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.

The latest MySQL Connector/Python version is recommended for use with MySQL Server version 8.0 and higher.

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

For legal information, see the Legal Notices.

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

Licensing information. This product may include third-party software, used under license. If you are using a Commercial release of MySQL Connector/Python, see the MySQL Connector/Python 9.5 Commercial License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Commercial release. If you are using a Community release of MySQL Connector/Python, see the MySQL Connector/Python 9.5 Community License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Document generated on: 2025-10-13 (revision: 83739)

Table of Contents

Preface and Legal Notices

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

Legal Notices

License Restrictions

Warranty Disclaimer

Restricted Rights Notice

Hazardous Applications Notice

Trademark Notice

International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

Third-Party Content, Products, and Services Disclaimer

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Use of This Documentation

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

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

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support for Accessibility

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

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).

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 version 8.0 and higher.
- Connector/Python supports X DevAPI. For X DevAPI specific documentation, see X DevAPI User Guide.

  > **Note**
  >
  > X DevAPI support was separated into its own package (`mysqlx-connector-python`) in Connector/Python 8.3.0. For related information, see Chapter 4, *Connector/Python Installation*.

- 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.

For information about which versions of Python can be used with different versions of MySQL Connector/Python, see Chapter 3, *Connector/Python 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 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 an SQL statement. Never concatenate a user input value into an 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 table and index data.
  - **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 Statement 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

This section describes both version releases, such as 8.0.34, along with notes specific to the two implementations (C Extension and Pure Python).

Connector/Python Releases

The following table summarizes the available Connector/Python versions. For series that have reached General Availability (GA) status, development releases in the series prior to the GA version are no longer supported.

> **Note**
>
> MySQL Connectors and other MySQL client tools and applications now synchronize the first digit of their version number with the (highest) MySQL server version they support. For example, MySQL Connector/Python 8.0.12 would be designed to support all features of MySQL server version 8 (or lower). This change makes it easy and intuitive to decide which client version to use for which server version.
>
> Connector/Python 8.0.4 is the first release to use the new numbering. It is the successor to Connector/Python 2.2.3.

Table 3.1 Connector/Python Version Reference

Connector/Python Version	MySQL Server Versions	Python Versions	Connector Status
9.5.0 and later	8.0 and later	3.14, 3.13*, 3.12, 3.11, 3.10	General Availability
9.10 - 9.4.0	8.0 and later	3.13*, 3.12, 3.11, 3.10, 3.9	General Availability
8.4.0 and 9.0.0	8.0 and later	3.12, 3.11, 3.10, 3.9, 3.8	General Availability
8.1.0 - 8.3.0	5.7 and later	3.12 (8.2.0+), 3.11, 3.10, 3.9, 3.8	General Availability
8.0	8.0, 5.7, 5.6, 5.5	3.11, 3.10, 3.9, 3.8, 3.7, (3.6 before 8.0.29), (2.7 and 3.5 before 8.0.24)	General Availability
2.2 (continues as 8.0)	5.7, 5.6, 5.5	3.5, 3.4, 2.7	Developer Milestone, No releases
2.1	5.7, 5.6, 5.5	3.5, 3.4, 2.7, 2.6	General Availability
2.0	5.7, 5.6, 5.5	3.5, 3.4, 2.7, 2.6	GA, final release on 2016-10-26
1.2	5.7, 5.6, 5.5 (5.1, 5.0, 4.1)	3.4, 3.3, 3.2, 3.1, 2.7, 2.6	GA, final release on 2014-08-22

> **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**
>
> Python 3.13 enables `ssl.VERIFY_X509_STRICT` SSL validation by default, which means SSL certificates must now be RFC-5280 compliant when using Python 3.13 and higher.

> **Note**
>
> On macOS x86_64 ARM: Python 3.7 is not supported with the c-ext implementation; note this is a non-default version of Python on macOS.

Connector/Python Implementations

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

- As pure Python; an implementation written in Python. It depends on the Python Standard Library.

  The X DevAPI variant of the connector requires Python Protobuf. The required version is 5.29.4.

- 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 MySQL Server packages (see MySQL C API Implementations).

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.

> **Note**
>
> Support for distutils was removed in Connector/Python 8.0.32.

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.

Chapter 4 Connector/Python Installation

Table of Contents

Connector/Python runs on any platform where Python is installed. Make sure Python is installed on your platform:

- Python comes preinstalled on most Unix and Unix-like systems, such as Linux, macOS, and FreeBSD. If your system does not have Python preinstalled for some reasons, use its software management system to install it.
- For Microsoft Windows, a Python installer is available at the Python Download website or via the Microsoft Store.

Also make sure Python in your system path.

Connector/Python includes the classic and X DevAPI APIs, which are installed separately. Each can be installed by a binary or source distribution.

Binaries of Connector/Python are distributed in the RPM and the wheel package formats. The source code, on the other hand, is distributed as a compressed archive of source files, from which a wheel package can be built.

4.1 Quick Installation Guide

The recommended way to install Connector/Python is by pip and wheel packages. If your system does not have `pip`, you can install it with your system's software manager, or with a standalone pip installer.

