# Contents

1 MySQLdb User's Guide 3  
1.1 Introduction .................................................. 3  
1.2 Installation .................................................. 4  
1.3 _mysql ....................................................... 4  
1.3.1 MySQL C API translation ................................. 4  
1.3.2 MySQL C API function mapping ......................... 4  
1.3.3 Some _mysql examples .................................. 5  
1.4 MySQLdb ..................................................... 7  
1.4.1 Functions and attributes ................................. 7  
1.4.2 Connection Objects ..................................... 9  
1.4.3 Cursor Objects ........................................... 9  
1.4.4 Some examples ........................................... 10  
1.5 Using and extending ........................................ 11  
1.6 Embedded Server ............................................ 12

2 MySQLdb Package 13  
2.1 MySQLdb Package ........................................... 13  
2.2 connections Module ........................................ 15  
2.3 converters Module .......................................... 17  
2.4 cursors Module .............................................. 18  
2.5 times Module ............................................... 18  
2.6 Subpackages ................................................ 19  
2.6.1 constants Package ...................................... 19

3 MySQLdb Frequently Asked Questions 21  
3.1 Build Errors ................................................ 21  
3.2 ImportError ............................................... 22  
3.3 My data disappeared! (or won’t go away!) ................ 22  
3.4 Other Errors ............................................... 23  
3.5 Other Resources ............................................ 23

4 Indices and tables 25

Python Module Index 27
Contents:
CHAPTER 1

MySQLdb User’s Guide

Contents

• MySQLdb User’s Guide
  – Introduction
  – Installation
  – _mysql
    * MySQL C API translation
    * MySQL C API function mapping
    * Some _mysql examples
  – MySQLdb
    * Functions and attributes
    * Connection Objects
    * Cursor Objects
    * Some examples
  – Using and extending
  – Embedded Server

1.1 Introduction

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

The `README` file has complete installation instructions.

### 1.3 `_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.

#### 1.3.1 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.

#### 1.3.2 MySQL C API function mapping

<table>
<thead>
<tr>
<th>C API</th>
<th><code>_mysql</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>mysql_affected_rows()</td>
<td>conn.affected_rows()</td>
</tr>
<tr>
<td>mysql_autocommit()</td>
<td>conn.autocommit()</td>
</tr>
<tr>
<td>mysql_character_set_name()</td>
<td>conn.character_set_name()</td>
</tr>
<tr>
<td>mysql_close()</td>
<td>conn.close()</td>
</tr>
<tr>
<td>mysql_commit()</td>
<td>conn.commit()</td>
</tr>
<tr>
<td>mysql_connect()</td>
<td><code>_mysql</code>.connect()</td>
</tr>
<tr>
<td>mysql_data_seek()</td>
<td>result.data_seek()</td>
</tr>
<tr>
<td>mysql_debug()</td>
<td><code>_mysql</code>.debug()</td>
</tr>
<tr>
<td>mysql_dump_debug_info()</td>
<td>conn.dump_debug_info()</td>
</tr>
<tr>
<td>mysql_escape_string()</td>
<td><code>_mysql</code>.escape_string()</td>
</tr>
<tr>
<td>mysql_fetch_row()</td>
<td>result.fetch_row()</td>
</tr>
<tr>
<td>mysql_get_character_set_info()</td>
<td>conn.get_character_set_info()</td>
</tr>
<tr>
<td>mysql_get_client_info()</td>
<td><code>_mysql</code>.get_client_info()</td>
</tr>
<tr>
<td>mysql_get_host_info()</td>
<td>conn.get_host_info()</td>
</tr>
<tr>
<td>mysql_get_proto_info()</td>
<td>conn.get_proto_info()</td>
</tr>
<tr>
<td>mysql_get_server_info()</td>
<td>conn.get_server_info()</td>
</tr>
<tr>
<td>mysql_info()</td>
<td>conn.info()</td>
</tr>
<tr>
<td>mysql_insert_id()</td>
<td>conn.insert_id()</td>
</tr>
<tr>
<td>mysql_num_fields()</td>
<td>result.num_fields()</td>
</tr>
<tr>
<td>mysql_num_rows()</td>
<td>result.num_rows()</td>
</tr>
<tr>
<td>mysql_options()</td>
<td><code>various options to </code>_mysql<code>.connect()</code></td>
</tr>
<tr>
<td>mysql_ping()</td>
<td><code>conn.ping()</code></td>
</tr>
</tbody>
</table>

