MySQL Connector/Python & ER

1 MySQL Connector/Python

This lab discusses how to use MySQL from within the context of a general-purpose programming language (Python). It covers basic application programming interface (API) operations: connecting to the MySQL server, executing statements, and retrieving the results.
There are many databases and many Python libraries for accessing databases. The **Python Database API Specification v2.0 (PEP 249)** is a community effort to unify the model of accessing different database systems. It defines:

- Module interface, connection objects, cursor objects, type objects and constructors
- It doesn't map database objects to Python structures in any way (as Pandas does)
  - Users are required to write SQL

We are going to use MySQL Connector/Python that enables Python programs to access MySQL databases, using an API that is compliant with PEP 249. Follow the next steps to obtain MySQL Connector/Python and test your connection to the database.

1. Activate your environment using the following commands. Ignore a possible warning of the first command "CondaValueError: prefix already exists:".

```
conda create -n DS_Py27 python=2.7
conda activate DS_Py27
```

2. Use Anaconda to install the MySQL Connector/Python

```
conda install -c anaconda mysql-connector-python=2.0.3
```

3. Open your favourite editor (e.g., Spyder) and write a simple program that connects to your database:

```
import mysql.connector
from mysql.connector import errorcode
try:
        conn = mysql.connector.connect(user='yourUserName', password='yourPassword',
                                       host='databaseServer',
                                       database='yourDatabase',port= '33306')
    print("Connected")
except:
    print("Cannot connect to server")
else:
    conn.close()
    print("Disconnected")
```

   Remember to replace the connection information with your own information.

   The import line tells Python to load the `mysql.connector module`. Then the script attempts to establish a connection to the MySQL server by calling `connect()` to obtain a connection object. Access to the database is made available through these connection objects. They allow: to create connections with the database, to close the connection, and to get cursor objects, etc.

4. Try the program

   If the `connect()` method fails, Connector/Python raises an exception. To handle exceptions, put the statements that might fail inside a try statement and use an except clause that contains the error-handling code. Exceptions that occur at the top level of a script (that is, outside of any try statement) are caught by the default exception handler, which prints a stack trace and exits.

1.1 Checking for Errors

Python signals errors by raising exceptions, and Python programs handle errors by catching exceptions in the except clause of a try statement. To obtain MySQL-specific error information, name an exception class, and provide a variable to receive the information. Here is an example:

```
import mysql.connector
from mysql.connector import errorcode

conn_params = {
        "database": "database",
        "host": "server",
        "user": "dbuser",
        "password": "dbpassword",
        "port":"33306 "
}
try:
        conn = mysql.connector.connect(**conn_params)
        print("Connected")
except mysql.connector.Error as e:
        print("Cannot connect to server")
        print("Error code: %s" % e.errno)
        print("Error message: %s" % e.msg)
        print("Error SQLSTATE: %s" % e.sqlstate)
```

Note that another way to connect is to specify the parameters using a Python dictionary and pass the dictionary to `connect()`.
If an exception occurs, the `errno`, `msg`, and `sqlstate` members of the exception object contain the error number, error message, and SQLSTATE values, respectively. Note that access to the Error class is through the driver module name.

1. Try the program above by providing incorrect usernames, database names, etc. and see what error codes are generated.

2. You can check for common error codes in your programs as follows:

```
try:
  conn = mysql.connector.connect(**conn_params)
except mysql.connector.Error as err:
  if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
    print("Wrong user name or password")
  elif err.errno == errorcode.ER_BAD_DB_ERROR:
    print("Database does not exist")
  else:
    print(err)
else:
  conn.close()
```

3. Modify your database connection program to check for most common errors using the code above.

1.2 Executing Statements and Retrieving Results

Note: Note for this section you need the limbs table we created last week in the NMS database. If you don't have it in the new database, please create and populate the limbs table as indicated in last week's lab.

The Python DB API uses the same calls for SQL statements that do not return a result set and those that do. To process a statement in Python, use your database connection object to get a cursor object. Cursor objects manage the context of a fetch operation. They allow to perform queries and other operations using SQL. To obtain a cursor use the `cursor()` method on the connection object. Then use the cursor's `execute()` method to send the statement to the server. If the statement fails with an error, `execute()` raises an exception. Otherwise, if there is no result set, statement execution is complete, and the cursor's rowcount attribute indicates how many rows were retrieved or affected by the query.

