MySQL Connector/Python Developer Guide

Abstract

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

For notes detailing the changes in each release of Connector/Python, see MySQL Connector/Python Release Notes.

Document generated on: 2015-06-12 (revision: 43509)

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

Chapter 1 Introduction to MySQL Connector/Python

MySQL Connector/Python enables Python programs to access MySQL databases, using an API that is compliant with the Python Database API Specification v2.0 (PEP 249). It is written in pure Python and does not have any dependencies except for the Python Standard Library.

For notes detailing the changes in each release of Connector/Python, see MySQL Connector/Python Release Notes.

MySQL Connector/Python includes support for:

- Almost all features provided by MySQL Server up to and including MySQL Server version 5.7.
- 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 2.0 supports Python 2.6 and 2.7, and Python 3.3 and later. MySQL Connector/Python 1.2 and 1.1 support Python from version 2.6 through 2.7, and Python 3.1 and later. MySQL Connector/Python 1.0 supports Python from version 2.4 through 2.7, and Python 3.1 and later.

> **Note**
>
> 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 Guidelines for Python Developers

The following guidelines cover aspects of developing MySQL applications that might not be immediately obvious to developers coming from a Python background:

- For security, do not hardcode the values needed to connect and log into the database in your main script. Python has the convention of a `config.py` module, where you can keep such values separate from the rest of your code.

- Python scripts often build up and tear down large data structures in memory, up to the limits of available RAM. Because MySQL often deals with data sets that are many times larger than available memory, techniques that optimize storage space and disk I/O are especially important. For example, in MySQL tables, you typically use numeric IDs rather than string-based dictionary keys, so that the key values are compact and have a predictable length. This is especially important for columns that make up the primary key for an `InnoDB` table, because those column values are duplicated within each secondary index.

- Any application that accepts input must expect to handle bad data.

  The bad data might be accidental, such as out-of-range values or misformatted strings. The application can use server-side checks such as unique constraints and NOT NULL constraints, to keep the bad data from ever reaching the database. On the client side, use techniques such as exception handlers to report any problems and take corrective action.

  The bad data might also be deliberate, representing an “SQL injection” attack. For example, input values might contain quotation marks, semicolons, `%` and `_` wildcard characters and other characters significant in SQL statements. Validate input values to make sure they have only the expected characters. Escape any special characters that could change the intended behavior when substituted into a SQL statement. Never concatenate a user input value into a SQL statement without doing validation and escaping first. Even when accepting input generated by some other program, expect that the other program could also have been compromised and be sending you incorrect or malicious data.

- Because the result sets from SQL queries can be very large, use the appropriate method to retrieve items from the result set as you loop through them. fetchone() retrieves a single item, when you know the result set contains a single row. fetchall() retrieves all the items, when you know the result set contains a limited number of rows that can fit comfortably into memory. fetchmany() is the general-purpose method when you cannot predict the size of the result set: you keep calling it and looping through the returned items, until there are no more results to process.

- Since Python already has convenient modules such as `pickle` and `cPickle` to read and write data structures on disk, data that you choose to store in MySQL instead is likely to have special characteristics:

  - **Too large to all fit in memory at one time.** You use `SELECT` statements to query only the precise items you need, and aggregate functions to perform calculations across multiple items. You configure the `innodb_buffer_pool_size` option within the MySQL server to dedicate a certain amount of RAM for caching query results.

  - **Too complex to be represented by a single data structure.** You divide the data between different SQL tables. You can recombine data from multiple tables by using a join query. You make sure that related data is kept in sync between different tables by setting up foreign key relationships.

  - **Updated frequently, perhaps by multiple users simultaneously.** The updates might only affect a small portion of the data, making it wasteful to write the whole structure each time. You use the SQL `INSERT`, `UPDATE`, and `DELETE` statements to update different items concurrently, writing only

the changed values to disk. You use `InnoDB` tables and transactions to keep write operations from conflicting with each other, and to return consistent query results even as the underlying data is being updated.

- Using MySQL best practices for performance can help your application to scale without requiring major rewrites and architectural changes. See Optimization for best practices for MySQL performance. It offers guidelines and tips for SQL tuning, database design, and server configuration.

- You can avoid reinventing the wheel by learning the MySQL SQL statements for common operations: operators to use in queries, techniques for bulk loading data, and so on. Some statements and clauses are extensions to the basic ones defined by the SQL standard. See Data Manipulation Statements, Data Definition Statements, and SELECT Syntax for the main classes of statements.

- Issuing SQL statements from Python typically involves declaring very long, possibly multi-line string literals. Because string literals within the SQL statements could be enclosed by single quotation, double quotation marks, or contain either of those characters, for simplicity you can use Python's triple-quoting mechanism to enclose the entire statement. For example:

  ```
  '''It doesn't matter if this string contains 'single'
  or "double" quotes, as long as there aren't 3 in a
  row.'''
  ```

  You can use either of the `'` or `"` characters for triple-quoting multi-line string literals.

- Many of the secrets to a fast, scalable MySQL application involve using the right syntax at the very start of your setup procedure, in the `CREATE TABLE` statements. For example, Oracle recommends the `ENGINE=INNODB` clause for most tables, and makes `InnoDB` the default storage engine in MySQL 5.5 and up. Using `InnoDB` tables enables transactional behavior that helps scalability of read-write workloads and offers automatic crash recovery. Another recommendation is to declare a numeric primary key for each table, which offers the fastest way to look up values and can act as a pointer to associated values in other tables (a foreign key). Also within the `CREATE TABLE` statement, using the most compact column data types that meet your application requirements helps performance and scalability because that enables the database server to move less data back and forth between memory and disk.

Chapter 3 Connector/Python Versions

The MySQL Connector/Python 2.0 is the current development series.

The MySQL Connector/Python 1.0, 1.1, and 1.2 series each went through a series of beta releases, leading to generally available (GA) versions. Any development releases in each series prior to the GA version are no longer supported.

The following table summarizes the available Connector/Python versions:

Table 3.1 Connector/Python Version Reference

Connector/Python Version	MySQL Server Versions	Python Versions	Support Status for Connector
2.0	5.7, 5.6, 5.5	3.3 and later, 2.7, 2.6	Recommended version
1.1, 1.2	5.7, 5.6, 5.5 (5.1, 5.0, 4.1)	3.1 and later, 2.7, 2.6	Recommended version
1.0	5.7, 5.6, 5.5 (5.1, 5.0, 4.1)	3.1 and later, 2.7, 2.6 (2.5, 2.4)	Recommended version

Note

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

Note

Connector/Python does not support the old MySQL Server authentication methods, which means that MySQL versions prior to 4.1 will not work.

Chapter 4 Connector/Python Installation

Table of Contents

Connector/Python runs on any platform where Python is installed. Python comes preinstalled on most Unix and Unix-like systems, such as Linux, OS X, and FreeBSD. On Microsoft Windows, a Python installer is available at the Python Download website. If necessary, download and install Python for Windows before attempting to install Connector/Python.

> **Note**
>
> Connector/Python requires `python` to be in the system's `PATH` and installation fails if `python` cannot be located. On Unix and Unix-like systems, `python` is normally located in a directory included in the default `PATH` setting. On Windows, if you install Python, either enable **Add python.exe to Path** during the installation process, or manually add the directory containing `python.exe` yourself.
>
> For more information about installation and configuration of Python on Windows, see Using Python on Windows in the Python documentation.

Connector/Python implements the MySQL client/server protocol two ways:

- As pure Python. This implementation of the protocol does not require any other MySQL client libraries or other components.
- As a C Extension that interfaces with the MySQL C client library. This implementation of the protocol is dependent on the client library, but can use the library provided by either MySQL Connector/C or MySQL Server packages (see MySQL C API Implementations). The C Extension is available as of Connector/Python 2.1.1.

Neither implementation of the client/server protocol has any third-party dependencies. However, if you need SSL support, verify that your Python installation has been compiled using the OpenSSL libraries.

Installation of Connector/Python is similar on every platform and follows the standard Python Distribution Utilities or Distutils. Distributions are available in native format for some platforms, such as RPM packages for Linux.

Python terminology regarding distributions:

- **Built Distribution**: A package created in the native packaging format intended for a given platform. It contains both sources and platform-independent bytecode. Connector/Python binary distributions are built distributions.
- **Source Distribution**: A distribution that contains only source files and is generally platform independent.

4.1 Obtaining Connector/Python

Packages are available at the Connector/Python download site. For some packaging formats, there are different packages for different versions of Python; choose the one appropriate for the version of Python installed on your system.

4.2 Installing Connector/Python from a Binary Distribution

Connector/Python installers in native package formats are available for Windows and for Unix and Unix-like systems:

- Windows: MSI installer package
- Linux: Yum repository for EL6 and EL7 and Fedora 20 and 21; RPM packages for Oracle Linux, Red Hat, and SuSE; Debian packages for Debian and Ubuntu
- OS X: Disk image package with PKG installer

You may need `root` or administrator privileges to perform the installation operation.

As of Connector/Python 2.1.1, binary distributions are available that include a C Extension that interfaces with the MySQL C client library. Some packaging types have a single distribution file that includes the pure-Python Connector/Python code together with the C Extension. (Windows MSI and OS X Disk Image packages fall into this category.) Other packaging types have two related distribution files: One that includes the pure-Python Connector/Python code, and one that includes only the C Extension. For packaging types that have separate distribution files, install both distributions if you want to use the C Extension. The two files have related names, the difference being that the one that contains the C Extension has “cext” in the distribution file name.

Binary distributions that provide the C Extension are either statically linked to MySQL Connector/C or link to an already installed C client library provided by a Connector/C or MySQL Server installation. For those distributions that are not statically linked, you must install Connector/C or MySQL Server if it is not already present on your system. To obtain either product, visit the MySQL download site.

Installing Connector/Python on Microsoft Windows Using an MSI Package

Connector/Python Windows MSI Installers (`.msi` files) are available from the Connector/Python download site (see Section 4.1, “Obtaining Connector/Python”). Choose an installer appropriate for the version of Python installed on your system. As of Connector/Python 2.1.1, MSI Installers include the C Extension; it need not be installed separately.

To use the MSI Installer, launch it and follow the prompts in the screens it presents to install Connector/Python in the location of your choosing.

Alternatively, to run the installer from the command line, use this command in a console window, where *VER* and *PYVER* are the respective Connector/Python and Python version numbers in the installer file name:

```
shell> msiexec /i mysql-connector-python-VER-pyPYVER.msi
```

Subsequent executions of Connector/Python using the MSI installer permit you to either repair or remove the existing Connector/Python installation.

Installing Connector/Python on Linux Using the MySQL Yum Repository

For EL6 or EL7-based platforms and Fedora 19 or 20, you can install Connector/Python using the MySQL Yum repository (see Installing Additional MySQL Products and Components with Yum). You must have the MySQL Yum repository on your system's repository list (for details, see Adding the MySQL Yum Repository). To make sure that your Yum repository is up-to-date, use this command:

```
```

```
shell> sudo yum update mysql-community-release
```

Then install Connector/Python as follows:

```
shell> sudo yum install mysql-connector-python
```

Installing Connector/Python on Linux Using an RPM Package

Connector/Python Linux RPM packages (`.rpm` files) are available from the Connector/Python download site (see Section 4.1, “Obtaining Connector/Python”).

To install a Connector/Python RPM package (denoted here as *`PACKAGE`*`.rpm`), use this command:

```
shell> rpm -i PACKAGE.rpm
```

To install the C Extension (available as of Connector/Python 2.1.1), install the corresponding package with “cext” in the package name.

RPM provides a feature to verify the integrity and authenticity of packages before installing them. To learn more, see Verifying Package Integrity Using MD5 Checksums or GnuPG.

Installing Connector/Python on Linux Using a Debian Package

Connector/Python Debian packages (`.deb` files) are available for Debian or Debian-like Linux systems from the Connector/Python download site (see Section 4.1, “Obtaining Connector/Python”).

To install a Connector/Python Debian package (denoted here as *`PACKAGE`*`.deb`), use this command:

```
shell> dpkg -i PACKAGE.deb
```

To install the C Extension (available as of Connector/Python 2.1.1), install the corresponding package with “cext” in the package name.

Installing Connector/Python on OS X Using a Disk Image

Connector/Python OS X disk images (`.dmg` files) are available from the Connector/Python download site (see Section 4.1, “Obtaining Connector/Python”). As of Connector/Python 2.1.1, OS X disk images include the C Extension; it need not be installed separately.

Download the `.dmg` file and install Connector/Python by opening it and double clicking the resulting `.pkg` file.

