Writing MySQL Scripts With Python's DB-API Interface

By Paul DuBois, NuSphere Corporation (October 2001)

TABLE OF CONTENTS

MySQLdb Installation
A Short DB-API Script
Writing the Script
Running the Script
A More Extensive DB-API Script
Error Handling
Methods for Issuing Queries
Portability Notes
Links
Appendix
About NuSphere

Python is one of the more popular Open Source programming languages, owing largely to its own native expressiveness as well as to the variety of support modules that are available to extend its capabilities. One of these modules is **DB-API**, which, as the name implies, provides a database application programming interface. DB-API is designed to be relatively independent of details specific to any given database engine; this helps you write database-access scripts that are portable between engines.

DB-API's design is similar to that used by Perl's DBI module, the PHP PEAR DB class, and the Java JDBC interface. It uses a two-level architecture in which the top level provides an abstract interface that is similar for all supported database engines, and a lower level consisting of drivers for specific engines that handle engine-dependent details. This means, of course, that to use DB-API for writing Python scripts, you must have a driver for your particular database system. For the NuSphere products, DB-API provides database access by means of the MySQLdb driver. This article begins by discussing driver installation (in case you don't have MySQLdb), then moves on to cover how to write DB-API scripts.

MySQLdb Installation

To write MySQL scripts that use DB-API, Python itself must be installed. That will almost certainly be true if you're using Unix, but is less likely for Windows. Installers for either platform can be found on the Python web site (see the "Links" section at the end of this article).

Next, verify that your version of Python is 1.5.2 or later, and that the MySQLdb module is installed. You can check both of these requirements by running Python in interactive mode from the command line prompt (something like `%` for Unix or `C:\>` for Windows):

```
% python
>>> import sys
>>> sys.version
'1.5.2 (#1, Aug 25 2000, 09:33:37)  [GCC 2.96 20000731
(experimental)]'
>>> import MySQLdb
```

Assuming that you have a recent enough version of Python and that no error occurs when you issue the `import MySQLdb` statement, you're ready to begin writing database-access scripts and you can skip to the next section. However, if you get the following error, you need to obtain and install MySQLdb first:

```
>>> import MySQLdb
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: No module named MySQLdb
```

To obtain MySQLdb, visit the "Links" section to see where to fetch a distribution appropriate for your system. Precompiled binaries are available for several platforms (RedHat Linux, Debian Linux, Windows), or you can install from source. If you use a binary distribution, install it using your platform's usual package installation procedure. To build and install MySQLdb from source, move into the top-level directory of the distribution and issue the following commands. (Under

Unix, it's likely that you'll need to run the second command as `root` so that the driver files can be copied into your Python installation.)

```
% python setup.py build
% python setup.py install
```

If the `setup.py` script fails because it can't find the Distutils module, one additional prerequisite you'll need to satisfy is to install Distutils. (MySQLdb supports Python 1.5.2 and up, but Distutils is included with Python only as of version 1.6.) The "Links" section indicates where to obtain this module. If you encounter other problems, check the `README` file included with the MySQLdb distribution.

A Short DB-API Script

Scripts that access MySQL through DB-API using MySQLdb generally perform the following steps:

- Import the MySQLdb module
- Open a connection to the MySQL database server
- Issue queries and retrieve their results
- Close the server connection

The rest of this section presents a short DB-API script that illustrates the basic elements of these steps. Later sections discuss specific aspects of script-writing in more detail.

