MySQLdb User's Guide

Contents

Introduction

MySQLdb is an thread-compatible interface to the popular MySQL database server that provides the Python database API.

Installation

The `README` file has complete installation instructions.

_mysql

If you want to write applications which are portable across databases, use MySQLdb, and avoid using this module directly. `_mysql` provides an interface which mostly implements the MySQL C API. For more information, see the MySQL documentation. The documentation for this module is intentionally weak because you probably should use the higher-level MySQLdb module. If you really need it, use the standard MySQL docs and transliterate as necessary.

MySQL C API translation

The MySQL C API has been wrapped in an object-oriented way. The only MySQL data structures which are implemented are the `MYSQL` (database connection handle) and `MYSQL_RES` (result handle) types. In general, any function which takes `MYSQL *mysql` as an argument is now a method of the connection object, and any function which takes `MYSQL_RES *result` as an argument is a method of the result object. Functions requiring none of the MySQL data structures are implemented as functions in the module. Functions requiring one of the other MySQL data structures are generally not implemented. Deprecated functions are not implemented. In all cases, the `mysql_` prefix is dropped from the name. Most of the `conn` methods listed are also available as MySQLdb Connection object methods. Their use is non-portable.

MySQL C API function mapping

C API	`_mysql`
`mysql_affected_rows()`	`conn.affected_rows()`
`mysql_autocommit()`	`conn.autocommit()`
`mysql_character_set_name()`	`conn.character_set_name()`
`mysql_close()`	`conn.close()`
`mysql_commit()`	`conn.commit()`
`mysql_connect()`	`_mysql.connect()`
`mysql_data_seek()`	`result.data_seek()`
`mysql_debug()`	`_mysql.debug()`
`mysql_dump_debug_info`	`conn.dump_debug_info()`
`mysql_escape_string()`	`_mysql.escape_string()`
`mysql_fetch_row()`	`result.fetch_row()`
`mysql_get_character_set_info()`	`conn.get_character_set_info()`
`mysql_get_client_info()`	`_mysql.get_client_info()`
`mysql_get_host_info()`	`conn.get_host_info()`
`mysql_get_proto_info()`	`conn.get_proto_info()`
`mysql_get_server_info()`	`conn.get_server_info()`
`mysql_info()`	`conn.info()`
`mysql_insert_id()`	`conn.insert_id()`
`mysql_num_fields()`	`result.num_fields()`
`mysql_num_rows()`	`result.num_rows()`
`mysql_options()`	various options to `_mysql.connect()`
`mysql_ping()`	`conn.ping()`
`mysql_query()`	`conn.query()`
`mysql_real_connect()`	`_mysql.connect()`
`mysql_real_query()`	`conn.query()`
`mysql_real_escape_string()`	`conn.escape_string()`
`mysql_rollback()`	`conn.rollback()`
`mysql_row_seek()`	`result.row_seek()`
`mysql_row_tell()`	`result.row_tell()`
`mysql_select_db()`	`conn.select_db()`
`mysql_set_character_set()`	`conn.set_character_set()`
`mysql_ssl_set()`	`ssl` option to `_mysql.connect()`
`mysql_stat()`	`conn.stat()`
`mysql_store_result()`	`conn.store_result()`
`mysql_thread_id()`	`conn.thread_id()`
`mysql_thread_safe_client()`	`conn.thread_safe_client()`
`mysql_use_result()`	`conn.use_result()`
`mysql_warning_count()`	`conn.warning_count()`
`CLIENT_*`	`MySQLdb.constants.CLIENT.*`
`CR_*`	`MySQLdb.constants.CR.*`
`ER_*`	`MySQLdb.constants.ER.*`
`FIELD_TYPE_*`	`MySQLdb.constants.FIELD_TYPE.*`

FLAG_*	MySQLdb.constants.FLAG.*

Some _mysql examples

Okay, so you want to use `_mysql` anyway. Here are some examples.

The simplest possible database connection is:

```
import _mysql
db=_mysql.connect()
```

This creates a connection to the MySQL server running on the local machine using the standard UNIX socket (or named pipe on Windows), your login name (from the USER environment variable), no password, and does not `USE` a database. Chances are you need to supply more information.:

```
db=_mysql.connect("localhost","joebob","moonpie","thangs")
```

This creates a connection to the MySQL server running on the local machine via a UNIX socket (or named pipe), the user name "joebob", the password "moonpie", and selects the initial database "thangs".