> **Note**
>
> You are strongly recommended to use the latest version of `pip` to install Connector/Python. Upgrade your `pip` version if needed.

Install the Connector/Python interfaces for the classic MySQL protocol and the X Protocol, respectively, with the following commands.

```
# classic API
$ pip install mysql-connector-python

# X DevAPI
$ pip install mysqlx-connector-python
```

Refer to the installation tutorial for alternate means to install X DevAPI.

4.2 Differences Between Binary And Source Distributions

Installing from a wheel (`bdist` package) is the recommended, except for Enterprise Linux systems, on which the RPM-based installation method may be preferred.

Wheels can be directly and easily installed without an extra build step. However, a wheel package is often specific to a particular platform and Python version, so there may be cases in which `pip` cannot find a suitable wheel package based on your platform or your Python version. When that happens, you can get the source distribution (`sdist`) and produce a wheel package from it for installing Connector/Python.

> **Note**
>
> Creating a wheel package from an `sdist` may fail for some older Python version, as the Connector/Python source code is only compatible with a specific subset of Python versions.

In summary, the recommendation is to use a `bdist` unless `pip` cannot find a suitable wheel package for your setup, or if you need to custom build a wheel package for some special reasons.

4.3 Obtaining Connector/Python

Using `pip` is the preferred method to obtain, install, and upgrade Connector/Python. For alternatives, see the Connector/Python download site.

> **Note**
>
> The `mysql-connector-python` package installs an interface to the classic MySQL protocol. The X DevAPI is available by its own `mysqlx-connector-python` package. Prior to Connector/Python 8.3.0, `mysql-connector-python` installed interfaces to both the X and classic protocols.

Most Linux installation packages (except RPMs for Enterprise Linux) are no longer available from Oracle since Connector/Python 9.0.0. Using `pip` to manage Connector/Python on those Linux distributions is recommended.

4.4 Installing Connector/Python from a Binary Distribution

Connector/Python includes the classic and X DevAPI connector APIs, which are installed separately. Each can be installed by a binary distribution.

Binaries are distributed in the RPM and the wheel package formats.

4.4.1 Installing Connector/Python with pip

Installation via `pip` is supported on Windows, macOS, and Linux platforms.

> **Note**
>
> For macOS platforms, DMG installer packages were available for Connector/Python 8.0 and earlier.

Use `pip` to install and upgrade Connector/Python:

```
# Installation
$> pip install mysql-connector-python

# Upgrade
$> pip install mysql-connector-python --upgrade

# Optionally, install X DevAPI
$> pip install mysqlx-connector-python

# Upgrade X DevAPI
$> pip install mysqlx-connector-python --upgrade
```

In case the wheel package you want to install is found in your local file system (for example, you produced a wheel package from a source distribution or downloaded it from somewhere), you can install it as follows:

```
# Installation
$ pip install /path/to/wheel/<wheel package name>.whl
```

Installation of Optional Features

Installation from wheels allow you to install optional dependencies to enable certain features with Connector/Python. For example:

```
# 3rd party packages to enable the telemetry functionality are installed
$ pip install mysql-connector-python[telemetry]
```

Similarly, for X DevAPI:

```
# 3rd party packages to enable the compression functionality are installed
$ pip install mysqlx-connector-python[compression]
```

These installation options are shortcuts to install all the dependencies needed by some particular features (they are only for your convenience, and you can always install the required dependencies for a feature by yourself):

- For the classic protocol:
  - dns-srv
  - gssapi
  - fido2
  - telemetry
- For X Protocol:
  - dns-srv
  - compression

You can specify a multiple of these options in your installation command, for example:

```
$ pip install mysql-connector-python[telemetry,dns-srv,...]
```

Or, if are installing a wheel package from your local file system:

```
$ pip install /path/to/wheel/<wheel package name>.whl[telemetry,dns-srv,...]
```

4.4.2 Installing by RPMs

Installation by RPMs is only supported on RedHat Enterprise Linux and Oracle Linux, and is performed using the MySQL Yum Repository or by using RPM packages downloaded directly from Oracle.

4.4.2.1 Using the MySQL Yum Repository

RedHat Enterprise Linux and Oracle Linux platforms can install Connector/Python using the MySQL Yum repository (see Adding the MySQL Yum Repository and Installing Additional MySQL Products and Components with Yum).

Prerequisites

- *For installing X DevAPI only:*Because the required `python3-protobuf` RPM package is not available for Python 3.8 on the RedHat Enterprise Linux and Oracle Linux platforms, it has to be

manually installed with, for example, `pip install protobuf`. This is required for Connector/Python 8.0.29 or later.

- The `mysql-community-client-plugins` package is required for using robust authentication methods like `caching_sha2_password`, which is the default authentication method for MySQL 8.0 and later. Install it using the Yum repository

```
$ sudo yum install mysql-community-client-plugins
```

Installation

Use the following commands to install Connector/Python:

```
$ sudo yum install mysql-connector-python

# Optionally, install also X DevAPI
$ sudo yum install mysqlx-connector-python
```

4.4.2.2 Using an RPM Package

Connector/Python RPM packages (`.rpm` files) are available from the Connector/Python download site.

