MySQL Connector/Python

MySQL Connector/Python

Abstract

This manual describes how to install, configure, and develop database applications using MySQL Connector/Python, a self-contained Python driver for communicating with MySQL servers.

Document generated on: 2012-09-14 (revision: 32235)

Table of Contents

Preface and Legal Notices

This manual describes how to install, configure, and develop database applications using MySQL Connector/Python, the a self-contained Python driver for communicating with MySQL servers.

Legal Notices

only. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this material is subject to the terms and conditions of your Oracle Software License and Service Agreement, which has been executed and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle or as specifically provided below. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

This documentation is NOT distributed under a GPL license. Use of this documentation is subject to the following terms:

You may create a printed copy of this documentation solely for your own personal use. Conversion to other formats is allowed as long as the actual content is not altered or edited in any way. You shall not publish or distribute this documentation in any form or on any media, except if you distribute the documentation in a manner similar to how Oracle disseminates it (that is, electronically for download on a Web site with the software) or on a CD-ROM or similar medium, provided however that the documentation is disseminated together with the software on the same medium. Any other use, such as any dissemination of printed copies or use of this documentation, in whole or in part, in another publication, requires the prior written consent from an authorized representative of Oracle. Oracle and/or its affiliates reserve any and all rights to this documentation not expressly granted above.

For more information on the terms of this license, or for details on how the MySQL documentation is built and produced, please visit MySQL Contact & Questions.

For additional licensing information, including licenses for third-party libraries used by MySQL products, see Preface and Legal Notices.

For help with using MySQL, please visit either the MySQL Forums or MySQL Mailing Lists where you can discuss your issues with other MySQL users.

For additional documentation on MySQL products, including translations of the documentation into other languages, and downloadable versions in variety of formats, including HTML and PDF formats, see the MySQL Documentation Library.

Chapter 1. MySQL Connector/Python

MySQL Connector/Python allows Python programs to access MySQL databases, using an API that is compliant with the Python DB API version 2.0. It is written in pure Python and does not have any dependencies except for the Python Standard Library.

MySQL Connector/Python includes support for:

- Almost all features provided by MySQL Server up to and including MySQL Server version 5.5.
- Converting parameter values back and forth between Python and MySQL data types, for example Python `datetime` and MySQL `DATETIME`. You can turn automatic conversion on for convenience, or off for optimal performance.
- All MySQL extensions to standard SQL syntax.
- Protocol compression, which enables compressing the data stream between the client and server.
- Connections using TCP/IP sockets and on Unix using Unix sockets.
- Secure TCP/IP connections using SSL.
- Self-contained driver. Connector/Python does not require the MySQL client library or any Python modules outside the standard library.

MySQL Connector/Python supports from Python version 2.4 through 2.7, and Python 3.1 and later. Note that Connector/Python does not support the old MySQL Server authentication methods, which means that MySQL versions prior to 4.1 will not work.

Chapter 2. Connector/Python Versions

MySQL Connector/Python v1.0.x series is going through a series of beta releases, leading to the first generally available (GA) version (not released yet). Any development releases prior to general availability will not be supported once the GA version is released.

The following table summarizes the available Connector/Python versions:

Connector/Python version	MySQL Server version	Python version	Status
1.0	5.6, 5.5 (5.1, 5.0, 4.1)	2.7, 2.6 (2.5, 2.4); 3.1 and later	Recommended version

> **Note**
>
> MySQL server and Python versions within brackets are known to work with Connector/Python, but are not officially supported. Bugs might not get fixed for those versions.

Chapter 3. Connector/Python Installation

Table of Contents

Connector/Python runs on any platform where Python is installed. Python comes pre-installed on almost any Linux distribution or UNIX-like system such as Apple Mac OS X and FreeBSD. On Microsoft Windows systems, you can install Python using the installer found on the Python Download website.

Connector/Python is a pure Python implementation of the MySQL Client/Server protocol, meaning it does not require any other MySQL client libraries or other components. It also has no third-party dependencies. If you need SSL support, verify that your Python installation has been compiled using the OpenSSL libraries.

The installation of Connector/Python is similar on every platform and follows the standard Python Distribution Utilities or Distutils. Some platforms have specific packaging, for example RPM, and, when made available, the installation of these will be covered in this manual.

Python terminology regarding distributions:

- **Source Distribution** is a distribution that contains only source files and is generally platform independent.
- **Built Distribution** can be regarded as a binary package. It contains both sources and platform-independent bytecode.

Installing Connector/Python Source Distribution on Linux, UNIX, or OS X

On UNIX-like systems such as Linux distributions, Solaris, Apple Mac OS X, and FreeBSD, you can download Connector/Python as a `tar` archive from http://dev.mysql.com/downloads/connector/python/.

To install Connector/Python from the `.tar.gz` file, download the latest version and follow these steps:

```
shell> gunzip mysql-connector-python-1.0.6b1.tar.gz
shell> tar xf mysql-connector-python-1.0.6b1.tar
shell> cd mysql-connector-python-1.0.6b1
shell> sudo python setup.py install
```

On UNIX-like systems, Connector/Python gets installed in the default location `/prefix/lib/pythonX.Y/site-packages/`, where *`prefix`* is the location where Python was installed and *`X.Y`* is the version of Python. See How installation works in the Python manual.

If you are not sure where Connector/Python was installed, do the following to retrieve the location:

```
shell> python
>>> from distutils.sysconfig import get_python_lib
>>> print get_python_lib()            # Python v2.x
/Library/Python/2.7/site-packages
>>> print(get_python_lib())           # Python v3.x
/Library/Frameworks/Python.framework/Versions/3.1/lib/python3.1/site-packages
```

> **Note**
>
> The above example shows the default installation location on Mac OS X 10.7.

Installing Connector/Python Source Distribution on Microsoft Windows

On Microsoft Windows systems, you can download Connector/Python as a `zip` archive from http://dev.mysql.com/downloads/connector/python/.

Make sure that the Python executable is available in the Windows `%PATH%` setting. For more information about installation and configuration of Python on Windows, see the section Using Python on Windows in the Python documentation.

To install Connector/Python from the `.zip` file, download the latest version and follow these steps:

1. Unpack the downloaded `zip` archive into a directory of your choice. For example, into the folder `C:\mysql-connector\`. Use the appropriate unzip command for your system, for example, `unzip`, `pkunzip`, and so on.

2. Start a console window (or a DOS window) and change to the folder where you unpacked the Connector/Python `zip` archive.

   ```
   shell> cd C:\mysql-connector\
   ```

3. Once inside the Connector/Python folder, do the following:

   ```
   shell> python setup.py install
   ```

On Windows, Connector/Python gets installed in the default location `C:\PythonX.Y\Lib\site-packages\` where *`X.Y`* is the Python version you used to install the connector.

If you are not sure where Connector/Python ended up, do the following to retrieve the location where packages get installed:

```
shell> python
>>> from distutils.sysconfig import get_python_lib
>>> print get_python_lib()            # Python v2.x
>>> print(get_python_lib())           # Python v3.x
```

Verifying Your Connector/Python Installation

To test that your Connector/Python installation is working and is able to connect to a MySQL database server, you can run a very simple program where you substitute the login credentials and host information of the MySQL server. See Connecting to MySQL Using Connector/Python for an example.

Chapter 4. Connector/Python Coding Examples

Table of Contents

These coding examples illustrate how to develop Python applications and scripts which connect to a MySQL Server using MySQL Connector/Python.

Connecting to MySQL Using Connector/Python

The `connect()` constructor is used for creating a connection to the MySQL server and returns a `MySQLConnection` object.

The following example shows how to connect to the MySQL server:

```
import mysql.connector
cnx = mysql.connector.connect(user='scott', password='tiger',
                              host='127.0.0.1',
                              database='employees')
cnx.close()
```

See Chapter 6, *Connector/Python Connection Arguments* for all possible connection arguments.

It is also possible to create connection objects using the connection.MySQLConnection() class. Both methods, using the `connect()` constructor, or the class directly, are valid and functionally equal, but using `connector()` is preferred and will be used in most examples in this manual.

To handle connection errors, use the `try` statement and catch all errors using the errors.Error exception:

```
import mysql.connector
from mysql.connector import errorcode
try:
  cnx = mysql.connector.connect(user='scott',
                                database='testt')
except mysql.connector.Error as err:
  if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
    print("Something is wrong your username or password")
  elif err.errno == errorcode.ER_BAD_DB_ERROR:
    print("Database does not exists")
  else:
    print(err)
else:
  cnx.close()
```

If you have lots of connection arguments, it's best to keep them in a dictionary and use the `**`-operator. Here is an example:

```
import mysql.connector
config = {
  'user': 'scott',
  'password': 'tiger',
  'host': '127.0.0.1',
  'database': 'employees',
  'raise_on_warnings': True,
```

```
}
cnx = mysql.connector.connect(**config)
cnx.close()
```

Creating Tables Using Connector/Python

All DDL (Data Definition Language) statements are executed using a handle structure known as a cursor. The following examples show how to create the tables of the `employees` database. You will need them for the other examples.