We haven't even begun to touch upon all the parameters `connect()` can take. For this reason, I prefer to use keyword parameters:

```
db=_mysql.connect(host="localhost",user="joebob",
                  passwd="moonpie",db="thangs")
```

This does exactly what the last example did, but is arguably easier to read. But since the default host is "localhost", and if your login name really was "joebob", you could shorten it to this:

```
db=_mysql.connect(passwd="moonpie",db="thangs")
```

UNIX sockets and named pipes don't work over a network, so if you specify a host other than localhost, TCP will be used, and you can specify an odd port if you need to (the default port is 3306):

```
db=_mysql.connect(host="outhouse",port=3307,passwd="moonpie",db="thangs")
```

If you really had to, you could connect to the local host with TCP by specifying the full host name, or 127.0.0.1.

Generally speaking, putting passwords in your code is not such a good idea:

```
db=_mysql.connect(host="outhouse",db="thangs",read_default_file="~/.my.cnf")
```

This does what the previous example does, but gets the username and password and other parameters from ~/.my.cnf (UNIX-like systems). Read about option files for more details.

So now you have an open connection as `db` and want to do a query. Well, there are no cursors in MySQL, and no parameter substitution, so you have to pass a complete query string to `db.query()`:

```
db.query("""SELECT spam, eggs, sausage FROM breakfast
         WHERE price < 5""")
```

There's no return value from this, but exceptions can be raised. The exceptions are defined in a separate module, `_mysql_exceptions`, but `_mysql` exports them. Read DB API specification PEP-249 to find out what they are, or you can use the catch-all `MySQLError`.

At this point your query has been executed and you need to get the results. You have two options:

```
r=db.store_result()
# ...or...
r=db.use_result()
```

Both methods return a result object. What's the difference? `store_result()` returns the entire result set to the client immediately. If your result set is really large, this could be a problem. One way around this is to add a `LIMIT` clause to your query, to limit the number of rows returned. The other is to use `use_result()`, which keeps the result set in the server and sends it row-by-row when you fetch. This does, however, tie up server resources, and it ties up the connection: You cannot do any more queries until you have fetched **all** the rows. Generally I recommend using `store_result()` unless your result set is really huge and you can't use `LIMIT` for some reason.

Now, for actually getting real results:

```
>>> r.fetch_row()
(('3','2','0'),)
```

This might look a little odd. The first thing you should know is, `fetch_row()` takes some additional parameters. The first one is, how many rows (`maxrows`) should be returned. By default, it returns one row. It may return fewer rows than you asked for, but never more. If you set `maxrows=0`, it returns all rows of the result set. If you ever get an empty tuple back, you ran out of rows.

The second parameter (`how`) tells it how the row should be represented. By default, it is zero which means, return as a tuple. `how=1` means, return it as a dictionary, where the keys are the column names, or `table.column` if there are two columns with the same name (say, from a join). `how=2` means the same as `how=1` except that the keys are *always* `table.column`; this is for compatibility with the old `Mysqldb` module.

OK, so why did we get a 1-tuple with a tuple inside? Because we implicitly asked for one row, since we didn't specify `maxrows`.

The other oddity is: Assuming these are numeric columns, why are they returned as strings? Because MySQL returns all data as strings and expects you to convert it yourself. This would be a real pain in the ass, but in fact, `_mysql` can do this for you. (And `MySQLdb` does do this for you.) To have automatic type conversion done, you need to create a type converter dictionary, and pass this to `connect()` as the `conv` keyword parameter.

The keys of `conv` should be MySQL column types, which in the C API are `FIELD_TYPE_*`. You can get these values like this:

```
from MySQLdb.constants import FIELD_TYPE
```

By default, any column type that can't be found in `conv` is returned as a string, which works for a lot of stuff. For our purposes, we probably want this:

```
my_conv = { FIELD_TYPE.LONG: int }
```

This means, if it's a `FIELD_TYPE_LONG`, call the builtin `int()` function on it. Note that `FIELD_TYPE_LONG` is an `INTEGER` column, which corresponds to a C `long`, which is also the type used for a normal Python integer. But beware: If it's really an `UNSIGNED INTEGER` column, this could cause overflows. For this reason, `MySQLdb` actually uses `long()` to do the conversion. But we'll ignore this potential problem for now.

Then if you use `db=_mysql.connect(conv=my_conv...)`, the results will come back `((3, 2, 0),)`, which is what you would expect.

MySQLdb

MySQLdb is a thin Python wrapper around `_mysql` which makes it compatible with the Python DB API interface (version 2). In reality, a fair amount of the code which implements the API is in `_mysql` for the sake of efficiency.

The DB API specification PEP-249 should be your primary guide for using this module. Only deviations from the spec and other database-dependent things will be documented here.

