Python 3 - MySQL Database Access

Previous Page Next Page

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
- SQLite

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.

The DB API provides a minimal standard for working with databases using Python structures and syntax wherever possible. This API includes the following –

- Importing the API module.
- Acquiring a connection with the database.
- Issuing SQL statements and stored procedures.

Closing the connection

Python has an in-built support for SQLite. In this section, we would learn all the concepts using MySQL. MySQLdb module, a popular interface with MySQL is not compatible with Python 3. Instead, we shall use PyMySQL module.

What is PyMySQL ?

PyMySQL is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and contains a pure-Python MySQL client library. The goal of PyMySQL is to be a drop-in replacement for MySQLdb.

How do I Install PyMySQL?

Before proceeding furthur, you make sure you have PyMySQL installed on your machine. Just type the following in your Python script and execute it –

```
#!/usr/bin/python3

import PyMySQL
```

If it produces the following result, then it means MySQLdb module is not installed –

```
Traceback (most recent call last):
   File "test.py", line 3, in <module>
      Import PyMySQL
ImportError: No module named PyMySQL
```

The last stable release is available on PyPI and can be installed with pip –

```
pip install PyMySQL
```

Alternatively (e.g. if pip is not available), a tarball can be downloaded from GitHub and installed with Setuptools as follows –

```
$ # X.X is the desired PyMySQL version (e.g. 0.5 or 0.6).
$ curl -L https://github.com/PyMySQL/PyMySQL/tarball/pymysql-X.X | tar xz
$ cd PyMySQL*
$ python setup.py install
$ # The folder PyMySQL* can be safely removed now.
```

Note – Make sure you have root privilege to install the above module.

Database Connection

Before connecting to a MySQL database, make sure of the following points –

You have created a database TESTDB.

You have created a table EMPLOYEE in TESTDB.

This table has fields FIRST_NAME, LAST_NAME, AGE, SEX and INCOME.

User ID "testuser" and password "test123" are set to access TESTDB.

Python module PyMySQL is installed properly on your machine.

You have gone through MySQL tutorial to understand MySQL Basics.

Example

Following is an example of connecting with MySQL database "TESTDB" –

```
#!/usr/bin/python3

import PyMySQL

# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# execute SQL query using execute() method.
cursor.execute("SELECT VERSION()")

# Fetch a single row using fetchone() method.
data = cursor.fetchone()
print ("Database version : %s " % data)

# disconnect from server
db.close()
```

While running this script, it produces the following result.

```
Database version : 5.5.20-log
```

If a connection is established with the datasource, then a Connection Object is returned and saved into **db** for further use, otherwise **db** is set to None. Next, **db** object is used to create a **cursor** object, which in turn is used to execute SQL queries. Finally, before coming out, it ensures that the database connection is closed and resources are released.

Creating Database Table

Once a database connection is established, we are ready to create tables or records into the database tables using **execute** method of the created cursor.

Example

Let us create a Database table EMPLOYEE –

```
#!/usr/bin/python3

import PyMySQL
```

```
# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Drop table if it already exist using execute() method.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# Create 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)

# disconnect from server
db.close()
```

INSERT Operation

The INSERT Operation is required when you want to create your records into a database table.

Example

The following example, executes SQL *INSERT* statement to create a record in the EMPLOYEE table –

```
#!/usr/bin/python3

import PyMySQL

# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

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

# disconnect from server
db.close()
```

The above example can be written as follows to create SQL queries dynamically –

```
#!/usr/bin/python3

import PyMySQL

# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to INSERT a record into the database.
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \
   LAST_NAME, AGE, SEX, INCOME) \
   VALUES ('%s', '%s', '%d', '%c', '%d' )" % \
   ('Mac', 'Mohan', 20, 'M', 2000)
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()

# disconnect from server
db.close()
```

Example

The following code segment is another form of execution where you can pass parameters directly –

```
.................................
user_id = "test123"
password = "password"

con.execute('insert into Login values("%s", "%s")' % \
            (user_id, password))
.................................
```

READ Operation

READ Operation on any database means to fetch some useful information from the database.