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

Prerequisites

- *For installing X DevAPI only:*Because the required `python3-protobuf` RPM package is not available for Python 3.8 on the RedHat Enterprise Linux and Oracle Linux platforms, it has to be manually installed with, for example, `pip install protobuf`. This is required for Connector/Python 8.0.29 or later.
- The `mysql-community-client-plugins` package is required for using robust authentication methods like `caching_sha2_password`, which is the default authentication method for MySQL 8.0 and later.

```
$ rpm -i mysql-community-client-plugins-ver.distro.architecture.rpm
```

Installation

To install Connector/Python using the downloaded RPM packages:

```
$ rpm -i mysql-connector-python-ver.distro.architecture.rpm

# Optionally, install X DevAPI
$ rpm -i mysqlx-connector-python-ver.distro.architecture.rpm
```

4.5 Installing Connector/Python from a Source Distribution

The Connector/Python source distribution is platform independent, and is packaged in the compressed `tar` archive format (`.tar.gz` file). See Obtaining Connector/Python) on how to download them.

Prerequisites for Compiling Connector/Python with the C Extension

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, the following prerequisites must be satisfied:

- Compiling tools:
  - *For Linux platforms:* A C/C++ compiler, such as `gcc`.
  - *For Windows platforms:* Current version of Visual Studio.

- Python development files.
- *For installing the classic interface only:* MySQL Server binaries (server may be installed or not installed on the system), including development files (to obtain the MySQL Server binaries, visit the MySQL download site).
- *For installing the X DevAPI interface only:* Protobuf C++ (version 5.29.4).

Installing Connector/Python from Source Code Using pip

> **Note**
>
> We recommend leveraging python virtual environments to encapsulate the package installation instead of installing packages directly into the Python system environment.

For installing the classic interface:

1. Download the latest version of the sdist of Connector/Python for the classic MySQL protocol, whose name is in the format of mysql_connector_python-x.y.z.tar.gz.
2. *Optional: To include the C Extension*, use these steps to provide the path to the installation directory of MySQL Server (or to the folder where the server binaries are located) with the MYSQL_CAPI system variable before running the installation step. On Linux platforms:

   ```
   $ export MYSQL_CAPI=<path to server binaries>
   ```

   On Windows platforms:

   ```
   > $env:MYSQL_CAPI=<path to server binaries>
   ```

   > **Note**
   >
   > It is not required that the server is actually installed on the system; for compiling the C-extension, the presence of libraries are sufficient

3. Perform the installation using this command:

   ```
   pip install ./mysql_connector_python-x.y.z.tar.gz
   ```

   > **Warning**
   >
   > DO NOT use mysql-connector-python instead of ./mysql_connector_python-x.y.z.tar.gz, as the former will install the WHEEL package from the PyPI repository, and the latter will install the local WHEEL that is compiled from the source code.

For installing X DevAPI:

1. Download the latest version of the sdist of Connector/Python for the MySQL X Protocol, whose name is in the format of mysqlx_connector_python-x.y.z.tar.gz.
2. *Optional: To include the Protobuf C-Extension,* use these commands on Linux platforms to provide the paths to the Protobuf folders by the MYSQLXPB_* system variables before the installation step:

   ```
   $ export MYSQLXPB_PROTOBUF=<path to protobuf binaries>
   $ export MYSQLXPB_PROTOBUF_INCLUDE_DIR="${MYSQLXPB_PROTOBUF}/include"
   $ export MYSQLXPB_PROTOBUF_LIB_DIR="${MYSQLXPB_PROTOBUF}/lib"
   $ export MYSQLXPB_PROTOC="${MYSQLXPB_PROTOBUF}/bin/protoc"
   ```

   Or these commands on Windows platforms:

   ```
   > $env:PROTOBUF=<path to protobuf binaries>
   > $env:PROTOBUF_INCLUDE_DIR=$env:PROTOBUF+"\include"
   ```

```
> $env:PROTOBUF_LIB_DIR=$env:PROTOBUF+"\lib"
> $env:PROTOC=$env:PROTOBUF+"\bin\protoc.exe"
```

3. Perform the installation using this command:

```
pip install ./mysqlx_connector_python-x.y.z.tar.gz
```

> **Warning**
>
> DO NOT use `mysqlx-connector-python` instead of `./mysqlx_connector_python-x.y.z.tar.gz`, as the former will install the WHEEL package from the PyPI repository, and the latter will install the local WHEEL that is compiled from the source code.

4.6 Verifying Your Connector/Python Installation

Verifying Installations by `pip`

To verify that a Connector/Python package has been installed successfully using `pip`, use the following command:

```
$ pip install list
```

If you have installed the classic interface, you should see an output similar to the following:

```
Package                               Version
------------------------------------- ----------
...                                   ...
mysql-connector-python                x.y.z
...                                   ...
```

If you have installed X DevAPI, you should see an output similar to the following:

```
Package                               Version
------------------------------------- ----------
...                                   ...
mysqlx-connector-python                x.y.z
...
```

Installed from an RPM

The default Connector/Python installation location is */prefix/*`python`*X.Y*`/site-packages/`, where *`prefix`* is the location where Python is installed and *`X.Y`* is the Python version.