4.3 Installing Connector/Python from a Source Distribution

Connector/Python source distributions are platform independent and can be used on any platform. Source distributions are packaged in two formats:

- Zip archive format (`.zip` file)
- Compressed `tar` archive format (`.tar.gz` file)

Either packaging format can be used on any platform, but Zip archives are more commonly used on Windows systems and `tar` archives on Unix and Unix-like systems.

Prerequisites for Compiling Connector/Python with the C Extension

As of Connector/Python 2.1.1, source distributions include the C Extension that interfaces with the MySQL C client library. You can build the distribution with or without support for this extension. To build Connector/Python with support for the C Extension, you must satisfy the following prerequisites.

- Prerequisites for Windows systems:
  - Correct version of Visual Studio: VS 2009 for Python 2.7, VS 2010 for Python 3.3
  - Python development files
  - MySQL Connector/C or MySQL Server installed, including development files
- Prerequisites for Unix and Unix-like systems:
  - A C/C++ compiler, such as `gcc`
  - Python development files
  - MySQL Connector/C or MySQL Server installed, including development files

You must install Connector/C or MySQL Server if it is not already present on your system. To obtain either product, visit the MySQL download site.

For certain platforms, MySQL development files are provided in separate packages. This is true for RPM and Debian packages, for example.

Installing Connector/Python from Source on Microsoft Windows

A Connector/Python Zip archive (`.zip` file) is available from the Connector/Python download site (see Section 4.1, “Obtaining Connector/Python”).

To install Connector/Python from a Zip archive, download the latest version and follow these steps:

1. Unpack the Zip archive in the intended installation directory (for example, `C:\mysql-connector\`) using `WinZip` or another tool that can read `.zip` files.
2. Start a console window and change location to the folder where you unpacked the Zip archive:

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

3. Inside the Connector/Python folder, perform the installation using this command:

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

   To include the C Extension (available as of Connector/Python 2.1.1), use this command instead:

   ```
   shell> python setup.py install --with-mysql-capi="path_name"
   ```

   The argument to `--with-mysql-capi` is the path to the installation directory of either MySQL Connector/C or MySQL Server.

To see all options and commands supported by `setup.py`, use this command:

```
```

```
shell> python setup.py --help
```

Installing Connector/Python from Source on Unix and Unix-Like Systems

For Unix and Unix-like systems such as Linux, Solaris, OS X, and FreeBSD, a Connector/Python `tar` archive (`.tar.gz` file) is available from the Connector/Python download site (see Section 4.1, “Obtaining Connector/Python”).

To install Connector/Python from a `tar` archive, download the latest version (denoted here as *`VER`*), and execute these commands:

```
shell> tar xzf mysql-connector-python-VER.tar.gz
shell> cd mysql-connector-python-VER
shell> sudo python setup.py install
```

To include the C Extension (available as of Connector/Python 2.1.1), use this command instead:

```
shell> sudo python setup.py install --with-mysql-capi=value
```

The argument to `--with-mysql-capi` is the path to the installation directory of either MySQL Connector/C or MySQL Server, or the path to the `mysql_config` command.

To see all options and commands supported by `setup.py`, use this command:

```
shell> python setup.py --help
```

4.4 Verifying Your Connector/Python Installation

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

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

The C Extension is installed as `_mysql_connector.so` in the `site-packages` directory, not in the `mysql/connector` directory.

If you are not sure where Connector/Python is installed, do the following to determine its location. The output here shows installation locations as might be seen on OS X:

```
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
```

To test that your Connector/Python installation is working and able to connect to MySQL Server, you can run a very simple program where you supply the login credentials and host information required for the connection. For an example, see Section 5.1, “Connecting to MySQL Using Connector/Python”.

Chapter 5 Connector/Python Coding Examples

Table of Contents

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

5.1 Connecting to MySQL Using Connector/Python

The `connect()` constructor creates 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()
```

Section 7.1, “Connector/Python Connection Arguments” describes the permitted connection arguments.

It is also possible to create connection objects using the connection.MySQLConnection() class:

```
from mysql.connector import (connection)

cnx = connection.MySQLConnection(user='scott', password='tiger',
                                 host='127.0.0.1',
                                 database='employees')
cnx.close()
```

Both methods, using the `connect()` constructor, or the class directly, are valid and functionally equal, but using `connector()` is preferred and is 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 with your user name or password")
  elif err.errno == errorcode.ER_BAD_DB_ERROR:
    print("Database does not exist")
  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:

```
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()
```

Using the Connector/Python C Extension

As of Connector/Python 2.1.1, the `use_pure` connection argument determines whether to connect using a pure Python interface to MySQL, or a C Extension that uses the MySQL C client library (see Chapter 8, *The Connector/Python C Extension*). The default is `True` (use the pure Python implementation). Setting `use_pure` to `False` causes the connection to use the C Extension if your Connector/Python installation includes it. The following examples are similar to others shown previously but with the includion of `use_pure=False`.

Connect by naming arguments in the `connect()` call:

```
import mysql.connector

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

Connect using an argument dictionary:

```
import mysql.connector

config = {
  'user': 'scott',
  'password': 'tiger',
  'host': '127.0.0.1',
  'database': 'employees',
  'raise_on_warnings': True,
  'use_pure': False,
}

cnx = mysql.connector.connect(**config)

cnx.close()
```

It is also possible to use the C Extension directly, by importing the `_mysql_connector` module rather than the `mysql.connector` module. For more information, see Section 8.2, “The _mysql_connector C Extension Module”.

5.2 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 Employee Sample Database. You 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 preceding 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 enables you to easily use a different schema.

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

A single MySQL server can manage 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.

After we successfully create or change to the target database, we create the tables by iterating over the items of the `TABLES` dictionary:

```
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.msg)
    else:
        print("OK")

cursor.close()
cnx.close()
```

To handle the error when the table already exists, we notify the user that it was already there. Other errors are printed, but we continue creating tables. (The example shows 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, execute the following commands, 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
```

5.3 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` (the default in MySQL 5.5 and later), you must commit the data after a sequence of `INSERT`, `DELETE`, and `UPDATE` statements.

This example shows how to insert new data. The second `INSERT` depends on the value of the newly created primary key of the first. The example also demonstrates 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 Section 5.2, “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 `emp_no` column (an `AUTO_INCREMENT` column) using the `lastrowid` property of the cursor object.

Next, we insert the new salary for the new employee, using the `emp_no` variable in the dictionary holding the data. This dictionary is passed to the `execute()` method of the cursor object if an error occurred.

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.

5.4 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 and hire dates 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 is 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 6 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.

6.1 Tutorial: Raise Employee's Salary Using a Buffered Cursor

The following example script gives a long-overdue 15% raise effective tomorrow to all employees who joined in the year 2000 and are still with the company.

To iterate through the selected employees, we use buffered cursors. (A buffered cursor fetches and buffers the rows of a result set after executing a query; see Section 10.6.1, “Class cursor.MySQLCursorBuffered”.) This way, it is unnecessary to fetch the rows in a new variables. Instead, the cursor can be used as an iterator.

> **Note**
>
> This 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 employees getting a raise
curA.execute(query, (date(2000, 1, 1), date(2000, 12, 31)))

# 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 7 Connector/Python Connection Establishment

Table of Contents

Connector/Python provides a `connect()` call used to establish connections to the MySQL server. The following sections describe the permitted arguments for `connect()` and describe how to use option files that supply additional arguments.

7.1 Connector/Python Connection Arguments

A connection with the MySQL server can be established using either the `mysql.connector.connect()` function or the `mysql.connector.MySQLConnection()` class:

```
cnx = mysql.connector.connect(user='joe', database='test')
cnx = MySQLConnection(user='joe', database='test')
```

The following table describes the arguments that can be used to initiate a connection. An asterisk (*) following an argument indicates a synonymous argument name, available only for compatibility with other Python MySQL drivers. Oracle recommends not to use these alternative names.

Table 7.1 Connection Arguments for Connector/Python

Argument Name	Default	Description
`user` (`username`*)		The user name used to authenticate with the MySQL server.
`password` (`passwd`*)		The password to authenticate the user with the MySQL server.
`database` (`db`*)		The database name to use when connecting with the MySQL server.
`host`	127.0.0.1	The host name or IP address of the MySQL server.
`port`	3306	The TCP/IP port of the MySQL server. Must be an integer.
`unix_socket`		The location of the Unix socket file.
`auth_plugin`		Authentication plugin to use. Added in 1.2.1.
`use_unicode`	`True`	Whether to use Unicode.
`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 time.
`sql_mode`		Set the `sql_mode` session variable at connection time.
`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.

Argument Name	Default	Description
client_flags		MySQL client flags.
buffered	False	Whether cursor objects fetch the results immediately after executing queries.
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.
ssl_verify_cert	False	When set to True, checks the server certificate against the certificate file specified by the ssl_ca option. Any mismatch causes a ValueError exception.
force_ipv6	False	When set to True, uses IPv6 when an address resolves to both IPv4 and IPv6. By default, IPv4 is used in such cases.
dsn		Not supported (raises NotSupportedError when used).
pool_name		Connection pool name. Added in 1.1.1.
pool_size	5	Connection pool size. Added in 1.1.1.
pool_reset_session	True	Whether to reset session variables when connection is returned to pool. Added in 1.1.5.
compress	False	Whether to use compressed client/server protocol. Added in 1.1.2.
converter_class		Converter class to use. Added in 1.1.2.
fabric		MySQL Fabric connection arguments. Added in 1.2.0.
failover		Server failover sequence. Added in 1.2.1.
option_files		Which option files to read. Added in 2.0.0.
option_groups	['client', 'connector_python']	Which groups to read from option files. Added in 2.0.0.
allow_local_infile	True	Whether to enable LOAD DATA LOCAL INFILE. Added in 2.0.0.
use_pure	True	Whether to use pure Python or C Extension. Added in 2.1.1.

MySQL Authentication Options

Authentication with MySQL uses username and password.

> **Note**
>
> MySQL Connector/Python does not support the old, less-secure password protocols of MySQL versions prior to 4.1.

When the database argument is given, the current database is set to the given value. To change the current database later, execute a USE SQL statement or set the database property of the MySQLConnection instance.

By default, Connector/Python tries to connect to a MySQL server running on the local host 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.

Connector/Python 1.2.1 and up supports authentication plugins found in MySQL 5.6. This includes `mysql_clear_password` and `sha256_password`, both of which require an SSL connection. The `sha256_password` plugin does not work over a non-SSL connection because Connector/Python does not support RSA encryption.

The `connect()` method supports an `auth_plugin` argument that can be used to force use of a particular plugin. For example, if the server is configured to use `sha256_password` by default and you want to connect to an account that authenticates using `mysql_native_password`, either connect using SSL or specify `auth_plugin='mysql_native_password'`.

Character Encoding

By default, strings coming from MySQL are 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 `SET NAMES` SQL statement directly. Similar to the `charset` property, you can set the `collation` for the current MySQL session.

Transactions

The `autocommit` value defaults to `False`, so 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.

Time Zones

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

SQL Modes

MySQL supports so-called SQL Modes. which 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.

Troubleshooting and Error Handling

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`.

Enabling and Disabling Features Using Client Flags

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 properly. A better way to set and unset flags individually 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)
```

Buffered Cursors for Result Sets

By default, MySQL Connector/Python does not buffer or prefetch results. This means that after a query is executed, your program is responsible for fetching the data. This avoids excessive memory use when queries return large result sets. If you know that the result set is small enough to handle all at once, you can fetch the results immediately by setting `buffered` to `True`. It is also possible to set this per cursor (see Section 10.2.6, “Method MySQLConnection.cursor()”).

Type Conversions

By default, MySQL types in result sets are converted automatically to Python types. For example, a `DATETIME` column value becomes a datetime.datetime object. To disable conversion, set the `raw` argument to `True`. You might do this to get better performance or perform different types of conversion yourself.

Connecting through SSL

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 `ssl_ca`, `ssl_key` and `ssl_cert` arguments, the connection switches to SSL, and the `client_flags` option includes the `ClientFlag.SSL` value automatically. You can use this in combination with the `compressed` argument set to `True`.