In a MySQL server, tables are very long-lived objects, and are often accessed by multiple applications written in different languages. You might typically work with tables that are already set up, rather than creating them within your own application. Avoid setting up and dropping tables over and over again, as that is an expensive operation. The exception is temporary tables, which can be created and dropped quickly within an application.

```
from __future__ import print_function
import mysql.connector
from mysql.connector import errorcode
DB_NAME = 'employees'
TABLES = {}
TABLES['employees'] = (
    "CREATE TABLE `employees` ("
    "  `emp_no` int(11) NOT NULL AUTO_INCREMENT,"
    "  `birth_date` date NOT NULL,"
    "  `first_name` varchar(14) NOT NULL,"
    "  `last_name` varchar(16) NOT NULL,"
    "  `gender` enum('M','F') NOT NULL,"
    "  `hire_date` date NOT NULL,"
    "  PRIMARY KEY (`emp_no`)"
    ") ENGINE=InnoDB")
TABLES['departments'] = (
    "CREATE TABLE `departments` ("
    "  `dept_no` char(4) NOT NULL,"
    "  `dept_name` varchar(40) NOT NULL,"
    "  PRIMARY KEY (`dept_no`), UNIQUE KEY `dept_name` (`dept_name`)"
    ") ENGINE=InnoDB")
TABLES['salaries'] = (
    "CREATE TABLE `salaries` ("
    "  `emp_no` int(11) NOT NULL,"
    "  `salary` int(11) NOT NULL,"
    "  `from_date` date NOT NULL,"
    "  `to_date` date NOT NULL,"
    "  PRIMARY KEY (`emp_no`,`from_date`), KEY `emp_no` (`emp_no`),"
    "  CONSTRAINT `salaries_ibfk_1` FOREIGN KEY (`emp_no`) "
    "     REFERENCES `employees` (`emp_no`) ON DELETE CASCADE"
    ") ENGINE=InnoDB")
TABLES['dept_emp'] = (
    "CREATE TABLE `dept_emp` ("
    "  `emp_no` int(11) NOT NULL,"
    "  `dept_no` char(4) NOT NULL,"
    "  `from_date` date NOT NULL,"
    "  `to_date` date NOT NULL,"
    "  PRIMARY KEY (`emp_no`,`dept_no`), KEY `emp_no` (`emp_no`),"
    "  KEY `dept_no` (`dept_no`),"
    "  CONSTRAINT `dept_emp_ibfk_1` FOREIGN KEY (`emp_no`) "
    "     REFERENCES `employees` (`emp_no`) ON DELETE CASCADE,"
    "  CONSTRAINT `dept_emp_ibfk_2` FOREIGN KEY (`dept_no`) "
    "     REFERENCES `departments` (`dept_no`) ON DELETE CASCADE"
    ") ENGINE=InnoDB")
TABLES['dept_manager'] = (
    "  CREATE TABLE `dept_manager` ("
    "  `dept_no` char(4) NOT NULL,"
    "  `emp_no` int(11) NOT NULL,"
```

```
    "  `from_date` date NOT NULL,"
    "  `to_date` date NOT NULL,"
    "  PRIMARY KEY (`emp_no`,`dept_no`),"
    "  KEY `emp_no` (`emp_no`),"
    "  KEY `dept_no` (`dept_no`),"
    "  CONSTRAINT `dept_manager_ibfk_1` FOREIGN KEY (`emp_no`) "
    "     REFERENCES `employees` (`emp_no`) ON DELETE CASCADE,"
    "  CONSTRAINT `dept_manager_ibfk_2` FOREIGN KEY (`dept_no`) "
    "     REFERENCES `departments` (`dept_no`) ON DELETE CASCADE"
    ") ENGINE=InnoDB")
TABLES['titles'] = (
    "CREATE TABLE `titles` ("
    "  `emp_no` int(11) NOT NULL,"
    "  `title` varchar(50) NOT NULL,"
    "  `from_date` date NOT NULL,"
    "  `to_date` date DEFAULT NULL,"
    "  PRIMARY KEY (`emp_no`,`title`,`from_date`), KEY `emp_no` (`emp_no`),"
    "  CONSTRAINT `titles_ibfk_1` FOREIGN KEY (`emp_no`)"
    "     REFERENCES `employees` (`emp_no`) ON DELETE CASCADE"
    ") ENGINE=InnoDB")
```

The above code shows how we are storing the `CREATE` statements in a Python dictionary called `TABLES`. We also define the database in a global variable called `DB_NAME`, which allows you to easily use a different schema.

```
cnx = mysql.connector.connect(user='scott')
cursor = cnx.cursor()
```

A single MySQL server can contain multiple databases. Typically, you specify the database to switch to when connecting to the MySQL server. This example does not connect to the database upon connection, so that it can make sure the database exists, and create it if not.

```
def create_database(cursor):
    try:
        cursor.execute(
            "CREATE DATABASE {} DEFAULT CHARACTER SET 'utf8'".format(DB_NAME))
    except mysql.connector.Error as err:
        print("Failed creating database: {}".format(err))
        exit(1)
try:
    cnx.database = DB_NAME
except mysql.connector.Error as err:
    if err.errno == errorcode.ER_BAD_DB_ERROR:
        create_database(cursor)
        cnx.database = DB_NAME
    else:
        print(err)
        exit(1)
```

We first try to change to a particular database using the `database` property of the connection object `cnx`. If there is an error, we examine the error number to check if the database does not exist. If so, we call the `create_database` function to create it for us.

On any other error, the application exits and displays the error message.

```
for name, ddl in TABLES.iteritems():
    try:
        print("Creating table {}: ".format(name), end='')
        cursor.execute(ddl)
    except mysql.connector.Error as err:
        if err.errno == errorcode.ER_TABLE_EXISTS_ERROR:
            print("already exists.")
        else:
            print(err.errmsg)
    else:
```

```
        print("OK")
cursor.close()
cnx.close()
```

After we succesfully created or changed to the target database, we create the tables by iterating over the items of the `TABLES` dictionary.

We handle the error when the table already exists by simply notifying the user that it was already there. Other errors are printed, but we simply continue creating tables. (We show how to handle the “table already exists” condition for illustration purposes. In a real application, we would typically avoid the error condition entirely by using the `IF NOT EXISTS` clause of the `CREATE TABLE` statement.)

The output would be something like this:

```
Creating table employees: already exists.
Creating table salaries: already exists.
Creating table titles: OK
Creating table departments: already exists.
Creating table dept_manager: already exists.
Creating table dept_emp: already exists.
```

To populate the employees tables, use the dump files of the Employee Sample Database. Note that you only need the data dump files that you will find in an archive named like `employees_db-dump-files-1.0.5.tar.bz2`. After downloading the dump files, do the following from the command line, adding connection options to the `mysql` commands if necessary:

```
shell> tar xzf employees_db-dump-files-1.0.5.tar.bz2
shell> cd employees_db
shell> mysql employees < load_employees.dump
shell> mysql employees < load_titles.dump
shell> mysql employees < load_departments.dump
shell> mysql employees < load_salaries.dump
shell> mysql employees < load_dept_emp.dump
shell> mysql employees < load_dept_manager.dump
```

Inserting Data Using Connector/Python

Inserting or updating data is also done using the handler structure known as a cursor. When you use a transactional storage engine such as `InnoDB` (which is the default in MySQL 5.5 and later), you must commit the data after a sequence of `INSERT`, `DELETE`, and `UPDATE` statements.

In this example we show how to insert new data. The second `INSERT` depends on the value of the newly created primary key of the first. We are also demonstrating how to use extended formats. The task is to add a new employee starting to work tomorrow with a salary set to 50000.

> **Note**
>
> The following example uses tables created in the example Creating Tables Using Connector/Python. The `AUTO_INCREMENT` column option for the primary key of the `employees` table is important to ensure reliable, easily searchable data.

```
from __future__ import print_function
from datetime import date, datetime, timedelta
import mysql.connector
cnx = mysql.connector.connect(user='scott', database='employees')
cursor = cnx.cursor()
tomorrow = datetime.now().date() + timedelta(days=1)
add_employee = ("INSERT INTO employees "
                "(first_name, last_name, hire_date, gender, birth_date) "
                "VALUES (%s, %s, %s, %s, %s)")
add_salary = ("INSERT INTO salaries "
```

```
              "(emp_no, salary, from_date, to_date) "
              "VALUES (%(emp_no)s, %(salary)s, %(from_date)s, %(to_date)s)")
data_employee = ('Geert', 'Vanderkelen', tomorrow, 'M', date(1977, 6, 14))
# Insert new employee
cursor.execute(add_employee, data_employee)
emp_no = cursor.lastrowid
# Insert salary information
data_salary = {
  'emp_no': emp_no,
  'salary': 50000,
  'from_date': tomorrow,
  'to_date': date(9999, 1, 1),
}
cursor.execute(add_salary, data_salary)
# Make sure data is committed to the database
cnx.commit()
cursor.close()
cnx.close()
```

We first open a connection to the MySQL server and store the connection object in the variable `cnx`. We then create a new cursor, by default a MySQLCursor object, using the connection's `cursor()` method.