The C Extension is installed as `_mysql_connector.so` and `_mysqlxpb.so` in the `site-packages` directory, not in the `mysql/connector` and `mysqlx` directories for the classic interface and X DevAPI, respectively.

Verify the C-extension

To verify the C-extension of the classic package is available, run this command:

```
$ python -c "import mysql.connector; assert mysql.connector.HAVE_CEXT; print(f'C-ext is {mysql.connector.HA
```

If no error is returned, the C-extension has been correctly built and installed.

Similarly, to verify the C-extension of the X DevAPI package is available, run this command and see if it returns any errors:

```
$ python -c "import mysqlx; assert mysqlx.protobuf.HAVE_MYSQLXPB_CEXT; print(f'C-ext is {mysqlx.protobuf.HA
```

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='password',
                              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='password',
                                 host='127.0.0.1',
                                 database='employees')
cnx.close()
```

Both forms (either using the `connect()` constructor or the class directly) are valid and functionally equal, but using `connect()` is preferred and used by 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='employ')
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()
```

Defining connection arguments in a dictionary and using the `**` operator is another option:

```
import mysql.connector

config = {
  'user': 'scott',
```

```
  'password': 'password',
  'host': '127.0.0.1',
  'database': 'employees',
  'raise_on_warnings': True
}

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

cnx.close()
```

Defining Logger options, a reconnection routine, and defined as a connection method named connect_to_mysql:

```
import logging
import time
import mysql.connector

# Set up logger
logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)
formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %(message)s")

# Log to console
handler = logging.StreamHandler()
handler.setFormatter(formatter)
logger.addHandler(handler)

# Also log to a file
file_handler = logging.FileHandler("cpy-errors.log")
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)

def connect_to_mysql(config, attempts=3, delay=2):
    attempt = 1
    # Implement a reconnection routine
    while attempt < attempts + 1:
        try:
            return mysql.connector.connect(**config)
        except (mysql.connector.Error, IOError) as err:
            if (attempts is attempt):
                # Attempts to reconnect failed; returning None
                logger.info("Failed to connect, exiting without a connection: %s", err)
                return None
            logger.info(
                "Connection failed: %s. Retrying (%d/%d)...",
                err,
                attempt,
                attempts-1,
            )
            # progressive reconnect delay
            time.sleep(delay ** attempt)
            attempt += 1
    return None
```

Connecting and using the Sakila database using the above routine, assuming it's defined in a file named `myconnection.py`:

```
from myconnection import connect_to_mysql

config = {
    "host": "127.0.0.1",
    "user": "user",
    "password": "pass",
    "database": "sakila",
}

cnx = connect_to_mysql(config, attempts=3)

if cnx and cnx.is_connected():
```

```
    with cnx.cursor() as cursor:

        result = cursor.execute("SELECT * FROM actor LIMIT 5")

        rows = cursor.fetchall()

        for rows in rows:

            print(rows)

    cnx.close()

else:

    print("Could not connect")
```

Using the Connector/Python Python or C Extension

Connector/Python offers two implementations: a pure Python interface and a C extension that uses the MySQL C client library (see Chapter 8, *The Connector/Python C Extension*). This can be configured at runtime using the `use_pure` connection argument. It defaults to `False` as of MySQL 8, meaning the C extension is used. If the C extension is not available on the system then `use_pure` defaults to `True`. Setting `use_pure=False` causes the connection to use the C Extension if your Connector/Python installation includes it, while `use_pure=True` to `False` means the Python implementation is used if available.

> **Note**
>
> The `use_pure` option and C extension were added in Connector/Python 2.1.1.

The following example shows how to set `use_pure` to False.

```
import mysql.connector

cnx = mysql.connector.connect(user='scott', password='password',
                              host='127.0.0.1',
                              database='employees',
                              use_pure=False)
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` ("
    "  `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_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:
    cursor.execute("USE {}".format(DB_NAME))
except mysql.connector.Error as err:
    print("Database {} does not exists.".format(DB_NAME))
    if err.errno == errorcode.ER_BAD_DB_ERROR:
        create_database(cursor)
        print("Database {} created successfully.".format(DB_NAME))
        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 table_name in TABLES:
    table_description = TABLES[table_name]
    try:
        print("Creating table {}: ".format(table_name), end='')
        cursor.execute(table_description)
    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:

```
Database employees does not exists.
Database employees created successfully.
Creating table employees: OK
Creating table departments: already exists.
Creating table salaries: already exists.
Creating table dept_emp: OK
Creating table dept_manager: OK
Creating table titles: OK
```

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:

```
$> tar xzf employees_db-dump-files-1.0.5.tar.bz2
$> cd employees_db
$> mysql employees < load_employees.dump
$> mysql employees < load_titles.dump
$> mysql employees < load_departments.dump
$> mysql employees < load_salaries.dump
$> mysql employees < load_dept_emp.dump
$> 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 higher), 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 higher 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, “cursor.MySQLCursorBuffered Class”.) 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.
`password1`, `password2`, and `password3`		For Multi-Factor Authentication (MFA); `password1` is an alias for `password`. Added in 8.0.28.
`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.
`unix_socket`		The location of the Unix socket file.
`port`	3306	The TCP/IP port of the MySQL server. Must be an integer.
`conn_attrs`		Standard `performance_schema.session_connect_attrs` values are sent; use `conn_attrs` to optionally set additional custom connection attributes as defined by a dictionary such as *config['conn_attrs'] = {"foo": "bar"}*. The c-ext and pure python implementations differ. The c-ext implementation depends on the mysqlclient library so its standard conn_attrs values originate from it. For example, '_client_name' is 'libmysql' with c-ext but 'mysql-connector-python' with pure python. C-ext adds these additional attributes: '_connector_version', '_connector_license', '_connector_name', and '_source_host'. This option was added in 8.0.17, as was the default session_connect_attrs behavior.