Functions and attributes

Only a few top-level functions and attributes are defined within MySQLdb.

connect(parameters...)
: Constructor for creating a connection to the database. Returns a Connection Object. Parameters are the same as for the MySQL C API. In addition, there are a few additional keywords that correspond to what you would pass `mysql_options()` before connecting. Note that some parameters must be specified as keyword arguments! The default value for each parameter is NULL or zero, as appropriate. Consult the MySQL documentation for more details. The important parameters are:

    host
    : name of host to connect to. Default: use the local host via a UNIX socket (where applicable)

    user
    : user to authenticate as. Default: current effective user.

    passwd
    : password to authenticate with. Default: no password.

    db
    : database to use. Default: no default database.

    port
    : TCP port of MySQL server. Default: standard port (3306).

    unix_socket
    : location of UNIX socket. Default: use default location or TCP for remote hosts.

    conv
    : type conversion dictionary. Default: a copy of `MySQLdb.converters.conversions`

    compress
    : Enable protocol compression. Default: no compression.

    connect_timeout
    : Abort if connect is not completed within given number of seconds. Default: no timeout (?)

    named_pipe
    : Use a named pipe (Windows). Default: don't.

    init_command
    : Initial command to issue to server upon connection. Default: Nothing.

    read_default_file
    : MySQL configuration file to read; see the MySQL documentation for `mysql_options()`.

    read_default_group
    : Default group to read; see the MySQL documentation for `mysql_options()`.

    cursorclass
    : cursor class that `cursor()` uses, unless overridden. Default: `MySQLdb.cursors.Cursor`. *This must be a keyword parameter.*

    use_unicode

If True, CHAR and VARCHAR and TEXT columns are returned as Unicode strings, using the configured character set. It is best to set the default encoding in the server configuration, or client configuration (read with read_default_file). If you change the character set after connecting (MySQL-4.1 and later), you'll need to put the correct character set name in connection.charset.

If False, text-like columns are returned as normal strings, but you can always write Unicode strings.

This must be a keyword parameter.

charset
: If present, the connection character set will be changed to this character set, if they are not equal. Support for changing the character set requires MySQL-4.1 and later server; if the server is too old, UnsupportedError will be raised. This option implies use_unicode=True, but you can override this with use_unicode=False, though you probably shouldn't.

  If not present, the default character set is used.

  This must be a keyword parameter.

sql_mode
: If present, the session SQL mode will be set to the given string. For more information on sql_mode, see the MySQL documentation. Only available for 4.1 and newer servers.

  If not present, the session SQL mode will be unchanged.

  This must be a keyword parameter.

ssl
: This parameter takes a dictionary or mapping, where the keys are parameter names used by the mysql_ssl_set MySQL C API call. If this is set, it initiates an SSL connection to the server; if there is no SSL support in the client, an exception is raised. *This must be a keyword parameter.*

apilevel
: String constant stating the supported DB API level. '2.0'

threadsafety
: Integer constant stating the level of thread safety the interface supports. This is set to 1, which means: Threads may share the module.

  The MySQL protocol can not handle multiple threads using the same connection at once. Some earlier versions of MySQLdb utilized locking to achieve a threadsafety of 2. While this is not terribly hard to accomplish using the standard Cursor class (which uses `mysql_store_result()`), it is complicated by SSCursor (which uses `mysql_use_result()`; with the latter you must ensure all the rows have been read before another query can be executed. It is further complicated by the addition of transactions, since transactions start when a cursor execute a query, but end when `COMMIT` or `ROLLBACK` is executed by the Connection object. Two threads simply cannot share a connection while a transaction is in progress, in addition to not being able to share it during query execution. This excessively complicated the code to the point where it just isn't worth it.

  The general upshot of this is: Don't share connections between threads. It's really not worth your effort or mine, and in the end, will probably hurt performance, since the MySQL server runs a separate thread for each connection. You can certainly do things like cache connections in a pool, and give those connections to one thread at a time. If you let two threads use a connection simultaneously, the MySQL client library will probably upchuck and die. You have been warned.

  For threaded applications, try using a connection pool. This can be done using the Pool module.

charset

The character set used by the connection. In MySQL-4.1 and newer, it is possible (but not recommended) to change the connection's character set with an SQL statement. If you do this, you'll also need to change this attribute. Otherwise, you'll get encoding errors.

paramstyle

String constant stating the type of parameter marker formatting expected by the interface. Set to 'format' = ANSI C printf format codes, e.g. '...WHERE name=%s'. If a mapping object is used for conn.execute(), then the interface actually uses 'pyformat' = Python extended format codes, e.g. '...WHERE name=%(name)s'. However, the API does not presently allow the specification of more than one style in paramstyle.

