" }

mycol.delete_one(myquery)
```

➤ **Delete Many Documents:**

- To delete more than one document, use the delete\_many() method.
- The first parameter of the delete\_many() method is a query object defining which documents to delete.

**Example:**

**Delete all documents where the address starts with the letter S:**

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$regex": "^S" } }
x = mycol.delete_many(myquery)

print(x.deleted_count, " documents deleted.")
```

➤ **Delete All Documents in a Collection:**

- To delete all documents in a collection, pass an empty query object to the delete\_many() method:

**Example:**

**Delete all documents in the "customers" collection:**

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

x = mycol.delete_many({})
```

```
print(x.deleted_count, " documents deleted.")
```

### **Python MongoDB Drop Collection:**

#### **➤ Delete Collection**

- You can delete a table, or collection as it is called in MongoDB, by using the drop() method.

#### **Example**

##### **Delete the "customers" collection:**

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
```

```
mycol.drop()
```

- The drop() method returns true if the collection was dropped successfully, and false if the collection does not exist.

### **Python MongoDB Limit:**

#### **➤ Limit the Result:**

- To limit the result in MongoDB, we use the limit() method.
- The limit() method takes one parameter, a number defining how many documents to return.
- Consider you have a "customers" collection:

#### **Example:**

##### **Limit the result to only return 5 documents:**

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
myresult = mycol.find().limit(5)
```

```
#print the result:
for x in myresult:
    print(x)
```