Argument Name	Default	Description
`init_command`		Command (SQL query) executed immediately after the connection is established as part of the initialization process. Added in 8.0.32.
`auth_plugin`		Authentication plugin to use. Added in 1.2.1.
`fido_callback`		Deprecated as of 8.2.0 and removed in 8.4.0; instead use `webauthn_callback`. A callable defined by the optional `fido_callback` option is executed when it's ready for user interaction with the hardware FIDO device. This option can be a callable object or a string path that the connector can import in runtime and execute. It does not block and is only used to notify the user of the need for interaction with the hardware FIDO device. This functionality was only available in the C extension. A *NotSupportedError* was raised when using the pure Python implementation.
`webauthn_callback`		A callable defined by the optional `webauthn_callback` option is executed when it's ready for user interaction with the hardware WebAuthn device. This option can be a callable object or a string path that the connector can import in runtime and execute. It does not block and is only used to notify the user of the need for interaction with the hardware FIDO device. Enable the `authentication_webauthn_client` auth_plugin in the connection configuration to use. This option was added in 8.2.0, and it deprecated the `fido_callback` option that was removed in version 8.4.0.
`openid_token_file`		Path to the file containing the OpenID JWT formatted identity token. Added in 9.1.0.
`use_unicode`	`True`	Whether to use Unicode.
`charset`	`utf8mb4`	Which MySQL character set to use.
`collation`	`utf8mb4_general_ci` (is `utf8_general_ci` in 2.x	Which MySQL collation to use. The 8.x default values are generated from the latest MySQL Server 8.0 defaults.
`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.
`read_timeout`	`None`	Time limit to receive a response from the server before raising a `ReadTimeoutError` level error. The default

Argument Name	Default	Description
		value (None) sets the wait time to indefinitely. Option added in 9.2.0.
write_timeout	None	Time limit to send data to the server before raising a WriteTimeoutError level error. The default value (None) sets the wait time to indefinitely. Option added in 9.2.0.
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.
consume_results	False	Whether to automatically read result sets.
tls_versions	["TLSv1.2", "TLSv1.3"]	TLS versions to support; allowed versions are TLSv1.2 and TLSv1.3. Versions TLSv1 and TLSv1.1 were removed in Connector/Python 8.0.28.
ssl_ca		File containing the SSL certificate authority.
ssl_cert		File containing the SSL certificate file.
ssl_disabled	False	True disables SSL/TLS usage. The TLSv1 and TLSv1.1 connection protocols are deprecated as of Connector/Python 8.0.26 and removed as of Connector/Python 8.0.28.
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.
ssl_verify_identity	False	When set to True, additionally perform host name identity verification by checking the host name that the client uses for connecting to the server against the identity in the certificate that the server sends to the client. Option added in Connector/Python 8.0.14.
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.
kerberos_auth_mode	SSPI	Windows-only, for choosing between SSPI and GSSAPI at runtime for the authentication_kerberos_client authentication plugin on Windows. Option added in Connector/Python 8.0.32.
oci_config_file	""	Optionally define a specific path to the authentication_oci server-side authentication configuration file. The profile name can be configured with oci_config_profile. The default file path on Linux and macOS is ~/.oci/config, and %HOMEDRIVE%%HOMEPATH%\.oci\config on Windows.
oci_config_profile	"DEFAULT"	Used to specify a profile to use from the OCI configuration file that contains the generated ephemeral key pair and security token. The OCI configuration file location can be defined by oci_config_file. Option

Argument Name	Default	Description
		`oci_config_profile` was added in Connector/Python 8.0.33.
`dsn`		Not supported (raises `NotSupportedError` when used).
`pool_name`		Connection pool name. 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).
`pool_size`	5	Connection pool size. The pool size must be greater than 0 and less than or equal to `pooling.CNX_POOL_MAXSIZE` (default 32).
`pool_reset_session`	`True`	Whether to reset session variables when connection is returned to pool.
`compress`	`False`	Whether to use compressed client/server protocol.
`converter_class`		Converter class to use.
`converter_str_fallback`	`False`	Enable the conversion to str of value types not supported by the Connector/Python converter class or by a custom converter class.
`failover`		Server failover sequence.
`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`	`False` as of 8.0.11, and `True` in earlier versions. If only one implementation (C or Python) is available, then then the default value is set to enable the available implementation.	Whether to use pure Python or C Extension. If `use_pure=False` and the C Extension is not available, then Connector/Python will automatically fall back to the pure Python implementation. Can be set with *mysql.connector.connect()* but not *MySQLConnection.connect()*. Added in 2.1.1.
`krb_service_principal`	The "@realm" defaults to the default realm, as configured in the `krb5.conf` file.	Must be a string in the form "primary/instance@realm" such as "ldap/ldapauth@MYSQL.COM" where "@realm" is optional. Added in 8.0.23.