As of Connector/Python 1.2.1, it is possible to establish an SSL connection using only the `ssl_ca` argument. The `ssl_key` and `ssl_cert` arguments are optional. However, when either is given, both must be given or an `AttributeError` is raised.

```
# Note (Example is valid for Python v2 and v3)
from __future__ import print_function

import sys

#sys.path.insert(0, 'python{0}/'.format(sys.version_info[0]))

import mysql.connector
from mysql.connector.constants import ClientFlag

config = {
    'user': 'ssluser',
    'password': 'asecret',
    'host': '127.0.0.1',
    'client_flags': [ClientFlag.SSL],
    'ssl_ca': '/opt/mysql/ssl/ca.pem',
    'ssl_cert': '/opt/mysql/ssl/client-cert.pem',
    'ssl_key': '/opt/mysql/ssl/client-key.pem',
}

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

```
cur = cnx.cursor(buffered=True)
cur.execute("SHOW STATUS LIKE 'Ssl_cipher'")
print(cur.fetchone())
cur.close()
cnx.close()
```

Connection Pooling

With either the `pool_name` or `pool_size` argument present, Connector/Python creates the new pool. If the `pool_name` argument is not given, the `connect()` call automatically generates the name, composed from whichever of the `host`, `port`, `user`, and `database` connection arguments are given, in that order. If the `pool_size` argument is not given, the default size is 5 connections.

The `pool_reset_session` permits control over whether session variables are reset when the connection is returned to the pool. The default is to reset them.

Connection pooling is supported as of Connector/Python 1.1.1. See Section 9.1, “Connector/Python Connection Pooling”.

Protocol Compression

The boolean `compress` argument indicates whether to use the compressed client/server protocol (default `False`). This provides an easier alternative to setting the `ClientFlag.COMPRESS` flag. This argument is available as of Connector/Python 1.1.2.

Converter Class

The `converter_class` argument takes a class and sets it when configuring the connection. An `AttributeError` is raised if the custom converter class is not a subclass of `conversion.MySQLConverterBase`. This argument is available as of Connector/Python 1.1.2. Before 1.1.2, setting a custom converter class is possible only after instantiating a new connection object.

The boolean `compress` argument indicates whether to use the compressed client/server protocol (default `False`). This provides an easier alternative to setting the `ClientFlag.COMPRESS` flag. This argument is available as of Connector/Python 1.1.2.

MySQL Fabric Support

To request a MySQL Fabric connection, provide a `fabric` argument that specifies to contact Fabric. For details, see Requesting a Fabric Connection.

Server Failover

As of Connector/Python 1.2.1, the `connect()` method accepts a `failover` argument that provides information to use for server failover in the event of connection failures. The argument value is a tuple or list of dictionaries (tuple is preferred because it is nonmutable). Each dictionary contains connection arguments for a given server in the failover sequence. Permitted dictionary values are: `user`, `password`, `host`, `port`, `unix_socket`, `database`, `pool_name`, `pool_size`.

Option File Support

As of Connector/Python 2.0.0, option files are supported using two options for `connect()`:

- `option_files`: Which option files to read. The value can be a file path name (a string) or a sequence of path name strings. By default, Connector/Python reads no option files, so this argument must be given explicitly to cause option files to be read. Files are read in the order specified.

- `option_groups`: Which groups to read from option files, if option files are read. The value can be an option group name (a string) or a sequence of group name strings. If this argument is not given, the default value is `['client, 'connector_python']` to read the `[client]` and `[connector_python]` groups.

For more information, see Section 7.2, “Connector/Python Option-File Support”.

LOAD DATA LOCAL INFILE

Prior to Connector/Python 2.0.0, to enable use of `LOAD DATA LOCAL INFILE`, clients had to explicitly set the `ClientFlag.LOCAL_FILES` flag. As of 2.0.0, this flag is enabled by default. To disable it, the `allow_local_infile` connection option can be set to `False` at connect time (the default is `True`).

Compatibitility with Other Connection Interfaces

`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.

Client/Server Protocol Implementation

Connector/Python can use a pure Python interface to MySQL, or a C Extension that uses the MySQL C client library. The `use_pure` connection argument determines which. The default is `True` (use the pure Python implementation). Setting `use_pure` to `False` causes the connection to use the C Extension if your Connector/Python installation includes it.

The `use_pure` argument is available as of Connector/Python 2.1.1. For more information, see Chapter 8, *The Connector/Python C Extension*.

7.2 Connector/Python Option-File Support

As of version 2.0.0, Connector/Python provides the capability of reading options from option files. (For general information about option files, see Using Option Files.) Two arguments for the `connect()` call control use of option files in Connector/Python programs:

- `option_files`: Which option files to read. The value can be a file path name (a string) or a sequence of path name strings. By default, Connector/Python reads no option files, so this argument must be given explicitly to cause option files to be read. Files are read in the order specified.

- `option_groups`: Which groups to read from option files, if option files are read. The value can be an option group name (a string) or a sequence of group name strings. If this argument is not given, the default value is `['client, 'connector_python']`, to read the `[client]` and `[connector_python]` groups.

Connector/Python also supports the `!include` and `!includedir` inclusion directives within option files. These directives work the same way as for other MySQL programs (see Using Option Files).

This example specifies a single option file as a string:

```
cnx = mysql.connector.connect(option_files='/etc/mysql/connectors.cnf')
```

This example specifies multiple option files as a sequence of strings:

```
mysql_option_files = [
```

```
    '/etc/mysql/connectors.cnf',
    './development.cnf',
]
cnx = mysql.connector.connect(option_files=mysql_option_files)
```

Connector/Python reads no option files by default, for backward compatibility with versions older than 2.0.0. This differs from standard MySQL clients such as `mysql` or `mysqldump`, which do read option files by default. To find out which option files the standard clients read on your system, invoke one of them with its `--help` option and examine the output. For example:

```
shell> mysql --help
...
Default options are read from the following files in the given order:
/etc/my.cnf /etc/mysql/my.cnf /usr/local/mysql/etc/my.cnf ~/.my.cnf
...
```

If you specify the `option_files` argument to read option files, Connector/Python reads the `[client]` and `[connector_python]` option groups by default. To specify explicitly which groups to read, use the `option_groups` connection argument. The following example causes only the `[connector_python]` group to be read:

```
cnx = mysql.connector.connect(option_files='/etc/mysql/connectors.cnf',
                              option_groups='connector_python')
```

Other connection arguments specified in the `connect()` call take precedence over options read from option files. Suppose that `/etc/mysql/connectors.conf` contains these lines:

```
[client]
database=cpyapp
```

The following `connect()` call includes no `database` connection argument, so the resulting connection uses `cpyapp`, the database specified in the option file:

```
cnx = mysql.connector.connect(option_files='/etc/mysql/connectors.cnf')
```

By contrast, the following `connect()` call specifies a default database different from the one found in the option file, so the resulting connection uses `cpyapp_dev` as the default database, not `cpyapp`:

```
cnx2 = mysql.connector.connect(option_files='/etc/mysql/connectors.cnf',
                               database='cpyapp_dev')
```

Connector/Python raises a `ValueError` if an option file cannot be read, or has already been read. This includes files read by inclusion directives.

For the `[connector_python]` group, only options supported by Connector/Python are accepted. Unrecognized options cause a `ValueError` to be raised.

For other option groups, Connector/Python ignores unrecognized options.

It is not an error for a named option group not to exist.

Chapter 8 The Connector/Python C Extension

Table of Contents

Connector/Python supports (as of version 2.1.1) a C Extension that interfaces with the MySQL C client library. For queries that return large result sets, using the C Extension can improve performance compared to a “pure Python” implementation of the MySQL client/server protocol. Section 8.1, “Application Development with the Connector/Python C Extension”, describes how applications that use the `mysql.connector` module can use the C Extension. It is also possible to use the C Extension directly, by importing the `_mysql_connector` module rather than the `mysql.connector` module. See Section 8.2, “The _mysql_connector C Extension Module”. For information about installing the C Extension, see Chapter 4, *Connector/Python Installation*.

8.1 Application Development with the Connector/Python C Extension

Installations of Connector/Python from version 2.1.1 on support a `use_pure` argument to `connect()` that indicates whether to use the pure Python interface to MySQL or the C Extension that uses the MySQL C client library:

- By default, `use_pure` is `True` and the C Extension is not used even for Connector/Python installations that include it.
- For Connector/Python installations that include the C Extension, enable it by passing a `use_pure=False` argument to `connect()`.
- For Connector/Python installations that do not include the C Extension, passing `use_pure=False` to `connect()` raises an exception.
- For older Connector/Python installations that know nothing of the C Extension (before version 2.1.1), passing `use_pure` to `connect()` raises an exception regardless of its value.

> **Note**
>
> On OS X, if your Connector/Python installation includes the C Extension, but Python scripts are unable to use it, try setting your `DYLD_LIBRARY_PATH` environment variable the directory containing the C client library. For example:
>
> ```
> export DYLD_LIBRARY_PATH=/usr/local/mysql/lib   (for sh)
> setenv DYLD_LIBRARY_PATH /usr/local/mysql/lib   (for tcsh)
> ```
>
> If you built the C Extension from source, this directory should be the one containing the C client library against which the extension was built.

If you need to check whether your Connector/Python installation is aware of the C Extension, test the `HAVE_CEXT` value. There are different approaches for this. Suppose that your usual arguments for `connect()` are specified in a dictionary:

```
config = {
```

```
  'user': 'scott',
  'password': 'tiger',
  'host': '127.0.0.1',
  'database': 'employees',
}
```

The following example illustrates one way to add `use_pure` to the connection arguments:

```
import mysql.connector

if mysql.connector.VERSION > (2, 1) and mysql.connector.HAVE_CEXT:
  config['use_pure'] = False
```

Alternatively, add `use_pure` to the configuration arguments as follows:

```
try:
  have_cext = mysql.connector.HAVE_CEXT
except AttributeError:
  have_cext = False

if have_cext:
  config['use_pure'] = False
```

8.2 The _mysql_connector C Extension Module

To use the C Extension directly, import the `_mysql_connector` module rather than `mysql.connector`, then use the `_mysql_connector.MySQL()` class to obtain a `MySQL` instance. For example:

```
import _mysql_connector

ccnx = _mysql_connector.MySQL()
ccnx.connect(user='scott', password='tiger',
             host='127.0.0.1', database='employees')

ccnx.query("SHOW VARIABLES LIKE 'version%'")
row = ccnx.fetch_row()
while row:
  print(row)
  row = ccnx.fetch_row()
ccnx.free_result()

ccnx.close()
```

For more information, see Chapter 11, *Connector/Python C Extension API Reference*.

Chapter 9 Connector/Python Other Topics

Table of Contents

This section describes additional Connection/Python features:

- Connection pooling: Section 9.1, “Connector/Python Connection Pooling”
- Django backend for MySQL: Section 9.3, “Connector/Python Django Backend”

9.1 Connector/Python Connection Pooling

MySQL Connector/Python 1.1.1 and up supports simple connection pooling that has these characteristics:

- The `mysql.connector.pooling` module implements pooling.
- A pool opens a number of connections and handles thread safety when providing connections to requesters.
- The size of a connection pool is configurable at pool creation time. It cannot be resized thereafter.
- A connection pool can be named at pool creation time. If no name is given, one is generated using the connection parameters.
- The connection pool name can be retrieved from the connection pool or connections obtained from it.
- It is possible to have multiple connection pools. This enables applications to support pools of connections to different MySQL servers, for example.
- For each connection request, the pool provides the next available connection. No round-robin or other scheduling algorithm is used. If a pool is exhausted, a `PoolError` is raised.
- It is possible to reconfigure the connection parameters used by a pool. These apply to connections obtained from the pool thereafter. Reconfiguring individual connections obtained from the pool by calling the connection `config()` method is not supported.

Applications that can benefit from connection-pooling capability include:

- Middleware that maintains multiple connections to multiple MySQL servers and requires connections to be readily available.
- Web sites that can have more “permanent” connections open to the MySQL server.

A connection pool can be created implicitly or explicitly.

To create a connection pool implicitly: Open a connection and specify one or more pool-related arguments (`pool_name`, `pool_size`). For example:

```
dbconfig = {
  "database": "test",
```

```
  "user":     "joe"
}

cnx = mysql.connector.connect(pool_name = "mypool",
                              pool_size = 3,
                              **dbconfig)
```

The pool name is restricted to alphanumeric characters and the special characters `.`, `_`, `*`, `$`, and `#`. The pool name must be no more than `pooling.CNX_POOL_MAXNAMESIZE` characters long (default 64).

The pool size must be greater than 0 and less than `pooling.CNX_POOL_MAXSIZE` (default 32).

With either the `pool_name` or `pool_size` argument present, Connector/Python creates the new pool. If the `pool_name` argument is not given, the `connect()` call automatically generates the name, composed from whichever of the `host`, `port`, `user`, and `database` connection arguments are given, in that order. If the `pool_size` argument is not given, the default size is 5 connections.

Subsequent calls to `connect()` that name the same connection pool return connections from the existing pool. Any `pool_size` or connection parameter arguments are ignored, so the following `connect()` calls are equivalent to the original `connect()` call shown earlier:

