Home

MySQL Python tutorial

This is a Python programming tutorial for the MySQL database. It covers the basics of MySQL programming with Python. It uses the `MySQLdb` module. The examples were created and tested on Ubuntu Linux.

There is a similar PostgreSQL Python tutorial, MySQL Visual Basic tutorial, or MySQL PHP tutorial on ZetCode. If you need to refresh your knowledge of the Python language, there is a full Python tutorial. You may also consider to look at the MySQL tutorial, too.

About MySQL database

MySQL is a leading open source database management system. It is a multi user, multithreaded database management system. MySQL is especially popular on the web. It is one part of the very popular *LAMP* platform which consists of Linux, Apache, MySQL, and PHP. Currently MySQL is owned by Oracle. MySQL database is available on most important OS platforms. It runs on BSD Unix, Linux, Windows, or Mac OS. Wikipedia and YouTube use MySQL. These sites manage millions of queries each day. MySQL comes in two versions: MySQL server system and MySQL embedded system.

Before we start

We need to install several packages to execute the examples in this tutorial.

If you do not already have MySQL installed, we must install it.

```
$ sudo apt-get install mysql-server
```

This command installs the MySQL server and various other packages. While installing the package, we are prompted to enter a password for the MySQL root account.

```
$ apt-cache search MySQLdb
python-mysqldb - A Python interface to MySQL
python-mysqldb-dbg - A Python interface to MySQL (debug extension)
bibus - bibliographic database
eikazo - graphical frontend for SANE designed for mass-scanning
```

We don not know the package name for the MySQLdb module. We use the `apt-cache` command to figure it out.

```
$ sudo apt-get install python-mysqldb
```

Here we install the Python interface to the MySQL database. Both `_mysql` and `MySQL` modules.

Next, we are going to create a new database user and a new database. We use the `mysql` client.

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 30
Server version: 5.0.67-0ubuntu6 (Ubuntu)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
+--------------------+
2 rows in set (0.00 sec)
```

We connect to the database using the root account. We show all available databases with the `SHOW DATABASES` statement.

```
mysql> CREATE DATABASE testdb;
Query OK, 1 row affected (0.02 sec)
```

We create a new `testdb` database. We will use this database throughout the tutorial.

```
mysql> CREATE USER 'testuser'@'localhost' IDENTIFIED BY 'test623';
Query OK, 0 rows affected (0.00 sec)

mysql> USE testdb;
Database changed

mysql> GRANT ALL ON testdb.* TO 'testuser'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> quit;
Bye
```

We create a new database user. We grant all privileges to this user for all tables of the `testdb` database.

_mysql module

The `_mysql` module implements the MySQL C API directly. It is not compatible with the Python DB API interface. Generally, the programmers prefer the object oriented `MySQLdb` module. We will concern ourself with the latter module. Here we present only one small example with the `_mysql` module.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import _mysql
import sys

try:
    con = _mysql.connect('localhost', 'testuser', 'test623', 'testdb')

    con.query("SELECT VERSION()")
    result = con.use_result()

    print "MySQL version: %s" % \
        result.fetch_row()[0]

except _mysql.Error, e:

    print "Error %d: %s" % (e.args[0], e.args[1])
    sys.exit(1)

finally:

    if con:
        con.close()
```

The example will get and print the version of the MySQL database. For this, we use the `SELECT VERSION()` SQL statement.

MySQLdb module

`MySQLdb` is a thin Python wrapper around `_mysql`. It is compatible with the Python DB API, which makes the code more portable. Using this model is the preferred way of working with the MySQL.

First example

In the first example, we will get the version of the MySQL database.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb
```

```
import sys

try:
    con = mdb.connect('localhost', 'testuser', 'test623', 'testdb');

    cur = con.cursor()
    cur.execute("SELECT VERSION()")

    ver = cur.fetchone()

    print "Database version : %s " % ver

except mdb.Error, e:

    print "Error %d: %s" % (e.args[0],e.args[1])
    sys.exit(1)

finally:

    if con:
        con.close()
```

In this script, we connect to the testdb database and execute the `SELECT VERSION()` statement. This will return the current version of the MySQL database. We print it to the console.

```
import MySQLdb as mdb
```

We import the `MySQLdb` module.

```
con = mdb.connect('localhost', 'testuser',
    'test623', 'testdb');
```

We connect to the database. The `connect()` method has four parameters. The first parameter is the host, where the MySQL database is located. In our case it is a localhost, e.g. our computer. The second parameter is the database user name. It is followed by the user's account password. The final parameter is the database name.