MySQL Authentication Options

Authentication with MySQL typically uses a `username` and `password`.

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 supports authentication plugins available as of MySQL 8.0, including the preferred caching_sha2_password authentication plugin.

The deprecated mysql_native_password plugin is supported, but it is disabled by default as of MySQL Server 8.4.0 and removed as of MySQL Server 9.0.0.

The `connect()` method supports an `auth_plugin` argument that can be used to force use of a particular authentication plugin.

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

Connector/Python supports the Kerberos authentication protocol for passwordless authentication. Linux clients are supported as of Connector/Python 8.0.26, and Windows support was added in Connector/Python 8.0.27 with the C extension implementation, and in Connector/Python 8.0.29 with the pure Python implementation. For Windows, the related `kerberos_auth_mode` connection option was added in 8.0.32 to configure the mode as either SSPI (default) or GSSAPI (via the pure Python implementation, or the C extension implementation as of 8.4.0). While Windows supports both modes, Linux only supports GSSAPI.

Optionally use the `[gssapi]` shortcut when installing the `mysql-connector-python` pip package to pull in specific GSSAPI versions as defined by the connector, which is v1.8.3 as of Connector/Python 9.1.0:

```
$ pip install mysql-connector-python[gssapi]
```

The following example assumes LDAP Pluggable Authentication is set up to utilize GSSAPI/Kerberos SASL authentication:

```
import mysql.connector as cpy
import logging

logging.basicConfig(level=logging.DEBUG)

SERVICE_NAME = "ldap"
LDAP_SERVER_IP = "server_ip or hostname"  # e.g., winexample01

config = {
    "host": "127.0.0.1",
    "port": 3306,
    "user": "myuser@example.com",
    "password": "s3cret",
    "use_pure": True,
    "krb_service_principal": f"{SERVICE_NAME}/{LDAP_SERVER_IP}"
}

with cpy.connect(**config) as cnx:
    with cnx.cursor() as cur:
        cur.execute("SELECT @@version")
        res = cur.fetchone()
        print(res[0])
```

Connector/Python supports Multi-Factor Authentication (MFA) as of v8.0.28 by utilizing the `password1` (alias of `password`), `password2`, and `password3` connection options.

Connector/Python supports WebAuthn Pluggable Authentication as of Connector/Python 8.2.0, which is supported in MySQL Enterprise Edition. Optionally use the Connector/Python webauthn_callback connection option to notify users that they need to touch the hardware device. This functionality is present in the C implementation (which uses libmysqlclient) but the pure Python implementation requires the FIDO2 dependency that is not provided with the MySQL connector and is assumed to already be present in your environment. It can be independently installed using:

```
$> pip install fido2
```

Previously, the now removed (as of version 8.4.0) `authentication_fido` MySQL Server plugin was supported using the fido_callback option that was available in the C extension implementation.

Connector/Python supports *OpenID Connect* as of Connector/Python 9.1.0. Functionality is enabled with the `authentication_openid_connect_client` client-side authentication plugin connecting to MySQL Enterprise Edition with the `authentication_openid_connect` authentication plugin. These examples enable the plugin with `auth_plugin` and defines the JWT Identity Token file location with `openid_token_file`:

```
# Standard connection
import mysql.connector as cpy
config = {
    "host": "localhost",
    "port": 3306,
    "user": "root",
    "openid_token_file": "{path-to-id-token-file}",
    "auth_plugin": "authentication_openid_connect_client",
    "use_pure": True, # Use False for C-Extension
}
with cpy.connect(**config) as cnx:
    with cnx.cursor() as cur:
        cur.execute("SELECT @@version")
        print(cur.fetchall())

# Or, using an async connection
import mysql.connector.aio as cpy_async
import asyncio
config = {
    "host": "localhost",
    "port": 3306,
    "user": "root",
    "auth_plugin": "authentication_openid_connect_client",
    "openid_token_file": "{path-to-id-token-file}",
}
async def test():
    async with await cpy_async.connect(**config) as cnx:
        async with await cnx.cursor() as cur:
            await cur.execute("SELECT @@version")
            print(await cur.fetchall())
asyncio.run(test())
```

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 4.1 and higher. 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)
```

Result Set Handling

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, “MySQLConnection.cursor() Method”).

Results generated by queries normally are not read until the client program fetches them. To automatically consume and discard result sets, set the `consume_results` option to `True`. The result is that all results are read, which for large result sets can be slow. (In this case, it might be preferable to close and reopen the connection.)

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` option 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` options, 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` option set to `True`.

As of Connector/Python 2.2.2, if the MySQL server supports SSL connections, Connector/Python attempts to establish a secure (encrypted) connection by default, falling back to an unencrypted connection otherwise.

From Connector/Python 1.2.1 through Connector/Python 2.2.1, it is possible to establish an SSL connection using only the `ssl_ca` opion. 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': 'password',
    '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.

For additional information about connection pooling, see Section 9.5, “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`.

Server Failover

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`. This failover option was added in Connector/Python 1.2.1.

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`).

Compatibility 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` *mysql.connector.connect()* connection argument determines which. The default changed in Connector/Python 8 from `True` (use the pure Python implementation) to `False`. Setting `use_pure` changes the implementation used.

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

7.2 Connector/Python Option-File Support

Connector/Python can read options from option files. (For general information about option files in MySQL, 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:

```
$> 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` connection 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. 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. 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.