We could calculate tomorrow by calling a database function, but for clarity we do it in Python using the `datetime` module.

Both `INSERT` statements are stored in the variables called `add_employee` and `add_salary`. Note that the second `INSERT` statement uses extended Python format codes.

The information of the new employee is stored in the tuple `data_employee`. The query to insert the new employee is executed and we retrieve the newly inserted value for the column `emp_no` using the `lastrowid` property of the cursor object.

Next, we insert the new salary for the new employee. We are using the `emp_no` variable in the directory holding the data. This directory is passed to the `execute()` method of the cursor object.

Since by default Connector/Python turns autocommit off, and MySQL 5.5 and later uses transactional `InnoDB` tables by default, it is necessary to commit your changes using the connection's `commit()` method. You could also roll back using the `rollback()` method.

Querying Data Using Connector/Python

The following example shows how to query data using a cursor created using the connection's `cursor()` method. The data returned is formatted and printed on the console.

The task is to select all employees hired in the year 1999 and print their names with their hire date to the console.

```
import datetime
import mysql.connector
cnx = mysql.connector.connect(user='scott', database='employees')
cursor = cnx.cursor()
query = ("SELECT first_name, last_name, hire_date FROM employees "
         "WHERE hire_date BETWEEN %s AND %s")
hire_start = datetime.date(1999, 1, 1)
hire_end = datetime.date(1999, 12, 31)
cursor.execute(query, (hire_start, hire_end))
for (first_name, last_name, hire_date) in cursor:
  print("{}, {} was hired on {:%d %b %Y}".format(
    last_name, first_name, hire_date))
cursor.close()
cnx.close()
```

We first open a connection to the MySQL server and store the connection object in the variable `cnx`. We then create a new cursor, by default a MySQLCursor object, using the connection's `cursor()` method.

In the preceding example, we store the `SELECT` statement in the variable `query`. Note that we are using unquoted `%s`-markers where dates should have been. Connector/Python converts `hire_start` and `hire_end` from Python types to a data type that MySQL understands and adds the required quotes. In this case, it replaces the first `%s` with `'1999-01-01'`, and the second with `'1999-12-31'`.

We then execute the operation stored in the `query` variable using the `execute()` method. The data used to replace the `%s`-markers in the query is passed as a tuple: `(hire_start, hire_end)`.

After executing the query, the MySQL server is ready to send the data. The result set could be zero rows, one row, or 100 million rows. Depending on the expected volume, you can use different techniques to process this result set. In this example, we use the `cursor` object as an iterator. The first column in the row will be stored in the variable `first_name`, the second in `last_name`, and the third in `hire_date`.

We print the result, formatting the output using Python's built-in `format()` function. Note that `hire_date` was converted automatically by Connector/Python to a Python `datetime.date` object. This means that we can easily format the date in a more human-readable form.

The output should be something like this:

```
..
Wilharm, LiMin was hired on 16 Dec 1999
Wielonsky, Lalit was hired on 16 Dec 1999
Kamble, Dannz was hired on 18 Dec 1999
DuBourdieux, Zhongwei was hired on 19 Dec 1999
Fujisawa, Rosita was hired on 20 Dec 1999
..
```

Chapter 5. Connector/Python Tutorials

Table of Contents

These tutorials illustrate how to develop Python applications and scripts that connect to a MySQL database server using MySQL Connector/Python.

Tutorial: Raise employee's salary using a buffering cursor

The following example script will give a long-overdue raise effective tomorrow to all employees who joined in the year 2000 and are still with the company.

We are using buffered cursors to iterate through the selected employees. This way we do not have to fetch the rows in a new variables, but can instead use the cursor as an iterator.

Note that the script is an example; there are other ways of doing this simple task.

```
from __future__ import print_function
from decimal import Decimal
from datetime import datetime, date, timedelta
import mysql.connector
# Connect with the MySQL Server
cnx = mysql.connector.connect(user='scott', database='employees')
# Get two buffered cursors
curA = cnx.cursor(buffered=True)
curB = cnx.cursor(buffered=True)
# Query to get employees who joined in a period defined by two dates
query = (
  "SELECT s.emp_no, salary, from_date, to_date FROM employees AS e "
  "LEFT JOIN salaries AS s USING (emp_no) "
  "WHERE to_date = DATE('9999-01-01')"
  "AND e.hire_date BETWEEN DATE(%s) AND DATE(%s)")
# UPDATE and INSERT statements for the old and new salary
update_old_salary = (
  "UPDATE salaries SET to_date = %s "
  "WHERE emp_no = %s AND from_date = %s")
insert_new_salary = (
  "INSERT INTO salaries (emp_no, from_date, to_date, salary) "
  "VALUES (%s, %s, %s, %s)")
# Select the employes getting a raise
curA.execute(query, (date(2000, 1, 1), date(2001, 1, 1)))
# Iterate through the result of curA
for (emp_no, salary, from_date, to_date) in curA:
  # Update the old and insert the new salary
  new_salary = int(round(salary * Decimal('1.15')))
  curB.execute(update_old_salary, (tomorrow, emp_no, from_date))
  curB.execute(insert_new_salary,
               (emp_no, tomorrow, date(9999, 1, 1,), new_salary))
  # Commit the changes
  cnx.commit()
cnx.close()
```

Chapter 6. Connector/Python Connection Arguments

The following lists the arguments which can be used to initiate a connection with the MySQL server using either:

- Function `mysql.connector.connect()`
- Class `mysql.connector.MySQLConnection()`

Argument Name	Default	Description
`user` (`username`*)		The username used to authenticate with the MySQL Server.
`password` (`passwd`*)		The password to authenticate the user with the MySQL Server.
`database` (`db`*)		Database name to use when connecting with the MySQL Server.
`host`	127.0.0.1	Hostname or IP address of the MySQL Server.
`port`	3306	TCP/IP port of the MySQL Server. Must be an integer.
`unix_socket`		The location of the Unix socket file.
`use_unicode`	True	Whether to use Unicode or not.
`charset`	utf8	Which MySQL character set to use.
`collation`	utf8_general_ci	Which MySQL collation to use.
`autocommit`	False	Whether to autocommit transactions.
`time_zone`		Set the `time_zone` session variable at connection.
`sql_mode`		Set the `sql_mode` session variable at connection.
`get_warnings`	False	Whether to fetch warnings.
`raise_on_warnings`	False	Whether to raise an exception on warnings.
`connection_timeout` (`connect_timeout`*)		Timeout for the TCP and Unix socket connections.
`client_flags`		MySQL client flags.
`buffered`	False	Whether cursor object fetches the result immediately after executing query.
`raw`	False	Whether MySQL results are returned as-is, rather than converted to Python types.
`ssl_ca`		File containing the SSL certificate authority.
`ssl_cert`		File containing the SSL certificate file.
`ssl_key`		File containing the SSL key.
`dsn`		Not supported (raises `NotSupportedError` when used).

* Synonymous argument name, available only for compatibility with other Python MySQL drivers. Oracle recommends not to use these alternative names.

Authentication with MySQL will use `username` and `password`. Note that MySQL Connector/Python does not support the old, insecure password protocols of MySQL versions prior to 4.1.

When the `database` parameter is given, the current database is set to the given value. To later change the database, execute the MySQL `USE` command or set the `database` property of the `MySQLConnection` instance.

By default, Connector/Python tries to connect to a MySQL server running on `localhost` using TCP/IP. The `host` argument defaults to IP address 127.0.0.1 and `port` to 3306. Unix sockets are supported by setting `unix_socket`. Named pipes on the Windows platform are not supported.

Strings coming from MySQL are by default returned as Python Unicode literals. To change this behavior, set `use_unicode` to `False`. You can change the character setting for the client connection through the `charset` argument. To change the character set after connecting to MySQL, set the `charset` property of the `MySQLConnection` instance. This technique is preferred over using the MySQL `SET NAMES` statement directly. Similar to the `charset` property, you can set the `collation` for the current MySQL session.

Transactions are not automatically committed; call the `commit()` method of the `MySQLConnection` instance within your application after doing a set of related insert, update, and delete operations. For data consistency and high throughput for write operations, it is best to leave the `autocommit` configuration option turned off when using `InnoDB` or other transactional tables.

