COMPUTER SCIENCE With python

Textbook for Class XII

- Programming & Computational Thinking
- Computer Networks
- Data Management (SQL, Django)
- Society, Law and Ethics

SUMITA ARORA

DHANPAT RAI & Co.

16 Interface Python with MySQL

511 – 524

16 Interface Python with MySQL

In This Chapter

16.1 INTRODUCTION

When you design real-life applications, you are bound to encounter situations wherein you need to manipulate data stored in a database through an application designed by you. Since you are developing Python applications, in this chapter our discussion will be based on how you can connect to a MySQL database from within a Python script.

In order to connect to a database from within Python, you need a library that provides connectivity functionality. There are many different libraries available for Python to accomplish this. We shall work with **mysql connector** library for the same.

NOTE

Before you start working with Python mysql connector, you need to install it on your computer. Both **pip** and **Conda** installation of Python MySQL connector is given in **Appendix E** of this book. So, please refer to **Appendix E** for the installation of the same.

16.2 CONNECTING TO MYSQL FROM PYTHON

After you have installed Python **MySQL connector** (as per *Appendix E*), you can write Python Scripts using **MySQL.connector** library that can connect to MySQL databases from within Python.

16.2.1 Steps for Creating Database Connectivity Applications

There are mainly *seven* steps that must be followed in order to create a database connectivity application.

Step 1 Start Python.

Step 2 Import the packages required for database programming.

Step 3 Open a connection to database.

Step 4 Create a cursor instance.

Step 5 Execute a query.

Step 6 Extract data from result set.

Step 7 Clean up the environment.

Let us talk about these steps in details.

Step 1. Start Python

Start Python's editor where you can create your Python scripts. It can be IDLE or Spyder IDE – whatever you feel comfortable in.

Step 2. Import mysql.connector Package

First of all you need to import **mysql.connector** package in your Python scripts. For this, write import command as shown below :

```
import mysql.connector
```

or

```
import mysql.connector as sqltor
```

You can use any identifier of your choice

Step 3. Open a Connection to MySQL Database

Next you need to establish connection to a MySQL database using **connect( )** function of **mysql.connector** package.

The **connect( )** function of **mysql.connector** establishes connection to a MySQL database and requires four parameters, which are :

```
<Connection-Object> = mysql.connector.connect (host = <host-name> , user = <username>,
                                               passwd = <password> [, database = <database>])
```

- *user* is the username on MySQL
- *password* is the password of the user
- *host-name* the database server hostname or IP address
- *database* is optional which provides the database name of a MySQL database.

Specify a Python identifier as **<connection object>** name. You shall be using this name for executing queries and for other tasks on connected database.

For example :

loginid and password of your MySQL database

```
import mysql.connector as sqltor
mycon = sqltor.connect(host = "localhost", user = "root", passwd = "MyPass",
                       database = "test")
```

The connection object

a MySQL database

Since we imported **mysql.connector** package as **sqltor**, we are using the name **sqltor** in place of **mysql.connector**. If you have imported the package as

```
import mysql.connector
```

Then you need to write above function as :

```
mycon = mysql.connector.connect(host = "localhost", user = "root",
                                passwd = "MyPass", database = "test")
```

The above command will establish connection to MySQL database with **user as "root", password as "MyPass"** and to the **MySQL database namely test** which exists on the MySQL.

Here, you must ensure that the user and password that you specify are valid user and password for your local MYSQL installation.