Option Parsing

Connector/Python reads the option values in option files as strings, and attempts to parse them using Python's `ast.literal_eval` function. This allows specifying values like numbers, tuples, lists, and booleans in the option files. If a value can't be parsed by `ast.literal_eval` then it's passed as a literal string.

For example, this option file has options with values using a number, a string, and a tuple of dictionaries that are correctly parsed for the `[connector_python]` group:

```
[connector_python]
database=cpyapp
port=3656
failover=({'host': '203.0.113.1', 'port': 3640}, {'host': '203.0.113.101', 'port': 3650})
```

For additional information, review Python's ast.literal_eval documentation including how to handle unsanitized data that could crash the Python interpreter. Confirm that the option file values are trustworthy and valid before parsing.

Chapter 8 The Connector/Python C Extension

Table of Contents

Connector/Python supports 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*.

> **Note**
>
> The C extension was added in version 2.1.1 and is enabled by default as of 8.0.11. The `use_pure` option determines whether the Python or C version of this connector is enabled and used.

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 `mysql.connector.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` (use the pure Python implementation) is `False` as of MySQL 8 and defaults to `True` in earlier versions. If the C extension is not available on the system then `use_pure` is `True`.
- On Linux, the C and Python implementations are available as different packages. You can install one or both implementations on the same system. On Windows and macOS, the packages include both implementations.

  For Connector/Python installations that include both implementations, it can optionally be toggled it by passing `use_pure=False` (to use C implementation) or `use_pure=True` (to use the Python implementation) as an argument to `mysql.connector.connect()`.
- For Connector/Python installations that do not include the C Extension, passing `use_pure=False` to `mysql.connector.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 `mysql.connector.connect()` raises an exception regardless of its value.

> **Note**
>
> On macOS, 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 `mysql.connector.connect()` are specified in a dictionary:

```
config = {
  'user': 'scott',
  'password': 'password',
  '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_info__ > (2, 1) and mysql.connector.HAVE_CEXT:
  config['use_pure'] = False
```

If `use_pure=False` and the C Extension is not available, then Connector/Python will automatically fall back to the pure Python implementation.

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='password',
             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

9.1 Connector/Python Logging

By default, logging functionality follows the default Python logging behavior. If logging functionality is not configured, only events with a severity level of WARNING and greater are printed to sys.stderr. For related information, see Python's Configuring Logging for a Library documentation.

Outputting additional levels requires configuration. For example, to output debug events to sys.stderr set logging.DEBUG and add the logging.StreamHandler handler. Additional handles can also be added, such as logging.FileHandler. This example sets both:

```
# Classic Protocol Example
import logging
import mysql.connector

logger = logging.getLogger("mysql.connector")
logger.setLevel(logging.DEBUG)

formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s- %(message)s")

stream_handler = logging.StreamHandler()
stream_handler.setFormatter(formatter)
logger.addHandler(stream_handler)

file_handler = logging.FileHandler("cpy.log")
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)

# XDevAPI Protocol Example
import logging
import mysqlx

logger = logging.getLogger("mysqlx")
logger.setLevel(logging.DEBUG)

formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s- %(message)s")

stream_handler = logging.StreamHandler()
stream_handler.setFormatter(formatter)
logger.addHandler(stream_handler)

file_handler = logging.FileHandler("cpy.log")
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)
```

9.2 Telemetry Support

MySQL Server added OpenTelemetry support in MySQL Enterprise Edition version 8.1.0, which is a commercial product. OpenTelemetry tracing support was added in Connector/Python 8.1.0.

Introduction to OpenTelemetry

OpenTelemetry is an observability framework and toolkit designed to create and manage telemetry data such as traces, metrics, and logs. Visit What is OpenTelemetry? for an explanation of what OpenTelemetry offers.

Connector/Python only supports tracing, so this guide does not include information about metric and log signals.

Installing Telemetry Support

Install the OpenTelemetry API, SDK, and OTLP Exporter packages on the system along with Connector/Python. Optionally use the `[telemetry]` shortcut when installing the `mysql-connector-python` pip package to pull in specific OpenTelemetry versions as defined by the connector.

Manual installation:

```
pip install opentelemetry-api
pip install opentelemetry-sdk
pip install opentelemetry-exporter-otlp-proto-http
pip install mysql-connector-python
```

Or pass in [telemetry] when installing Connector/Python to perform the same actions except it installs a specific and tested OpenTelemetry version, which for Connector/Python 9.4.0 and later is OpenTelemetry v1.33.1:

```
pip install mysql-connector-python[telemetry]
```

Connector/Python 8.1.0 through 8.4.0 included an [opentelemetry] option that installed a bundled version of the OpenTelemetry SDK/API libraries. Doing so in those versions was not recommended.

