Python Programming

MySQL Connectivity With Python
Contributed by icarus, (c) Melonfire
2002−08−21
[*Send Me Similar Content When Posted*]
[*Add Developer Shed Headlines To Your Site*]

DISCUSS NEWS SEND PRINT PDF
advertisement

Article Index:

One of the cool things about Python is that, as an object−oriented language, it forces developers to think in terms of abstraction and code modularization when writing code, and thereby results in cleaner, better−packaged applications. However, packaging your code into functions is just the beginning; Python also allows you to create collections of shared code fragments, or "modules", which may be imported and used by any Python program. Powerful, flexible and very interesting, modules can be one of the most potent weapons in a Python developer's arsenal...so long as they're used correctly.

Over the course of this article, I'll be exploring one of Python's more useful modules, the MySQLdb module. This MySQLdb module allows developers to connect their Python code up to a MySQL database server, execute SQL commands on it, and massage the resulting data set into a useful and usable format...and it comes in very handy when you're developing database−driven Python applications. But don't take my word for it − see for yourself! In the words of its author, MySQLdb is "a thread−compatible interface to the popular MySQL database server that provides the Python database API." Developed by Andy Dustman, this module is needed for MySQL database connectivity under Python.

The first thing to do is make sure that you have the MySQLdb module installed on your Python development system. The easiest way to check this via the interactive Python command line interpreter − as the following example demonstrates:

```
Python 1.5.2 (#1, Feb 1 2000, 16:32:16) [GCC egcs−2.91.66
19990314/Linux (egcs− on linux−i386 Copyright 1991−1995 Stichting
Mathematisch Centrum, Amsterdam >>> import MySQLdb Traceback
(innermost last): File "<stdin>", line 1, in ? ImportError: No module named
MySQLdb >>>
```

If you see something like the error message above, you can safely assume that the module is not installed on your system. Drop by http://sourceforge.net/projects/mysql−python, download the latest copy of the MySQLdb distribution, and extract and install it on your system.

```
$ tar −xzvf MySQL−python−0.9.2.tar.gz $ cd MySQL−python−0.9.2 $
```

```
python setup.py build $ python setup.py install
```

Now, you can verify that the module has been installed by trying to import it again via the command line.

```
Python 1.5.2 (#1, Feb 1 2000, 16:32:16) [GCC egcs−2.91.66 19990314/Linux (egcs− on linux−i386 Copyright 1991−1995 Stichting Mathematisch Centrum, Amsterdam >>> import MySQLdb >>>
```

No problems this time round − which means we can proceed to actually writing some code. With that out of the way, here's a simple example that demonstrates some of the functionality of the DBI. Consider the following database table,

```
mysql> SELECT * FROM animals; +−−−−−−−−−+−−−−−−−−−−+ | name | species | +−−−−−−−−−+−−−−−−−−−−+ | Wallace | Walrus | | Polly | Parrot | | Freddie | Frog | | Tusker | Elephant | | Sammy | Skunk | +−−−−−−−−−+−−−−−−−−−−+ 5 rows in set (0.01 sec)
```

and then consider this short Python script, which connects to the database and prints out the data within the table.

```
#!/usr/bin/python # import MySQL module import MySQLdb # connect db = MySQLdb.connect(host="localhost", user="joe", passwd="secret", db="db56a") # create a cursor cursor = db.cursor() # execute SQL statement cursor.execute("SELECT * FROM animals") # get the resultset as a tuple result = cursor.fetchall() # iterate through resultset for record in result: print record[0] , "−−>", record[1]
```

Most of this is self−explanatory, but let me go through it with you briefly anyway.

The first step is to import the MySQLdb module, via Python's "import" function.

```
# import MySQL module import MySQLdb
```

Once that's done, you can open up a connection to the MySQL database server, by passing the module's connect() method a series of connection parameters − the server name, the database user name and password, and the database name.

```
# connect db = MySQLdb.connect(host="localhost", user="joe", passwd="secret", db="db56a")
```

A successful connection returns a Connection object, which you can use to create a cursor.

```
# create a cursor cursor = db.cursor()
```

This cursor is needed to execute an SQL statement, and to retrieve the generated resultset.

```
# execute SQL statement cursor.execute("SELECT * FROM animals") # get
the resultset as a tuple result = cursor.fetchall()
```

A number of methods are available to retrieve the SQL resultset − the one used here is the fetchall() method, which returns a tuple of tuples, each inner tuple representing a row of the resultset. This tuple can then be iterated over with a regular "for" loop, and its elements printed to the standard output.

```
# iterate through resultset for record in result: print record[0] , "−−>",
record[1]
```

You can also use the fetchone() method of the cursor object to get and display rows one at a time, instead of all at once. Consider the following example, which demonstrates how this might work:

```
#!/usr/bin/python # import MySQL module import MySQLdb # connect db
= MySQLdb.connect(host="localhost", user="joe", passwd="secret",
db="db56a") # create a cursor cursor = db.cursor() # execute SQL statement
cursor.execute("SELECT * FROM animals") # get the number of rows in
the resultset numrows = int(cursor.rowcount) # get and display one row at a
time for x in range(0,numrows): row = cursor.fetchone() print row[0],
"−−>", row[1]
```

In this case, the rowcount() method of the cursor object is first used to find out the number of rows in the resultset. This number is then used in a "for" loop to iterate over the resultset, with the fetchone() method used to move forward through the resultset and sequentially display the contents of each row.

You can get MySQLdb to give you a specific subset of the resultset with the cursor object's fetchmany() method, which allows you to specify the size of the returned resultset. Consider the following example, which demonstrates:

```
#!/usr/bin/python # import MySQL module import MySQLdb # connect db
= MySQLdb.connect(host="localhost", user="joe", passwd="secret",
db="db56a") # create a cursor cursor = db.cursor() # execute SQL statement
cursor.execute("SELECT * FROM animals") # limit the resultset to 3 items
result = cursor.fetchmany(3) # iterate through resultset for record in result:
print record[0] , "−−>", record[1]
```

Obviously, you can also perform INSERT, UPDATE and DELETE queries via the MySQLdb module. Consider the following example, which illustrates:

```
#!/usr/bin/python # import MySQL module import MySQLdb # connect db
= MySQLdb.connect(host="localhost", user="joe", passwd="secret",
db="db56a") # create a cursor cursor = db.cursor() # execute SQL statement
cursor.execute("""INSERT INTO animals (name, species) VALUES
("Harry", "Hamster")""")
```

You can modify this so that the values for the query string are input by the user − take a look at this variant of the example above, which demonstrates:

```
#!/usr/bin/python # import MySQL module import MySQLdb # get user
input name = raw_input("Please enter a name: ") species =
raw_input("Please enter a species: ") # connect db =
MySQLdb.connect(host="localhost", user="joe", passwd="secret",
db="db56a") # create a cursor cursor = db.cursor() # execute SQL statement
cursor.execute("INSERT INTO animals (name, species) VALUES (%s,
%s)", (name, species))
```

This time, when you run the script, you'll be asked for the values to be inserted into the database.

```
Please enter a name: Rollo Please enter a species: Rat
```

Notice the manner in which variables have been integrated into the SQL query in the example above. The %s placeholder is used to represent each variable in the query string, with the actual values stored in a tuple and passed as second argument.

In case you have auto−increment fields in your database, you can use the cursor object's insert_id() method to obtain the ID of the last inserted record − this comes in handy when you're dealing with linked tables in an RDBMS, as newly−inserted IDs from one table often serve as keys into other tables. The following code snippet should demonstrate how this works:

```
#!/usr/bin/python # import MySQL module import MySQLdb # connect db
= MySQLdb.connect(host="localhost", user="joe", passwd="secret",
db="db56a") # create a cursor cursor = db.cursor() # execute SQL statement
cursor.execute("""INSERT INTO test (field1, field2) VALUES ("val1",
"val2")""") # get ID of last inserted record print "ID of inserted record is ",
int(cursor.insert_id())
```

Many database scripts involve preparing a single query (an INSERT, for example) and then executing it again and again with different values. MySQLdb comes with an executemany() method, which simplifies this task and can also reduce performance overhead.

In order to understand how this works, consider the following example, which demonstrates:

```
#!/usr/bin/python # import MySQL module import MySQLdb # connect db
```

```
= MySQLdb.connect(host="localhost", user="joe", passwd="secret",
db="db56a") # create a cursor cursor = db.cursor() # dynamically generate
SQL statements from list cursor.executemany("INSERT INTO animals
(name, species) VALUES (%s, %s)", [ ('Rollo', 'Rat'), ('Dudley', 'Dolphin'),
('Mark', 'Marmoset') ])
```

In this case, the same query is repeated multiple times, with a different set of values each time. The values for each iteration are provided to the executemany() method as a Python list; each element of the list is a tuple containing the values for that iteration.

Using this technique, it's possible to write a script that asks the user to enter a series of data values, and then inserts them all into the database in one swell foop using the executemany() method. Which is just what I've done below:

```
#!/usr/bin/python # import MySQL module import MySQLdb # initialize
some variables name = "" data = [] # loop and ask for user input while (1):
name = raw_input("Please enter a name (EOF to end): ") if name == "EOF":
break species = raw_input("Please enter a species: ") # put user input into a
tuple tuple = (name, species) # and append to data[] list data.append(tuple) #
connect db = MySQLdb.connect(host="localhost", user="joe",
passwd="secret", db="db56a") # create a cursor cursor = db.cursor() #
dynamically generate SQL statements from data[] list
cursor.executemany("INSERT INTO animals (name, species) VALUES
(%s, %s)", data)
```

In this case, a "while" loop is used to continuously throw up user prompts, with each set of values entered by the user being packaged into a tuple and added to the "data" list. Once the user completes entering all the data, an executemany() statement is used, in combination with the various input values, to INSERT the values into the database. A number of other methods come bundled with the MySQLdb class − here's a brief list of the more interesting ones:

connection.begin() − start a transaction

connection.apilevel() − returns the current DB API level

connection.conv() − set type conversion options between MySQL and Python

connection.commit() − commit a transaction

connection.rollback() − roll back a transaction

And that's about all for the moment. In this article, I showed you how to configure and install the Python MySQLdb module, and use it to hook your Python scripts up to a MySQL database. I demonstrated the different techniques available for iterating over a resultset, showed you the basics of using variable placeholders and prepared queries, and illustrated some of the ancillary methods available in the module.

While this tutorial should get you and running with Python and MySQL, you shouldn't stop reading right away. Here are a few links worth checking out:

The MySQLdb project page on SourceForge, at
http://sourceforge.net/projects/mysql−python

The Python home page, at http://www.python.org/

The MySQL home page, at http://www.mysql.com/

Till next time...be good!

Note: All examples in this article have been tested on Linux/i586 with Python 1.5.2, MySQL 3.23 and MySQLdb 0.9.2. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!