Writing the Script

Use a text editor to create a file named `server_version.py` that contains the following script. This script uses MySQLdb to interact with the MySQL database server, albeit in relatively rudimentary fashion—all it does is ask the server for its version string:

```
# server_version.py - retrieve and display database server
version
```

```
import MySQLdb

conn = MySQLdb.connect (host = "localhost",
                        user = "testuser",
                        passwd = "testpass",
                        db = "test")
cursor = conn.cursor ()
cursor.execute ("SELECT VERSION()")
row = cursor.fetchone ()
print "server version:", row[0]
cursor.close ()
conn.close ()
```

The `import` statement tells Python that the script needs to use the code in the MySQLdb module. This statement must precede any attempt to connect to the MySQL database server. Then the connection is established by invoking the `connect()` method of the MySQLdb driver and specifying the proper connection parameters. These include the hostname where the server is running, the user name and password for your MySQL account, and the name of the database you want to use. `connect()` argument list syntax varies among drivers; for MySQLdb, the arguments are

allowed to be given in *`name`* = *`value`* format, which has the advantage that you can specify them in any order. `server_version.py` makes a connection to the MySQL database server on the local host to access the `test` database with a user name and password of `testuser` and `testpass`:

```
conn = MySQLdb.connect (host = "localhost",
                        user = "testuser",
                        passwd = "testpass",
                        db = "test")
```

If the `connect()` call is successful, it returns a connection object that serves as the basis for further interaction with the MySQL database. If the call fails, an exception is raised.

(`server_version.py` doesn't handle the exception, so an error at this point terminates the script. Error handling is covered later in this article.)

After the connection object has been obtained successfully, `server_version.py` invokes its `cursor()` method to create a cursor object for processing queries. The script uses this cursor to issue a `SELECT VERSION()` statement, which returns a string containing server version information:

```
cursor = conn.cursor ()
cursor.execute ("SELECT VERSION()")
row = cursor.fetchone ()
print "server version:", row[0]
cursor.close ()
```

The cursor object's `execute()` method sends the query to the server and `fetchone()` retrieves a row as a tuple. For the query shown here, the tuple contains a single value, which the script prints. (If no row is available, `fetchone()` actually will return the value `None`; `server_version.py` blithely assumes that this won't happen, an assumption that you normally should not make. In later examples, we'll handle this case.) Cursor objects can be used to issue multiple queries, but `server_version.py` has no more need for `cursor` after getting the version string, so it closes it.

Finally, the script invokes the connection object's `close()` method to disconnect from the server:

```
conn.close ()
```

After that, `conn` becomes invalid and should not be used to access the server.

Running the Script

To execute the `server_version.py` script, invoke Python from the command line prompt and tell it the script name. You should see a result something like this:

```
% python server_version.py
server version: 3.23.39-log
```

This indicates that the MySQL server version is 3.23.39, and the `-log` suffix tells us that query logging is enabled. (If you have debugging enabled, you'll see a `-debug` suffix.)

It's possible to set up the script so that it can be run by name without invoking Python explicitly. Under Unix, add an initial `#!` line to the script that specifies the full pathname of the Python interpreter. This tells the system what program should execute the script. For example, if Python lives at `/usr/bin/python` on your system, add the following as the first line of the script:

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

Then use chmod to make the script executable, and you'll be able to run it directly:

```
% chmod +x server_version.py
% ./server_version.py
```

(The leading `./` tells your command interpreter explicitly that the script is located in your current directory. Many Unix accounts are set up not to search the current directory when looking for commands.)

Under Windows, the `#!` line is unnecessary (although it's harmless, so you need not remove it if you write the script on a Unix system and then move it to a Windows box). Also, instead of using **chmod** to make the script executable, open the Folder Options item in the Control Panel and select its File Types tab. File Types allows you to set up an association for files that end with `.py` to tell Windows to execute them with Python. Then you can invoke the script by name:

```
C:\> server_version.py
```

A More Extensive DB-API Script

`server_version.py` has a number of shortcomings. For example, it doesn't catch exceptions or indicate what went wrong if an error occurs, and it doesn't allow for the possibility that the query it runs may not return any results. This section shows how to address these issues using a more elaborate script, `animal.py`, that uses a table containing animal names and categories:

```
CREATE TABLE animal
(
    name CHAR(40),
    category CHAR(40)
```

```
)
```

If you've read the PEAR DB article available at the NuSphere Tech Library, you may recognize this table and some of the queries issued by `animal.py`; they were used in that article, too.