Continued on next page
Table 1.1 – continued from previous page

<table>
<thead>
<tr>
<th>C API</th>
<th>_mysql</th>
</tr>
</thead>
<tbody>
<tr>
<td>mysql_query()</td>
<td>conn.query()</td>
</tr>
<tr>
<td>mysql_real_connect()</td>
<td>_mysql.connect()</td>
</tr>
<tr>
<td>mysql_real_query()</td>
<td>conn.query()</td>
</tr>
<tr>
<td>mysql_real_escape_string()</td>
<td>conn.escape_string()</td>
</tr>
<tr>
<td>mysql_rollback()</td>
<td>conn.rollback()</td>
</tr>
<tr>
<td>mysql_row_seek()</td>
<td>result.row_seek()</td>
</tr>
<tr>
<td>mysql_row_tell()</td>
<td>result.row_tell()</td>
</tr>
<tr>
<td>mysql_select_db()</td>
<td>conn.select_db()</td>
</tr>
<tr>
<td>mysql_set_character_set()</td>
<td>conn.set_character_set()</td>
</tr>
<tr>
<td>mysql_ssl_set()</td>
<td>ssl option to _mysql.connect()</td>
</tr>
<tr>
<td>mysql_stat()</td>
<td>conn.stat()</td>
</tr>
<tr>
<td>mysql_store_result()</td>
<td>conn.store_result()</td>
</tr>
<tr>
<td>mysql_thread_id()</td>
<td>conn.thread_id()</td>
</tr>
<tr>
<td>mysql_thread_safe_client()</td>
<td>conn.thread_safe_client()</td>
</tr>
<tr>
<td>mysql_use_result()</td>
<td>conn.use_result()</td>
</tr>
<tr>
<td>mysql_warning_count()</td>
<td>conn.warning_count()</td>
</tr>
<tr>
<td>CLIENT_*</td>
<td>MySQLdb.constants.CLIENT.*</td>
</tr>
<tr>
<td>CR_*</td>
<td>MySQLdb.constants.CR.*</td>
</tr>
<tr>
<td>ER_*</td>
<td>MySQLdb.constants.ER.*</td>
</tr>
<tr>
<td>FIELD_TYPE_*</td>
<td>MySQLdb.constants.FIELD_TYPE.*</td>
</tr>
<tr>
<td>FLAG_*</td>
<td>MySQLdb.constants.FLAG.*</td>
</tr>
</tbody>
</table>

### 1.3.3 Some _mysql examples

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

The simplest possible database connection is:

```python
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:

```python
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:

```python
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:

```python
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):
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:

```python
db=_mysql.connect(host="outhouse",db="thangs")
```

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()`:

```python
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:

```python
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:

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

\begin{verbatim}
my_conv = { FIELD_TYPE.LONG: int }
\end{verbatim}

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

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

### 1.4 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.

#### 1.4.1 Functions and attributes

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

\begin{verbatim}
connect(parameters...)
\end{verbatim}

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 \texttt{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:

- \texttt{host} name of host to connect to. Default: use the local host via a UNIX socket (where applicable)
- \texttt{user} user to authenticate as. Default: current effective user.
- \texttt{passwd} password to authenticate with. Default: no password.
- \texttt{db} database to use. Default: no default database.
- \texttt{port} TCP port of MySQL server. Default: standard port (3306).
- \texttt{unix_socket} location of UNIX socket. Default: use default location or TCP for remote hosts.
- \texttt{conv} type conversion dictionary. Default: a copy of MySQLdb.converters.conversions
- \texttt{compress} Enable protocol compression. Default: no compression.
- \texttt{connect_timeout} Abort if connect is not completed within given number of seconds. Default: no timeout (?)
- \texttt{named_pipe} Use a named pipe (Windows). Default: don’t.
- \texttt{init_command} Initial command to issue to server upon connection. Default: Nothing.
- \texttt{read_default_file} MySQL configuration file to read; see the MySQL documentation for \texttt{mysql_options()}.
- \texttt{read_default_group} Default group to read; see the MySQL documentation for \texttt{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 con-  
figuration (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, Un-  
supportedError 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’

threadsaftety 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.