Note that any literal percent signs in the query string passed to execute() must be escaped, i.e. %%.

Parameter placeholders can **only** be used to insert column values. They can **not** be used for other parts of SQL, such as table names, statements, etc.

conv

A dictionary or mapping which controls how types are converted from MySQL to Python and vice versa.

If the key is a MySQL type (from `FIELD_TYPE.*`), then the value can be either:

- a callable object which takes a string argument (the MySQL value),' returning a Python value
- a sequence of 2-tuples, where the first value is a combination of flags from `MySQLdb.constants.FLAG`, and the second value is a function as above. The sequence is tested until the flags on the field match those of the first value. If both values are None, then the default conversion is done. Presently this is only used to distinquish TEXT and BLOB columns.

If the key is a Python type or class, then the value is a callable Python object (usually a function) taking two arguments (value to convert, and the conversion dictionary) which converts values of this type to a SQL literal string value.

This is initialized with reasonable defaults for most types. When creating a Connection object, you can pass your own type converter dictionary as a keyword parameter. Otherwise, it uses a copy of `MySQLdb.converters.conversions`. Several non-standard types are returned as strings, which is how MySQL returns all columns. For more details, see the built-in module documentation.

Connection Objects

Connection objects are returned by the `connect()` function.

commit()

If the database and the tables support transactions, this commits the current transaction; otherwise this method successfully does nothing.

rollback()

If the database and tables support transactions, this rolls back (cancels) the current transaction; otherwise a `NotSupportedError` is raised.

cursor([cursorclass])

MySQL does not support cursors; however, cursors are easily emulated. You can supply an alternative cursor class as an optional parameter. If this is not present, it defaults to the value given when creating the connection object, or the standard `Cursor` class. Also see the additional supplied cursor classes in the usage section.

There are many more methods defined on the connection object which are MySQL-specific. For more information on them, consult the internal documentation using `pydoc`.

Cursor Objects

callproc(procname, args)

Calls stored procedure procname with the sequence of arguments in args. Returns the original arguments. Stored procedure support only works with MySQL-5.0 and newer.

Compatibility note: PEP-249 specifies that if there are OUT or INOUT parameters, the modified values are to be returned. This is not consistently possible with MySQL. Stored procedure arguments must be passed as server variables, and can only be returned with a SELECT statement. Since a stored procedure may return zero or more result sets, it is impossible for MySQLdb to determine if there are result sets to fetch before the modified parmeters are accessible.

The parameters are stored in the server as @_*procname*_*n*, where *n* is the position of the parameter. I.e., if you cursor.callproc('foo', (a, b, c)), the parameters will be accessible by a SELECT statement as @_foo_0, @_foo_1, and @_foo_2.

Compatibility note: It appears that the mere act of executing the CALL statement produces an empty result set, which appears after any result sets which might be generated by the stored procedure. Thus, you will always need to use nextset() to advance result sets.

close()

Closes the cursor. Future operations raise `ProgrammingError`. If you are using server-side cursors, it is very important to close the cursor when you are done with it and before creating a new one.

info()

Returns some information about the last query. Normally you don't need to check this. If there are any MySQL warnings, it will cause a Warning to be issued through the Python warning module. By default, Warning causes a message to appear on the console. However, it is possible to filter these out or cause Warning to be raised as exception. See the MySQL docs for `mysql_info()`, and the Python warning module. (Non-standard)

setinputsizes()

Does nothing, successfully.

setoutputsizes()

Does nothing, successfully.

nextset()

Advances the cursor to the next result set, discarding the remaining rows in the current result set. If there are no additional result sets, it returns None; otherwise it returns a true value.

Note that MySQL doesn't support multiple result sets until 4.1.

Some examples

The `connect()` method works nearly the same as with _mysql:

```
import MySQLdb
db=MySQLdb.connect(passwd="moonpie",db="thangs")
```

To perform a query, you first need a cursor, and then you can execute queries on it:

```
c=db.cursor()
max_price=5
c.execute("""SELECT spam, eggs, sausage FROM breakfast
          WHERE price < %s""", (max_price,))
```

In this example, `max_price=5` Why, then, use `%s` in the string? Because MySQLdb will convert it to a SQL literal value, which is the string '5'. When it's finished, the query will actually say, "...WHERE price < 5".

Why the tuple? Because the DB API requires you to pass in any parameters as a sequence. Due to the design of the parser, (max_price) is interpreted as using algebraic grouping and simply as max_price and not a tuple. Adding a comma, i.e. (max_price,) forces it to make a tuple.