```
cnx = mysql.connector.connect(pool_name = "mypool", pool_size = 3)
cnx = mysql.connector.connect(pool_name = "mypool", **dbconfig)
cnx = mysql.connector.connect(pool_name = "mypool")
```

Pooled connections obtained by calling `connect()` with a pool-related argument have a class of `PooledMySQLConnection` (see Section 10.4, “Class pooling.PooledMySQLConnection”). `PooledMySQLConnection` pooled connection objects are similar to `MySQLConnection` unpooled connection objects, with these differences:

- To release a pooled connection obtained from a connection pool, invoke its `close()` method, just as for any unpooled connection. However, for a pooled connection, `close()` does not actually close the connection but returns it to the pool and makes it available for subsequent connection requests.
- A pooled connection cannot be reconfigured using its `config()` method. Connection changes must be done through the pool object itself, as described shortly.
- A pooled connection has a `pool_name` property that returns the pool name.

To create a connection pool explicitly: Create a `MySQLConnectionPool` object (see Section 10.3, “Class pooling.MySQLConnectionPool”):

```
dbconfig = {
  "database": "test",
  "user":     "joe"
}

cnxpool = mysql.connector.pooling.MySQLConnectionPool(pool_name = "mypool",
                                                      pool_size = 3,
                                                      **dbconfig)
```

To request a connection from the pool, use its `get_connection()` method:

```
cnx1 = cnxpool.get_connection()
cnx2 = cnxpool.get_connection()
```

When you create a connection pool explicitly, it is possible to use the pool object's `set_config()` method to reconfigure the pool connection parameters:

```
dbconfig = {
  "database": "performance_schema",
  "user":     "admin",
  "password": "secret"
}

cnxpool.set_config(**dbconfig)
```

Connections requested from the pool after the configuration change use the new parameters. Connections obtained before the change remain unaffected, but when they are closed (returned to the pool) are reopened with the new parameters before being returned by the pool for subsequent connection requests.

9.2 Connector/Python Fabric Support

MySQL Fabric is a system for managing a farm of MySQL servers (and other components). Fabric provides an extensible and easy to use system for managing a MySQL deployment for sharding and high-availability.

Connector/Python 1.2.0 and up supports Fabric and provides these capabilities:

- Requesting a connection to a MySQL server managed by Fabric is as transparent as possible to users already familiar with Connector/Python.
- Connector/Python is able to get a MySQL server connection given a high-availability group and a mode specifying whether the connection is read-only or also permits updates (read-write).
- Connector/Python supports sharding and is able to find the correct MySQL server for a given table or tables and key based on scope (local or global) and mode (read-only or read-write). `RANGE` and `HASH` mechanisms are supported transparently to the user.
- Among secondary MySQL servers in the same group, load balancing of read-only operations occurs based on server weight.
- Faulty MySQL servers are reported to Fabric, and failover is supported when failure occurs for a server in a group.
- Connector/Python caches information coming from Fabric to speed up operations. Failures connecting to a MySQL server reset this cache. The time to live for cached information can be set by Fabric, or a default is used otherwise.
- Fabric support applies to versions of Python supported by Connector/Python itself (see Chapter 3, *Connector/Python Versions*). In particular, you can use Connector/Python with Python 3.1 and later to establish Fabric connections, even though Fabric does not support Python 3.

For more information, see MySQL Fabric.

9.3 Connector/Python Django Backend

Connector/Python 1.1.1 and up includes a `mysql.connector.django` module that provides a Django backend for MySQL. This backend supports new features found in MySQL 5.6 such as fractional seconds support for temporal data types.

Django Configuration

Django uses a configuration file named `settings.py` that contains a variable called `DATABASES` (see https://docs.djangoproject.com/en/1.5/ref/settings/#std:setting-DATABASES). To configure Django to use

Connector/Python as the MySQL backend, the example found in the Django manual can be used as a basis:

```
DATABASES = {
    'default': {
        'NAME': 'user_data',
        'ENGINE': 'mysql.connector.django',
        'USER': 'mysql_user',
        'PASSWORD': 'priv4te',
        'OPTIONS': {
          'autocommit': True,
        },
    }
}
```

It is possible to add more connection arguments using `OPTIONS`.

Support for MySQL Features

Django can launch the MySQL client application `mysql`. When the Connector/Python backend does this, it arranges for the `sql_mode` system variable to be set to `TRADITIONAL` at startup.

Some MySQL features are enabled depending on the server version. For example, support for fractional seconds precision is enabled when connecting to a server from MySQL 5.6.4 or higher. Django's `DateTimeField` is stored in a MySQL column defined as `DATETIME(6)`, and `TimeField` is stored as `TIME(6)`. For more information about fractional seconds support, see Fractional Seconds in Time Values.

Django Unit Testing

Django comes with an extensive set of unit tests. To run these, use the `run_django_tests.py` script located in the Connector/Python distribution `support/django` directory. For example, to run the basic tests using Django 1.5, run the following commands:

```
shell> cd support/django
shell> python run_django_tests.py --django 1.5 --tests basic
```

The script can be run using Python 2 or Python 3. It downloads Django, unpacks it and starts the tests. To avoid the download and use an already-fetched version, use the `--offline` option.

To see the script requirements, run it with the `--help` option, or examine the script itself. Here is an overview:

- Two MySQL servers, configured as a master/slave pair
- A database named `django_tests` on each server

To alter the settings of the MySQL servers, modify the file `test_mysqlconnector_settings.py`, also located in the `support/django` directory.

Chapter 10 Connector/Python API Reference

Table of Contents

This chapter contains the public API reference for Connector/Python. Examples should be considered working for Python 2.7, and Python 3.1 and greater. They might also work for older versions (such as Python 2.4) unless they use features introduced in newer Python versions. For example, exception handling using the `as` keyword was introduced in Python 2.6 and will not work in Python 2.4.

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

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

10.1 Module mysql.connector

The `mysql.connector` module provides top-level methods and properties.

10.1.1 Method mysql.connector.connect()

This method sets up a connection, establishing a session with the MySQL server. If no arguments are given, it uses the already configured or default values. For a complete list of possible arguments, see Section 7.1, “Connector/Python Connection Arguments”.

A connection with the MySQL server can be established using either the `mysql.connector.connect()` method or the `mysql.connector.MySQLConnection()` class:

```
cnx = mysql.connector.connect(user='joe', database='test')
cnx = MySQLConnection(user='joe', database='test')
```

For descriptions of connection methods and properties, see Section 10.2, “Class connection.MySQLConnection”.

10.1.2 Property mysql.connector.apilevel

This property is a string that indicates the supported DB API level.

```
>>> mysql.connector.apilevel
'2.0'
```

10.1.3 Property mysql.connector.paramstyle

This property is a string that indicates the Connector/Python default parameter style.

```
>>> mysql.connector.paramstyle
'pyformat'
```

10.1.4 Property mysql.connector.threadsafety

This property is an integer that indicates the supported level of thread safety provided by Connector/Python.

```
>>> mysql.connector.threadsafety
1
```

10.1.5 Property mysql.connector.__version__

This property indicates the Connector/Python version as a string. It is available as of Connector/Python 1.1.0.

```
>>> mysql.connector.__version__
'1.1.0'
```

10.1.6 Property mysql.connector.__version_info__

This property indicates the Connector/Python version as an array of version components. It is available as of Connector/Python 1.1.0.

```
>>> mysql.connector.__version_info__
(1, 1, 0, 'a', 0)
```

10.2 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 statements and read the results.

10.2.1 Constructor connection.MySQLConnection(**kwargs)

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

For a complete list of arguments, see Section 7.1, “Connector/Python Connection Arguments”.

10.2.2 Method MySQLConnection.close()

Syntax:

```
cnx.close()
```

`close()` is a synonym for `disconnect()`. See Section 10.2.20, “Method MySQLConnection.disconnect()”.

For a connection obtained from a connection pool, `close()` does not actually close it but returns it to the pool and makes it available for subsequent connection requests. See Section 9.1, “Connector/Python Connection Pooling”.

10.2.3 Method MySQLConnection.commit()

This method sends a `COMMIT` statement to the MySQL server, committing the current transaction. Since by default Connector/Python does not autocommit, it is important to call this method after every transaction that modifies data for tables that use transactional storage engines.

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

To roll back instead and discard modifications, see the rollback() method.

10.2.4 Method MySQLConnection.config(**kwargs)

Syntax:

```
cnx.config(**kwargs)
```

Configures a `MySQLConnection` instance after it has been instantiated. For a complete list of possible arguments, see Section 7.1, “Connector/Python Connection Arguments”.

Arguments:

- `kwargs`: Connection arguments.

You could use the `config()` method to change (for example) the user name, then call `reconnect()`.

Example:

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

For a connection obtained from a connection pool, `config()` raises an exception. See Section 9.1, “Connector/Python Connection Pooling”.

10.2.5 Method MySQLConnection.connect()

Syntax:

```
MySQLConnection.connect(**kwargs)
```

This method sets up a connection, establishing a session with the MySQL server. If no arguments are given, it uses the already configured or default values. For a complete list of possible arguments, see Section 7.1, “Connector/Python Connection Arguments”.

Arguments:

- `kwargs`: Connection arguments.

Example:

```
cnx = MySQLConnection(user='joe', database='test')
```

For a connection obtained from a conection pool, the connection object class is `PooledMySQLConnection`. A pooled connection differs from an unpooled connection as described in Section 9.1, “Connector/Python Connection Pooling”.

10.2.6 Method MySQLConnection.cursor()

Syntax:

```
cursor = cnx.cursor([arg=value[, arg=value]...])
```

This method returns a `MySQLCursor()` object, or a subclass of it depending on the passed arguments. The returned object is a `cursor.CursorBase` instance. For more information about cursor objects, see Section 10.5, “Class cursor.MySQLCursor”, and Section 10.6, “cursor.MySQLCursor Subclasses”.

Arguments may be passed to the `cursor()` method to control what type of cursor to create:

- If `buffered` is `True`, the cursor fetches all rows from the server after an operation is executed. This is useful when queries return small result sets. `buffered` can be used alone, or in combination with the `dictionary` or `named_tuple` argument.

  `buffered` can also be passed to `connect()` to set the default buffering mode for all cursors created from the connection object. See Section 7.1, “Connector/Python Connection Arguments”.

  For information about the implications of buffering, see Section 10.6.1, “Class cursor.MySQLCursorBuffered”.

- If `raw` is `True`, the cursor skips the conversion from MySQL data types to Python types when fetching rows. A raw cursor is usually used to get better performance or when you want to do the conversion yourself.

  `raw` can also be passed to `connect()` to set the default raw mode for all cursors created from the connection object. See Section 7.1, “Connector/Python Connection Arguments”.

- If `dictionary` is `True`, the cursor returns rows as dictionaries. This argument is available as of Connector/Python 2.0.0.

- If `named_tuple` is `True`, the cursor returns rows as named tuples. This argument is available as of Connector/Python 2.0.0.

- If `prepared` is `True`, the cursor is used for executing prepared statements. This argument is available as of Connector/Python 1.1.2.

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

The returned object depends on the combination of the arguments. Examples:

- If not buffered and not raw: `MySQLCursor`
- If buffered and not raw: `MySQLCursorBuffered`
- If not buffered and raw: `MySQLCursorRaw`
- If buffered and raw: `MySQLCursorBufferedRaw`

10.2.7 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.

10.2.8 Method MySQLConnection.cmd_debug()

Instructs the server to write debugging information to the error log. The connected user must have the `SUPER` privilege.

Returns a dictionary containing the OK packet information.

10.2.9 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 include no explicit database qualifier.

Returns a dictionary containing the OK packet information.

10.2.10 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.

10.2.11 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`.

10.2.12 Method MySQLConnection.cmd_process_kill(mysql_pid)

Asks the server to kill the thread specified by `mysql_pid`. Although still available, it is better to use the `KILL` SQL statement.

Returns a dictionary containing the OK packet information.

The following two lines have the same effect:

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

10.2.13 Method MySQLConnection.cmd_query(statement)

This method sends the given `statement` to the MySQL server and returns a result. To send multiple statements, use the cmd_query_iter() method instead.

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.

10.2.14 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_iter(statement):
  if 'columns' in result:
    columns = result['columns']
    rows = cnx.get_rows()
  else:
    # do something useful with INSERT result
```

Returns a generator object.

10.2.15 Method MySQLConnection.cmd_quit()

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

10.2.16 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 bitmask value constructed using constants from the `constants.RefreshOption` class.

For a list of options, see Section 10.11, “Class constants.RefreshOption”.

Example:

```
>>> from mysql.connector import RefreshOption
>>> refresh = RefreshOption.LOG | RefreshOption.THREADS
```

```
>>> cnx.cmd_refresh(refresh)
```

10.2.17 Method MySQLConnection.cmd_reset_connection()

Syntax:

```
cnx.cmd_reset_connection()
```

Resets the connection by sending a `COM_RESET_CONNECTION` command to the server to clear the session state.

This method permits the session state to be cleared without reauthenticating. For MySQL servers older than 5.7.3 (when `COM_RESET_CONNECTION` was introduced), the `reset_session()` method can be used instead. That method resets the session state by reauthenticating, which is more expensive.