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 distinguish 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.

### 1.4.2 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.

### 1.4.3 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 parameters are accessible.

The parameters are stored in the server as `@_*proname*_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.

### 1.4.4 Some examples

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

```python
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:

```python
c=db.cursor()
max_price=5
c.execute("\n\nSELECT spam, eggs, sausage FROM breakfast
WHERE price < %s\n\n", (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:

```python
>>> 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.fetchall(n) or c.fetchall(). These do exactly what you think they do. On c.fetchall(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 Want 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.

1.5 Using and extending

In general, it is probably wise to not directly interact with the DB API except for small applications. 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.
1.6 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** $Revision$
2.1 MySQLdb Package

MySQLdb - A DB API v2.0 compatible interface to MySQL.

This package is a wrapper around _mysql, which mostly implements the MySQL C API.

connect() – connects to server

See the C API specification and the MySQL documentation for more info on other items.

For information on how MySQLdb handles type conversion, see the MySQLdb.converters module.

MySQLdb.Binary(x)

MySQLdb.Connect(*args, **kwargs)
    Factory function for connections.Connection.

MySQLdb.Connection(*args, **kwargs)
    Factory function for connections.Connection.

MySQLdb.Date
    alias of date

MySQLdb.Time
    alias of time

MySQLdb.Timestamp
    alias of datetime

MySQLdb.DateFromTicks(ticks)
    Convert UNIX ticks into a date instance.

MySQLdb.TimeFromTicks(ticks)
    Convert UNIX ticks into a time instance.

MySQLdb.TimestampFromTicks(ticks)
    Convert UNIX ticks into a datetime instance.
exception MySQLdb.DataError
    Bases: _mysql_exceptions.DatabaseError
    Exception raised for errors that are due to problems with the processed data like division by zero, numeric value out of range, etc.

exception MySQLdb.DatabaseError
    Bases: _mysql_exceptions.Error
    Exception raised for errors that are related to the database.

exception MySQLdb.Error
    Bases: _mysql_exceptions.MySQLError
    Exception that is the base class of all other error exceptions (not Warning).

exception MySQLdb.IntegrityError
    Bases: _mysql_exceptions.DatabaseError
    Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails, duplicate key, etc.

exception MySQLdb.InterfaceError
    Bases: _mysql_exceptions.Error
    Exception raised for errors that are related to the database interface rather than the database itself.

exception MySQLdb.InternalError
    Bases: _mysql_exceptions.DatabaseError
    Exception raised when the database encounters an internal error, e.g. the cursor is not valid anymore, the transaction is out of sync, etc.

exception MySQLdb.MySQLError
    Bases: Exception
    Exception related to operation with MySQL.

exception MySQLdb.NotSupportedError
    Bases: _mysql_exceptions.DatabaseError
    Exception raised in case a method or database API was used which is not supported by the database, e.g. requesting a .rollback() on a connection that does not support transaction or has transactions turned off.

class MySQLdb.DBAPISet
    Bases: frozenset
    A special type of set for which A == x is true if A is a DBAPISet and x is a member of that set.

exception MySQLdb.OperationalError
    Bases: _mysql_exceptions.DatabaseError
    Exception raised for errors that are related to the database’s operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, a memory allocation error occurred during processing, etc.

exception MySQLdb.ProgrammingError
    Bases: _mysql_exceptions.DatabaseError
    Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc.

exception MySQLdb.Warning
    Bases: Warning, _mysql_exceptions.MySQLError
Exception raised for important warnings like data truncations while inserting, etc.

```python
MySQLdb.connect(*args, **kwargs)
```
Factory function for connections.Connection.

```python
MySQLdb.debug()
```
Does a DBUG_PUSH with the given string. mysql_debug() uses the Fred Fish debug library. To use this function, you must compile the client library to support debugging.