The time zone can be set per connection using the `time_zone` argument. This is useful if the MySQL server is set, for example, to UTC and `TIMESTAMP` values should be returned by MySQL converted to the `PST` time zone.

MySQL supports so called SQL Modes. which will change the behavior of the server globally or per connection. For example, to have warnings raised as errors, set `sql_mode` to `TRADITIONAL`. For more information, see Server SQL Modes.

Warnings generated by queries are fetched automatically when `get_warnings` is set to `True`. You can also immediately raise an exception by setting `raise_on_warnings` to `True`. Consider using the MySQL sql_mode setting for turning warnings into errors.

To set a timeout value for connections, use `connection_timeout`.

MySQL uses client flags to enable or disable features. Using the `client_flags` argument, you have control of what is set. To find out what flags are available, use the following:

```
from mysql.connector.constants import ClientFlag
print '\n'.join(ClientFlag.get_full_info())
```

If `client_flags` is not specified (that is, it is zero), defaults are used for MySQL v4.1 and later. If you specify an integer greater than `0`, make sure all flags are set. A better way to set and unset flags is to use a list. For example, to set `FOUND_ROWS`, but disable the default `LONG_FLAG`:

```
flags = [ClientFlag.FOUND_ROWS, -ClientFlag.LONG_FLAG]
mysql.connector.connect(client_flags=flags)
```

By default, MySQL Connector/Python does not buffer or pre-fetch results. This means that after a query is executed, your program is responsible of fetching the data. This avoids using excessive memory when queries return large result sets. If you know that the result set is small enough to handle all at once, fetching the results immediately by setting `buffered` to `True`. It is also possible to set this per cursor (see cursor manual).

MySQL types will be converted automatically to Python types. For example, a `DATETIME` column becomes a datetime.datetime object. When conversion should be done differently, for example to get better performance, set `raw` to `True`.

Using SSL connections is possible when your Python installation supports SSL, that is, when it is compiled against the OpenSSL libraries. When you provide the arguments `ssl_ca`, `ssl_key` and `ssl_cert`, the connection switches to SSL. You can use this in combination with the `compressed` argument set to `True`.

`passwd`, `db` and `connect_timeout` are valid for compatibility with other MySQL interfaces and are respectively the same as `password`, `database` and `connection_timeout`. The latter take precedence. Data source name syntax or `dsn` is not used; if specified, it raises a `NotSupportedError` exception.

Chapter 7. Connector/Python API Reference

Table of Contents

This section contains the public API reference of Connector/Python. Although valid for both Python 2 and Python 3, examples should be considered working for Python 2.7, and Python 3.1 and greater.

The following overview shows the mysql.connector package with its modules. Currently, only the most useful modules, classes and functions for end users are documented.

```
mysql.connector
  errorcode
  errors
  connection
  constants
  conversion
  cursor
  dbapi
  locales
    eng
      client_error
  protocol
  utils
```

Errors and Exceptions

The `mysql.connector.errors` module defines exception classes for errors and warnings raised by MySQL Connector/Python. Most classes defined in this module are available when you import `mysql.connector`.

The exception classes defined in this module follow mostly the Python Database Specification v2.0 (PEP-249). For some MySQL client or server errors it is not always clear which exception to raise. It is good to discuss whether an error should be reclassified by opening a bug report.

MySQL Server errors are mapped with Python exception based on their SQLState (see Server Error Codes and Messages). The following list shows the `SQLState` classes and the exception Connector/Python will raise. It is, however, possible to redefine which exception is raised for each server error. Note that the default exception is `DatabaseError`.

- `02`: DataError
- `07`: DatabaseError
- `08`: OperationalError
- `0A`: NotSupportedError
- `21`: DataError
- `22`: DataError
- `23`: IntegrityError
- `24`: ProgrammingError
- `25`: ProgrammingError
- `26`: ProgrammingError
- `27`: ProgrammingError
- `28`: ProgrammingError
- `2A`: ProgrammingError
- `2B`: DatabaseError
- `2C`: ProgrammingError
- `2D`: DatabaseError
- `2E`: DatabaseError
- `33`: DatabaseError
- `34`: ProgrammingError
- `35`: ProgrammingError
- `37`: ProgrammingError
- `3C`: ProgrammingError
- `3D`: ProgrammingError

- 3F: ProgrammingError
- 40: InternalError
- 42: ProgrammingError
- 44: InternalError
- HZ: OperationalError
- XA: IntegrityError
- 0K: OperationalError
- HY: DatabaseError

Module errorcode

This module contains both MySQL server and client error codes defined as module attributes with the error number as value. Using error codes instead of error numbers could make reading the source code a bit easier.

```
>>> from mysql.connector import errorcode
>>> errorcode.ER_BAD_TABLE_ERROR
1051
```

See Server Error Codes and Messages and Client Error Codes and Messages.

Exception errors.Error

This exception is the base class for all other exceptions in the errors module. It can be used to catch all errors in a single except statement.

The following example shows how we could catch syntax errors:

```
import mysql.connector
try:
  cnx = mysql.connector.connect(user='scott', database='employees')
  cursor = cnx.cursor()
  cursor.execute("SELECT * FORM employees")   # Syntax error in query
  cnx.close()
except mysql.connector.Error as err:
  print("Something went wrong: {}".format(err))
```

Initializing the exception supports a few optional arguments, namely msg, errno, values and sqlstate. All of them are optional and default to None. errors.Error isinternally used by Connector/Python to raise MySQL client and server errors and should not be used by your application to raise exceptions.

The following examples show the result when using no or a combination of the arguments:

```
>>> from mysql.connector.errors import Error
>>> str(Error())
'Unknown error'
>>> str(Error("Oops! There was an error."))
'Oops! There was an error.'
>>> str(Error(errno=2006))
'2006: MySQL server has gone away'
>>> str(Error(errno=2002, values=('/tmp/mysql.sock', 2)))
"2002: Can't connect to local MySQL server through socket '/tmp/mysql.sock' (2)"
>>> str(Error(errno=1146, sqlstate='42S02', msg="Table 'test.spam' doesn't exist"))
```

```
"1146 (42S02): Table 'test.spam' doesn't exist"
```

The example which uses error number 1146 is used when Connector/Python receives an error packet from the MySQL Server. The information is parsed and passed to the `Error` exception as shown.

Each exception subclassing from `Error` can be initialized using the above mentioned arguments. Additionally, each instance has the attributes `errno`, `msg` and `sqlstate` which can be used in your code.

The following example shows how to handle errors when dropping a table which does not exists (when you do not want to use the `IF EXISTS` clause):

```
import mysql.connector
from mysql.connector import errorcode
cnx = mysql.connector.connect(user='scott', database='test')
try:
  cur.execute("DROP TABLE spam")
except mysql.connector.Error as err:
  if err.errno == errorcode.ER_BAD_TABLE_ERROR:
    print("Creating table spam")
  else:
    raise
```

`errors.Error` is a subclass of the Python `StandardError`.

Exception `errors.Warning`

This exception is used for reporting important warnings, however, Connector/Python does not use it. It is included to be compliant with the Python Database Specification v2.0 (PEP-249).

Consider using either more strict Server SQL Modes or the raise_on_warnings connection argument to make Connector/Python raise errors when your queries produce warnings.

`errors.Warning` is a subclass of the Python `StandardError`.

Exception `errors.InterfaceError`

This exception is raised for errors originating from Connector/Python itself, not related to the MySQL server.

`errors.InterfaceError` is a subclass of `errors.Error`.

Exception `errors.DatabaseError`

This exception is the default for any MySQL error which does not fit the other exceptions.

`errors.DatabaseError` is a subclass of `errors.Error`.

Exception `errors.InternalError`

This exception is raised when the MySQL server encounters an internal error, for example, when a deadlock occurred.

`errors.InternalError` is a subclass of `errors.DatabaseError`.

Exception `errors.OperationalError`

This exception is raised for errors which are related to MySQL's operations. For example, to many connections, a hostname could not be resolved, bad handshake, server is shutting down, communication errors, and so on.

errors.OperationalError is a subclass of errors.DatabaseError.

Exception errors.ProgrammingError

This exception is raised on programming errors, for example when you have a syntax error in your SQL or a table was not found.

The following example shows how to handle syntax errors:

```
try:
  cursor.execute("CREATE DESK t1 (id int, PRIMARY KEY (id))")
except mysql.connector.ProgrammingError as err:
  if err.errno == errorcode.ER_SYNTAX_ERROR:
    print("Check your syntax!")
  else:
    print("Error: {}".format(err))
```

errors.ProgrammingError is a subclass of errors.DatabaseError.

Exception errors.IntegrityError

This exception is raised when the relational integrity of the data is affected. For example, a duplicate key was inserted or a foreign key constraint would fail.