This method was added in Connector/Python 1.2.1.

10.2.18 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.

10.2.19 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.

10.2.20 Method MySQLConnection.disconnect()

This method tries to send a `QUIT` command and close the socket. It raises no exceptions.

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

To shut down the connection without sending a `QUIT` command first, use `shutdown()`.

10.2.21 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.

10.2.22 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 sequences and the EOF packet information. The count argument can be used to obtain a given number 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.

MySQLCursor uses the `get_rows()` method to fetch rows.

Returns a tuple.

10.2.23 Method MySQLConnection.get_server_info()

This method returns the MySQL server information verbatim as a string, for example `'5.6.11-log'`, or `None` when not connected.

10.2.24 Method MySQLConnection.get_server_version()

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

10.2.25 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.

10.2.26 Method MySQLConnection.isset_client_flag(flag)

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

10.2.27 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.

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

Attempt to reconnect to 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.

10.2.29 Method MySQLConnection.reset_session()

Syntax:

```
cnx.reset_session(user_variables = None, session_variables = None)
```

Resets the connection by reauthenticating to clear the session state. `user_variables`, if given, is a dictionary of user variable names and values. `session_variables`, if given, is a dictionary of system variable names and values. The method sets each variable to the given value.

Example:

```
user_variables = {'var1': '1', 'var2': '10'}
session_variables = {'wait_timeout': 100000, 'sql_mode': 'TRADITIONAL'}
self.cnx.reset_session(user_variables, session_variables)
```

This method resets the session state by reauthenticating, which is expensive. For MySQL servers 5.7.3 or later, the `cmd_reset_connection()` method can be used instead. It is more lightweight because it permits the session state to be cleared without reauthenticating.

This method was added in Connector/Python 1.2.1.

10.2.30 Method MySQLConnection.rollback()

This method sends a `ROLLBACK` statement to the MySQL server, undoing all data changes from the current transaction. By default, Connector/Python does not autocommit, so it is possible to cancel transactions when using transactional storage engines such as `InnoDB`.

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

To commit modifications, see the commit() method.

10.2.31 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 collation for the character set is used.

In the following example, we set the character set to `latin1` and the collation to `latin1_swedish_ci` (the default collation for: `latin1`):

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

Specify a given collation as follows:

```
```

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

10.2.32 Method MySQLConnection.set_client_flags(flags)

This method sets the client flags to use when connecting to the MySQL server, and returns the new value as an integer. The `flags` argument can be either an integer or a sequence of valid client flag values (see Section 10.7, “Class constants.ClientFlag”).

If `flags` is a sequence, each item in the sequence sets the flag when the value is positive or unsets 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**
>
> Client flags are only set or used when connecting to the MySQL server. It is therefore necessary to reconnect after making changes.

10.2.33 Method MySQLConnection.shutdown()

This method closes the socket. It raises no exceptions.

Unlike `disconnect()`, `shutdown()` closes the client connection without attempting to send a `QUIT` command to the server first. Thus, it will not block if the connection is disrupted for some reason such as network failure.

`shutdown()` was added in Connector/Python 2.0.1.

10.2.34 Method MySQLConnection.start_transaction()

This method starts a transaction. It accepts arguments indicating whether to use a consistent snapshot, which transaction isolation level to use, and the transaction access mode:

```
cnx.start_transaction(consistent_snapshot=bool,
                      isolation_level=level,
                      readonly=access_mode)
```

The default `consistent_snapshot` value is `False`. If the value is `True`, Connector/Python sends `WITH CONSISTENT SNAPSHOT` with the statement. MySQL ignores this for isolation levels for which that option does not apply.

The default `isolation_level` value is `None`, and permitted values are `'READ UNCOMMITTED'`, `'READ COMMITTED'`, `'REPEATABLE READ'`, and `'SERIALIZABLE'`. If the `isolation_level` value is `None`, no isolation level is sent, so the default level applies.

The `readonly` argument can be `True` to start the transaction in `READ ONLY` mode or `False` to start it in `READ WRITE` mode. If `readonly` is omitted, the server's default access mode is used. For details about transaction access mode, see the description for the `START TRANSACTION` statement at START TRANSACTION, COMMIT, and ROLLBACK Syntax. If the server is older than MySQL 5.6.5, it does not support setting the access mode and Connector/Python raises a `ValueError`.

Invoking `start_transaction()` raises a `ProgrammingError` if invoked while a transaction is currently in progress. This differs from executing a `START TRANSACTION` SQL statement while a transaction is in progress; the statement implicitly commits the current transaction.

To determine whether a transaction is active for the connection, use the in_transaction property.

`start_transaction()` was added in MySQL Connector/Python 1.1.0. The `readonly` argument was added in Connector/Python 1.1.5.

10.2.35 Property MySQLConnection.autocommit

This property can be assigned a value of `True` or `False` to enable or disable the autocommit feature of MySQL. The property can be invoked to retrieve the current autocommit setting.

> **Note**
>
> Autocommit is disabled by default when connecting through Connector/Python. This can be enabled using the `autocommit` connection parameter.

When the autocommit is turned off, you must commit transactions when using transactional storage engines such as `InnoDB` or `NDBCluster`.

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

10.2.36 Property MySQLConnection.charset_name

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

10.2.37 Property MySQLConnection.collation_name

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

10.2.38 Property MySQLConnection.connection_id

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

10.2.39 Property MySQLConnection.database

This property sets the current (default) database by executing a `USE` statement. 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.

10.2.40 Property MySQLConnection.get_warnings

This property can be assigned a value of `True` or `False` to enable or disable whether warnings should be fetched automatically. The default is `False` (default). The property can be invoked to retrieve the current warnings setting.

Fetching warnings automatically can be useful when debugging queries. Cursors 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`.

10.2.41 Property MySQLConnection.in_transaction

This property returns `True` or `False` to indicate whether a transaction is active for the connection. The value is `True` regardless of whether you start a transaction using the `start_transaction()` API call or by directly executing a SQL statement such as `START TRANSACTION` or `BEGIN`.

```
>>> cnx.start_transaction()
>>> cnx.in_transaction
True
>>> cnx.commit()
>>> cnx.in_transaction
False
```

`in_transaction` was added in MySQL Connector/Python 1.1.0.

10.2.42 Property MySQLConnection.raise_on_warnings

This property can be assigned a value of `True` or `False` to enable or disable whether warnings should raise exceptions. The default is `False` (default). The property can be invoked to retrieve the current exceptions setting.

Setting `raise_on_warnings` also sets `get_warnings` because warnings need to be fetched so they can be raised as exceptions.

> **Note**
>
> You might always want to set the SQL mode if you would like to have the MySQL server directly report warnings as errors (see Section 10.2.45, “Property MySQLConnection.sql_mode”). 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 that 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`.

10.2.43 Property MySQLConnection.server_host

This read-only property returns the host name or IP address used for connecting to the MySQL server.

Returns a string.

10.2.44 Property MySQLConnection.server_port

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

Returns an integer.

10.2.45 Property MySQLConnection.sql_mode

This property is used to retrieve and set the SQL Modes for the current connection. The value should be a 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.

10.2.46 Property MySQLConnection.time_zone

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

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

Returns a string.

10.2.47 Property MySQLConnection.unix_socket

This read-only property returns the Unix socket file for connecting to the MySQL server.

Returns a string.

10.2.48 Property MySQLConnection.user

This read-only property returns the user name used for connecting to the MySQL server.

Returns a string.

10.3 Class pooling.MySQLConnectionPool

This class provides for the instantiation and management of connection pools.

10.3.1 Constructor pooling.MySQLConnectionPool

Syntax:

```
MySQLConnectionPool(pool_name=None,
                    pool_size=5,
                    pool_reset_session=True,
                    **kwargs)
```

This constructor instantiates an object that manages a connection pool.

Arguments:

- `pool_name`: The pool name. If this argument is not given, Connector/Python automatically generates the name, composed from whichever of the `host`, `port`, `user`, and `database` connection arguments are given in `kwargs`, in that order.

  It is not an error for multiple pools to have the same name. An application that must distinguish pools by their `pool_name` property should create each pool with a distinct name.

- `pool_size`: The pool size. If this argument is not given, the default is 5.

- `pool_reset_session`: Whether to reset session variables when the connection is returned to the pool. This argument was added in Connector/Python 1.1.5. Before 1.1.5, session variables are not reset.

- `kwargs`: Optional additional connection arguments, as described in Section 7.1, “Connector/Python Connection Arguments”.

Example:

```
dbconfig = {
  "database": "test",
  "user":     "joe",
}

cnxpool = mysql.connector.pooling.MySQLConnectionPool(pool_name = "mypool",
                                                      pool_size = 3,
                                                      **dbconfig)
```

10.3.2 Method MySQLConnectionPool.add_connection()

Syntax:

```
cnxpool.add_connection(cnx = None)
```

This method adds a new or existing `MySQLConnection` to the pool, or raises a `PoolError` if the pool is full.

Arguments:

- `cnx`: The `MySQLConnection` object to be added to the pool. If this argument is missing, the pool creates a new connection and adds it.

Example:

```
cnxpool.add_connection()    # add new connection to pool
cnxpool.add_connection(cnx) # add existing connection to pool
```

10.3.3 Method MySQLConnectionPool.get_connection()

Syntax:

```
cnxpool.get_connection()
```

This method returns a connection from the pool, or raises a `PoolError` if no connections are available.

Example:

```
cnx = cnxpool.get_connection()
```

10.3.4 Method MySQLConnectionPool.set_config()

Syntax:

```
cnxpool.set_config(**kwargs)
```

This method sets the configuration parameters for connections in the pool. Connections requested from the pool after the configuration change use the new parameters. Connections obtained before the change remain unaffected, but when they are closed (returned to the pool) are reopened with the new parameters before being returned by the pool for subsequent connection requests.

Arguments:

- `kwargs`: Connection arguments.

Example:

```
dbconfig = {
  "database": "performance_schema",
  "user":     "admin",
  "password": "secret",
}

cnxpool.set_config(**dbconfig)
```

10.3.5 Property MySQLConnectionPool.pool_name

Syntax:

```
cnxpool.pool_name
```

This property returns the connection pool name.

Example:

```
name = cnxpool.pool_name
```

10.4 Class pooling.PooledMySQLConnection

This class is used by `MySQLConnectionPool` to return a pooled connection instance. It is also the class used for connections obtained with calls to the `connect()` method that name a connection pool (see Section 9.1, “Connector/Python Connection Pooling”).

`PooledMySQLConnection` pooled connection objects are similar to `MySQLConnection` unpooled connection objects, with these differences:

- To release a pooled connection obtained from a connection pool, invoke its `close()` method, just as for any unpooled connection. However, for a pooled connection, `close()` does not actually close the connection but returns it to the pool and makes it available for subsequent connection requests.
- A pooled connection cannot be reconfigured using its `config()` method. Connection changes must be done through the pool object itself, as described shortly.
- A pooled connection has a `pool_name` property that returns the pool name.

10.4.1 Constructor pooling.PooledMySQLConnection

Syntax:

```
PooledMySQLConnection(cnxpool, cnx)
```

This constructor takes connection pool and connection arguments and returns a pooled connection. It is used by the `MySQLConnectionPool` class.

Arguments:

- `cnxpool`: A `MySQLConnectionPool` instance.
- `cnx`: A `MySQLConnection` instance.

Example:

```
pcnx = mysql.connector.pooling.PooledMySQLConnection(cnxpool, cnx)
```

10.4.2 Method PooledMySQLConnection.close()

Syntax:

```
cnx.close()
```

Returns a pooled connection to its connection pool.

For a pooled connection, `close()` does not actually close it but returns it to the pool and makes it available for subsequent connection requests.

If the pool configuration parameters are changed, a returned connection is closed and reopened with the new configuration before being returned from the pool again in response to a connection request.

10.4.3 Method PooledMySQLConnection.config()

For pooled connections, the `config()` method raises a `PoolError` exception. Configuration for pooled connections should be done using the pool object.

10.4.4 Property PooledMySQLConnection.pool_name

Syntax:

```
cnx.pool_name
```

This property returns the name of the connection pool to which the connection belongs.

Example:

```
cnx = cnxpool.get_connection()
name = cnx.pool_name
```

10.5 Class cursor.MySQLCursor

The `MySQLCursor` class instantiates objects that can execute operations such as SQL statements. Cursor objects interact with the MySQL server using a `MySQLConnection` object.