```python
MySQLdb.escape()
```
escape(obj, dict) – escape any special characters in object obj using mapping dict to provide quoting functions for each type. Returns a SQL literal string.

```python
MySQLdb.escape_dict()
```
escape_sequence(d, dict) – escape any special characters in dictionary d using mapping dict to provide quoting functions for each type. Returns a dictionary of escaped items.

```python
MySQLdb.escape_sequence()
```
escape_sequence(seq, dict) – escape any special characters in sequence seq using mapping dict to provide quoting functions for each type. Returns a tuple of escaped items.

```python
MySQLdb.escape_string()
```
escape_string(s) – quote any SQL-interpreted characters in string s.

Use connection.escape_string(s), if you use it at all. _mysql.escape_string(s) cannot handle character sets. You are probably better off using connection.escape(o) instead, since it will escape entire sequences as well as strings.

```python
MySQLdb.get_client_info()
```
get_client_info() – Returns a string that represents the client library version.

```python
MySQLdb.string_literal()
```
string_literal(obj) – converts object obj into a SQL string literal. This means, any special SQL characters are escaped, and it is enclosed within single quotes. In other words, it performs:

```
"%s" % escape_string(str(obj))
```

Use connection.string_literal(obj), if you use it at all. _mysql.string_literal(obj) cannot handle character sets.

## 2.2 connections Module

This module implements connections for MySQLdb. Presently there is only one class: Connection. Others are unlikely. However, you might want to make your own subclasses. In most cases, you will probably override Connection.default_cursor with a non-standard Cursor class.

```python
class MySQLdb.connections.Connection(*args, **kwargs)
```
Bases: _mysql.connection

MySQL Database Connection Object

```python
exception DataError
```
Bases: _mysql_exceptions.DatabaseError

Exception raised for errors that are due to problems with the processed data like division by zero, numeric value out of range, etc.