The following example shows a duplicate key error raised as IntegrityError:

```
cursor.execute("CREATE TABLE t1 (id int, PRIMARY KEY (id))")
try:
  cursor.execute("INSERT INTO t1 (id) VALUES (1)")
  cursor.execute("INSERT INTO t1 (id) VALUES (1)")
except mysql.connector.IntegrityError as err:
  print("Error: {}".format(err))
```

errors.IntegrityError is a subclass of errors.DatabaseError.

Exception errors.DataError

This exception is raised when there were problems with the data. Examples are a column set to NULL when it can not, out of range values for a column, division by zero, column count does not match value count, and so on.

errors.DataError is a subclass of errors.DatabaseError.

Exception errors.NotSupportedError

This exception is raised is case some feature was used but not supported by the version of MySQL which returned the error. It is also raised when using functions or statements which are not supported by stored routines.

errors.NotSupportedError is a subclass of errors.DatabaseError.

Function errors.custom_error_exception(error=None, exception=None)

This function defines custom exceptions for MySQL server errors and returns current customizations.

If error is a MySQL Server error number, then you have to pass also the exception class. The error argument can also be a dictionary in which case the key is the server error number, and value the class of the exception to be raised.

To reset the customizations, simply supply an empty dictionary.

```
import mysql.connector
from mysql.connector import errorcode
# Server error 1028 should raise a DatabaseError
mysql.connector.custom_error_exception(1028, mysql.connector.DatabaseError)
# Or using a dictionary:
mysql.connector.custom_error_exception({
  1028: mysql.connector.DatabaseError,
  1029: mysql.connector.OperationalError,
})
# To reset, pass an empty dictionary:
mysql.connector.custom_error_exception({})
```

Class connection.MySQLConnection

The MySQLConnection class is used to open and manage a connection to a MySQL server. It also used to send commands and SQL queries and read result.

Constructor connection.MySQLConnection(**kwargs)

The MySQLConnection constructor initializes the attributes and when at least one argument is passed, it tries to connect with the MySQL server.

For a complete list or arguments, see Chapter 6, *Connector/Python Connection Arguments*.

Method MySQLConnection.close()

See disconnect().

Returns a tuple.

Method MySQLConnection.config(**kwargs)

Allows to configure a MySQLConnection instance after it was instantiated. See Chapter 6, *Connector/Python Connection Arguments* for a complete list of possible arguments.

You could use the `config()` method to change, for example, the username and call `reconnect()`.

```
cnx = MySQLConnection(user='joe', database='test')
# Connected as 'joe'
cnx.config(user='jane')
cnx.reconnect()
# Now connected as 'jane'
```

Method MySQLConnection.connect(**kwargs)

This method sets up the connection to the MySQL server. If no arguments are given, it uses the already configured or default values. See Chapter 6, *Connector/Python Connection Arguments* for a complete list of possible arguments.

Method MySQLConnection.commit()

This method sends the `COMMIT` command to the MySQL server, committing the current transaction. Since by default, Connector/Python does not auto commit, it is important to call this method after every transaction which updates data for tables using transactional storage engines.

See the rollback() method for rolling back transactions.

```
>>> cursor.execute("INSERT INTO employees (first_name) VALUES (%s)", ('Jane'))
>>> cnx.commit()
```

Method MySQLConnection.cursor(buffered=None, raw=None, cursor_class=None)

This method returns a MySQLCursor() object, or a subclass of it depending the passed arguments.

When buffered is True, the cursor will fetch all rows after the operation is executed. This is useful when queries return small result sets. Setting raw will skip the conversion from MySQL data types to Python types when fetching rows. Raw is usually used when you want to have more performance and/or you want to do the conversion yourself.

The cursor_class argument can be used to pass a class to use for instantiating a new cursor. It has to be a subclass of cursor.CursorBase.

The returned object depends on the combination of the buffered and raw arguments.

- If not buffered and not raw: cursor.MySQLCursor
- If buffered and not raw: cursor.MySQLCursorBuffered
- If buffered and raw: cursor.MySQLCursorBufferedRaw
- If not buffered and raw: cursor.MySQLCursorRaw

Returns a CursorBase instance.

Method MySQLConnection.cmd_change_user(username='', password='', database='', charset=33)

Changes the user using username and password. It also causes the specified database to become the default (current) database. It is also possible to change the character set using the charset argument.

Returns a dictionary containing the OK packet information.

Method MySQLConnection.cmd_debug()

Instructs the server to write some debug information to the log. For this to work, the connected user must have the SUPER privilege.

Returns a dictionary containing the OK packet information.

Method MySQLConnection.cmd_init_db(database)

This method makes specified database the default (current) database. In subsequent queries, this database is the default for table references that do not include an explicit database specifier.

Returns a dictionary containing the OK packet information.

Method MySQLConnection.cmd_ping()

Checks whether the connection to the server is working.

This method is not to be used directly. Use ping() or is_connected() instead.

Returns a dictionary containing the OK packet information.

Method MySQLConnection.cmd_process_info()

This method raises the NotSupportedError exception. Instead, use the SHOW PROCESSLIST statement or query the tables found in the database INFORMATION_SCHEMA.

Method MySQLConnection.cmd_process_kill(mysql_pid)

Asks the server to kill the thread specified by mysql_pid. Although still available, it's better to use the SQL KILL command.

Returns a dictionary containing the OK packet information.

The following two lines do the same:

```
>>> cnx.cmd_process_kill(123)
>>> cnx.cmd_query('KILL 123')
```

Method MySQLConnection.cmd_quit()

This method sends the QUIT command to the MySQL server, closing the current connection. Since there is no response from the MySQL, the packet that was sent is returned.

Method MySQLConnection.cmd_query(statement)

This method sends the given statement to the MySQL server and returns a result. If you need to send multiple statements, you have to use the cmd_query_iter() method.

The returned dictionary contains information depending on what kind of query was executed. If the query is a SELECT statement, the result contains information about columns. Other statements return a dictionary containing OK or EOF packet information.

Errors received from the MySQL server are raised as exceptions. An InterfaceError is raised when multiple results are found.

Returns a dictionary.

Method MySQLConnection.cmd_query_iter(statement)

Similar to the cmd_query() method, but returns a generator object to iterate through results. Use cmd_query_iter() when sending multiple statements, and separate the statements with semicolons.

The following example shows how to iterate through the results after sending multiple statements:

```
statement = 'SELECT 1; INSERT INTO t1 VALUES (); SELECT 2'
for result in cnx.cmd_query(statement, iterate=True):
  if 'columns' in result:
    columns = result['columns']
    rows = cnx.get_rows()
  else:
    # do something useful with INSERT result
```

Returns a generator object.

Method MySQLConnection.cmd_refresh(options)

This method flushes tables or caches, or resets replication server information. The connected user must have the RELOAD privilege.

The options argument should be a bitwise value using constants from the class constants.RefreshOption.

See Class constants.RefreshOption for a list of options.

Example:

```
>>> from mysql.connector import RefreshOption
>>> refresh = RefreshOption.LOG | RefreshOption.THREADS
>>> cnx.cmd_refresh(refresh)
```

Method MySQLConnection.cmd_shutdown()

Asks the database server to shut down. The connected user must have the SHUTDOWN privilege.

Returns a dictionary containing the OK packet information.

Method MySQLConnection.cmd_statistics()

Returns a dictionary containing information about the MySQL server including uptime in seconds and the number of running threads, questions, reloads, and open tables.

Method MySQLConnection.disconnect()

This method tries to send the QUIT command and close the socket. It does not raise any exceptions.

MySQLConnection.close() is a synonymous method name and more commonly used.

Method MySQLConnection.get_rows(count=None)

This method retrieves all or remaining rows of a query result set, returning a tuple containing the rows as sequence and the EOF packet information. The count argument can be used to get a given amount of rows. If count is not specified or is None, all rows are retrieved.

The tuple returned by get_rows() consists of:

- A list of tuples containing the row data as byte objects, or an empty list when no rows are available.
- EOF packet information as a dictionary containing status_flag and warning_count.

An InterfaceError is raised when all rows have been retrieved.

The get_rows() method is used by MySQLCursor to fetch rows.

Returns a tuple.

Method MySQLConnection.get_row()

This method retrieves the next row of a query result set, returning a tuple.

The tuple returned by get_row() consists of:

- The row as a tuple containing byte objects, or None when no more rows are available.
- EOF packet information as a dictionary containing status_flag and warning_count, or None when the row returned is not the last row.

The get_row() method is used by MySQLCursor to fetch rows.

Method MySQLConnection.get_server_info()

This method returns the MySQL server information verbatim, for example '5.5.24-log', or None when not connected.

Returns a string or None.

Method MySQLConnection.get_server_version()

This method returns the MySQL server version as a tuple, or None when not connected.

Returns a tuple or None.