The `animal.py` script begins like this (including the `#!` line, should you intend to run the script on a Unix system):

```
#! /usr/bin/python
# animal.py - create animal table and
# retrieve information from it

import sys
import MySQLdb
```

As with `server_version.py`, the script imports MySQLdb, but it also imports the `sys` module for use in error handling. (`animal.py` uses `sys.exit()` to return 0 or 1 to indicate normal termination or that an error occurred.)

Error Handling

After importing the requisite modules, `animal.py` establishes a connection to the server using the `connect()` call. To allow for the possibility of connection failure (for example, so that you can display the reason for the failure), it's necessary to catch exceptions. To handle exceptions in Python, put your code in a `try` block and include an `except` block that contains the error-handling code. The resulting connection sequence looks like this:

```
try:
    conn = MySQLdb.connect (host = "localhost",
                            user = "testuser",
                            passwd = "testpass",
                            db = "test")
except MySQLdb.Error, e:
    print "Error %d: %s" % (e.args[0], e.args[1])
    sys.exit (1)
```

The `except` line names an exception class (`MySQLdb.Error` in this example) to obtain the database-specific error information that MySQLdb can provide, as well as a variable (`e`) in which to store the information. If an exception occurs, MySQLdb makes this information available in `e.args`, a two-element tuple containing the numeric error code and a string describing the error. The `except` block shown in the example prints both values and exits.

Any database-related statements can be placed in a similar `try/except` structure to trap and report errors. However, for brevity, the following discussion doesn't show the exception-handling code. (The complete text of `animal.py` is listed in the appendix.)

Methods for Issuing Queries

The next section of `animal.py` creates a cursor object and uses it to issue queries that set up and populate the `animal` table:

```
cursor = conn.cursor ()
cursor.execute ("DROP TABLE IF EXISTS animal")
cursor.execute ("""
        CREATE TABLE animal
        (
            name CHAR(40),
            category CHAR(40)
        )
    """)
cursor.execute ("""
        INSERT INTO animal (name, category)
        VALUES
            ('snake', 'reptile'),
            ('frog', 'amphibian'),
            ('tuna', 'fish'),
            ('racoon', 'mammal')
    """)
print "%d rows were inserted" % cursor.rowcount
```

Note that this code includes no error-checking. (Remember that it will be placed in a `try` block; errors will trigger exceptions that are caught and handled in the corresponding `except` block, which allows the main flow of the code to read more smoothly.) The queries perform the following actions:

- Drop the `animal` table if it already exists, to begin with a clean slate.
- Create the `animal` table.
- Insert some data into the table and report the number of rows added.

Each query is issued by invoking the cursor object's `execute()` method. The first two queries produce no result, but the third produces a count indicating the number of rows inserted. The count is available in the cursor's `rowcount` attribute. (Some database interfaces provide this count as the return value of the query-execution call, but that is not true for DB-API.)

The `animal` table is set up at this point, so we can issue `SELECT` queries to retrieve information from it. As with the preceding statements, `SELECT` queries are issued using `execute()`. However, unlike statements such as `DROP` or `INSERT`, `SELECT` queries generate a result set that you must retrieve. That is, `execute()` only issues the query, it does not return the result set. You can use `fetchone()` to get the rows one at a time, or `fetchall()` to get them all at once. `animal.py` uses both approaches. Here's how to use `fetchone()` for row-at-a-time retrieval:

```
cursor.execute ("SELECT name, category FROM animal")
while (1):
    row = cursor.fetchone ()
    if row == None:
        break
    print "%s, %s" % (row[0], row[1])
print "%d rows were returned" % cursor.rowcount
```

`fetchone()` returns the next row of the result set as a tuple, or the value `None` if no more rows are available. The loop checks for this and exits when the result set has been exhausted. For each row returned, the tuple contains two values (that's how many columns the `SELECT` query asked for), which `animal.py` prints. The `print` statement shown above accesses the individual

tuple elements. However, because they are used in order of occurrence within the tuple, the `print` statement could just as well have been written like this:

```
print "%s, %s" % row
```

After displaying the query result, the script also prints the number of rows returned (available as the value of the `rowcount` attribute).

`fetchall()` returns the entire result set all at once as a tuple of tuples, or as an empty tuple if the result set is empty. To access the individual row tuples, iterate through the row set that `fetchall()` returns:

```
cursor.execute ("SELECT name, category FROM animal")
rows = cursor.fetchall ()
for row in rows:
    print "%s, %s" % (row[0], row[1])
print "%d rows were returned" % cursor.rowcount
```

This code prints the row count by accessing `rowcount`, just as for the `fetchone()` loop. Another way to determine the row count when you use `fetchall()` is by taking the length of the value that it returns:

```
print "%d rows were returned" % len (rows)
```

The fetch loops shown thus far retrieve rows as tuples. It's also possible to fetch rows as dictionaries, which allows column values to be accessed by name. The following code shows how to do this. Note that dictionary access requires a different kind of cursor, so the example closes the cursor and obtains a new one that uses a different cursor class:

```
cursor.close ()
cursor = conn.cursor (MySQLdb.cursors.DictCursor)
cursor.execute ("SELECT name, category FROM animal")
result_set = cursor.fetchall ()
for row in result_set:
    print "%s, %s" % (row["name"], row["category"])
print "%d rows were returned" % cursor.rowcount
```

MySQLdb supports a placeholder capability that allows you to bind data values to special markers within the query string. This provides an alternative to embedding the values directly into the query. The placeholder mechanism handles adding quotes around data values, and it escapes any special characters that occur within values. The following examples demonstrate an `UPDATE` query that changes `snake` to `turtle`, first using literal values and then using placeholders. The literal-value query looks like this:

```
cursor.execute ("""
            UPDATE animal SET name = 'turtle'
            WHERE name = 'snake'
        """)
print "%d rows were updated" % cursor.rowcount
```

If the values are stored in variables, you can issue the same query by using placeholders and binding the appropriate values to them:

```
cur_name = "snake"
new_name = "turtle"
cursor.execute ("""
            UPDATE animal SET name = %s
            WHERE name = %s
        """, (new_name, cur_name))
print "%d rows were updated" % cursor.rowcount
```

Note the following points about the form of the preceding `execute()` call:

- The placeholder marker is `%s`; it should occur once per value to be inserted into the query string.
- No quotes should be placed around the `%s` markers; MySQLdb supplies them for you as necessary.
- Following the query string argument to `execute()`, provide a tuple containing the values to be bound to the placeholders, in the order they should appear within the string. If you have only a single value `x`, specify it as `(x,)` to indicate a single-element tuple.

After issuing the queries, `animal.py` closes the cursor, disconnects from the server, and exits:

```
cursor.close ()
conn.close ()
sys.exit (0)
```

Portability Notes

If you want to port a MySQLdb-based DB-API script for use with a different database, the following things may cause problems. Sources of non-portability occur anywhere that the driver name might be used. These include:

- The `import` statement that imports the driver module. This must be changed to import a different driver.
- The `connect()` call that connects to the database server. The `connect()` method is accessed through the name of the driver modules, so the driver name needs to be changed. In addition, the `connect()` argument syntax may vary between drivers.
- Exception handling. The exception class named on `except` statements is referenced through the driver name.

Another type of non-portability that does not involve the driver name concerns the use of placeholders. The DB-API specification allows for several placeholder syntaxes, and some drivers use a syntax that differs from the one supported by MySQLdb.