If Python reports no error, it means you have successfully connected to MySQL database. You can also check for successful connection using function **is_connected( )** with connected object (which returns *True*, if connection is successful, *i.e.*, if you may write following additional code to check if the connection is successful or not)

CONNECTION OBJECT

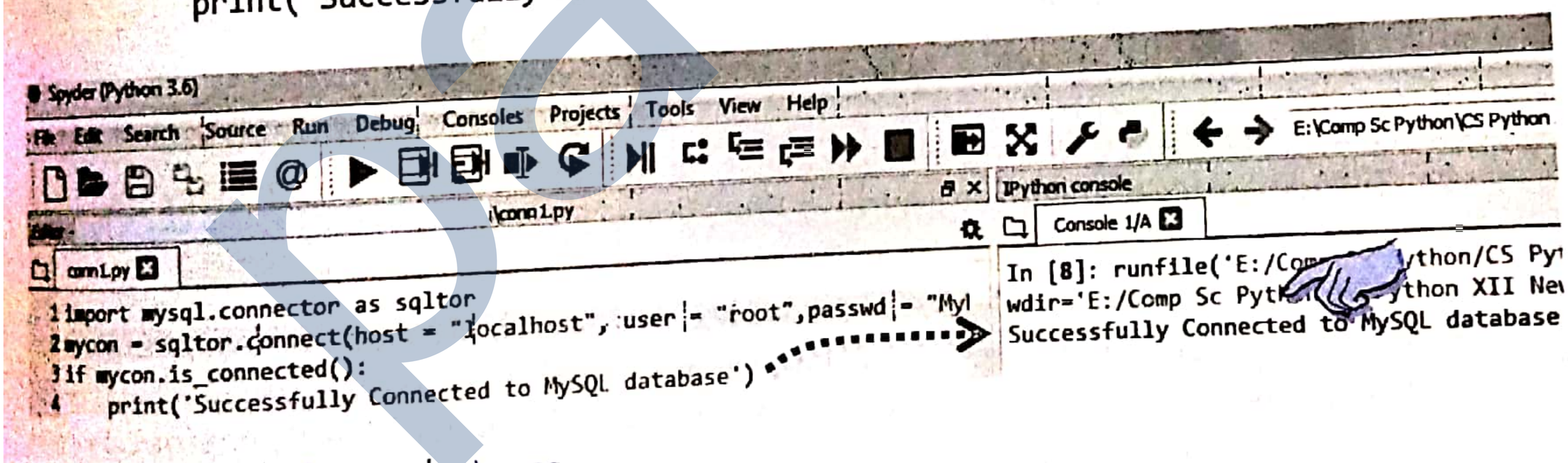
A database **Connection object** controls the connection to the database. It represents a unique session with a database connected from within a script/program.

The same connection object with which we connected to MySQL database

```
if mycon.is_connected():
    print('Successfully Connected to MySQL database')
```

Following screenshot shows you the output produced by code shown below :

```
import mysql.connector as sqltor
mycon = sqltor.connect(host = "localhost", user = "root", passwd = "MyPass", database ="test")
if mycon.is_connected():
    print('Successfully Connected to MySQL database')
```



Step 4. Create a Cursor Instance

A database cursor is a useful control structure of database connectivity. Normally when you connect to a database from within a script/program, then the query gets sent to the server, where it gets executed, and the *resultset* (the set of records retrieved as per query) is sent over the connection to you, in **one burst of activity** *i.e.*, in one go. But you may want to access the retrieved data, one row at a time. But query processing cannot happen as one row at a time, so a special type of control structure called **database cursor** can be created that gets the access of all the records retrieved as per query (called the **resultset**) and allows you to traverse the *resultset* row by row.

In this step(step 4), you create an instance of cursor by using **cursor( )** function as per following syntax :

```
<cursorobject> = <connectionobject>.cursor()
```

That is, for our connection established in earlier steps, we can create **cursor( )** by writing :

```
cursor = mycon.cursor()
```

Cursor object created — *Connection object*

Since we established database connection through connection object **mycon** earlier, we have created a *cursor object* using the same connection object **mycon.**

DATABASE CURSOR

A **Database Cursor** is a special control structure that facilitates the row by row processing of records in the resultset, *i.e.*, the set of records retrieved as per query.