Method MySQLConnection.is_connected()

Reports whether the connection to MySQL Server is available.

This method checks whether the connection to MySQL is available using the ping() method, but unlike ping(), is_connected() returns True when the connection is available, False otherwise.

Returns True or False.

Method MySQLConnection.isset_client_flag(flag)

This method returns True if the client flag was set, False otherwise.

Returns True or False.

Method MySQLConnection.ping(attempts=1, delay=0)

Check whether the connection to the MySQL server is still available.

When reconnect is set to True, one or more attempts are made to try to reconnect to the MySQL server using the reconnect() method. Use the delay argument (seconds) if you want to wait between each retry.

When the connection is not available, an InterfaceError is raised. Use the is_connected() method to check the connection without raising an error.

Raises InterfaceError on errors.

Method MySQLConnection.reconnect(attempts=1, delay=0)

Attempt to reconnect with the MySQL server.

The argument attempts specifies the number of times a reconnect is tried. The delay argument is the number of seconds to wait between each retry.

You might set the number of attempts higher and use a longer delay when you expect the MySQL server to be down for maintenance, or when you expect the network to be temporarily unavailable.

Method MySQLConnection.rollback()

This method sends the ROLLBACK command to the MySQL server, undoing all data changes from the current transaction. Since by default, Connector/Python does not auto commit, it is possible to cancel transactions when using transactional storage engines such as InnoDB.

See the commit() method for committing transactions.

```
>>> cursor.execute("INSERT INTO employees (first_name) VALUES (%s)", ('Jane'))
>>> cnx.rollback()
```

Method MySQLConnection.set_charset_collation(charset=None, collation=None)

This method sets the character set and collation to be used for the current connection. The charset argument can be either the name of a character set, or the numerical equivalent as defined in constants.CharacterSet.

When collation is None, the default will be looked up and used.

The charset argument then be either:

In the following example, we set the character set to latin1 and the collation will be set to the default latin1_swedish_ci:

```
>>> cnx = mysql.connector.connect(user='scott')
>>> cnx.set_charset('latin1')
```

Specify a specific collation as follows:

```
>>> cnx = mysql.connector.connect(user='scott')
>>> cnx.set_charset('latin1', 'latin1_general_ci')
```

Method MySQLConnection.set_client_flags(flags)

This method sets the client flags which are used when connecting with the MySQL server and returns the new value. The flags argument can be either an integer or a sequence of valid client flag values (see Class constants.ClientFlag).

If flags is a sequence, each item in the sequence will set the flag when the value is positive or unset it when negative. For example, to unset LONG_FLAG and set the FOUND_ROWS flags:

```
>>> from mysql.connector.constants import ClientFlag
>>> cnx.set_client_flags([ClientFlag.FOUND_ROWS, -ClientFlag.LONG_FLAG])
>>> cnx.reconnect()
```

Note that client flags are only set or used when connecting with the MySQL server. It is therefor necessary to reconnect after making changes.

Returns an integer.

Property MySQLConnection.autocommit

This property is used to toggle the auto commit feature of MySQL and retrieve the current state. When the value evaluates to True, auto commit will be turned, otherwise it is turned off.

Note that auto commit is disabled by default when connecting through Connector/Python. This can be toggled using the connection parameter `autocommit`.

When the auto commit is turned off, you have to commit transactions when using transactional storage engines such as InnoDB or NDBCluster.

```
>>> cnx.autocommit
False
>>> cnx.autocommit = True
>>> cnx.autocommit
True
```

Returns True or False.

Property MySQLConnection.charset_name

This property returns which character set is used for the connection whether it is connected or not.

Returns a string.

Property MySQLConnection.collation_name

This property returns which collation is used for the connection whether it is connected or not.

Returns a string.

Property MySQLConnection.connection_id

This property returns the connection ID (thread ID or session ID) for the current connection or None when not connected.

Returns a integer or None.

Property MySQLConnection.database

This property is used to set current (active) database executing the `USE` command. The property can also be used to retrieve the current database name.

```
>>> cnx.database = 'test'
>>> cnx.database = 'mysql'
>>> cnx.database
u'mysql'
```

Returns a string.

Property MySQLConnection.get_warnings

This property is used to toggle whether warnings should be fetched automatically or not. It accepts True or False (default).

Fetching warnings automatically could be useful when debugging queries. Cursors will make warnings available through the method MySQLCursor.fetchwarnings().

```
>>> cnx.get_warnings = True
>>> cursor.execute('SELECT "a"+1')
>>> cursor.fetchall()
```

```
[(1.0,)]
>>> cursor.fetchwarnings()
[(u'Warning', 1292, u"Truncated incorrect DOUBLE value: 'a'")]
```

Returns True or False.

Property MySQLConnection.raise_on_warnings

This property is used to toggle whether warnings should raise exceptions or not. It accepts True or False (default).

Toggling raise_on_warnings will also toggle get_warnings since warnings need to be fetched so they can be raised as exceptions.

Note that you might always want to check setting SQL Mode if you would like to have the MySQL server directly report warnings as errors. It is also good to use transactional engines so transactions can be rolled back when catching the exception.

Result sets needs to be fetched completely before any exception can be raised. The following example shows the execution of a query which produces a warning

```
>>> cnx.raise_on_warnings = True
>>> cursor.execute('SELECT "a"+1')
>>> cursor.fetchall()
..
mysql.connector.errors.DataError: 1292: Truncated incorrect DOUBLE value: 'a'
```

Returns True or False.

Property MySQLConnection.server_host

This read-only property returns the hostname or IP address used for connecting with the MySQL server.

Returns a string.

Property MySQLConnection.server_port

This read-only property returns the TCP/IP port used for connecting with the MySQL server.

Returns a integer.

Property MySQLConnection.sql_mode

This property is used to retrieve and set the SQL Modes for the current. The value should be list of different modes separated by comma (","), or a sequence of modes, preferably using the constants.SQLMode class.

To unset all modes, pass an empty string or an empty sequence.

```
>>> cnx.sql_mode = 'TRADITIONAL,NO_ENGINE_SUBSTITUTION'
>>> cnx.sql_mode.split(',')
[u'STRICT_TRANS_TABLES', u'STRICT_ALL_TABLES', u'NO_ZERO_IN_DATE',
u'NO_ZERO_DATE', u'ERROR_FOR_DIVISION_BY_ZERO', u'TRADITIONAL',
u'NO_AUTO_CREATE_USER', u'NO_ENGINE_SUBSTITUTION']
>>> from mysql.connector.constants import SQLMode
>>> cnx.sql_mode = [ SQLMode.NO_ZERO_DATE, SQLMode.REAL_AS_FLOAT]
>>> cnx.sql_mode

u'REAL_AS_FLOAT,NO_ZERO_DATE'
```

Returns a string.

Property MySQLConnection.time_zone

This property is used to set the time zone session variable for the current connection and retrieve it.

```
>>> cnx.time_zone = '+00:00'
>>> cur.execute('SELECT NOW()') ; cur.fetchone()
(datetime.datetime(2012, 6, 15, 11, 24, 36),)
>>> cnx.time_zone = '-09:00'
>>> cur.execute('SELECT NOW()') ; cur.fetchone()
(datetime.datetime(2012, 6, 15, 2, 24, 44),)
>>> cnx.time_zone
u'-09:00'
```

Returns a string.

Property MySQLConnection.unix_socket

This read-only property returns the UNIX socket user for connecting with the MySQL server.

Returns a string.

Property MySQLConnection.user

This read-only property returns the username used for connecting with the MySQL server.

Returns a string.

Class cursor.MySQLCursor

The MySQLCursor class is used to instantiate object which can execute operation such as SQL queries. They interact with the MySQL server using a MySQLConnection object.

Constructor cursor.MySQLCursor

The constructor initializes the instance with the optional `connection`, which should be an instance of MySQLConnection.

In most cases, the MySQLConnection method `cursor()` is used to instantiate a MySQLCursor object.

Method MySQLCursor.callproc(procname, args=())

This method calls a stored procedure with the given name. The `args` sequence of parameters must contain one entry for each argument that the routine expects. The result is returned as modified copy of the input sequence. Input parameters are left untouched, output and input/output parameters replaced with possibly new values.

Result set provided by the stored procedure are automatically fetched and stored as MySQLBufferedCursor instances. See stored_results() for more information.

The following example shows how to execute a stored procedure which takes two parameters, multiplies the values and returns the product:

```
# Definition of the multiply stored procedure:
```

```
# CREATE PROCEDURE multiply(IN pFac1 INT, IN pFac2 INT, OUT pProd INT)
# BEGIN
#  SET pProd := pFac1 * pFac2;
# END
>>> args = (5, 5, 0) # 0 is to hold value of the OUT parameter pProd
>>> cursor.callproc('multiply', args)
('5', '5', 25L)
```

Method MySQLCursor.close()

This method will close the MySQL cursor, resetting all results and removing the connection.

