UNIT 4

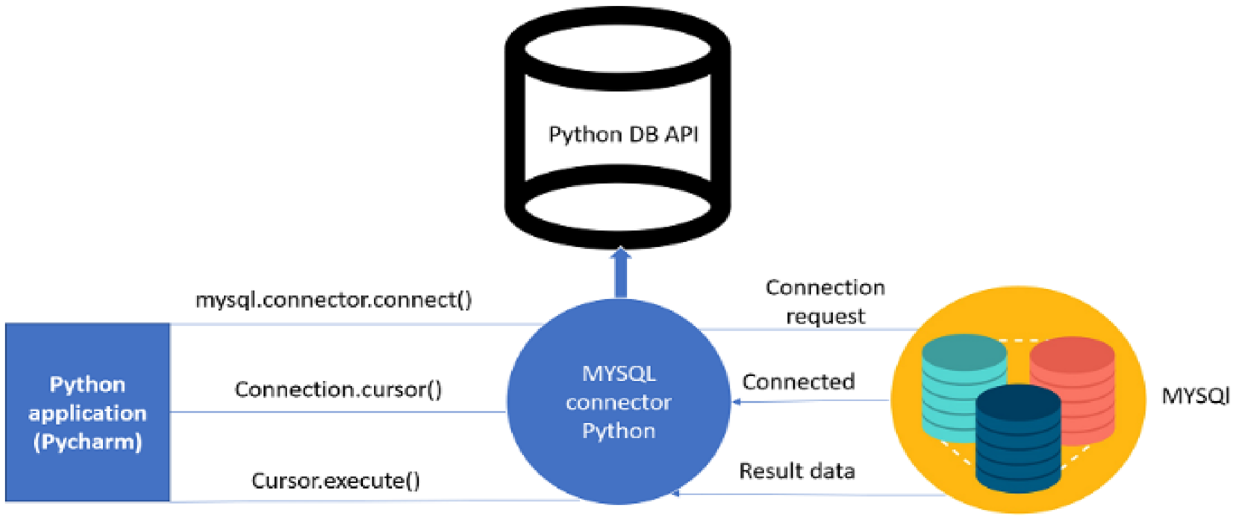
Connecting to Database

Python allows us to connect to various databases though database interfaces. Python's database interface is DB-API (Data Base - Application Programming Interface). We can choose the database which we want to connect to Python. Python DB-API supports a large number of databases like Oracle, MS-SQL server 2000, MySQL, mSQL, Sybase etc. For using each database we need to download separate DB-API modules.

The API includes the following steps which include

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

Here explains all the concepts and examples using MySQL. MySQL is an open source database. For connecting to MySQL database, we need pymysql, which is an interface connecting MySOL and Python. Before establishing a connection between Python and MySQL, we should make sure that pymysql is installed in our machine.



1 CONNECTING TO A DATABASE

Before connecting to a database, following things should be done to assure the right connectivity.

1. Create a database called SAMPLE in MySQL.
2. The user id and password used to access SAMPLE database is "user" and "pass" respectively.
3. Create a table STUDENT in SAMPLE.
4. The table STUDENT has fields ROLLNO, NAME, AGE, COURSE, GRADE.

The following code shows how to connect MySQL database with Python.

Example Program

```
import pymysql
# Open database connection
db = pymysql.connect("localhost", "user", "pass" , "SAMPLE" )
# 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:", data)
# disconnect from server
db.close()
```

While running this script, it is producing the following output in Windows machine.

Output

```
Database version: 5.6.36-log
```

In the above example, a connection is established with the data source and a connection object is returned. In our example, the connection object is saved to db. If connection is not properly established, db will store the value None. The database object db is used to create a cursor object cursor and this cursor is used for executing SQL queries. At the end, database connection is closed and resources are released.

2 CREATING TABLES

Once a connection is successfully established, we can create the tables. Creation of tables, insertion, Updation and deletion operations are performed in Python with the help of **execute() statement**. The following example shows how to create a table in MySQL from Python.

Example Program

```
import pymysql
# Open database connection
db = pymysql.connect ("localhost", "user", "pass" , "SAMPLE" )
# prepare a cursor object using cursor() method
cursor = db.cursor()
 # Create table as per requirement
sql = " CREATE TABLE DEPT( DEPTNO INT, DEPT_NAME CHAR (20), LOCATION CHAR (25)) "
cursor.execute(sql)
```

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

The program creates a table called DEPT with fields DEPTNO, DEPT NAME and TOCATION. The SQL statement for creating the table is stored in sql and the sql is passed to execute() method for creating the table.

3 INSERT OPERATION

Once a table is created, we need to insert values to the table. The values can be inserted using the INSERT statement of SQL. The following example shows an example for insert operation.

Example Program

```
import pymysql
# Open database connection
db = pymysql.connect("localhost","user", "pass" , "SAMPLE" )
# prepare a cursor object using cursor() method
cursor = db.cursor()
# Prepare SQL query to INSERT a record into the database.
sql = " INSERT INTO DEPT (DEPTNO, DEPT_NAME, LOCATION)
        VALUES (10, 'Sales', 'Chennai) "
cursor.execute(sql)
db.commit()
# disconnect from server
db.close()
```

A record with specified values will be inserted into the DEPT table.

4 UPDATE OPERATION

UPDATE operation is done to modify existing values available in the table. We can update e or more records at the same time. The following example shows how to update records in a table from Python.