Step 5. Execute SQL Query

Once you have created a cursor, you can execute SQL query using **execute( )** function with cursor object as per following syntax :

```
<cursorobject>.execute(<sql query string>)
```

For example, if you want to view all the records of table *data* which is a table in the database *test* to which you established connection in step 2, you can execute SQL query "select * from data" by writing :

```
cursor.execute("select * from data")
```

cursor object name — *Give SQL query in quotes*

The above code will execute the given SQL query and store the retrieved records (*i.e.*, the *resultset*) in the cursor object (namely **cursor**) which you can then use in your programs/scripts as required.

RESULT SET

The **result set** refers to a logical set of records that are fetched from the database by executing an SQL query and made available to the application program.

Step 6. Extract Data from Resultset

You will need this step (step 6) if you have retrieved records from the database using SQL SELECT query and need to extract records from the retrieved resultset. Once the result of query is available in the form of a resultset stored in a cursor object, you can extract data from the resultset using any of the following *fetch...( )* functions.

(*i*) **<data> = <cursor>.fetchall( ).** It will return all the records retrieved as per query in a tuple form (*i.e.*, now <data> will be a tuple.)

(*ii*) **<data> = <cursor>.fetchone( ).** It will return one record from the *resultset* as a tuple or a list. First time it will return the first record, next time it will fetch the next record and so on.

This method returns one record as a tuple : if there are no more records then it returns **None.**

(*iii*) **<data> = <cursor>.fetchmany(<n>).** This method accepts number of records to fetch and returns a tuple where each record itself is a tuple. If there are not more records then it returns an empty tuple.

(*iv*) **<variable> = <coursor>.rowcount.** The *rowcount* is a property of cursor object that returns the number of rows retrieved from the cursor so far. Following examples will make it more clear.

Let us now see with following code examples how these functions work. For the following code examples, we shall be connecting to database *test*, table *student* that has the following data in it :

Table *student* of MySQL database *test*

Rollno	Name	Marks	Grade	Section	Project
101	Ruhani	76.80	A	A	Pending
102	George	71.20	B	A	Submitted
103	Simran	81.20	A	B	Evaluated
104	Ali	61.20	B	C	Assigned
105	Kushal	51.60	C	C	Evaluated
106	Arsiya	91.60	A+	B	Submitted
107	Raunak	32.50	F	B	Submitted

Following code examples assume that the connection to the database has been established using **connect( )** method of *mysql.connector* as discussed in earlier steps. That is, all the following code examples of fetch functions have following code pre-executed for them :

```
import mysql.connector as sqltor
mycon = sqltor.connect(host = "localhost", user = "root", passwd = "MyPass",
                       database = "test")

if mycon.is_connected() == False:
    print('Error connecting to MySQL database')
cursor = mycon.cursor()
cursor.execute("select * from student")
```

*The SQL query retrieves all the data of table **student** of database **test***

Let us now see that in addition to above code how all *fetch..() methods* will behave.

(i) The fetchall( ) method

The **fetchall( )** method will return all the rows from the resultset in the form of a tuple containing the records.

```
:    # database connected established and cursor object created
     data = cursor.fetchall()
     count = cursor.rowcount
     print("Total number of rows retrieved in resultset :", count)

     for row in data :
         print(row)
```

Fetch all the records in the resultset

How many records returned by SQL query in the resultset

*The **data** variable will store the retrieved records from the resultset in the form of a tuple (a tuple of re-*

*Now you can process the **data** tuple one row at a time.*

The output produced by above code is :

```
Total number of rows retrieved in resultset : 7
(101, 'Ruhani', Decimal('76.80'), 'A', 'A', 'Pending')
(102, 'George', Decimal('71.20'), 'B', 'A', 'Submitted')
(103, 'Simran', Decimal('81.20'), 'A', 'B', 'Evaluated')
(104, 'Ali',    Decimal('61.20'), 'B', 'C', 'Assigned')
(105, 'Kushal', Decimal('51.60'), 'C', 'C', 'Evaluated')
(106, 'Arsiya', Decimal('91.60'), 'A+', 'B', 'Submitted')
(107, 'Raunaq', Decimal('32.50'), 'F', 'B', 'Submitted')
```