Use `close()` every time you are done using the cursor.

Method MySQLCursor.execute(operation, params=None, multi=False)

This method prepare the given database `operation` (query or command). The parameters found in the tuple or dictionary `params` will be bound to the variables in the operation. Variables are specified using %s markers or named markers %(name)s.

For example, insert information about a new employee and selecting again the data of this person:

```
insert = (
"INSERT INTO employees (emp_no, first_name, last_name, hire_date) "
"VALUES (%s, %s, %s, %s)")
data = (2, 'Jane', 'Doe', datetime.date(2012, 3, 23))
cursor.execute(insert, data)
select = "SELECT * FROM employees WHERE emp_no = %(emp_no)s"
cursor.execute(select, { 'emp_no': 2 })
```

Note that the data is converted from Python object to something MySQL understand. In the above example, the datetime.date() instance is converted to '2012-03-23' in the above example.

When `multi` is set to True, `execute()` will be able to execute multiple statements. It will return an iterator which makes it possible to go through all results for each statement. Note that using parameters is not working well in this case, and it's usually a good idea to execute each statement on its own.

In the following example we select and insert data in one operation and display the result:

```
operation = 'SELECT 1; INSERT INTO t1 VALUES (); SELECT 2'
for result in cursor.execute(operation):
  if result.with_rows:
    print("Statement '{}' has following rows:".format(
      result.statement))
    print(result.fetchall())
  else:
    print("Affected row(s) by query '{}' was {}".format(
      result.statement, result.rowcount))
```

If the connection was configured to fetch warnings, warnings generated by the operation will be available through the method MySQLCursor.fetchwarnings().

Returns an iterator when multi is True.

Method MySQLCursor.executemany(operation, seq_params)

This method prepares a database operation (query or command) and then execute it against all parameter sequences or mappings found in the sequence `seq_of_params`.

The executemany() is simply iterating through the sequence of parameters calling the execute() method. Inserting data, however, is optimized by batching them using the multiple rows syntax.

In the following example we are inserting 3 records:

```
data = [
  ('Jane', date(2005, 2, 12)),
  ('Joe', date(2006, 5, 23)),
  ('John', date(2010, 10, 3)),
]
stmt = "INSERT INTO employees (first_name, hire_date) VALUES (%s, %s)"
cursor.executemany(stmt, data)
```

In the above example, the INSERT statement sent to MySQL would be as follows: `INSERT INTO employees (first_name, hire_date) VALUES ('Jane', '2005-02-12'), ('Joe', '2006-05-23'), ('John', '2010-10-03')`.

Note that it is not possible to execute multiple statements using the executemany() method. Doing so will raise an InternalError exception.

Method MySQLCursor.fetchall()

The method fetches all or remaining rows of a query result set, returning a list of tuples. An empty list is returned when no rows are (anymore) available.

The following examples shows how to retrieve the first 2 rows of a result set, and then retrieve the remaining rows:

```
>>> cursor.execute("SELECT * FROM employees ORDER BY emp_no")
>>> head_rows = cursor.fetchmany(size=2)
>>> remaining_rows = cursor.fetchall()
```

Note that you have to fetch all rows before being able to execute new queries using the same connection.

Returns a list of tuples or empty list when no rows available.

Method MySQLCursor.fetchmany(size=1)

This method fetches the next set of rows of a query results, returning a list of tuples. An empty list is returned when no more rows are available.

The number of rows returned can be specified using the size argument, which defaults to one. Fewer rows might be returned, when there are not more rows available than specified by the argument.

Note that you have to fetch all rows before being able to execute new queries using the same connection.

Returns a list of tuples or empty list when no rows available.

Method MySQLCursor.fetchone()

This method retrieves the next row of a query result set, returning a single sequence, or None when no more data is available.The returned tuple consists of data returned by the MySQL server converted to Python objects.

The fetchone() method is used by fetchmany() and fetchall(). It is also used when using the MySQLCursor instance as an iterator.

The following examples show how to iterate through the result of a query using fetchone():

```
# Using a while-loop
cursor.execute("SELECT * FROM employees")
row = cursor.fetchone()
while row is not None:
  print(row)
  row = cursor.fetchone()
# Using the cursor as iterator
cursor.execute("SELECT * FROM employees")
for row in cursor:
  print(row)
```

Note that you have to fetch all rows before being able to execute new queries using the same connection.

Returns a tuple or None.

Method MySQLCursor.fetchwarnings()

This method returns a list of tuples containing warnings generated by previously executed statement. Use the connection's `get_warnings` property to toggle whether warnings has to be fetched.

The following example shows a SELECT statement which generated a warning:

```
>>> cnx.get_warnings = True
>>> cursor.execute('SELECT "a"+1')
>>> cursor.fetchall()
[(1.0,)]
>>> cursor.fetchwarnings()
[(u'Warning', 1292, u"Truncated incorrect DOUBLE value: 'a'")]
```

It is also possible to raise errors when warnings are found. See the MySQLConnection property raise_on_warnings.

Returns a list of tuples.

Method MySQLCursor.stored_results()

This method returns an list iterator object which can be used to go through result sets provided by stored procedures after calling them using the callproc() method.

In the following example we execute a stored procedure which will provide two result sets. We use `stored_results()` to retrieve them:

```
>>> cursor.callproc('sp1')
()
>>> for result in cursor.stored_results():
...     print result.fetchall()
...
[(1,)]
[(2,)]
```

Note that the result sets stay available until you executed another operation or call another stored procedure.

Returns a listiterator.

Property MySQLCursor.column_names

This read-only property returns the column names of a result set as sequence of (unicode) strings.

The following example shows how you can create a dictionary out of a tuple containing data with keys using column_names:

```
cursor.execute("SELECT last_name, first_name, hire_date "
  "FROM employees WHERE emp_no = %s", (123,))
row = dict(zip(cursor.column_names, cursor.fetchone())
print("{last_name}, {first_name}: {hire_date}".format(row))
```

Returns a tuple.

Property MySQLCursor.statement

This read-only property returns the last executed statement. In case multiple statements where executed, it will show the actual statement.

The statement property might be useful for debugging and showing what was send to the MySQL server.

Returns a string.

Property MySQLCursor.with_rows

This read-only property will return True when the result of the executed operation provides rows.

The with_rows property is useful when executing multiple statements and you need to fetch rows. In the following example we only report the affected rows by the UPDATE statement:

```
import mysql.connector
cnx = mysql.connector.connect(user='scott', database='test')
cursor = cnx.cursor()
operation = 'SELECT 1; UPDATE t1 SET c1 = 2; SELECT 2'
for result in cursor.execute(operation, multi=True):
  if result.with_rows:
    result.fetchall()
  else:
    print("Updated row(s): {}".format(result.rowcount))
```

Class cursor.MySQLCursorBuffered

This class is inheriting from cursor.MySQLCursor and if needed automatically retrieves rows after an operation has been executed.

MySQLCursorBuffered can be useful in situations where two queries, with small result sets, need to be combined or computed with each other.

You can either use the buffered argument when using the connection's cursor() method, or you can use the buffered connection option to make all created cursors by default buffering.

```
import mysql.connector
cnx = mysql.connector.connect()
# Only this particular cursor will be buffering results
cursor.cursor(buffered=True)
# All cursors by default buffering
cnx = mysql.connector.connect(buffered=True)
```

See Tutorial: Raise employee's salary using a buffering cursor for a practical use case.

Class constants.ClientFlag

This class provides constants defining MySQL client flags which can be used upon connection to configure the session. The ClientFlag class is available when importing mysql.connector.

```
>>> import mysql.connector
>>> mysql.connector.ClientFlag.FOUND_ROWS
2
```

See Method MySQLConnection.set_client_flags(flags) and the connection argument client_flag.

Note that the ClientFlag class can not be instantiated.

Class constants.FieldType

This class provides all supported MySQL field or data types. They can be useful when dealing with raw data or defining your own converters. The field type is stored with every cursor in the description for each column.

The following example shows how you can print the name of the data types for each of the columns in the result set.

```
from __future__ import print_function
import mysql.connector
from mysql.connector import FieldType
cnx = mysql.connector.connect(user='scott', database='test')
cursor = cnx.cursor()
cursor.execute(
  "SELECT DATE(NOW()) AS `c1`, TIME(NOW()) AS `c2`, "
  "NOW() AS `c3`, 'a string' AS `c4`, 42 AS `c5`")
rows = cursor.fetchall()
for desc in cursor.description:
  colname = desc[0]
  coltype = desc[1]
  print("Column {} has type {}".format(
    colname, FieldType.get_info(coltype)))
cursor.close()
cnx.close()
```

Note that the FieldType class can not be instantiated.