Example Program

```
import pymysql
#Open database connection
db=pymysql.connect("localhost", "user", "pass". "SAMPLE")
# prepare a cursor object using cursor() method
cursor = db.cursor()
 # Prepare SQL query to UPDATE records into a table.
```

```
sql = "UPDATE DEPT SET LOCATION = 'Delhi'   WHERE DEPT NAME = 'SALES' "
cursor.execute (sql)
db.commit()
# disconnect from server
db.close()
```

The above example updates the location of sales department from Chennai to Delhi. If there is more than one record with the satisfied condition, then all those records will be updated.

5 DELETE OPERATION

DELETE operation is required when we want to delete some undesired or unwanted records from a table. We can specify DELETE operation with or without conditions. The following example shows how to delete records from a table.

Example Program

```
import pymysql
# Open database connection
db = pymysql.connect ("localhost","user", "pass" , "SAMPLE" )
# prepare a cursor object using cursor() method
cursor = db.cursor()
# Prepare SQL query to DELETE records from a table.
sql = "DELETE FROM DEPT WHERE LOCATION = 'Delhi' "
cursor.execute(sql)
db.commit()
# disconnect from server
db.close()
```

The above example will delete all the records with location Delhi.

6 READ OPERATION

READ operation is used to **fetch desired records from a database**. There are several methods for fetching records from a database. Once a connection is established, we can make queries to a database. The following methods are used in READ operation.

1. 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.

Example Program

```
import pymysql
```

```
# Open database connection
pomysql.connect("localhost", "user", "pass" , "SAMPLE”
#prepare a cursor object using cursor() method
cursor = db.cursor()
# Prepare SQL query
sql = "SELECT LOCATION FROM DEPT"
cursor.execute(sql)
# Fetches the next Record. In this case first Record
row = cursor.fetchone ()
if row:
        print ("Location:", row [0] )
# disconnect from server
db.close()
```

Output

```
Location: Delhi
```

2. 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.

Example Program

```
import pymysql
# Open database connection
db = pymysql.connect("localhost", "user", "pass" , "SAMPLE")
* prepare a cursor object using cursor() method
cursor = db.cursor()
# Prepare SQL query
sql = "SELECT LOCATION FROM DEPT"
cursor.execute(sal)
# Fetches all the records.
rows = cursor.fetchall()
for row in rows:
        print('Location:", row[0])
# disconnect from server
db.close()
```

Output

```
Location: Delhi
```

```
Location: Chennai
Location: Mumbai
```

3. rowcount This is a read only abuse and affected by an execute() method

Example Program

```
import prysai
cursor = d.cursor()
# Prepare SOL query
sql = "SELECT LOCATION FROM DEPT "
cursor.execute(sql)
# Displays number of records
numrows=cursor.rowcount
print("Number of Records:", numrows.rowcount)
#disconnect from server
db.close()
```

Output

```
Number of Records:3
```

7 TRANSACTION CONTROL

A transaction is a logical unit of work that contains one or more SQL statements. A transaction is an atomic unit. The effects of all the SQL statements in a transaction can be either all committed (applied to the database) or all rolled back (undone from the database). Transaction is a mechanism to ensure data consistency. Transaction ensures 4 properties generally referred to as ACID properties.

Atomicity: ensures that all operations within the work unit are completed success otherwise, the transaction is aborted at the point of failure, and previous operations are rolled back to their former state.

Consistency: ensures that the database properly change states upon a success committed transaction.

Isolation: enable transactions to operate independently and transparent to each other

Durability: ensures that the result or effect of a committed transaction persists in case of a system failure

7.1 COMMIT Operation

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command. The syntax for a commit statement db.commit().

7.2 ROLLBACK Operation

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued. The syntax for rollback is db.rollback().

Example Program for COMMIT and ROLLBACK

```
import pymysql
# Open database connection
db = pymysql.connect("localhost","user", "pass" , "SAMPLE" )
# prepare a cursor object using cursor() method
cursor = db.cursor()
 # Prepare SQL query to INSERT a record into the database.
sql = "INSERT INTO DEPT (DEPTNO, DEPT_NAME, LOCATION)
        VALUES (10,'Sales', 'Chennai') "
cursor.execute(sql)
# Commit changes in the database
db.commit()
except:
        # Rollback in case there is any error
        db.rollback()
# disconnect from server
db.close()
```

In the above example, commit() statement makes changes to the database permanently. If there is any error, rollback() is called to undo the changes.

8 DISCONNECTING FROM A DATABASE

A close() method is called to disconnect from the database. The syntax for closing is 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 database. However, instead of depending on any of database lower level implementation details, our application would be better to call commit or rollback explicitly.

9 EXCEPTION HANDLING IN DATABASES

There are many sources of errors in databases. A few examples are a syntax error in an executed SOL statement, a connection failure, or calling the fetch method for an already canceled or finished statement handle. The database API defines a number of errors that must exist in each database module. The following Table 1 list the Exceptions and their descriptions related to databases.

Table 1 Exceptions related to databases.

Exception	Description
DatabaseError	Used for errors in the database. Must subclass Error.
DataError	Subclass of DatabaseError that refers to errors in the data.
Error	Base class for errors. Must subclass Standard Error.
IntegrityError	Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys.
InterfaceError	Used for errors in the database module, not the database itself. Must subclass Error.
InternalError	Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active.
NotSupportedError	Subclass of DatabaseError that refers to trying to call unsupported functionality.
OperationalError	Subclass of Database Error that refers to errors such as the loss of a connection to the database. These errors are generally outside the control of the Python scripter.
ProgrammingError	Subclass of DatabaseError that refers to errors such as a bad table name
Warning	Used for non-fatal issues. Must subclass Standard Error.