To create a cursor, use the `cursor()` method of a connection object:

```
import mysql.connector

cnx = mysql.connector.connect(database='world')
cursor = cnx.cursor()
```

Several related classes inherit from `MySQLCursor`. To create a cursor of one of these types, pass the appropriate arguments to `cursor()`:

- `MySQLCursorBuffered` creates a buffered cursor. See Section 10.6.1, “Class cursor.MySQLCursorBuffered”.

  ```
  cursor = cnx.cursor(buffered=True)
  ```

- `MySQLCursorRaw` creates a raw cursor. See Section 10.6.2, “Class cursor.MySQLCursorRaw”.

  ```
  cursor = cnx.cursor(raw=True)
  ```

- `MySQLCursorBufferedRaw` creates a buffered raw cursor. See Section 10.6.3, “Class cursor.MySQLCursorBufferedRaw”.

  ```
  cursor = cnx.cursor(raw=True, buffered=True)
  ```

- `MySQLCursorDict` creates a cursor that returns rows as dictionaries. See Section 10.6.4, “Class cursor.MySQLCursorDict”.

```
cursor = cnx.cursor(dictionary=True)
```

- `MySQLCursorBufferedDict` creates a buffered cursor that returns rows as dictionaries. See Section 10.6.5, “Class cursor.MySQLCursorBufferedDict”.

```
cursor = cnx.cursor(dictionary=True, buffered=True)
```

- `MySQLCursorNamedTuple` creates a cursor that returns rows as named tuples. See Section 10.6.6, “Class cursor.MySQLCursorNamedTuple”.

```
cursor = cnx.cursor(named_tuple=True)
```

- `MySQLCursorBufferedNamedTuple` creates a buffered cursor that returns rows as named tuples. See Section 10.6.7, “Class cursor.MySQLCursorBufferedNamedTuple”.

```
cursor = cnx.cursor(named_tuple=True, buffered=True)
```

- `MySQLCursorPrepared` creates a cursor for executing prepared statements. See Section 10.6.8, “Class cursor.MySQLCursorPrepared”.

```
cursor = cnx.cursor(prepared=True)
```

10.5.1 Constructor cursor.MySQLCursor

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

```
import mysql.connector

cnx = mysql.connector.connect(database='world')
cursor = cnx.cursor()
```

It is also possible to instantiate a cursor by passing a `MySQLConnection` object to `MySQLCursor`:

```
import mysql.connector
from mysql.connector.cursor import MySQLCursor

cnx = mysql.connector.connect(database='world')
cursor = MySQLCursor(cnx)
```

The connection argument is optional. If omitted, the cursor is created but its `execute()` method raises an exception.

10.5.2 Method MySQLCursor.callproc()

Syntax:

```
result_args = cursor.callproc(proc_name, args=())
```

This method calls the stored procedure named by the `proc_name` argument. The `args` sequence of parameters must contain one entry for each argument that the procedure expects. `callproc()` returns

a modified copy of the input sequence. Input parameters are left untouched. Output and input/output parameters may be replaced with new values.

Result sets produced by the stored procedure are automatically fetched and stored as MySQLCursorBuffered instances. For more information about using these result sets, see `stored_results()`.

Suppose that a stored procedure takes two parameters, multiplies the values, and returns the product:

```
CREATE PROCEDURE multiply(IN pFac1 INT, IN pFac2 INT, OUT pProd INT)
BEGIN
  SET pProd := pFac1 * pFac2;
END;
```

The following example shows how to execute the `multiply()` procedure:

```
>>> args = (5, 6, 0) # 0 is to hold value of the OUT parameter pProd
>>> cursor.callproc('multiply', args)
('5', '6', 30L)
```

Connector/Python 1.2.1 and up permits parameter types to be specified. To do this, specify a parameter as a two-item tuple consisting of the parameter value and type. Suppose that a procedure `sp1()` has this definition:

```
CREATE PROCEDURE sp1(IN pStr1 VARCHAR(20), IN pStr2 VARCHAR(20),
                     OUT pConCat VARCHAR(100))
BEGIN
  SET pConCat := CONCAT(pStr1, pStr2);
END;
```

To execute this procedure from Connector/Python, specifying a type for the `OUT` parameter, do this:

```
args = ('ham', 'eggs', (0, 'CHAR'))
result_args = cursor.callproc('sp1', args)
print(result_args[2])
```

10.5.3 Method MySQLCursor.close()

Syntax:

```
cursor.close()
```

Use `close()` when you are done using a cursor. This method closes the cursor, resets all results, and ensures that the cursor object has no reference to its original connection object.

10.5.4 Method MySQLCursor.execute()

Syntax:

```
cursor.execute(operation, params=None, multi=False)
iterator = cursor.execute(operation, params=None, multi=True)
```

This method executes the given database `operation` (query or command). The parameters found in the tuple or dictionary `params` are bound to the variables in the operation. Specify variables using `%s` or

`%(name)s` parameter style (that is, using `format` or `pyformat` style). `execute()` returns an iterator if `multi` is `True`.

This example inserts information about a new employee, then selects the data for that person. The statements are executed as separate `execute()` operations:

```
insert_stmt = (
  "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_stmt, data)

select_stmt = "SELECT * FROM employees WHERE emp_no = %(emp_no)s"
cursor.execute(select_stmt, { 'emp_no': 2 })
```

The data values are converted as necessary from Python objects to something MySQL understands. In the preceding example, the `datetime.date()` instance is converted to `'2012-03-23'`.

If `multi` is set to `True`, `execute()` is able to execute multiple statements specified in the `operation` string. It returns an iterator that enables processing the result of each statement. However, using parameters does not work well in this case, and it is usually a good idea to execute each statement on its own.

The following example selects and inserts data in a single `execute()` operation and displays the result of each statement:

```
operation = 'SELECT 1; INSERT INTO t1 VALUES (); SELECT 2'
for result in cursor.execute(operation, multi=True):
  if result.with_rows:
    print("Rows produced by statement '{}':".format(
      result.statement))
    print(result.fetchall())
  else:
    print("Number of rows affected by statement '{}': {}".format(
      result.statement, result.rowcount))
```

If the connection is configured to fetch warnings, warnings generated by the operation are available through the MySQLCursor.fetchwarnings() method.

10.5.5 Method MySQLCursor.executemany()

Syntax:

```
cursor.executemany(operation, seq_of_params)
```

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

In most cases, the `executemany()` method iterates through the sequence of parameters, each time passing the current parameters to the the `execute()` method.

An optimization is applied for inserts: The data values given by the parameter sequences are batched using multiple-row syntax. The following example inserts three 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)
```

For the preceding example, the `INSERT` statement sent to MySQL is:

```
INSERT INTO employees (first_name, hire_date)
VALUES ('Jane', '2005-02-12'), ('Joe', '2006-05-23'), ('John', '2010-10-03')
```

With the `executemany()` method, it is not possible to specify multiple statements to execute in the `operation` argument. Doing so raises an `InternalError` exception. Consider using `execute()` with `multi=True` instead.

10.5.6 Method MySQLCursor.fetchall()

Syntax:

```
rows = cursor.fetchall()
```

The method fetches all (or all remaining) rows of a query result set and returns a list of tuples. If no more rows are available, it returns an empty list.

The following example shows how to retrieve the first two rows of a result set, and then retrieve any remaining rows:

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

You must fetch all rows for the current query before executing new statements using the same connection.

10.5.7 Method MySQLCursor.fetchmany()

Syntax:

```
rows = cursor.fetchmany(size=1)
```

This method fetches the next set of rows of a query result and returns a list of tuples. If no more rows are available, it returns an empty list.

The number of rows returned can be specified using the `size` argument, which defaults to one. Fewer rows are returned if fewer rows are available than specified.

You must fetch all rows for the current query before executing new statements using the same connection.

10.5.8 Method MySQLCursor.fetchone()

Syntax:

```
row = cursor.fetchone()
```

This method retrieves the next row of a query result set and returns a single sequence, or `None` if no more rows are available. By default, the returned tuple consists of data returned by the MySQL server, converted

to Python objects. If the cursor is a raw cursor, no such conversion occurs; see Section 10.6.2, “Class cursor.MySQLCursorRaw”.

The `fetchone()` method is used by fetchall() and fetchmany(). It is also used when a cursor is used as an iterator.

The following example shows two equivalent ways to process a query result. The first uses `fetchone()` in a `while` loop, the second uses the cursor as an iterator:

```
# 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)
```

You must fetch all rows for the current query before executing new statements using the same connection.

10.5.9 Method MySQLCursor.fetchwarnings()

Syntax:

```
tuples = cursor.fetchwarnings()
```

This method returns a list of tuples containing warnings generated by the previously executed operation. To set whether to fetch warnings, use the connection's `get_warnings` property.

The following example shows a `SELECT` statement that generates 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'")]
```

When warnings are generated, it is possible to raise errors instead, using the connection's `raise_on_warnings` property.

10.5.10 Method MySQLCursor.stored_results()

Syntax:

```
iterator = cursor.stored_results()
```

This method returns a list iterator object that can be used to process result sets produced by a stored procedure executed using the callproc() method. The result sets remain available until you use the cursor to execute another operation or call another stored procedure.

The following example executes a stored procedure that produces two result sets, then uses `stored_results()` to retrieve them:

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

10.5.11 Property MySQLCursor.column_names

Syntax:

```
sequence = cursor.column_names
```

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

The following example shows how to create a dictionary from 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))
```

Alternatively, as of Connector/Python 2.0.0, you can fetch rows as dictionaries directly; see Section 10.6.4, “Class cursor.MySQLCursorDict”.

10.5.12 Property MySQLCursor.description

Syntax:

```
tuples = cursor.description
```

This read-only property returns a list of tuples describing the columns in a result set. Each tuple in the list contains values as follows:

```
(column_name,
 type,
 None,
 None,
 None,
 None,
 null_ok,
 column_flags)
```

The following example shows how to interpret `description` tuples:

```
import mysql.connector
from mysql.connector import FieldType

...

cursor.execute("SELECT emp_no, last_name, hire_date "
               "FROM employees WHERE emp_no = %s", (123,))
for i in range(len(cursor.description)):
  print("Column {}:".format(i+1))
  desc = cursor.description[i]
```

```
  print("  column_name = {}".format(desc[0]))
  print("  type = {} ({})".format(desc[1], FieldType.get_info(desc[1])))
  print("  null_ok = {}".format(desc[6]))
  print("  column_flags = {}".format(desc[7]))
```

The output looks like this:

```
Column 1:
  column_name = emp_no
  type = 3 (LONG)
  null_ok = 0
  column_flags = 20483
Column 2:
  column_name = last_name
  type = 253 (VAR_STRING)
  null_ok = 0
  column_flags = 4097
Column 3:
  column_name = hire_date
  type = 10 (DATE)
  null_ok = 0
  column_flags = 4225
```

The `column_flags` value is an instance of the `constants.FieldFlag` class. To see how to interpret it, do this:

```
>>> from mysql.connector import FieldFlag
>>> FieldFlag.desc
```

10.5.13 Property MySQLCursor.lastrowid

Syntax:

```
id = cursor.lastrowid
```

This read-only property returns the value generated for an `AUTO_INCREMENT` column by the previous `INSERT` or `UPDATE` statement or `None` when there is no such value available. For example, if you perform an `INSERT` into a table that contains an `AUTO_INCREMENT` column, `lastrowid` returns the `AUTO_INCREMENT` value for the new row. For an example, see Section 5.3, “Inserting Data Using Connector/Python”.

The `lastrowid` property is like the `mysql_insert_id()` C API function; see mysql_insert_id().

10.5.14 Property MySQLCursor.statement

Syntax:

```
str = cursor.statement
```

This read-only property returns the last executed statement as a string. The `statement` property can be useful for debugging and displaying what was sent to the MySQL server.

The string can contain multiple statements if a multiple-statement string was executed. This occurs for `execute()` with `multi=True`. In this case, the `statement` property contains the entire statement string and the `execute()` call returns an iterator that can be used to process results from the individual statements. The `statement` property for this iterator shows statement strings for the individual statements.

10.5.15 Property MySQLCursor.with_rows

Syntax:

```
boolean = cursor.with_rows
```

This read-only property returns `True` or `False` to indicate whether the most recently executed operation produced rows.

The `with_rows` property is useful when it is necessary to determine whether a statement produces a result set and you need to fetch rows. The following example retrieves the rows returned by the `SELECT` statements, but reports only the affected-rows value for 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("Number of affected rows: {}".format(result.rowcount))
```

10.6 cursor.MySQLCursor Subclasses

The cursor classes described in the following sections inherit from the `MySQLCursor` class, which is described in Section 10.5, “Class cursor.MySQLCursor”.

10.6.1 Class cursor.MySQLCursorBuffered

The `MySQLCursorBuffered` class inherits from `MySQLCursor`.

After executing a query, a `MySQLCursorBuffered` cursor fetches the entire result set from the server and buffers the rows.