```
cur = con.cursor()
cur.execute("SELECT VERSION()")
```

From the connection, we get the cursor object. The cursor is used to traverse the records from the result set. We call the `execute()` method of the cursor and execute the SQL statement.

```
ver = cur.fetchone()
```

We fetch the data. Since we retrieve only one record, we call the `fetchone()` method.

```
print "Database version : %s " % ver
```

We print the data that we have retrieved to the console.

```
except mdb.Error, e:

    print "Error %d: %s" % (e.args[0],e.args[1])
    sys.exit(1)
```

We check for errors. This is important, since working with databases is error prone.

```
finally:

    if con:
        con.close()
```

In the final step, we release the resources.

```
$ ./version.py
Database version : 5.5.9
```

The output might look like the above.

Creating and populating a table

We create a table and populate it with some data.

```
#!/usr/bin/python
```

```
# -*- coding: utf-8 -*-

import MySQLdb as mdb

con = mdb.connect('localhost', 'testuser', 'test623', 'testdb');

with con:

    cur = con.cursor()
    cur.execute("DROP TABLE IF EXISTS Writers")
    cur.execute("CREATE TABLE Writers(Id INT PRIMARY KEY AUTO_INCREMENT, \
                 Name VARCHAR(25))")
    cur.execute("INSERT INTO Writers(Name) VALUES('Jack London')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Honore de Balzac')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Lion Feuchtwanger')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Emile Zola')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Truman Capote')")
```

We create a `Writers` table and add five authors to it.

```
with con:
```

With the with keyword, the Python interpreter automatically releases the resources. It also provides error handling.

```
cur.execute("CREATE TABLE Writers(Id INT PRIMARY KEY AUTO_INCREMENT, \
                Name VARCHAR(25))")
```

This SQL statement creates a new database table called `Writers`. It has two columns: `Id` and `Name`.

```
cur.execute("INSERT INTO Writers(Name) VALUES('Jack London')")
cur.execute("INSERT INTO Writers(Name) VALUES('Honore de Balzac')")
...
```

We use the `INSERT` statement to insert authors to the table. Here we add two rows.

```
mysql> SELECT * FROM Writers;
+----+------------------+
| Id | Name             |
+----+------------------+
|  1 | Jack London      |
|  2 | Honore de Balzac |
|  3 | Lion Feuchtwanger |
|  4 | Emile Zola       |
|  5 | Truman Capote    |
+----+------------------+
5 rows in set (0.00 sec)
```

After executing the script, we use the `mysql` client tool to select all data from the Writers table.

Retrieving data

Now that we have inserted some data into the database, we want to get it back.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb

con = mdb.connect('localhost', 'testuser', 'test623', 'testdb');

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM Writers")

    rows = cur.fetchall()

    for row in rows:
        print row
```

In this example, we retrieve all data from the `Writers` table.

```
cur.execute("SELECT * FROM Writers")
```

This SQL statement selects all data from the Writers table.

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

The `fetchall()` method gets all records. It returns a result set. Technically, it is a tuple of tuples. Each of the inner tuples represent a row in the table.

```
for row in rows:
    print row
```

We print the data to the console, row by row.

```
$ ./retrieve.py
(1L, 'Jack London')
(2L, 'Honore de Balzac')
(3L, 'Lion Feuchtwanger')
(4L, 'Emile Zola')
(5L, 'Truman Capote')
```

This is the output of the example.

Returning all data at a time may not be feasible. We can fetch rows one by one.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb

con = mdb.connect('localhost', 'testuser', 'test623', 'testdb');

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM Writers")

    for i in range(cur.rowcount):

        row = cur.fetchone()
        print row[0], row[1]
```

We again print the data from the `Writers` table to the console. This time, we fetch the rows one by one.

```
for i in range(cur.rowcount):

    row = cur.fetchone()
    print row[0], row[1]
```

We fetch the rows one by one using the `fetchone()` method. The rowcount property gives the number of rows returned by the SQL statement.