Links

- Andy Dustman, author of the MySQLdb module, has a site at:

  `http://dustman.net/andy/python/`

  That site is the best place to read the MySQLdb documentation and FAQ online. It also has links to Debian and Windows binary distributions. To get source code or Linux RPMs, visit the MySQLdb SourceForge repository at:

```
http://sourceforge.net/projects/mysql-python
```

- The Python web site has installers for the Python language processor, should you be running on a system that doesn't already have it installed:

  ```
  http://www.python.org/
  ```

- If your version of Python doesn't include it, the Distutils distribution that is needed for building and installing MySQLdb from source can be obtained at:

  ```
  http://www.python.org/sigs/distutils-sig/download.html
  ```

- The database SIG (special interest group) area on the Python web site contains additional DB-API information:

  ```
  http://www.python.org/sigs/db-sig/
  ```

- The `animal` table used by the `animal.py` script is also used in the PEAR DB article at the NuSphere Tech Library:

  ```
  http://www.nusphere.com/products/tech_library.htm
  ```

  You might find it instructive to compare that article with this one to see where DB-API and PEAR DB are similar or different in their approaches to database access.

Appendix

The full source code for the `animal.py` script is shown here:

```
#!/usr/bin/python
# animal.py - create animal table and
# retrieve information from it

import sys
import MySQLdb
```

```
# connect to the MySQL server

try:
    conn = MySQLdb.connect (host = "localhost",
                            user = "testuser",
                            passwd = "testpass",
                            db = "test")
except MySQLdb.Error, e:
    print "Error %d: %s" % (e.args[0], e.args[1])
    sys.exit (1)

# create the animal table and populate it

try:
    cursor = conn.cursor ()
    cursor.execute ("DROP TABLE IF EXISTS animal")
    cursor.execute ("""
            CREATE TABLE animal
            (
                name CHAR(40),
                category CHAR(40)
            )
        """)
    cursor.execute ("""
            INSERT INTO animal (name, category)
            VALUES
                ('snake', 'reptile'),
                ('frog', 'amphibian'),
                ('tuna', 'fish'),
                ('racoon', 'mammal')
        """)
    print "%d rows were inserted" % cursor.rowcount

# perform a fetch loop using fetchone()
```

```
    cursor.execute ("SELECT name, category FROM animal")
    while (1):
        row = cursor.fetchone ()
        if row == None:
            break
        print "%s, %s" % (row[0], row[1])
    print "%d rows were returned" % cursor.rowcount

# perform a fetch loop using fetchall()

    cursor.execute ("SELECT name, category FROM animal")
    rows = cursor.fetchall ()
    for row in rows:
        print "%s, %s" % (row[0], row[1])
    print "%d rows were returned" % cursor.rowcount

# issue a query that includes data values literally in
# the query string, then do same thing using placeholders

    cursor.execute ("""
                UPDATE animal SET name = 'turtle'
                WHERE name = 'snake'
            """)
    print "%d rows were updated" % cursor.rowcount

    cur_name = "snake"
    new_name = "turtle"
    cursor.execute ("""
                UPDATE animal SET name = %s
                WHERE name = %s
            """, (new_name, cur_name))
    print "%d rows were updated" % cursor.rowcount

# create a dictionary cursor so that column values
# can be accessed by name rather than by position
```

```
    cursor.close ()
    cursor = conn.cursor (MySQLdb.cursors.DictCursor)
    cursor.execute ("SELECT name, category FROM animal")
    result_set = cursor.fetchall ()
    for row in result_set:
        print "%s, %s" % (row["name"], row["category"])
    print "%d rows were returned" % cursor.rowcount

    cursor.close ()

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

conn.close ()
sys.exit (0)
```

About NuSphere Corporation

NuSphere delivers the first Internet Application Platform (IAP) based on open source components, providing an integrated foundation that allows companies to deploy reliable, cost-effective, enterprise-class applications across Windows, UNIX and Linux environments. NuSphere® Advantage is an integrated software suite that pairs the reliability and cost-effectiveness of PHP, Apache, Perl and open source databases with new technology for building business-critical web applications and web services. Based in Bedford, Mass., the company's commercial software services include technical support, consulting and training. For more information, visit www.nusphere.com or call +1-781-280-4600.