For queries executed using a buffered cursor, row-fetching methods such as `fetchone()` return rows from the set of buffered rows. For nonbuffered cursors, rows are not fetched from the server until a row-fetching method is called. In this case, you must be sure to fetch all rows of the result set before executing any other statements on the same connection, or an `InternalError` (Unread result found) exception will be raised.

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

To create a buffered cursor, use the `buffered` argument when calling a connection's `cursor()` method. Alternatively, to make all cursors created from the connection buffered by default, use the `buffered` connection argument.

Example:

```
import mysql.connector

cnx = mysql.connector.connect()

# Only this particular cursor will buffer results
cursor = cnx.cursor(buffered=True)
```

```
# All cursors created from cnx2 will be buffered by default
cnx2 = mysql.connector.connect(buffered=True)
```

For a practical use case, see Section 6.1, “Tutorial: Raise Employee's Salary Using a Buffered Cursor”.

10.6.2 Class cursor.MySQLCursorRaw

The `MySQLCursorRaw` class inherits from `MySQLCursor`.

A `MySQLCursorRaw` cursor skips the conversion from MySQL data types to Python types when fetching rows. A raw cursor is usually used to get better performance or when you want to do the conversion yourself.

To create a raw cursor, use the `raw` argument when calling a connection's `cursor()` method. Alternatively, to make all cursors created from the connection raw by default, use the `raw` connection argument.

Example:

```
import mysql.connector

cnx = mysql.connector.connect()

# Only this particular cursor will be raw
cursor = cnx.cursor(raw=True)

# All cursors created from cnx2 will be raw by default
cnx2 = mysql.connector.connect(raw=True)
```

10.6.3 Class cursor.MySQLCursorBufferedRaw

The `MySQLCursorBufferedRaw` class inherits from `MySQLCursor`.

A `MySQLCursorBufferedRaw` cursor is like a `MySQLCursorRaw` cursor, but is buffered: After executing a query, it fetches the entire result set from the server and buffers the rows. For information about the implications of buffering, see Section 10.6.1, “Class cursor.MySQLCursorBuffered”.

To create a buffered raw cursor, use the `raw` and `buffered` arguments when calling a connection's `cursor()` method. Alternatively, to make all cursors created from the connection raw and buffered by default, use the `raw` and `buffered` connection arguments.

Example:

```
import mysql.connector

cnx = mysql.connector.connect()

# Only this particular cursor will be raw and buffered
cursor = cnx.cursor(raw=True, buffered=True)

# All cursors created from cnx2 will be raw and buffered by default
cnx2 = mysql.connector.connect(raw=True, buffered=True)
```

10.6.4 Class cursor.MySQLCursorDict

The `MySQLCursorDict` class inherits from `MySQLCursor`. This class is available as of Connector/Python 2.0.0.

A `MySQLCursorDict` cursor returns each row as a dictionary. The keys for each dictionary object are the column names of the MySQL result.

Example:

```
cnx = mysql.connector.connect(database='world')
cursor = cnx.cursor(dictionary=True)
cursor.execute("SELECT * FROM country WHERE Continent = 'Europe'")

print("Countries in Europe:")
for row in cursor:
    print("* {Name}".format(Name=row['Name']
```

The preceding code produces output like this:

```
Countries in Europe:
* Albania
* Andorra
* Austria
* Belgium
* Bulgaria
...
```

It may be convenient to pass the dictionary to `format()` as follows:

```
cursor.execute("SELECT Name, Population FROM country WHERE Continent = 'Europe'")

print("Countries in Europe with population:")
for row in cursor:
    print("* {Name}: {Population}".format(**row))
```

10.6.5 Class cursor.MySQLCursorBufferedDict

The `MySQLCursorBufferedDict` class inherits from `MySQLCursor`. This class is available as of Connector/Python 2.0.0.

A `MySQLCursorBufferedDict` cursor is like a `MySQLCursorDict` cursor, but is buffered: After executing a query, it fetches the entire result set from the server and buffers the rows. For information about the implications of buffering, see Section 10.6.1, “Class cursor.MySQLCursorBuffered”.

To get a buffered cursor that returns dictionaries, add the `buffered` argument when instantiating a new dictionary cursor:

```
cursor = cnx.cursor(dictionary=True, buffered=True)
```

10.6.6 Class cursor.MySQLCursorNamedTuple

The `MySQLCursorNamedTuple` class inherits from `MySQLCursor`. This class is available as of Connector/Python 2.0.0.

A `MySQLCursorNamedTuple` cursor returns each row as a named tuple. The attributes for each named-tuple object are the column names of the MySQL result.

Example:

```
cnx = mysql.connector.connect(database='world')
```

```
cursor = cnx.cursor(named_tuple=True)
cursor.execute("SELECT * FROM country WHERE Continent = 'Europe'")

print("Countries in Europe with population:")
for row in cursor:
    print("* {Name}: {Population}".format(
        Name=row.Name,
        Population=row.Population
    ))
```

10.6.7 Class cursor.MySQLCursorBufferedNamedTuple

The `MySQLCursorBufferedNamedTuple` class inherits from `MySQLCursor`. This class is available as of Connector/Python 2.0.0.

A `MySQLCursorBufferedNamedTuple` cursor is like a `MySQLCursorNamedTuple` cursor, but is buffered: After executing a query, it fetches the entire result set from the server and buffers the rows. For information about the implications of buffering, see Section 10.6.1, “Class cursor.MySQLCursorBuffered”.

To get a buffered cursor that returns named tuples, add the `buffered` argument when instantiating a new named-tuple cursor:

```
cursor = cnx.cursor(named_tuple=True, buffered=True)
```

10.6.8 Class cursor.MySQLCursorPrepared

The `MySQLCursorPrepared` class inherits from `MySQLCursor`. This class is available as of Connector/Python 1.1.0.

In MySQL, there are two ways to execute a prepared statement:

- Use the `PREPARE` and `EXECUTE` statements.
- Use the binary client/server protocol to send and receive data. To repeatedly execute the same statement with different data for different executions, this is more efficient than using `PREPARE` and `EXECUTE`. For information about the binary protocol, see C API Prepared Statements.

In Connector/Python, there are two ways to create a cursor that enables execution of prepared statements using the binary protocol. In both cases, the `cursor()` method of the connection object returns a `MySQLCursorPrepared` object:

- The simpler syntax uses a `prepared=True` argument to the `cursor()` method. This syntax is available as of Connector/Python 1.1.2.

  ```
  import mysql.connector

  cnx = mysql.connector.connect(database='employees')
  cursor = cnx.cursor(prepared=True)
  ```

- Alternatively, create an instance of the `MySQLCursorPrepared` class using the `cursor_class` argument to the `cursor()` method. This syntax is available as of Connector/Python 1.1.0.

  ```
  import mysql.connector
  from mysql.connector.cursor import MySQLCursorPrepared

  cnx = mysql.connector.connect(database='employees')
  cursor = cnx.cursor(cursor_class=MySQLCursorPrepared)
  ```

A cursor instantiated from the `MySQLCursorPrepared` class works like this:

- The first time you pass a statement to the cursor's `execute()` method, it prepares the statement. For subsequent invocations of `execute()`, the preparation phase is skipped if the statement is the same.
- The `execute()` method takes an optional second argument containing a list of data values to associate with parameter markers in the statement. If the list argument is present, there must be one value per parameter marker.

Example:

```
cursor = cnx.cursor(prepared=True)
stmt = "SELECT fullname FROM employees WHERE id = %s" # (1)
cursor.execute(stmt, (5,))                            # (2)
# ... fetch data ...
cursor.execute(stmt, (10,))                           # (3)
# ... fetch data ...
```

1. The `%s` within the statement is a parameter marker. Do not put quote marks around parameter markers.
2. For the first call to the `execute()` method, the cursor prepares the statement. If data is given in the same call, it also executes the statement and you should fetch the data.
3. For subsequent `execute()` calls that pass the same SQL statement, the cursor skips the preparation phase.

Prepared statements executed with `MySQLCursorPrepared` can use the `format` (`%s`) or `qmark` (`?`) parameterization style. This differs from nonprepared statements executed with `MySQLCursor`, which can use the `format` or `pyformat` parameterization style.

To use multiple prepared statements simultaneously, instantiate multiple cursors from the `MySQLCursorPrepared` class.

10.7 Class constants.ClientFlag

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

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

See Section 10.2.32, “Method MySQLConnection.set_client_flags(flags)” and the connection argument `client_flag`.

The `ClientFlag` class cannot be instantiated.

10.8 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 to print the name of the data type for each column in a 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()
```

The `FieldType` class cannot be instantiated.

10.9 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 `sql_mode` property. See Section 10.2.45, “Property MySQLConnection.sql_mode”.

The `SQLMode` class cannot be instantiated.

10.10 Class constants.CharacterSet

This class provides all known MySQL characters sets and their default collations. For examples, see Section 10.2.31, “Method MySQLConnection.set_charset_collation(charset=None, collation=None)”.

The `CharacterSet` class cannot be instantiated.

10.11 Class constants.RefreshOption

This class performs various flush operations.

- `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`.

10.12 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 mostly follow the Python Database API 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 value (see Server Error Codes and Messages). The following table shows the SQLSTATE classes and the exception Connector/Python raises. It is, however, possible to redefine which exception is raised for each server error. The default exception is `DatabaseError`.

Table 10.1

SQLSTATE Class	Connector/Python Exception
02	DataError
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

SQLSTATE Class	Connector/Python Exception
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

10.12.1 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.

10.12.2 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` is internally 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 arguments 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 previously 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 exist (when the `DROP TABLE` statement does not include a `IF EXISTS` clause):

```
import mysql.connector
from mysql.connector import errorcode

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

Prior to Connector/Python 1.1.1, the original message passed to `errors.Error()` is not saved in such a way that it could be retrieved. Instead, the `Error.msg` attribute was formatted with the error number and SQLSTATE value. As of 1.1.1, only the original message is saved in the `Error.msg` attribute. The formatted value together with the error number and SQLSTATE value can be obtained by printing or getting the string representation of the error object. Example:

```
try:
  conn = mysql.connector.connect(database = "baddb")
except mysql.connector.Error as e:
  print "Error code:", e.errno        # error number
  print "SQLSTATE value:", e.sqlstate # SQLSTATE value
  print "Error message:", e.msg       # error message
  print "Error:", e                   # errno, sqlstate, msg values
  s = str(e)
```

```
  print "Error:", s                  # errno, sqlstate, msg values
```

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

10.12.3 Exception errors.DataError

This exception is raised when there were problems with the data. Examples are a column set to `NULL` that cannot be `NULL`, 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`.

10.12.4 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`.

10.12.5 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`.

10.12.6 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`.

10.12.7 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`.

10.12.8 Exception errors.NotSupportedError

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

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

10.12.9 Exception errors.OperationalError

This exception is raised for errors which are related to MySQL's operations. For example: too many connections; a host name could not be resolved; bad handshake; server is shutting down, communication errors.

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

10.12.10 Exception errors.PoolError

This exception is raised for connection pool errors. `errors.PoolError` is a subclass of `errors.Error`.

10.12.11 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`.

10.12.12 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`.

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

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

If `error` is a MySQL Server error number, you must also pass the `exception` class. The `error` argument can 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, 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({})
```

Chapter 11 Connector/Python C Extension API Reference

Table of Contents

This chapter contains the public API reference for the Connector/Python C Extension, also known as the `_mysql_connector` Python module.

The `_mysql_connector` C Extension module can be used directly without any other code of Connector/Python. One reason to use this module directly is for performance reasons.

> **Note**
>
> Examples in this reference use `ccnx` to represent a connector object as used with the `_mysql_connector` C Extension module. `ccnx` is an instance of the `_mysql_connector.MySQL()` class. It is distinct from the `cnx` object used in examples for the `mysql.connector` Connector/Python module described in Chapter 10, *Connector/Python API Reference*. `cnx` is an instance of the object returned by the `connect()` method of the `MySQLConnection` class.

> **Note**
>
> The C Extension is not part of the pure Python installation. It is an optional module that must be installed using a binary distribution of Connector/Python that includes it, or compiled using a source distribution. See Chapter 4, *Connector/Python Installation*.

11.1 Module _mysql_connector

The `_mysql_connector` module provides classes.

11.2 Class _mysql_connector.MySQL()

Syntax:

```
ccnx = _mysql_connector.MySQL(args)
```

The `MySQL` class is used to open and manage a connection to a MySQL server (referred to elsewhere in this reference as “the `MySQL` instance”). It is also used to send commands and SQL statements and read results.

The `MySQL` class wraps most functions found in the MySQL C Client API and adds some additional convenient functionality.