```
$ ./retrieve2.py
1 Jack London
2 Honore de Balzac
3 Lion Feuchtwanger
4 Emile Zola
5 Truman Capote
```

Output of the example.

The dictionary cursor

There are multiple cursor types in the `MySQLdb` module. The default cursor returns the data in a tuple of tuples. When we use a dictionary cursor, the data is sent in a form of Python dictionaries. This way we can refer to the data by their column names.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb

con = mdb.connect('localhost', 'testuser', 'test623', 'testdb')
```

```
with con:

    cur = con.cursor(mdb.cursors.DictCursor)
    cur.execute("SELECT * FROM Writers LIMIT 4")

    rows = cur.fetchall()

    for row in rows:
        print row["Id"], row["Name"]
```

In this example, we get the first four rows of the `Writers` table using the dictionary cursor.

```
cur = con.cursor(mdb.cursors.DictCursor)
```

We use the `DictCursor` dictionary cursor.

```
cur.execute("SELECT * FROM Writers LIMIT 4")
```

We fetch four rows from the `Writers` table.

```
for row in rows:
    print row["Id"], row["Name"]
```

We refer to the data by column names of the `Writers` table.

```
$ ./dictcur.py
1 Jack London
2 Honore de Balzac
3 Lion Feuchtwanger
4 Emile Zola
```

Example output.

Column headers

Next we will show, how to print column headers with the data from the database table.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb

con = mdb.connect('localhost', 'testuser', 'test623', 'testdb')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM Writers LIMIT 5")

    rows = cur.fetchall()

    desc = cur.description

    print "%s %3s" % (desc[0][0], desc[1][0])

    for row in rows:
        print "%2s %3s" % row
```

Again, we print the contents of the `Writers` table to the console. Now, we include the names of the columns too. The column names are considered to be the 'meta data'. It is obtained from the cursor object.

```
desc = cur.description
```

A description attribute of the cursor returns information about each of the result columns of a query.

```
print "%s %3s" % (desc[0][0], desc[1][0])
```

Here we print and format the table column names.

```
for row in rows:
    print "%2s %3s" % row
```

And here, we traverse and print the data.

```
$ ./columnheaders.py
Id Name
 1 Jack London
 2 Honore de Balzac
 3 Lion Feuchtwanger
 4 Emile Zola
 5 Truman Capote
```

Ouput of the script.

Prepared statements

Now we will concern ourselves with prepared statements. When we write prepared statements, we use placeholders instead of directly writing the values into the statements. Prepared statements increase security and performance. The Python DB API specification suggests 5 different ways how to build prepared statements. The `MySQLdb` module supports one of them, the ANSI `printf` format codes.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb

con = mdb.connect('localhost', 'testuser', 'test623', 'testdb')

with con:

    cur = con.cursor()

    cur.execute("UPDATE Writers SET Name = %s WHERE Id = %s",
        ("Guy de Maupasant", "4"))

    print "Number of rows updated:",  cur.rowcount
```

We change the name of an author on the fourth row.

```
 cur.execute("UPDATE Writers SET Name = %s WHERE Id = %s",
     ("Guy de Maupasant", "4"))
```

We use the two placeholders identified by the `%s` markers. Before the SQL statement is executed, the values are bound to their placeholders.

```
$ ./prepared.py
Number of rows updated: 1
```

We have updated one row.

```
mysql> SELECT Name FROM Writers WHERE Id=4;
+------------------+
| Name             |
+------------------+
| Guy de Maupasant |
+------------------+
1 row in set (0.00 sec)
```

The author on the fourth row was successfully changed.

Inserting images

People often look for ways to insert images into databases. We will show how it can be done in SQLite and Python. Note that some people do not recommend to put images into databases. Images are binary data. MySQL database has a special data type to store binary data called `BLOB` (Binary Large Object). `TINYBLOB`, `BLOB`, `MEDIUMBLOB` and `LONGBLOB` are variants of the binary object type.

```
mysql> CREATE TABLE Images(Id INT PRIMARY KEY, Data MEDIUMBLOB);
Query OK, 0 rows affected (0.08 sec)
```

For this example, we create a new table called `Images`.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb


def read_image():

    fin = open("woman.jpg")
    img = fin.read()

    return img


con = mdb.connect('localhost', 'testuser', 'test623', 'testdb')

with con:

    cur = con.cursor()
    data = read_image()
    cur.execute("INSERT INTO Images VALUES(1, %s)", (data, ))
```

In the above script, we read a JPG image from the disk and insert it into the `Images` table.

```
def read_image():

    fin = open("woman.jpg")
    img = fin.read()

    return img
```

The `read_image()` method reads binary data from the JPG file, located in the current working directory.