1. Try the code below:

```
import mysql.connector
from mysql.connector import errorcode
try:
  conn = mysql.connector.connect(user='yourUserName', password='yourPassword',
                                 host='databaseServer',
                                 database='yourDatabase',port= '33306')
  cursor = conn.cursor()
  cursor.execute("SELECT * FROM limbs")
  rows = cursor.fetchall()
  print('Total Row(s):', cursor.rowcount)
  for row in rows:
     print(row)

except mysql.connector.Error as e:
  print(e)

finally:
  cursor.close()
  conn.close()
```

Result sets from SQL queries can be very large, and there are different methods to retrieve items from the result set:

- `fetchone()` retrieves a single item
- `fetchall()` retrieves all the items
  - Use it only when you know the result set contains a limited number of rows that can fit into memory
- `fetchmany(n)` retrieves the next number of rows (`n`) of the result set
  - Use it when you cannot predict the size of the result set

2. Try the code below using `fetchone()`:

```
cursor = conn.cursor()
cursor.execute("SELECT thing, legs, arms FROM limbs")
while True:
        row = cursor.fetchone()
        if row is None:
                break
        print("Thing: %s, legs: %s, arms: %s" % (row[0], row[1], row[2]))
print("Number of rows returned: %d" % cursor.rowcount)
cursor.close()
```

The `fetchone()` method returns the next row as a sequence, or None when there are no more rows

3. Alternatively, you can use the cursor itself as an iterator that returns each row in turn:

```
cursor = conn.cursor()
cursor.execute("SELECT thing, legs, arms FROM limbs")
for (thing, legs, arms) in cursor:
        print("Thing: %s, legs: %s, arms: %s" % (thing, legs, arms))
print("Number of rows returned: %d" % cursor.rowcount)
cursor.close()
```

4. The fetchall() method returns the entire result set as a sequence of row sequences. Iterate through the sequence to access the rows:

```
cursor = conn.cursor()
cursor.execute("SELECT thing, legs, arms FROM limbs")
rows = cursor.fetchall()
for row in rows:
        print("Thing: %s, legs: %s, arms: %s" % (row[0], row[1], row[2]))
print("Number of rows returned: %d" % cursor.rowcount)
cursor.close()
```

DB API provides no way to rewind a result set, so `fetchall()` can be convenient when you must iterate through the rows of the result set more than once or access individual values directly. For example, if `rows` holds the result set, you can access the value of the third column in the second row as `rows[1][2]` (indexes begin at 0, not 1).

5. As described above `fetchall()` cannot be suitable for large databases, use `fetchmany` instead to retrieve the information in the limbs table in chunks of 4 rows.

1.3 Handling Special Characters and NULL Values in Statements

The Connector/Python module implements placeholders using %s format specifiers in the SQL statement string. To use `placeholders`, invoke the `execute()` method with two arguments:

- a statement string containing format specifiers
- a list containing the values to bind to the statement string. Use None to bind a NULL value to a placeholder.

1. use the following code to retrieve information about things with 2 legs:

```
cursor = conn.cursor()
cursor.execute("SELECT thing, legs, arms FROM limbs WHERE arms = %s", [2])
for (thing, legs, arms) in cursor:
        print("thing: %s, leg: %s, arms: %s" % (thing, legs, arms))
cursor.close()
```

   The statement sent to the server by the preceding execute() call looks like this:
   SELECT thing, legs, arms FROM limbs WHERE arms = 2

2. Modify the previous code to retrieve information about things with 2 arms and 1 leg.
3. The Connector/Python placeholder mechanism provides quotes around data values as necessary when they are bound to the statement string, so don't put quotes around the %s format specifiers in the string. Modify the previous code to retrieve information about humans.

1.4 Identifying NULL Values in Result Sets

Python DB API programs represent `NULL` in result sets using None. The following example shows how to detect `NULL` values:

```
cursor = conn.cursor()
cursor.execute("SELECT thing,arms,legs FROM limbs")
for row in cursor:
    row = list(row) # convert nonmutable tuple to mutable list
    for i, value in enumerate(row):
        if value is None: # is the column value NULL?
            row[i] = "NULL"
    print("Thing: %s, arms: %s, legs: %s" % (row[0], row[1], row[2]))
cursor.close()
```

The inner loop checks for `NULL` column values by looking for None and converts them to the string `"NULL"`. The example converts row to a mutable object prior to the loop because `fetchall()` returns rows as sequence values, which are nonmutable (read only).

2 Drawing ER Models

Create a complete ERD in Crow's Foot notation that can be implemented in the relational model using the following description.

Each region has a name and code. Each store has a code and an address. One region can be the location for many stores. Each store is located in only one region. Each store sells several products and each product can be sold in several stories. Each product has a code and price and may have a description.

Use a diagramming program such as Microsoft Visio, OneNote, Powerpoint, Illustrator etc... for you ER diagram. You can also use web apps such as http://draw.io to help you sketch out your diagrams. Hand-drawn diagrams are usually unreadable!

3 Personal Study

1. Please complete the Interim Module Evaluation Form:

   `https://keats.kcl.ac.uk/mod/feedback/view.php?id=1964111`

2. Have a look at the tutorials contained here:

   `https://dev.mysql.com/doc/connector-python/en/connector-python-tutorials.html`