```python
exception DatabaseError
```
Bases: _mysql_exceptions.Error

Exception raised for errors that are related to the database.
exception Error
   Bases: _mysql_exceptions.MySQLError
   Exception that is the base class of all other error exceptions (not Warning).

exception IntegrityError
   Bases: _mysql_exceptions.DatabaseError
   Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails, duplicate key, etc.

exception InterfaceError
   Bases: _mysql_exceptions.Error
   Exception raised for errors that are related to the database interface rather than the database itself.

exception InternalError
   Bases: _mysql_exceptions.DatabaseError
   Exception raised when the database encounters an internal error, e.g. the cursor is not valid anymore, the transaction is out of sync, etc.

exception NotSupportedError
   Bases: _mysql_exceptions.DatabaseError
   Exception raised in case a method or database API was used which is not supported by the database, e.g. requesting a .rollback() on a connection that does not support transaction or has transactions turned off.

exception OperationalError
   Bases: _mysql_exceptions.DatabaseError
   Exception raised for errors that are related to the database’s operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, a memory allocation error occurred during processing, etc.

exception ProgrammingError
   Bases: _mysql_exceptions.DatabaseError
   Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc.

exception Warning
   Bases: Warning, _mysql_exceptions.MySQLError
   Exception raised for important warnings like data truncations while inserting, etc.

autocommit (on)

begin ()

cursor (cursorclass=None)
   Create a cursor on which queries may be performed. The optional cursorclass parameter is used to create the Cursor. By default, self.cursorclass=cursors.Cursor is used.

default_cursor
   alias of Cursor

errorhandler (connection, cursor, errorclass, errorvalue)
   If cursor is not None, (errorclass, errorvalue) is appended to cursor.messages; otherwise it is appended to connection.messages. Then errorclass is raised with errorvalue as the value.
   You can override this with your own error handler by assigning it to the instance.
literal(o)

If o is a single object, returns an SQL literal as a string. If o is a non-string sequence, the items of the sequence are converted and returned as a sequence.

Non-standard. For internal use; do not use this in your applications.

query(query)

set_character_set(charset)

Set the connection character set to charset. The character set can only be changed in MySQL-4.1 and newer. If you try to change the character set from the current value in an older version, NotSupportedError will be raised.

set_sql_mode(sql_mode)

Set the connection sql_mode. See MySQL documentation for legal values.

show_warnings()

Return detailed information about warnings as a sequence of tuples of (Level, Code, Message). This is only supported in MySQL-4.1 and up. If your server is an earlier version, an empty sequence is returned.

waiter = None

2.3 converters Module

MySQLdb type conversion module

This module handles all the type conversions for MySQL. If the default type conversions aren’t what you need, you can make your own. The dictionary conversions maps some kind of type to a conversion function which returns the corresponding value:

Key: FIELD_TYPE.* (from MySQLdb.constants)

Conversion function:

Arguments: string
Returns: Python object

Key: Python type object (from types) or class

Conversion function:

Arguments: Python object of indicated type or class AND conversion dictionary
Returns: SQL literal value

Notes: Most conversion functions can ignore the dictionary, but it is a required parameter. It is necessary for converting things like sequences and instances.

Don’t modify conversions if you can avoid it. Instead, make copies (with the copy() method), modify the copies, and then pass them to MySQL.connect().

MySQLdb.converters.Bool2Str(s, d)
MySQLdb.converters.Float2Str(o, d)
MySQLdb.converters.None2NULL(o, d)
    Convert None to NULL.
MySQLdb.converters.Set2Str(s, d)
MySQLdb.converters.Str2Set(s)
MySQLdb.converters.Thing2Literal(o, d)
Convert something into a SQL string literal. If using MySQL-3.23 or newer, string_literal() is a method of the
_mysql.MYSQL object, and this function will be overridden with that method when the connection is created.