```
import _mysql_connector

ccnx = _mysql_connector.MySQL()
ccnx.connect(user='scott', password='tiger',
             host='127.0.0.1', database='employees')
ccnx.close()
```

Permitted arguments for the `MySQL` class are `auth_plugin`, `buffered`, `charset_name`, `connection_timeout`, `raw`, `use_unicode`. Those arguments correspond to the arguments of the same names for `MySQLConnection.connect()` as described at Section 7.1, “Connector/Python Connection Arguments”, except that `charset_name` corresponds to `charset`.

11.3 Method _mysql_connector.MySQL.affected_rows()

Syntax:

```
count = ccnx.afffected_rows()
```

Returns the number of rows changed, inserted, or deleted by the most recent `UPDATE`, `INSERT`, or `DELETE` statement.

11.4 Method _mysql_connector.MySQL.autocommit()

Syntax:

```
ccnx.autocommit(bool)
```

Sets the autocommit mode.

Raises a `ValueError` exception if `mode` is not `True` or `False`.

11.5 Method _mysql_connector.MySQL.buffered()

Syntax:

```
is_buffered = ccnx.buffered()     # getter
ccnx.buffered(bool)               # setter
```

With no argument, returns `True` or `False` to indicate whether the `MySQL` instance buffers (stores) the results.

With a boolean argument, sets the `MySQL` instance buffering mode.

For the setter syntax, raises a `TypeError` exception if the value is not `True` or `False`.

11.6 Method _mysql_connector.MySQL.change_user()

Syntax:

```
ccnx.change_user(user='user_name,
                 password='password_val',
                 database='db_name')
```

Changes the user and sets a new default database. Permitted arguments are `user`, `password`, and `database`.

11.7 Method _mysql_connector.MySQL.character_set_name()

Syntax:

```
charset = ccnx.character_set_name()
```

Returns the name of the default character set for the current MySQL session.

Some MySQL character sets have no equivalent names in Python. When this is the case, a name usable by Python is returned. For example, the `'utf8mb4'` MySQL character set name is returned as `'utf8'`.

11.8 Method _mysql_connector.MySQL.close()

Syntax:

```
```

```
ccnx.close()
```

Closes the MySQL connection.

11.9 Method _mysql_connector.MySQL.commit()

Syntax:

```
ccnx.commit()
```

Commits the current transaction.

11.10 Method _mysql_connector.MySQL.connect()

Syntax:

```
ccnx.connect(args)
```

Connects to a MySQL server.

```
import _mysql_connector

ccnx = _mysql_connector.MySQL()
ccnx.connect(user='scott', password='tiger',
             host='127.0.0.1', database='employees')
ccnx.close()
```

`connect()` supports the following arguments: `host`, `user`, `password`, `database`, `port`, `unix_socket`, `client_flags`, `ssl_ca`, `ssl_cert`, `ssl_key`, `ssl_verify_cert`, `compress`. See Section 7.1, “Connector/Python Connection Arguments”.

If `ccnx` is already connected, `connect()` discards any pending result set and closes the connection before reopening it.

Raises a `TypeError` exception if any argument is of an invalid type.

11.11 Method _mysql_connector.MySQL.connected()

Syntax:

```
is_connected = ccnx.connected()
```

Returns `True` or `False` to indicate whether the `MySQL` instance is connected.

11.12 Method _mysql_connector.MySQL.consume_result()

Syntax:

```
ccnx.consume_result()
```

Consumes the stored result set, if there is one, for this `MySQL` instance, by fetching all rows. If the statement that was executed returned multiple result sets, this method loops over and consumes all of them.

11.13 Method _mysql_connector.MySQL.convert_to_mysql()

Syntax:

```
converted_obj = ccnx.convert_to_mysql(obj))
```

Converts a Python object to a MySQL value based on the Python type of the object. The converted object is escaped and quoted.

```
ccnx.query('SELECT CURRENT_USER(), 1 + 3, NOW()')
row = ccnx.fetch_row()
for col in row:
  print(ccnx.convert_to_mysql(col))
ccnx.consume_result()
```

Raises a `MySQLInterfaceError` exception if the Python object cannot be converted.

11.14 Method _mysql_connector.MySQL.escape_string()

Syntax:

```
str = ccnx.escape_string(str_to_escape)
```

Uses the `mysql_escape_string()` C API function to create a SQL string that you can use in an SQL statement.

Raises a `TypeError` exception if the value does not have a `Unicode`, `bytes`, or (for Python 2) `string` type. Raises a `MySQLError` exception if the string could not be escaped.

11.15 Method _mysql_connector.MySQL.fetch_fields()

Syntax:

```
field_info = ccnx.fetch_fields()
```

Fetches column information for the active result set. Returns a list of tuples, one tuple per column

Raises a `MySQLInterfaceError` exception for any MySQL error returned by the MySQL server.

```
ccnx.query('SELECT CURRENT_USER(), 1 + 3, NOW()')
field_info = ccnx.fetch_fields()
for fi in field_info:
  print(fi)
ccnx.consume_result()
```

11.16 Method _mysql_connector.MySQL.fetch_row()

Syntax:

```
row = ccnx.fetch_row()
```

Fetches the next row from the active result set. The row is returned as a tuple that contains the values converted to Python objects, unless `raw` was set.

```
ccnx.query('SELECT CURRENT_USER(), 1 + 3, NOW()')
row = ccnx.fetch_row()
print(row)
ccnx.free_result()
```

Raises a `MySQLInterfaceError` exception for any MySQL error returned by the MySQL server.

11.17 Method _mysql_connector.MySQL.field_count()

Syntax:

```
count = ccnx.field_count()
```

Returns the number of columns in the active result set.

11.18 Method _mysql_connector.MySQL.free_result()

Syntax:

```
ccnx.free_result()
```

Frees the stored result set, if there is one, for this `MySQL` instance. If the statement that was executed returned multiple result sets, this method loops over and consumes all of them.

11.19 Method _mysql_connector.MySQL.get_character_set_info()

Syntax:

```
info = ccnx.get_character_set_info()
```

Returns information about the default character set for the current MySQL session. The returned dictionary has the keys `number`, `name`, `csname`, `comment`, `dir`, `mbminlen`, and `mbmaxlen`.

11.20 Method _mysql_connector.MySQL.get_client_info()

Syntax:

```
info = ccnx.get_client_info()
```

Returns the MySQL client library version as a string.

11.21 Method _mysql_connector.MySQL.get_client_version()

Syntax:

```
info = ccnx.get_client_version()
```

Returns the MySQL client library version as a tuple.

11.22 Method _mysql_connector.MySQL.get_host_info()

Syntax:

```
info = ccnx.get_host_info()
```

Returns a description of the type of connection in use as a string.

11.23 Method _mysql_connector.MySQL.get_proto_info()

Syntax:

```
info = ccnx.get_proto_info()
```

Returns the protocol version used by the current session.

11.24 Method _mysql_connector.MySQL.get_server_info()

Syntax:

```
info = ccnx.get_server_info()
```

Returns the MySQL server version as a string.

11.25 Method _mysql_connector.MySQL.get_server_version()

Syntax:

```
info = ccnx.get_server_version()
```

Returns the MySQL server version as a tuple.

11.26 Method _mysql_connector.MySQL.get_ssl_cipher()

Syntax:

```
info = ccnx.get_ssl_cipher()
```

Returns the SSL cipher used for the current session, or `None` if SSL is not in use.

11.27 Method _mysql_connector.MySQL.hex_string()

Syntax:

```
str = ccnx.hex_string(string_to_hexify)
```

Encodes a value in hexadecimal format and wraps it within `X''`. For example, `"ham"` becomes `X'68616D'`.

11.28 Method _mysql_connector.MySQL.insert_id()

Syntax:

```
```

```
insert_id = ccnx.insert_id()
```

Returns the `AUTO_INCREMENT` value generated by the most recent executed statement, or 0 if there is no such value.

11.29 Method _mysql_connector.MySQL.more_results()

Syntax:

```
more = ccnx.more_results()
```

Returns `True` or `False` to indicate whether any more result sets exist.

11.30 Method _mysql_connector.MySQL.next_result()

Syntax:

```
ccnx.next_result()
```

Initiates the next result set for a statement string that produced multiple result sets.

Raises a `MySQLInterfaceError` exception for any MySQL error returned by the MySQL server.

11.31 Method _mysql_connector.MySQL.num_fields()

Syntax:

```
count = ccnx.num_fields()
```

Returns the number of columns in the active result set.

11.32 Method _mysql_connector.MySQL.num_rows()

Syntax:

```
count = ccnx.num_rows()
```

Returns the number of rows in the active result set.

Raises a `MySQLError` exception if there is no result set.

11.33 Method _mysql_connector.MySQL.ping()

Syntax:

```
alive = ccnx.ping()
```

Returns `True` or `False` to indicate whether the connection to the MySQL server is working.

11.34 Method _mysql_connector.MySQL.query()

Syntax:

```
ccnx.query(args)
```

Executes an SQL statement. The permitted arguments are `statement`, `buffered`, `raw`, and `raw_as_string`.

```
ccnx.query('DROP TABLE IF EXISTS t')
ccnx.query('CREATE TABLE t (i INT NOT NULL AUTO_INCREMENT PRIMARY KEY)')
ccnx.query('INSERT INTO t (i) VALUES (NULL),(NULL),(NULL)')
ccnx.query('SELECT LAST_INSERT_ID()')
row = ccnx.fetch_row()
print('LAST_INSERT_ID(): ', row)
ccnx.consume_result()
```

`buffered` and `raw`, if not provided, take their values from the `MySQL` instance. `raw_as_string` is a special argument for Python v2 and returns `str` instead of `bytearray` (compatible with Connector/Python v1.x).

To check whether the query returns rows, check the `have_result_set` property of the `MySQL` instance.

`query()` returns `True` if the query executes, and raises an exception otherwise. It raises a `TypeError` exception if any argument has an invalid type, and a `MySQLInterfaceError` exception for any MySQL error returned by the MySQL server.

11.35 Method _mysql_connector.MySQL.raw()

Syntax:

```
is_raw = ccnx.raw()     # getter
ccnx.raw(bool)          # setter
```

With no argument, returns `True` or `False` to indicate whether the `MySQL` instance return the rows as is (without conversion to Python objects).

With a boolean argument, sets the `MySQL` instance raw mode.

11.36 Method _mysql_connector.MySQL.refresh()

Syntax:

```
ccnx.refresh(flags)
```

Flushes or resets the tables and caches indicated by the argument. The only argument currently permitted is an integer.

Raises a `TypeError` exception if the first argument is not an integer.

11.37 Method _mysql_connector.MySQL.rollback()

Syntax:

```
ccnx.rollback()
```

Rolls back the current transaction.

Raises a `MySQLInterfaceError` exception on errors.

11.38 Method _mysql_connector.MySQL.select_db()

Syntax:

```
ccnx.select_db(db_name)
```

Sets the default (current) database for the current session.

Raises a `MySQLInterfaceError` exception for any MySQL error returned by the MySQL server.

11.39 Method _mysql_connector.MySQL.set_character_set()

Syntax:

```
ccnx.set_character_set(charset_name)
```

Sets the default character set for the current session. The only argument permitted is a string that contains the character set name.

Raises a `TypeError` exception if the argument is not a `PyString_type`.

11.40 Method _mysql_connector.MySQL.shutdown()

Syntax:

```
ccnx.shutdown(flags)
```

Shuts down the MySQL server. The only argument currently permitted is an integer that describes the shutdown type.

Raises a `TypeError` exception if the first argument is not an integer. Raises a `MySQLErrorInterface` exception if an error is retured by the MySQL server.

11.41 Method _mysql_connector.MySQL.stat()

Syntax:

```
info = ccnx.stat()
```

Returns the server status as a string.

Raises a `MySQLErrorInterface` exception if an error is retured by the MySQL server.

11.42 Method _mysql_connector.MySQL.thread_id()

Syntax:

```
thread_id = ccnx.thread_id()
```

Returns the current thread or connection ID.

11.43 Method _mysql_connector.MySQL.use_unicode()

Syntax:

```
is_unicode = ccnx.use_unicode()      # getter
ccnx.use_unicode(bool)               # setter
```

With no argument, returns `True` or `False` to indicate whether the `MySQL` instance returns nonbinary strings as Unicode.

With a boolean argument, sets whether the `MySQL` instance returns nonbinary strings as Unicode.

11.44 Method _mysql_connector.MySQL.warning_count()

Syntax:

```
count = ccnx.warning_count()
```

Returns the number of errors, warnings, and notes produced by the previous SQL statement.

11.45 Property _mysql_connector.MySQL.have_result_set

Syntax:

```
has_rows = ccnx.have_result_set
```

After execution of the `query()` method, this property indicates whether the query returns rows.

Appendix A Licenses for Third-Party Components

Table of Contents

MySQL Connector/Python 1.1

A.1 Django 1.5.1 License

The following software may be included in this product: