Unit 5 – Python Programming to Access MariaDB Databases

The topic of this unit is programming a user interface to interact with MariaDB databases. The textbook contains a full chapter devoted to this topic, including the use of Python Application Programming Interfaces (API). Unfortunately this information may be outdated, therefore, the purpose of this document is to fill that void. Please disregard the instructions in the textbook and follow the ones in this document.

Students taking this course are expected to have programming experience, either in Java or Python. There is another document that shows how to program the access to a database using Java. In this document will attempt to explain how to write Python programs that interact with MariaDB databases, in as much detail as needed for students to perform this task.

Students are provided with the following sample Python programs in this unit:

- CreateTable.py
- PopulateTable.py
- AlterTable.py
- SelectStatement.py
- SelectStatement2.py

These programs may be downloaded from the Canvas environment and be used to follow the explanations in this document.

General Python Structures

Python programs are written as text files (also known as ASCII files) with the file extension **.py**. They may contain the following sections in order:

1. An introductory section were all the language extensions are attached to the program (also known as the import section),
2. The definition of various user created functions, and
3. The main program.

In this course we will only need two sections to write programs: the import section and the main program. To write the program and test it, Python provides with a program writing environment named IDLE. Students may open this environment and load the example Python programs that were provided. Once the program is in IDLE, it can be compiled and executed. This is achieved by pressing the **F5** button in IDLE. The compilation process will translate the program written in Python into machine code that the computer can understand and execute. A Python program will compile only if the statements inside the file are written in correct Python syntax. IDLE will compile and immediately execute the program line by line, but it will stop if a compilation or runtime error is detected.

Let us see an example of these ideas with a Python program. Let us use the beginning of the file named **CreateTable.py**. It will contain a Python program to create a table in a MariaDB database. In these examples, the database to be used is ***ForLecture05***, and it must be created with no tables, before any program is run. The beginning of the **CreateFile.py** file will contain the following text (without the line numbers):

```
 1: #  Program Name: CreateTable.py
 2: #   Author: Guillermo Tonsmann
 3: #   Description: This program will create the Customer Table
 4: #                in the MariaDB database named ForLecture05
 5:
 6: import mariadb
 7:
 8: userAux = input("Enter MariaDB user:")
 9: passwordAux = input("Enter password:")
```

Let us explain the meaning of these sentences so-far:

1. Lines 1 to 4 are comments. Comments are used to make remarks about a program. These remarks will not be executed and they are just notes for whoever is reading the program (like a colleague working in a same project, or your instructor reading your program for grading). We can put comments anywhere in the program after the hash symbol (***#***). The remaining characters within the same line will be considered comments and the compiler will not pay attention to them. At the minimum, students must write initial comments at the beginning of the file as shown in the example, so that the Instructor will be able to identify the program. The name of the program, the student's name as the author and a brief description of what the program does must be included in these initial comments, known as the header of the program.

2. Lines 6 is the import section. It tells the compiler that some libraries from the Python language will be used by the program and they should be loaded for the program to compile correctly. The library in line 6 (***mariadb***) is the one that contains all statements used by Python to interact with MariaDB databases, so it is a must for all our programs in this course.

3. Line 8 is where the main program is actually starting, and it will continue up to the end of the file. Lines 8 and 9 use the Python's input statement to request the user for the names of the MariaDB logonid that will have access to the MariaDB database and its password (usually **root** and **password,** respectively, in default installations). The program's user will be able to answer the questions prompted in the input statements and her/his answers will be stored in the variables ***userAux*** and ***passwordAux*** respectively. It is a good security practice to have these statements to request information to user about the access to the database, rather than writing this information in the program itself.

After these instructions for basic setup, we need to write Python statements to establish a connection to the database, create a SQL statement and execute that statement in the database. All these steps will be explained in the following sections.

Creating a Connection from Python to a MariaDB database

To create a connection to MariaDB from a Python program two things must happen in sequence:

1. Create a connection object.
2. Obtain a cursor from the connection object.

To create a connection object we use the following command:

```
10: conn = mariadb.connect(
11:     user = userAux,
12:     password = passwordAux,
13:     host="localhost",
14:     database="ForLecture05")
```

This is a single sentence that uses the ***mariadb.connect*** function to establish the connection. It takes 4 parameters, each one with its own name that needs to be included in the command. They should be separated from others with commas:

1. **`user:`** The name of the MariaDB logonid to access the database. This should be a string with the name of the login account that will be used to work with the database. We obtained this name from the program's user in the variable ***userAux*** from the input statement in line 8. Our Citrix Park Virtual Lab is working with the "***root***" user. Students working in their own PCs should know the name of the logon id for the user they installed in their environments (also "***root***" by default). Students must change the value of this variable if they are using a different login account. One could write the word "***root***" directly in the command instead of using the ***userAux*** variable. However, due to security reasons, it is best to obtain the value from the program's user rather than write the name directly in the command itself.

2. **`password:`** This is a string that contains the password for the login account that will be used to work with the database. The password we are using in our Citrix Park Virtual Lab is also the word "**password**". Students working in their own PCs should know their own password. However, as with the logon id, it is not a good practice to code the actual password directly into a program. It is best to ask the user of the program for a password. We did this in line 9 where we obtained the value from the user in the ***passwordAux*** variable. This variable is used in the sentence above.

3. **`host:`** The host of the database (usually the ***localhost*** for systems installed in the same computer). and

4. **`database`**: A string with the name of the database to be used. In the example above this database is ***ForLecture05***. This database has to be previously created in MariaDB to be able to access it.

All these parameters should be provided as shown for the function to create a connection to the database. The link to this connection will be stored in the variable ***conn*** that will be used when we need to access the database.

Notice this is a long command. Whenever you would like to break a long Python sentence into various lines, you will need to surround the block to be broken with parentheses. The ***connect*** command already uses parentheses, so the break up appears very natural. If the command does not have parentheses, they will need to be added before the break up.

To obtain a cursor from the connection object we use the following command:

```
15: cur = conn.cursor()
```

This line obtains a cursor object (***cur***) from the connection. A cursor will be used to send commands to the database in the connection and to receive results from the database.

Creating and Executing SQL Statements

Once the connection is established, we can create an SQL statement inside a string variable as shown in the following lines:

```
16: stmt = ("create table Customer "+
17: "(CustomerID INT PRIMARY KEY, "+
18: "CustomerName TEXT,"+
19: "CustomerAddress TEXT, "+
20: "State TEXT)")
21: # Notice there is no semicolon at the end of the SQL command
```

These lines declare a string ***stmt*** variable containing a create-SQL command. Because this command is too long to be seen properly at a glance in one single line, it has been broken up along many lines. Each line contains a consecutive piece of the command and all pieces are put together in the Python command using the plus sign at the end of each line in between. These plus symbols are called the **concatenation operators**. Concatenation is an operation for strings that put them together side by side. Each piece begins and end with double quotes. As before, to indicate that the command continues for various lines, all the pieces were surrounded by parentheses. The command only needs the external parentheses that surrounds the SQL command to achieve the breaking up of the statement. They will not be needed if the whole statement were written in one line. Notice also that there is no semicolon at the end of the SQL command itself, inside the quotes, as indicated by the comment in line 21. The connection will add the semicolon on its own when executing the command, so there is no need to add it ourselves. The SQL command will create a table named Customer with some fields and no records.

To execute this statement we can use the following command:

```
    cur.execute(stmt)
```

This command will execute the statement in the database and return any results in the cursor named ***cur***. However, because this command may not perform as expected in the database for many reasons (maybe the SQL command was not properly written or the DBMS cannot satisfy the request), it would be good to surround these statements with a try/except structure also as shown below:

```
try:
    cur.execute(stmt)
except mariadb.Error as e:
    print(f"Error: {e}")
```

If the program receives an error when trying to execute the statement in the database, it will print the error and the program will continue with the lines after this section. No indication will be given if the statement was run without an error.

The last sentence of the program will close the connection:

```
conn.close()
```

This is just a good programming technique that avoids keeping resources opened after a program ends.

The complete program may look something like this (highlighted portions are the ones that students may change/update when writing programs to create other tables):

```
#  Program Name: CreateTable.py
#   Author: Guillermo Tonsmann
#   Description: This program will create the Customer Table
#                in the MariaDB database named ForLecture05

import mariadb

userAux = input("Enter MariaDB user:")
passwordAux = input("Enter password:")
conn = mariadb.connect(
    user = userAux,
    password = passwordAux,
    host="localhost",
    database="ForLecture05")
cur = conn.cursor()
stmt = ("create table Customer "+
"(CustomerID INT PRIMARY KEY, "+
"CustomerName TEXT,"+
"CustomerAddress TEXT, "+
"State TEXT)")
# Notice there is no semicolon at the end of the SQL command
try:
    cur.execute(stmt)
except mariadb.Error as e:
    print(f"Error: {e}")
conn.close()/*
stmt = "insert into Customer (CustomerID, CustomerName,
CustomerAddress, State)
```

Other embedded SQL commands

All Python programs accessing MariaDB datbases will have similar programming structures as the ones shown in the '**CreateTable.py**' program above. For example, once the table is constructed, we can populate it with records using another similar program. With this in mind, the program '**PopulateTable.py**' creates the following statement:

```
stmt = ("insert into Customer (CustomerID, CustomerName, " +
        "CustomerAddress, State) values (?,?,?,?)")
```

This statement will insert values into the four indicated fields of the Customer table, but the values are not yet defined. The question mark symbols are used as placeholders to be filled later with actual values.

The program also has a list of lists where the four values for these fields per record are stored, as follows:

```
records = [
    [5101,"John Smith","PO 2335 TX 78799","Texas"],
    [5102,"Quincey Adams","PO 4337 DC 20010","Washington DC"],
    [5103,"Jay Leno","PO 2675 CA 90210","California"],
    [5104,"Barry Manilow","PO 3726 CA 94205","California"],
    [5105,"Jerry Seinfeld","PO 8788 NY 10047","New York"],
    [5106,"Salem Municipal Library","PO 7 WA 98008","Washington"],
    [5107,"Clark Kent","PO 10101 NY 10007","New York"],
    [5108,"Tom Sawyer","PO 46213 MS 39532","Mississippi"],
    [5109,"Alejandro Toledo","PO 85391 CA 94204","California"],
    [5110,"Ming Zhao","PO 45667 OR 97202","Oregon"]
    ]
```

Each list is created with the square brackets ([ ]) and each one contains the four values for the fields to be added in the correct format (text fields surrounded by quotes, and numbers without them) and separated by commas. All of these lists are further surrounded by a square bracket that creates the list of lists named ***records***. Notice that we do not need to add parentheses to break this sentence between many lines because the square bracket fulfills the same purpose.

Having the statement and its values, we can execute this command. Notice that the command inserts only one record, but we have several records to add in the records variable. We need to use a for-statement to run the command as many times as needed with the correct parameters. This is achieved with the following:

```
        for r in records:
            cur.execute(stmt,(r[0], r[1], r[2], r[3]))
```

The sentence will take each list inside records in turn and execute the command with the variables provided. For example, the first time the for-statement runs, it will use the following list in place of ***r***:

```
      r = [5101,"John Smith","PO 2335 TX 78799","Texas"]
```

Where `r[0]= 5101, r[1] = "John Smith", r[2]= "PO 2335 TX 78799",and r[3] - "Texas"`. These values will replace the questions marks in the ***stmt***, in the same order before the statement is executed in the database.

As before, the for-statement should be included between a try/except structure as follows:

```
     try:
         for r in records:
             cur.execute(stmt,(r[0], r[1], r[2], r[3]))
     except mariadb.Error as e:
         print[f"Error: {e}"]
```

Even though the previous commands may have been executed in the computer's memory, they will not modify the actual database unless we explicitly commit them. Therefore, the last sentence before the connection is closed should be:

```
        conn.commit()
```

This command will perform the actual modification of the records on the database permanently.

The complete program "**PopulateTable.py**" uses the previous sentences to populate the Customer Table with records, assuming the table was previously created with the program "**CreateTable.py**".

Once the Customer table is filled with records, the program "**AlterTable.py**" shows how to use a Python program to alter the table and update values of some fields. This is done with the following statements that contain embedded SQL commands as strings:

```
    alterStmt = "alter table Customer add column phone TEXT"
    update1Stmt = "update Customer set phone = ? "
    update2Stmt = "update Customer set state = ? where state = ? "
```

As before, the update statements contain questions marks as placeholders for values. These values can be obtained from the user, or as in the case of this program, stored in string variables as follows:

```
    phone = "999-999"
    stateIn = "Colorado"
    stateOut = "California"
```

The statements can be executed with the as following try/except and commit statements:

```
  try:
      cur.execute(alterStmt)
      cur.execute(update1Stmt,(phone,))
      cur.execute(update2Stmt,(stateIn, stateOut))
  except mariadb.Error as e:
      print[f"Error: {e}"]
  conn.commit()
```

As it can be seen, one can have multiple SQL statements executed consecutively in the same Python program and also within the same try/except statement.

Handling Queries

Once the steps to perform SQL commands in MariaDB from Python are understood, performing queries will require only a slight addition of these procedures. We still need to establish a connection from Python to Maria DBMS, create the query statement as a string and execute it, as the following statements from the '**SelectStatement.py**' program show:

```
      userAux = input("Enter MariaDB user:")
      passwordAux = input("Enter password:")
      conn = mariadb.connect(
          user = userAux,
          password = passwordAux,
          host="localhost",
          database="forLecture05")
      cur = conn.cursor()
      selectStmt = "select * from Customer"
      cur.execute(selectStmt)
```

The cursor now contains the results of the query. To retrieve this information we may use a for-statement that pulls the results record by record, as follows:

```
for CustomerID, CustomerName, CustomerAddress, State, phone in cur:
   print(f"CustomerID:{CustomerID},Customer Name: {CustomerName},"+
         f"Address:{CustomerAddress}State: {State} Phone: {phone}")
```

This for-statement expects four values for every record from the query. This is something the programmer should know, the number and types of the values that will be obtained from the query. The for-statement assigns each of these values a variable with name (***CustomerID***, ***CustomerName***, ***CustomerAddress***, and ***State*** in this example). These names are independent of any names the fields may have had in the database. At every pass of the for-statement a new set of values will be obtained until all records retrieved are read.

Within the for-statement we are free to use the new variables for whichever purpose we have in our program. In our case, we are printing these values preceded by a label. For example, the label "**Customer Name**" precedes the value obtained for the variable ***CustomerName***. The curly brackets around {CustomerName} indicate the place where the value is going to be printed. Notice that for this notation to work, each individual piece of the parameter to be printed must begin with the letter f, followed by the string. The first record obtained in this for-statement will look as follows:

```
CustomerID: 5101, Customer Name: John Smith, Address: PO 2335
TX 78799 State: Texas Phone: 999-999
```

The output produced by the "**SelectStatement.py**" program is very simple. More sophisticated reports can be constructed if we obtain more information of the results of the query from the cursor. The "**SelectStatement2.py**" program shows some of these capabilities. For example, the number of columns read in a query can be obtained from the cursor variable (***cur***) with the following command:

```
        numberOfColumns = cur.fieldcount()
```

The variable ***cur.description*** also contains information about each of the columns it retrieves. This is a list of lists. It has a list for each of the fields and in that list it stores the name, type and size of the field. For example, the following piece of code retrieves the names of all the retrieved fields from a query and display them as headers of size ***cSize*** and separated by the '**|**' symbol:

```
    cSize = 15
    nDashes = 0
    for field in cur.description:
            h = field[0]
            if cSize < len(h):
                    h = h[:cSize]
            print (h.center(cSize), ' | ', end='')
            nDashes += cSize+4
```

The for-statement goes to every field in cur-description and retrieves its name from the first position of the list (***field[0]***). The variable ***nDashes*** counts the length of the header to create an underlining with the following sentence:

```
        print('-'*nDashes)
```

A similar for-statement can later be used to print each of the records properly aligned under these headers. We can extract all records in the cursor with the following command:

```
rows= cur.fetchall()
```

rows is a list of records, so to display each one of them we need a for-statement to navigate thru the records and inside another for-statement to print the various columns. This will be something like the following:

```
for r in rows:
   for value in r:
         v = value
         if type(v)== str:
               if cSize < len(v):
                     v = v[:cSize]
               print(v.rjust(cSize), ' | ', end='')
         elif type(v) == int:
               f="{:"+str(cSize)+"d}"
               print(f.format(v), ' | ', end='')
         elif type(v) == float:
               f="{:"+str(cSize)+".f}"
               print(f.format(v), ' | ', end='')
   print()
```

The code inside the for-statements decides how to print each value according to its type (string, integer or decimal number) with a maximum size of ***cSize***.

Guillermo Tonsmann Ph.D.
July 2021