MySQLdb.converters.Thing2Str(s, d)
Convert something into a string via str().

MySQLdb.converters.Unicode2Str(s, d)
Convert a unicode object to a string using the default encoding. This is only used as a placeholder for the real
function, which is connection-dependent.

MySQLdb.converters.array2Str(o, d)
MySQLdb.converters.char_array(s)
MySQLdb.converters.quote_tuple(t, d)

2.4 cursors Module

MySQLdb Cursors
This module implements Cursors of various types for MySQLdb. By default, MySQLdb uses the Cursor class.

class MySQLdb.cursors.Cursor(connection)
Bases: MySQLdb.cursors.CursorStoreResultMixIn, MySQLdb.cursors.CursorTupleRowsMixIn, MySQLdb.cursors.BaseCursor

This is the standard Cursor class that returns rows as tuples and stores the result set in the client.

2.5 times Module

times module
This module provides some Date and Time classes for dealing with MySQL data.

Use Python datetime module to handle date and time columns.

MySQLdb.times.DateFromTicks(ticks)
Convert UNIX ticks into a date instance.

MySQLdb.times.DateTime2literal(d, c)
Format a DateTime object as an ISO timestamp.

MySQLdb.times.DateTimeDelta2literal(d, c)
Format a DateTimeDelta object as a time.

MySQLdb.times.DateTime_or_None(s)
MySQLdb.times.Date_or_None(s)
MySQLdb.times.TimeDelta_or_None(s)
MySQLdb.times.TimeFromTicks(ticks)
Convert UNIX ticks into a time instance.

MySQLdb.times.Time_or_None(s)
MySQLdb.times.TimestampFromTicks(ticks)
Convert UNIX ticks into a datetime instance.

MySQLdb.times.format_TIMEDELTA(v)
MySQLdb times \texttt{format\_TIMESTAMP} \((d)\)

MySQLdb times \texttt{mysql\_timestamp\_converter} \((s)\)

Convert a MySQL TIMESTAMP to a Timestamp object.

\section*{2.6 Subpackages}

\subsection*{2.6.1 constants Package}

\textbf{constants Package}

\textbf{CLIENT Module}

MySQL CLIENT constants

These constants are used when creating the connection. Use bitwise-OR (\(|\)) to combine options together, and pass them as the client_flags parameter to MySQLdb.Connection. For more information on these flags, see the MySQL C API documentation for mysql_real_connect().

\textbf{CR Module}

MySQL Connection Errors

Nearly all of these raise OperationalError. COMMANDS_OUT_OF_SYNC raises ProgrammingError.

\textbf{ER Module}

MySQL ER Constants

These constants are error codes for the bulk of the error conditions that may occur.

\textbf{FIELD\_TYPE Module}

MySQL FIELD\_TYPE Constants

These constants represent the various column (field) types that are supported by MySQL.

\textbf{FLAG Module}

MySQL FLAG Constants

These flags are used along with the FIELD\_TYPE to indicate various properties of columns in a result set.

\textbf{REFRESH Module}

MySQL REFRESH Constants

These constants seem to mostly deal with things internal to the MySQL server. Forget you saw this.
MySQLdb Frequently Asked Questions

3.1 Build Errors

ld: fatal: library -lmysqlclient_r: not found

mysqlclient_r is the thread-safe library. It’s not available on all platforms, or all installations, apparently. You’ll need to reconfigure site.cfg (in MySQLdb-1.2.1 and newer) to have threadsafe = False.

mysql.h: No such file or directory

This almost always mean you don’t have development packages installed. On some systems, C headers for various things (like MySQL) are distributed as a separate package. You’ll need to figure out what that is and install it, but often the name ends with -devel.

Another possibility: Some older versions of mysql_config behave oddly and may throw quotes around some of the path names, which confused MySQLdb-1.2.0. 1.2.1 works around these problems. If you see things like -I’/usr/local/include/mysql’ in your compile command, that’s probably the issue, but it shouldn’t happen any more.
3.2 ImportError

ImportError: No module named _mysql

If you see this, it’s likely you did some wrong when installing MySQLdb; re-read (or read) README. _mysql is the low-level C module that interfaces with the MySQL client library.

Various versions of MySQLdb in the past have had build issues on “weird” platforms; “weird” in this case means “not Linux”, though generally there aren’t problems on Unix/POSIX platforms, including BSDs and Mac OS X. Windows has been more problematic, in part because there is no mysql_config available in the Windows installation of MySQL. 1.2.1 solves most, if not all, of these problems, but you will still have to edit a configuration file so that the setup knows where to find MySQL and what libraries to include.