```
cur.execute("INSERT INTO Images VALUES(1, %s)", (data, ))
```

We insert the image data into the `Images` table.

Reading images

In the previous example, we have inserted an image into the database table. Now we are going to read the image back from the table.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb

def writeImage(data):

    fout = open('woman2.jpg', 'wb')

    with fout:

        fout.write(data)

con = mdb.connect('localhost', 'testuser', 'test623', 'testdb')

with con:

    cur = con.cursor()

    cur.execute("SELECT Data FROM Images WHERE Id=1")
    data = cur.fetchone()[0]
    writeImage(data)
```

We read one image from the `Images` table.

```
cur.execute("SELECT Data FROM Images WHERE Id=1")
```

We select one record from the table.

```
fout = open('woman2.jpg', 'wb')
```

We open a writable binary file.

```
fout.write(data)
```

We write the data to the disk.

Now we should have an image called `woman2.jpg` in our current directory. We can check if it is the same image that we have inserted into the table.

Transaction support

A *transaction* is an atomic unit of database operations against the data in one or more databases. The effects of all the SQL statements in a transaction can be either all committed to the database or all rolled back.

For databases that support transactions, the Python interface silently starts a transaction when the cursor is created. The `commit()` method commits the updates made using that cursor, and the `rollback()` method discards them. Each method starts a new transaction.

The MySQL database has different types of storage engines. The most common are the MyISAM and the InnoDB engines. Since MySQL 5.5, InnoDB becomes the default storage engine. There is a trade-off between data security and database speed. The MyISAM tables are faster to process and they do not support transactions. The `commit()` and `rollback()` methods are not implemented. They do nothing. On the other hand, the InnoDB tables are more safe against the data loss. They support transactions. They are slower to process.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb
import sys


try:
    con = mdb.connect('localhost', 'testuser', 'test623', 'testdb')

    cur = con.cursor()
    cur.execute("DROP TABLE IF EXISTS Writers")
    cur.execute("CREATE TABLE Writers(Id INT PRIMARY KEY AUTO_INCREMENT, \
                 Name VARCHAR(25)) ENGINE=INNODB")
    cur.execute("INSERT INTO Writers(Name) VALUES('Jack London')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Honore de Balzac')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Lion Feuchtwanger')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Emile Zola')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Truman Capote')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Terry Pratchett')")

    con.commit()


except mdb.Error, e:

    if con:
        con.rollback()

    print "Error %d: %s" % (e.args[0],e.args[1])
    sys.exit(1)

finally:

    if con:
        con.close()
```

We recreate the `Writers` table. We explicitly work with transactions.

```
cur = con.cursor()
```

In Python DB API, we do not call the BEGIN statement to start a transaction. A transaction is started when the cursor is created.

```
cur.execute("CREATE TABLE Writers(Id INT PRIMARY KEY AUTO_INCREMENT, \
                Name VARCHAR(25)) ENGINE=INNODB")
```

We are dealing with a InnoDB table type. For older MySQL versions (<5.5), we need to specify the engine type with the `ENGINE=INNODB` option.

```
con.commit()
```

We must end a transaction with either a `commit()` or a `rollback()` method. If we comment this line,

the table is created but the data is not written to the table.

In this tutorial, we have been working with transactions without explicitly stating it. We used context managers. The context manager handles the entry and the exit from the desired runtime context for the execution of the block of code. Context managers are normally invoked using the `with` statement.

Connection objects in `MySQLdb` module can be used as context managers. They automatically commit or rollback transactions. Connection context managers clean up code by factoring out `try`, `except`, and `finally` statements.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb

con = mdb.connect('localhost', 'testuser', 'test623', 'testdb')

with con:

    cur = con.cursor()
    cur.execute("DROP TABLE IF EXISTS Writers")
    cur.execute("CREATE TABLE Writers(Id INT PRIMARY KEY AUTO_INCREMENT, \
                 Name VARCHAR(25))")
    cur.execute("INSERT INTO Writers(Name) VALUES('Jack London')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Honore de Balzac')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Lion Feuchtwanger')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Emile Zola')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Truman Capote')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Terry Pratchett')")
```

In our code example, the context manager deals with all the work necessary for error handling. It automatically commits or rolls back a transaction.

This was MySQL Python tutorial. ZetCode has a complete *e-book* for SQLite Python: SQLite Python e-book.

Home ‡ Top of Page