Class constants.SQLMode

This class provides all known MySQL Server SQL Modes. It is mostly used when setting the SQL modes at connection time using the connection's property sql_mode. See Property MySQLConnection.sql_mode.

Note that the SQLMode class can not be instantiated.

Class constants.CharacterSet

This class provides all known MySQL characters sets and their default collations. See Method MySQLConnection.set_charset_collation(charset=None, collation=None) for examples.

Note that the CharacterSet class can not be instantiated.

Class constants.RefreshOption

- RefreshOption.GRANT

  Refresh the grant tables, like FLUSH PRIVILEGES.

- `RefreshOption.LOG`

  Flush the logs, like `FLUSH LOGS`.

- `RefreshOption.TABLES`

  Flush the table cache, like `FLUSH TABLES`.

- `RefreshOption.HOSTS`

  Flush the host cache, like `FLUSH HOSTS`.

- `RefreshOption.STATUS`

  Reset status variables, like `FLUSH STATUS`.

- `RefreshOption.THREADS`

  Flush the thread cache.

- `RefreshOption.SLAVE`

  On a slave replication server, reset the master server information and restart the slave, like `RESET SLAVE`.

- `RefreshOption.MASTER`

  On a master replication server, remove the binary log files listed in the binary log index and truncate the index file, like `RESET MASTER`.

Chapter 8. MySQL Connector/Python Change History

Changes in MySQL Connector/Python 1.0.6 (30 August 2012, beta)

Second beta release.

Functionality Added or Changed

- Changed name and version of distributions to align with other MySQL projects:
  - The version now includes the suffix 'b' for beta and 'a' for alpha followed by a number. This version is used in the source and built distributions. GA versions will have no suffix.
  - The RPM spec files have been updated to create packages whose names are aligned with RPMs from other MySQL projects.
- Changed how MySQL server errors are mapped to Python exceptions. We now use the `SQLState` (when available) to raise a better error.
  - Incompatibility: some server errors are now raised with a different exception.
  - It is possible to override how errors are raised using the `mysql.connector.custom_error_exception()` function, defined in the `mysql.connector.errors` module. This can be useful for certain frameworks to align with other database drivers.

Bugs Fixed

- Fixed version-specific code so Connector/Python works with Python 3.3. (Bug #14524942)
- Fixed `MySQLCursorRaw.fetchall()` so it does not raise an exception when results are available. (Bug #14517262, Bug #66465)
- Fixed installation of `version.py` on OS X:
  - `version.py` is now correctly installed on OS X in the `mysql.connector` package. Previously, it was installed through `data_files`, and `version.py` ended up in the system-wide package location of Python, from which it could not be imported.
  - `data_files` is not used any longer in `setup.py` and is removed. Extra files like `version.py` are now copied in the custom `Distutils` commands.

  (Bug #14483142)
- Timeout for unit tests has been set to 10 seconds. Test cases can individually adjust it to be higher or lower. (Bug #14487502)
- Fixed test cases in `test_mysql_database.py` that failed when using `YEAR(2)` with MySQL 5.6.6 and greater. (Bug #14460680)
- Fixed SSL unit testing for source distributions:
  - The SSL keys and certificates were missing and are now added to the source distribution. Now SSL testing works properly.
  - Additionally for the Windows platform, forward slashes were added to the option file creation so the MySQL server can pick up the needed SSL files.

(Bug #14402737)

Changes in MySQL Connector/Python 1.0.5 (17 July 2012, beta)

First beta release.

Functionality Added or Changed

- Added SQLMode class in the constants module to make it easier to set modes. For example:

```
cnx.sql_mode = [SQLMode.REAL_AS_FLOAT, SQLMode.NO_ZERO_DATE]
```

- Added descriptive error codes for both client and server errors in the module `errorcode`. A new sub-package `locales` has been added, which currently only supports English client error messages.

  For example, `errorcode.CR_CONNECTION_ERROR` is 2002.

Changes in MySQL Connector/Python 1.0.4 (07 July 2012, alpha)

Internal alpha release.

Bugs Fixed

- **Incompatible Change**: The `MySQLConnection` methods `unset_client_flag()` and `set_client_flag()` have been removed. Use the`set_client_flags()` method instead using a sequence. (Bug #14259996)

- **Incompatible Change**: The method `MySQLConnection.set_charset()` has been removed and replaced by `MySQLConnection.set_charset_collation()` to simplify setting and retrieving character set and collation information. The `MySQLConnection` properties `collation` and `charset` are now read-only. (Bug #14260052)

- **Incompatible Change**: Fixed `MySQLConnection.cmd_query()` to raise an error when the operation has multiple statements. We introduced a new method `MySQLConnection.cmd_query_iter()` which needs to be used when multiple statements send to the MySQL server. It returns a generator object to iterate through results.

  When executing single statements, `MySQLCursor.execute()` will always return `None`. You can use the `MySQLCursor` property `with_rows` to check whether a result could have rows or not.

  `MySQLCursor.execute()` returns a generator object with which you can iterate over results when executing multiple statements.

  The `MySQLCursor.next_resultset()` became obsolete and was removed and the `MySQLCursor.next_proc_result()` method has been renamed to `MySQLCursor.proc_results()`, which returns a generator object. The `MySQLCursor.with_rows` property can be used to check if a result could return rows. The `multiple_resultset.py` example script shows how to go through results produced by sending multiple statements. (Bug #14208326)

- Fixed `MySQLCursor.executemany()` when `INSERT` statements use the `ON DUPLICATE KEY` clause with a function such as `VALUES()`. (Bug #14259954)

- Fixed unit testing on the Microsoft Windows platform. (Bug #14236592)

- Fixed converting a `datetime.time` to a MySQL type using Python 2.4 and 2.5. The `strftime()` function has no support for the `%f` mark in those Python versions. (Bug #14231941)

- Fixed `cursor.CursorBase` attributes `description`, `lastrowid` and `rowcount` to be read-only properties. (Bug #14231160)
- Fixed `MySQLConnection.cmd_query()` and other methods so they check first whether there are unread results. (Bug #14184643)

Changes in MySQL Connector/Python 1.0.3 (08 June 2012, alpha)

Internal alpha release.

Functionality Added or Changed

- Adding new `Distutils` commands to create Windows Installers using WiX and RPM packages.
- Adding support for time values with a fractional part, for MySQL 5.6.4 and greater. A new example script `microseconds.py` was added to show this functionality.

Changes in MySQL Connector/Python 1.0.2 (19 May 2012, alpha)

Internal alpha release.

Functionality Added or Changed

- Added more unit tests for modules like `connection` and `network` as well as testing the SSL functionality.

Bugs Fixed

- Fixed bootstrapping MySQL 5.6 running unit tests.

  Messages send by the bootstrapped MySQL server to `stdout` and `stderr` are now discarded. (Bug #14048685)
- Fixing and refactoring the `mysql.connector.errors` module. (Bug #14039339)

Changes in MySQL Connector/Python 1.0.1 (26 April 2012, alpha)

Internal alpha release.

Functionality Added or Changed

- Change the version so it only contain integers. The 'a' or 'alpha' suffix will not be present in packages, but it will be mentioned in the `_version.py` module since `metasetupinfo.py` uses this information to set, for example, the Trove classifiers dynamically.

Changes in MySQL Connector/Python 1.0.0 (22 April 2012, alpha)

Internal alpha release.

Functionality Added or Changed

- **Incompatible Change**: `MySQLConnection.reconnect()` can be used to reconnect to the MySQL server. It accepts number of retries and an optional delay between attempts.

  `MySQLConnectiong.ping()` is now a method and works the way the MySQL C API `mysql_ping()` function works: it raises an error. It can also optionally reconnect.

`MySQLConnection.is_connected()` now returns `True` when connection is available, `False` otherwise.

`ping()` and `is_connected()` are backwards incompatible. (Bug #13392739)

- Refactored the modules connection and protocol and created a new module `network`. The `MySQLProtocol` does not keep a reference to the connection object any more and deals only with creating and parsing MySQL packets. Network interaction is now done by the `MySQLConnection` objects (with the exception of `MySQLProtocol.read_text_result()`).

Bugs Fixed

- Fixed `metasetupinfo.py` to use the Connector/Python which is being installed instead of the version already installed. (Bug #13962765)
- Fixed `MySQLCursor.description` so it stores column names as Unicode. (Bug #13792575)
- Fixed `dbapi.Binary` to be a bytes types for Python 3.x. (Bug #13780676)
- Fixed automatic garbage collection which caused memory usage to grow over time. Note that `MySQLConnection` does not keep track of its cursors any longer. (Bug #13435186)
- Fixed setting time zone for current MySQL session. (Bug #13395083)
- Fixed setting and retrieving character set and collation. (Bug #13375632)
- Fixed handling of errors after authentication for Python 3. (Bug #13364285)