And now, the results:

```
>>> c.fetchone()
(3L, 2L, 0L)
```

Quite unlike the `_mysql` example, this returns a single tuple, which is the row, and the values are properly converted by default... except... What's with the L's?

As mentioned earlier, while MySQL's INTEGER column translates perfectly into a Python integer, UNSIGNED INTEGER could overflow, so these values are converted to Python long integers instead.

If you wanted more rows, you could use `c.fetchmany(n)` or `c.fetchall()`. These do exactly what you think they do. On `c.fetchmany(n)`, the `n` is optional and defaults to `c.arraysize`, which is normally 1. Both of these methods return a sequence of rows, or an empty sequence if there are no more rows. If you use a weird cursor class, the rows themselves might not be tuples.

Note that in contrast to the above, `c.fetchone()` returns `None` when there are no more rows to fetch.

The only other method you are very likely to use is when you have to do a multi-row insert:

```
c.executemany(
      """INSERT INTO breakfast (name, spam, eggs, sausage, price)
      VALUES (%s, %s, %s, %s, %s)""",
      [
      ("Spam and Sausage Lover's Plate", 5, 1, 8, 7.95 ),
      ("Not So Much Spam Plate", 3, 2, 0, 3.95 ),
      ("Don't Wany ANY SPAM! Plate", 0, 4, 3, 5.95 )
      ] )
```

Here we are inserting three rows of five values. Notice that there is a mix of types (strings, ints, floats) though we still only use `%s`. And also note that we only included format strings for one row. MySQLdb picks those out and duplicates them for each row.

Using and extending

In general, it is probably wise to not directly interact with the DB API except for small applicatons. Databases, even SQL databases, vary widely in capabilities and may have non-standard features. The DB API does a good job of providing a reasonably portable interface but some methods are non-portable. Specifically, the parameters accepted by `connect()` are completely implementation-dependent.

If you believe your application may need to run on several different databases, the author recommends the following approach, based on personal experience: Write a simplified API for your application which implements the specific queries and operations your application needs to perform. Implement this API as a base class which should be have few database dependencies, and then derive a subclass from this which implements the necessary dependencies. In this way, porting your application to a new database should be a relatively simple matter of creating a new subclass, assuming the new database is reasonably standard.

Because MySQLdb's Connection and Cursor objects are written in Python, you can easily derive your own subclasses. There are several Cursor classes in MySQLdb.cursors:

BaseCursor
: The base class for Cursor objects. This does not raise Warnings.

CursorStoreResultMixIn
: Causes the Cursor to use the `mysql_store_result()` function to get the query result. The entire result set is stored on the client side.

CursorUseResultMixIn
: Causes the cursor to use the `mysql_use_result()` function to get the query result. The result set is stored on the server side and is transferred row by row using fetch operations.

CursorTupleRowsMixIn
: Causes the cursor to return rows as a tuple of the column values.

CursorDictRowsMixIn
: Causes the cursor to return rows as a dictionary, where the keys are column names and the values are column values. Note that if the column names are not unique, i.e., you are selecting from two tables that share column names, some of them will be rewritten as `table.column`. This can be avoided by using the SQL `AS` keyword. (This is yet-another reason not to use `*` in SQL queries, particularly where `JOIN` is involved.)

Cursor
: The default cursor class. This class is composed of `CursorWarningMixIn`, `CursorStoreResultMixIn`, `CursorTupleRowsMixIn`, and `BaseCursor`, i.e. it raises `Warning`, uses `mysql_store_result()`, and returns rows as tuples.

DictCursor
: Like `Cursor` except it returns rows as dictionaries.

SSCursor
: A "server-side" cursor. Like `Cursor` but uses `CursorUseResultMixIn`. Use only if you are dealing with potentially large result sets.

SSDictCursor
: Like `SSCursor` except it returns rows as dictionaries.

Embedded Server

Instead of connecting to a stand-alone server over the network, the embedded server support lets you run a full server right in your Python code or application server.

If you have built MySQLdb with embedded server support, there are two additional functions you will need to make use of:

server_init(args, groups)
: Initialize embedded server. If this client is not linked against the embedded server library, this function does nothing.

  args
  : sequence of command-line arguments

  groups
  : sequence of groups to use in defaults files

server_end()
: Shut down embedded server. If not using an embedded server, this does nothing.

See the MySQL documentation for more information on the embedded server.

Title:	MySQLdb: a Python interface for MySQL
Author:	Andy Dustman
Version:	$Revision: 421 $