*Result of **cursor.rowcount***

***cursor.fetchall()** 's result was stored in data, which is then printed row by row*

(ii) The fetchmany( ) method

The **fetchmany(<n>)** method will return only the <n> number of rows from the resultset in the form of a tuple containing the records.

```
:    # database connected established and cursor object created
     data = cursor.fetchmany(4)      <-- Fetch 4 records in the resultset
     count = cursor.rowcount         <-- How many records returned by SQL query in the resultset
     print("Total number of rows retrieved from resultset :", count)
     for row in data :               <-- Now you can process the data tuple one row at a time
         print(row)
```

The data variable will store the retrieved records from the resultset in the form of a tuple (a tuple of records)

The output produced by above code is :

```
Total number of rows retrieved from resultset : 4   <-- Result of cursor.rowcount
(101, 'Ruhani', Decimal('76.80'), 'A', 'A', 'Pending')
(102, 'George', Decimal('71.20'), 'B', 'A', 'Submitted')
(103, 'Simran', Decimal('81.20'), 'A', 'B', 'Evaluated')
(104, 'Ali',    Decimal('61.20'), 'B', 'C', 'Assigned')
```

NOTE

The **cursor.rowcount** returns how many rows have been so far retrieved through **fetch..( )** methods from the cursor.

(iii) The fetchone( ) method

The **fetchone( )** method will return only one row from the resultset in the form of a tuple containing a record. A pointer is initialized which points to the first record of the resultset as soon as you execute a query. The **fetchone( )** returns the record pointed to by this pointer. When you fetch one record, the pointer moves to next record of the recordset. So next time , if you execute the **fetchone( )** metod, it will return only one record pointed to by the pointer and after fetching, the pointer will move to the next record of the resultset.

Also, carefully notice the behaviour of **cursor.rowcount** that always returns how many records have been **retrieved so far** using any of the *fetch..( ) methods*.

```
:    # database connected established and cursor object created
     data = cursor.fetchone()        <-- Fetch 1 records in the resultset (first time, only the first record is retrieved)
     count = cursor.rowcount
     print("Total number of rows retrieved from resultset :", count)
     print(data)
     print("\nAgain fetching one record")
     data = cursor.fetchone()        <-- Next fetchone( ) will fetch the next record from the resultset
     count = cursor.rowcount
     print("Total number of rows retrieved from resultset :", count)
     print(data)
```

```
Total number of rows retrieved in resultset : 1   <-- Result of cursor.rowcount
(101, 'Ruhani', Decimal('76.80'), 'A', 'A', 'Pending')
Again fetching one record
Total number of rows retrieved from resultset : 2   <-- Result of cursor.rowcount
(102, 'George', Decimal('71.20'), 'B', 'A', 'Submitted')
```

*Result of **cursor.rowcount***
This time it is 1 because fetchone() method retrieved only 1 record from the cursor

*Result of **cursor.rowcount***
This time it is 2 because fetchone() method retrieved only 1 record (next record) from the cursor but SO FAR 2 records have been retrieved.

Now can you guess the output of following code?

```
import mysql.connector as sqltor
mycon = sqltor.connect(host = "localhost", user = "root", passwd = "MyPass",
                       database ="test")
if mycon.is_connected() == False:
    print('Error connecting to MySQL database')
cursor = mycon.cursor()
cursor.execute("select * from student")
data = cursor.fetchone()
count = cursor.rowcount
print("Total number of rows retrieved so far from resultset :", count)
data = cursor.fetchone()
count = cursor.rowcount
print("Total number of rows retrieved so far from resultset :", count)
data = cursor.fetchmany(3)
count = cursor.rowcount
print("Total number of rows retrieved so far from resultset :", count)
```

Well, you guessed it right. The output is :

```
Total number of rows retrieved so far from resultset : 1
Total number of rows retrieved so far from resultset : 2
Total number of rows retrieved so far from resultset : 5
```

Step 7. Clean Up the Environment

After you are through all the processing, in this final step, you need to close the connection established.

This, you can do as follows :

```
<connection object>.close( )
```

e.g., we established the database connection via connection object **mycon** above, so

We shall write :

```
mycon.close( )
```

Following example lists the complete code wherein only first three records of the database table **student** of MySQL database **test** are listed row by row.

Program 16.1 Write a Python program that displays first three rows fetched from **student** table of MySQL database "test'.
(**Note.** user is "learner" and password is "fast")

```
import mysql.connector as sqltor
mycon = sqltor.connect(host = "localhost", user = "learner", passwd = "fast", database = "test")
if mycon.is_connected() == False:
    print('Error connecting to MySQL database')
```

```
cursor = mycon.cursor()
cursor.execute("select * from student")
data = cursor.fetchmany(3)
count = cursor.rowcount
for row in data :
      print(row)
mycon.close()
```

```
(101, 'Ruhani', Decimal('76.80'), 'A', 'A', 'Pending')
(102, 'George', Decimal('71.20'), 'B', 'A', 'Submitted')
(103, 'Simran', Decimal('81.20'), 'A', 'B', 'Evaluated')
```

16.3 PARAMETERISED QUERIES

In previous section, you learnt about how you can execute simple SQL queries after establishing connection to MySQL database. But SQL queries that were used, were simple queries. Sometimes, you may need to run queries which are based on some parameters or values that you provide from outside, *e.g.*,

```
inp = 70
```

```
SELECT * FROM student WHERE MARKS > inp ;
```

Such queries are called parameterised queries. To execute parameterised queries in a *mysql.connector* connection, you need to form SQL query strings that include values of parameters. Let us talk about how you can form such query strings.

Forming Query Strings

Note, for all the following code examples, we are again connecting to the *student* table of MySQL database *test*, the one we have used earlier. The connection object name is **mycon** and the cursor object name is **cursor** as earlier. That is, the code for connection is :

```
import mysql.connector as sqltor
mycon = sqltor.connect(host = "localhost", user = "learner", passwd = "fast", database = "test")
if mycon.is_connected() == False:
    print('Error connecting to MySQL database')
cursor = mycon.cursor()
```

Let us now discuss about how you can form parametrised query strings. To form query strings based on some parameters, you can go for one of the following *two* methods:

(i) Old Style : String Templates with % formatting

In this style, string formatting uses this general form : f % v

where **f** is a template string and **v** specifies the value or values to be formatted using that template. If multiple values are to be formatted, **v** must be a tuple. For this you can write the SQL query in a string but use a **%s** code in place of the value to be provided as a parameter.

For example,

```
"select * from student where marks > %s"
```

← *This means it is a parameter and its value must be provided*

The above string is an incomplete string (only **f**, no **%v** here) as it contains a placeholder for parameter as **%<code>**, but its parameter value is not provided here. To complete it, you must provide a tuple of values with% prefixed to it next to the string (called the value tuple, **% v**),

e.g., if you want to provide value 70 for %s placeholder in above query string, you can form the value tuple as follows :

```
(70,)
```
← *For single value to be stored in a tuple, comma must follow it*

Now you can complete the above SQL query string as follows :

```
"select * from student where marks > %s"   %(70,)
```
f v

Now you can store this query string in variable and then execute that variable through **cursor.execute( )** method as shown below :

```
:    # database connected established and cursor object created
st = "select * from student where marks > %s" %(70,)
cursor.execute( st )
data = cursor.fetchall()
for row in data :
    print(row)
```

```
(101, 'Ruhani', Decimal('76.80'), 'A', 'A', 'Pending')
(102, 'George', Decimal('71.20'), 'B', 'A', 'Submitted')
(103, 'Simran', Decimal('81.20'), 'A', 'B', 'Evaluated')
(106, 'Arsiya', Decimal('91.60'), 'A+', 'B', 'Submitted')
```

NOTE

Do not forget to enclose placeholder %s in quotes for string parameters in string template.