ImportError: libmysqlclient_r.so.14: cannot open shared object file: No such file or directory

The number after .so may vary, but this means you have a version of MySQLdb compiled against one version of MySQL, and are now trying to run it against a different version. The shared library version tends to change between major releases.

Solution: Rebuilt MySQLdb, or get the matching version of MySQL.

Another thing that can cause this: The MySQL libraries may not be on your system path.

Solutions:

• set the LD_LIBRARY_PATH environment variable so that it includes the path to the MySQL libraries.
• set static=True in site.cfg for static linking
• reconfigure your system so that the MySQL libraries are on the default loader path. In Linux, you edit /etc/ld.so.conf and run ldconfig. For Solaris, see Linker and Libraries Guide.

ImportError: ld.so.1: python: fatal: libmtmalloc.so.1: DF_1_NOOPEN tagged object may not be dlopen()ed

This is a weird one from Solaris. What does it mean? I have no idea. However, things like this can happen if there is some sort of a compiler or environment mismatch between Python and MySQL. For example, on some commercial systems, you might have some code compiled with their own compiler, and other things compiled with GCC. They don’t always mesh together. One way to encounter this is by getting binary packages from different vendors.

Solution: Rebuild Python or MySQL (or maybe both) from source.

ImportError: dlopen(_mysql.so, 2): Symbol not found: _sprintf$LDBLStub Referenced from:
_/mysql.so Expected in: dynamic lookup

This is one from Mac OS X. It seems to have been a compiler mismatch, but this time between two different versions of GCC. It seems nearly every major release of GCC changes the ABI in some way, so linking code compiled with GCC-3.3 and GCC-4.0, for example, can be problematic.

3.3 My data disappeared! (or won’t go away!)

Starting with 1.2.0, MySQLdb disables autocommit by default, as required by the DB-API standard (PEP-249). If you are using InnoDB tables or some other type of transactional table type, you’ll need to do connection.commit() before closing the connection, or else none of your changes will be written to the database.

Conversely, you can also use connection.rollback() to throw away any changes you’ve made since the last commit.

Important note: Some SQL statements – specifically DDL statements like CREATE TABLE – are non-transactional, so they can’t be rolled back, and they cause pending transactions to commit.
3.4 Other Errors

OperationalError: (1251, 'Client does not support authentication protocol requested by server; consider upgrading MySQL client')

This means your server and client libraries are not the same version. More specifically, it probably means you have a 4.1 or newer server and 4.0 or older client. You can either upgrade the client side, or try some of the workarounds in Password Hashing as of MySQL 4.1.

3.5 Other Resources

- Help forum. Please search before posting.
- Google
- READ README!
- Read the User’s Guide
- Read PEP-249
CHAPTER 4

Indices and tables

- genindex
- modindex
- search
m
MySQLdb, 13
MySQLdb.connections, 15
MySQLdb.constants, 19
MySQLdb.constants.CLIENT, 19
MySQLdb.constants.CR, 19
MySQLdb.constants.ER, 19
MySQLdb.constants.FIELD_TYPE, 19
MySQLdb.constants.FLAG, 19
MySQLdb.constants.REFRESH, 19
MySQLdb.converters, 17
MySQLdb.cursors, 18
MySQLdb.times, 18
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>array2Str() (in module MySQLdb.converters), 18</td>
<td>autocommit() (MySQLdb.connections.Connection method), 16</td>
<td>begin() (MySQLdb.connections.Connection method), 16</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Binary() (in module MySQLdb), 13</td>
<td>char_array() (in module MySQLdb.converters), 18</td>
<td>Connect() (in module MySQLdb), 13</td>
</tr>
<tr>
<td>Bool2Str() (in module MySQLdb.converters), 17</td>
<td>Connect() (in module MySQLdb), 13</td>
<td>connect() (in module MySQLdb), 15</td>
</tr>
<tr>
<td>Connection (class in MySQLdb.connections), 15</td>
<td>Connection (class in MySQLdb.connections), 15</td>
<td>Connection (class in MySQLdb.connections), 15</td>
</tr>
<tr>
<td>Connection() (in module MySQLdb), 13</td>
<td>Connection() (in module MySQLdb), 13</td>
<td>Connection.DatabaseError, 15</td>