Once the database connection is established, you are ready to make a query into this database. You can use either **fetchone()** method to fetch a single record or **fetchall()** method to fetch multiple values from a database table.

> **fetchone()** – It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.

fetchall() – It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.

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

Example

The following procedure queries all the records from EMPLOYEE table having salary more than 1000 –

```
#!/usr/bin/python3

import PyMySQL

# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to INSERT a record into the database.
sql = "SELECT * FROM EMPLOYEE \
       WHERE INCOME > '%d'" % (1000)
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Fetch all the rows in a list of lists.
   results = cursor.fetchall()
   for row in results:
      fname = row[0]
      lname = row[1]
      age = row[2]
      sex = row[3]
      income = row[4]
      # Now print fetched result
      print ("fname = %s,lname = %s,age = %d,sex = %s,income = %d" % \
             (fname, lname, age, sex, income ))
except:
   print ("Error: unable to fetch data")

# disconnect from server
db.close()
```

Output

This will produce the following result –

```
fname = Mac, lname = Mohan, age = 20, sex = M, income = 2000
```

Update Operation

UPDATE Operation on any database means to update one or more records, which are already available in the database.

The following procedure updates all the records having SEX as **'M'**. Here, we increase the AGE of all the males by one year.

Example

```
#!/usr/bin/python3

import PyMySQL

# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to UPDATE required records
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1
                          WHERE SEX = '%c'" % ('M')
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()

# disconnect from server
db.close()
```

DELETE Operation

DELETE operation is required when you want to delete some records from your database. Following is the procedure to delete all the records from EMPLOYEE where AGE is more than 20 −

Example

```
#!/usr/bin/python3

import PyMySQL

# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
   # Execute the SQL command
```

```
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()

# disconnect from server
db.close()
```

Performing Transactions

Transactions are a mechanism that ensures data consistency. Transactions have the following four properties –

Atomicity – Either a transaction completes or nothing happens at all.

Consistency – A transaction must start in a consistent state and leave the system in a consistent state.

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

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

The Python DB API 2.0 provides two methods to either *commit* or *rollback* a transaction.

Example

You already know how to implement transactions. Here is a similar example –

```
# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()
```

COMMIT Operation

Commit is an operation, which gives a green signal to the database to finalize the changes, and after this operation, no change can be reverted back.

Here is a simple example to call the **commit** method.

```
db.commit()
```

ROLLBACK Operation

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use the **rollback()** method.

Here is a simple example to call the **rollback()** method.

```
db.rollback()
```

Disconnecting Database

To disconnect the Database connection, use the close() method.

```
db.close()
```

If the connection to a database is closed by the user with the close() method, any outstanding transactions are rolled back by the DB. However, instead of depending on any of the DB lower level implementation details, your application would be better off calling commit or rollback explicitly.

Handling Errors

There are many sources of errors. A few examples are a syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already canceled or finished statement handle.

The DB API defines a number of errors that must exist in each database module. The following table lists these exceptions.

S.No.	Exception & Description
1	**Warning** Used for non-fatal issues. Must subclass StandardError.
2	**Error** Base class for errors. Must subclass StandardError.
3	**InterfaceError** Used for errors in the database module, not the database itself. Must subclass Error.
4	**DatabaseError** Used for errors in the database. Must subclass Error.

5	**DataError** Subclass of DatabaseError that refers to errors in the data.
6	**OperationalError** Subclass of DatabaseError that refers to errors such as the loss of a connection to the database. These errors are generally outside of the control of the Python scripter.
7	**IntegrityError** Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys.
8	**InternalError** Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active.
9	**ProgrammingError** Subclass of DatabaseError that refers to errors such as a bad table name and other things that can safely be blamed on you.
10	**NotSupportedError** Subclass of DatabaseError that refers to trying to call unsupported functionality.

Your Python scripts should handle these errors, but before using any of the above exceptions, make sure your MySQLdb has support for that exception. You can get more information about them by reading the DB API 2.0 specification.

Previous Page Next Page