In the similar manner, you may add multiple parameter values, but one caution you must exercise is that do not forget to enclose placeholder %s in quotes for string parameters *e.g.,*

```
:    # database connected established and cursor object created
st = "select * from student where marks > %s and section = '%s' " % (70, 'B')
cursor.execute(st)
data = cursor.fetchall()
for row in data :
    print(row)
```

See, '%s' enclosed in quotes for string value

V tuple containing a number and a string value

```
(103, 'Simran', Decimal('81.20'), 'A', 'B', 'Evaluated')
(106, 'Arsiya', Decimal('91.60'), 'A+', 'B', 'Submitted')
```

Alternatively, you can provide the tuple with the execute( ) function as :

```
cursor.execute(<parameterised sql query string>, <values tuple>)
```

e.g.,

```
ram = 20
id = 2
input = (ram, id)
query = "Update computers set ram = %s where id = %s"
cursor.execute(query , input)
```

(ii) New Style : String Templates with % formatting

The new style of creating SQL query strings involves the use of the .format( ) method of the str type. Let us first understand this with a normal string and then we shall apply it on SQL query strings. The general form for using.**format( )** is :

```
template.format(p0, p1, ..., k0 = v0, k1 = v1, ...)
```

The **template** is a string containing a mixture of one or more format codes embedded in constant text. The format method uses its arguments to substitute an appropriate value for each format code in the template.

The arguments to the **.format( )** method are of two types. The list starts with zero or more positional arguments p_i, followed by zero or more keyword arguments of the form $k_i v_i$ where each k_i is a name with an associated value v_i.

Consider following example. In this example, the format code "{0}" is replaced by the first positional argument (49), and "{1}" is replaced by the second positional argument, the string "okra"

These are place holders

Values tuple V. Values are substituted from here

```
"We have {0} hectares planted to {1}." .format (49, "okra")
```

The above string template will yield following string

```
'We have 49 hectares planted to okra.
```

The values are picked from the values tuple V from left to right, *i.e.*, the first value of tuple will be placed in the first { } placeholder, second value in the second { } placeholder, and so on.

You may even skip the numbers 0, 1 in { }. That is, following string will also yield the same string as above.

```
"We have {} hectares planted to {} ." .format (49, "okra")
```

In the next example, we supply the values using **named arguments**. The arguments may be supplied in any order. The keyword names must be valid Python names, *e.g.*,

Values tuple V. Values are substituted from here

```
"{monster} has eaten {city}".format(city ='Tokyo', monster ='Tsunami')
```

In the above string template **{monster}** and **{city}** are named arguments. Named arguments will replace the value of the arguments as per the names not as per the order. So even though in values tuple above, the first value is **city ='Tokyo'**, it will not replace the first placeholder**{monster}**, rather it will replace the placeholder having its name *i.e.*, **{city}**. So the string generated will be :

```
'Tsunami has eaten Tokyo
```

Based on above information, you can create your SQL query strings. Again, make sure to enclose a string value place holder in quotes, *i.e.*, as '{ },]. Now consider following example :

```
st = "select * from student where marks > {} and section = '{}' ".format(70,'B')
```

Place holder enclosed in quotes for string value

The above query string **st** stores :

```
"select * from student where marks > 70 and section = 'B' "
```

And it will yield the same query result as earlier.

Common Cursor Methods

execute(operation [, params]) – Executes an SQL statement.

fetchone( ) – Fetches the next row as a sequence ; returns **None** when no more data

fetchall( ) – Fetches all remaining rows, current pointer position onwards

fetchmany([size = cursor.arraysize]) – Fetches the next set of rows as a sequence of sequences (default is set using *cursor.arraysize*); returns an empty sequence when no more rows to fetch.

16.4 PERFORMING INSERT AND UPDATE QUERIES

Since INSERT and UPDATE are also SQL commands, you can execute them just the way you have executed SELECT queries earlier. That is, store the query in a string as a simple query or as a parameterised query and execute it using **cursor.execute( )** method.

BUT one important thing you must ensure with INSERT and UPDATE queries (which make changes to the database unlike SELECT) you must **commit** your query after executing INSERT and UPDATE queries. For this your must run **commit( )** method of connection in the end *i.e.*

```
<connection object>.commit( )
```

This is done to reflect the change in the database physically.

Consider following examples (assuming that connection has been established with **student** table of **test** database as earlier ; the connection object is **mycon** and the cursor object name is **cursor**)

(*i*) INSERT query example

```
st = "INSERT INTO student(rollno, name, marks, grade, section)
     VALUES({}, '{}', {}, '{}', '{}')".format(108, 'Eka', 84.0, 'A', 'B')
cursor.execute(st)
mycon.commit()
```

You can run a SELECT query to check if your data has been added to the table or not.

(*ii*) UPDATE query example

```
st = "UPDATE student SET marks = {}
     WHERE marks = {}".format(77, 76.8)
cursor.execute(st)
mycon.commit()
```

In the same manner, you can also run DELETE queries, but DO NOT forget to **commit( )** after running the query. (Refer to solved problem 10 that executes a DELETE query)

With this we have come to the end of this chapter. Let us quickly revise that we have learnt so far.

Check Point 11.1

1. How is database connectivity useful ?
2. What is a connection ?
3. What is a result set ?
4. What is the package used for creating a Python database connectivity application.
5. Which function/method do you use for establishing connection to database ?
6. Which function/method do you use for executing an SQL query ?
7. Which method do you use to fetch records from the resultset ?

NOTE

Please note, you need to run **commit( )** with connection object for queries that change the data of the database table so that changes are reflected in the database.

LET US REVISE

- *To connect to a database from within a programming application, you need a framework that facilitates communication between two different generes of software (programming application and DBMS).*
- *To connect from Python to MySQL, you need a library called* ***mysql connector.***
- *You must import* ***mysql.connector*** *in the Python program/script before writing code for connectivity.*
- *Steps to create a database connectivity Python application are :*

  Step 1 *Start Python.* **Step 2** *Import the packages required for database programming.*

  Step 3 *Open a connection.* **Step 4** *Create a cursor instance.*

  Step 5 *Execute a query.* **Step 6** *Extract data from result set.* **Step 7** *Clean up the environment.*
- *A database Connection object controls the connection to the database. It represents a unique session with a database connected from within a script/program.*
- *A Database Cursor is a special control structure that facilitates the row by row processing of records in the resultset, i.e., the set of records retrieved as per query.*
- *The result set refers to a logical set of records that are fetched from the database by executing an SQL query and made available to the application program.*
- *You can use* ***connect( )*** *method for establishing database connection,* ***cursor( )*** *to create a cursor and* ***execute( )*** *to execute an SQL query.*
- *To fetch records from a result set, you can use fetchone( ), fetchmany( ) and fetchall( ) methods.*
- *For INSERT, UPDATE and DELETE queries, you must run commit( ) with the connection object.*

Solved Problems

1. *What is database connectivity ?*

   Solution. Database connectivity refers to connection and communication between an application and a database system.

2. *What is Connection ? What is its role ?*

   Solution. A Connection (represented through a connection object) is the session between the application program and the database. To do anything with database, one must have a connection object.

3. *What is a result set ?*

   Solution. A **result set** refers to a logical set of records that are fetched from the database by executing a query and made available to the application-program.

4. *Which package must be imported in Python to create a database connectivity application?*

   Solution. There are multiple packages available through which database connectivity applications can be created in Python. One such package is **mysql.connector**.

5. *What will be the generated query string ?*

```
query = "INSERT INTO books(title, isbn) VALUES(%s, %s)".% ('Ushakaal','12678987036')
```

   Solution. `"INSERT INTO books(title, isbn) VALUES('Ushakaal','12678987036')"`

6. *Which record will get inserted in the table by following code?*

```
import mysql.connector as sqltor
mycon = sqltor.connect(host = "localhost", user = "learner", passwd = "fast", database="test")
cursor = mycon.cursor()
```

```
query = "INSERT INTO books(title, isbn) VALUES(%s, %s)".% ('Ushakaal','12678987036')
cursor.execute(query)
mycon.commit()
```

Solution. 'Ushakaal',12678987036

7. *What will be the generated query string ?*

```
query = "INSERT INTO books(title, isbn) VALUES('{}', {})". format('Ushakiran','42568987036')
```

Solution. "INSERT INTO books(title, isbn) VALUES(('Ushakiran','42568987036')"

8. *Which record will get inserted in the table by following code :*

```
import mysql.connector as sqltor
mycon = sqltor.connect(host = "localhost", user = "learner", passwd = "fast", database ="test")
cursor = mycon.cursor()
query = "INSERT INTO books(title, isbn) VALUES('{}', {})".format('Ushakiran','42568987036')
cursor.execute(query)
mycon.commit()
```

Solution. 'Ushakiran', 42568987036

9. *The books table of test database contains the records shown here.*

Title	ISBN
Die to Live	78127873915
Again?	23686286243
Ushakaal	12678987036
Ushakiran'	42568987036

What will be the output produced by following code :

```
import mysql.connector as sqltor
conn = sqltor.connect(host = "localhost", user = "learner", passwd = "fast", database ="test")
cursor = conn.cursor()
cursor.execute("SELECT * FROM books")
row = cursor.fetchone()
while row is not None:
    print(row)
    row = cursor.fetchone()
```

Solution.

```
(Die to Live 78127873915)
(Again?      23686286243)
(Ushakaal    12678987036)
(Ushakiran   42568987036)
```

10. *Write a Python database connectivity script that deletes records from* **category** *table of database* **items** *that have name = 'Stockable'*

Solution.

```
import mysql.connector as ctor
dbcon = ctor.connect(host = "localhost", user = "learner", passwd = "fast",
database = "items")
cursor = dbcon.cursor()
sql1 = "DELETE FROM category WHERE name= '%s' "
data1 = ('Stockable',)
cursor.execute(sql1, data1)
dbcon.commit()   # commit the changes
print("Rows affected:", cursor.rowcount)
dbcon.close()
```

GLOSSARY

Connection	Session between the application program and the database.
Result Set	A logical set of records fetched from database based on query and made available to the application

Assignment

Type A : Short Answer Questions/Conceptual Questions

1. What are the steps to connect to a database from with Java application ?
2. Write code to connect to a MySQL database namely **School** and then fetch all those records from table **Student** where *grade* is *'A'*.
3. Predict the output of the following code :

```
import mysql.connector
db = mysql.connector.connect(.....)
cursor = db.cursor()
sql1 = "update category set name= '%s' WHERE ID= %s" % ('CSS',2)
cursor.execute(sql1)
db.commit()
print("Rows affected:", cursor.rowcount)
db.close()
```

4. Explain what the following query will do?

```
import mysql.connector
db = mysql.connector.connect(.....)
cursor = db.cursor()
person_id = input("Enter required person id")
lastname = input("Enter required lastname")
db.execute("INSERT INTO staff (person_id, lastname) VALUES ({}, '{}') ".
format(person_id, lastname))
db.commit()
db.close()
```

5. Explain what the following query will do?

```
import mysql.connector
db = mysql.connector.connect(.....)
cursor = db.cursor()
db.execute("SELECT * FROM staff WHERE person_id in {}".format((1,3,4)))
db.commit()
db.close()
```

Type B : Application Based Questions

1. Design a Python application that fetches all the records from **Pet** table of **menagerie** database.
2. Design a Python application that fetches only those records from **Event** table of **menagerie** database where *type* is **Kennel**.
3. Design a Python application to obtain a search criteria from user and then fetch records based on that from **empl** table. (given in chapter 13)