</tr>
<tr>
<td>Connection.DatabaseError, 15</td>
<td>Connection.DatabaseError, 15</td>
<td>Connection.DataError, 15</td>
</tr>
<tr>
<td>Connection.Error, 15</td>
<td>Connection.Error, 15</td>
<td>Connection.Error, 15</td>
</tr>
<tr>
<td>Connection.IntegrityError, 16</td>
<td>Connection.InterfaceError, 16</td>
<td>Connection.InterfaceError, 16</td>
</tr>
<tr>
<td>Connection.InternalError, 16</td>
<td>Connection.NotSupportedError, 16</td>
<td>Connection.NotSupportedError, 16</td>
</tr>
<tr>
<td>Connection.OperationalError, 16</td>
<td>Connection.ProgrammingError, 16</td>
<td>Connection.OperationalError, 16</td>
</tr>
<tr>
<td>Connection.Warning, 16</td>
<td>Cursor (class in MySQLdb.cursors), 18</td>
<td>cursor() (MySQLdb.connections.Connection method), 16</td>
</tr>
<tr>
<td>Cursor (class in MySQLdb.cursors), 18</td>
<td>DatabaseError, 14</td>
<td>Debug() (module MySQLdb), 14</td>
</tr>
<tr>
<td>Literal() (MySQLdb.connections.Connection method), 16</td>
<td>DataError, 13</td>
<td>Error, 14</td>
</tr>
<tr>
<td>Literal() (MySQLdb.connections.Connection method), 16</td>
<td>Date (module MySQLdb), 13</td>
<td>Errorhandler() (MySQLdb.connections.Connection method), 16</td>
</tr>
<tr>
<td>Literal() (module MySQLdb.times), 18</td>
<td>Date_or_None() (module MySQLdb.times), 18</td>
<td>Escapesh() (module MySQLdb), 15</td>
</tr>
<tr>
<td>Literal() (module MySQLdb.times), 18</td>
<td>Datetime2literal() (module MySQLdb.times), 18</td>
<td>Escapesh_dict() (module MySQLdb), 15</td>
</tr>
<tr>
<td>Literal() (module MySQLdb.times), 18</td>
<td>Datetime2literal() (module MySQLdb.times), 18</td>
<td>Escapesh_sequence() (module MySQLdb), 15</td>
</tr>
<tr>
<td>Literal() (module MySQLdb.times), 18</td>
<td>Datetime2literal() (module MySQLdb.times), 18</td>
<td>Escapesh_string() (module MySQLdb), 15</td>
</tr>
<tr>
<td>Literal() (module MySQLdb.times), 18</td>
<td>Latest() (module MySQLdb), 13</td>
<td>F</td>
</tr>
<tr>
<td>Literal() (module MySQLdb.times), 18</td>
<td>Mysql_timestamp_converter() (module MySQLdb.times), 19</td>
<td>F</td>
</tr>
<tr>
<td>Literal() (module MySQLdb.times), 18</td>
<td>Friends() (module MySQLdb), 13</td>
<td>Forexample() (module MySQLdb), 13</td>
</tr>
<tr>
<td>Literal() (module MySQLdb.times), 18</td>
<td>Mysqldb (module), 13</td>
<td>Forexample() (module MySQLdb), 13</td>
</tr>
<tr>
<td>Literal() (module MySQLdb.times), 18</td>
<td>Mysqldb.connections (module), 15</td>
<td>Forexample() (module MySQLdb), 13</td>
</tr>
<tr>
<td>Literal() (module MySQLdb.times), 18</td>
<td>Mysqldb.constants (module), 19</td>
<td>Forexample() (module MySQLdb), 13</td>
</tr>
<tr>
<td>Literal() (module MySQLdb.times), 18</td>
<td>Mysqldb.constants.CLIENT (module), 19</td>
<td>Forexample() (module MySQLdb), 13</td>
</tr>
<tr>
<td>Literal() (module MySQLdb.times), 18</td>
<td>Mysqldb.constants.CR (module), 19</td>
<td>Forexample() (module MySQLdb), 13</td>
</tr>
<tr>
<td>Literal() (module MySQLdb.times), 18</td>
<td>Mysqldb.constants.ER (module), 19</td>
<td>Forexample() (module MySQLdb), 13</td>
</tr>
</tbody>
</table>

---

29
MySQLdb.constants.FIELD_TYPE (module), 19
MySQLdb.constants.FLAG (module), 19
MySQLdb.constants.REFRESH (module), 19
MySQLdb.converters (module), 17
MySQLdb.cursors (module), 18
MySQLdb.times (module), 18
MySQLdb.Error, 14

N
None2NULL() (in module MySQLdb.converters), 17
NotSupportedError, 14

O
OperationalError, 14

P
ProgrammingError, 14

Q
query() (MySQLdb.connections.Connection method), 17
quote_tuple() (in module MySQLdb.converters), 18

S
Set2Str() (in module MySQLdb.converters), 17
set_character_set() (MySQLdb.connections.Connection method), 17
set_sql_mode() (MySQLdb.connections.Connection method), 17
show_warnings() (MySQLdb.connections.Connection method), 17
Str2Set() (in module MySQLdb.converters), 17
string_literal() (in module MySQLdb), 15

T
Thing2Literal() (in module MySQLdb.converters), 17
Thing2Str() (in module MySQLdb.converters), 18
Time (in module MySQLdb), 13
Time_or_None() (in module MySQLdb.times), 18
TimeDelta_or_None() (in module MySQLdb.times), 18
TimeFromTicks() (in module MySQLdb), 13
TimeFromTicks() (in module MySQLdb.times), 18
Timestamp (in module MySQLdb), 13
TimestampFromTicks() (in module MySQLdb), 13
TimestampFromTicks() (in module MySQLdb.times), 18

U
Unicode2Str() (in module MySQLdb.converters), 18

W
waiter (MySQLdb.connections.Connection attribute), 17
Warning, 14