Instrumentation

For instrumenting an application, Connector/Python utilizes the official OpenTelemetry SDK to initialize OpenTelemetry, and the official OpenTelemetry API to instrument the application's code. This emits telemetry from the application and from utilized libraries that include instrumentation.

An application can be instrumented as demonstrated by this generic example:

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.sdk.trace.export import ConsoleSpanExporter

provider = TracerProvider()
processor = BatchSpanProcessor(ConsoleSpanExporter())
provider.add_span_processor(processor)
trace.set_tracer_provider(provider)
tracer = trace.get_tracer(__name__)

with tracer.start_as_current_span("app"):
    my_app()
```

To better understand and get started using OpenTelemetry tracing for Python, see the official OpenTelemetry Python Instrumentation guide.

MySQL Connector/Python

Connector/Python includes a MySQL instrumentor to instrument MySQL connections. This instrumentor provides an API and usage similar to OpenTelemetry's own MySQL package named opentelemetry-instrumentation-mysql.

An exception is raised if a system does not support OpenTelemetry when attempting to use the instrumentor.

An example that utilizes the system's OpenTelemetry SDK/API and implements tracing with MySQL Connector/Python:

```
import os
import mysql.connector

# An instrumentor that comes with mysql-connector-python
from mysql.connector.opentelemetry.instrumentation import (
    MySQLInstrumentor as OracleMySQLInstrumentor,
)

# Loading SDK from the system
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.sdk.trace.export import ConsoleSpanExporter

provider = TracerProvider()
processor = BatchSpanProcessor(ConsoleSpanExporter())
provider.add_span_processor(processor)
trace.set_tracer_provider(provider)
tracer = trace.get_tracer(__name__)

config = {
    "host": "127.0.0.1",
    "user": "root",
    "password": os.environ.get("password"),
    "use_pure": True,
    "port": 3306,
    "database": "test",
}

# Global instrumentation: all connection objects returned by
# mysql.connector.connect will be instrumented.
OracleMySQLInstrumentor().instrument()

with tracer.start_as_current_span("client_app"):
    with mysql.connector.connect(**config) as cnx:
        with cnx.cursor() as cur:
            cur.execute("SELECT @@version")
            _ = cur.fetchall()
```

Morphology of the Emitted Traces

A trace generated by the Connector/Python instrumentor contains one connection span, and zero or more query spans as described in the rest of this section.

Connection Span

- Time from connection initialization to the moment the connection ends. The span is named `connection`.
- If the application does not provide a span, the connection span generated is a ROOT span, originating in the connector.
- If the application does provide a span, the query span generated is a CHILD span, originating in the connector.

Query Span

- Time from when an SQL statement is requested (on the connector side) to the moment the connector finishes processing the server's reply to this statement.
- A query span is created for each query request sent to the server. If the application does not provide a span, the query span generated is a ROOT span, originating in the connector.
- If the application does provide a span, the query span generated is a CHILD span, originating in the connector.

- The query span is linked to the existing connection span of the connection the query was executed.
- Query attributes with prepared statements is supported as of MySQL Enterprise Edition 8.3.0.
- Query spans for the connection object is supported as of Connector/Python 8.3.0, which includes methods such as commit(), rollback(), and cmd_change_user().

Context Propagation

By default, the trace context of the span in progress (if any) is propagated to the MySQL server.

Propagation has no effect when the MySQL server either disabled or does not support OpenTelemetry (the trace context is ignored by the server), however, when connecting to a server with OpenTelemetry enabled and configured, the server processes the propagated traces and creates parent-child relationships between the spans from the connector and those from the server. In other words, this provides trace continuity.

> **Note**
>
> Context propagation with prepared statements is supported as of MySQL Enterprise Edition 8.3.0.

- The trace context is propagated for statements with query attributes defined in the MySQL client/server protocol, such as COM_QUERY.

  The trace context is not propagated for statements without query attributes defined in the MySQL client/server protocol, statements such as COM_PING.

- Trace context propagation is done via query attributes where a new attribute named "traceparent" is defined. Its value is based on the current span context. For details on how this value is computed, read the traceparent header W3C specification.

  If the "traceparent" query attribute is manually set for a query, then it is not be overwritten by the connector; it's assumed that it provides OTel context intended to forward to the server.

Disabling Trace Context Propagation

The boolean connection property named `otel_context_propagation` is `True` by default. Setting it to `False` disables context propagation.

Since `otel_context_propagation` is a connection property that can be changed after a connection is established (a connection object is created), setting such property to `False` does not have an effect over the spans generated during the connection phase. In other words, spans generated during the connection phase are always propagated since `otel_context_propagation` is `True` by default.

This implementation is distinct from the implementation provided through the MySQL client library (or the related `telemetry_client` client-side plugin).

9.3 Executing Multiple Statements

Connector/Python can execute either a single or multiple statements, this section references multiple statement and associated delimiter support.

> **Note**
>
> Before Connector/Python 9.2.0, the `multi` option was required to execute multiple statements. This option provided inconsistent results and was removed in 9.2.0.

Basic usage example:

```
sql_operation = """
SET @a=1, @b='2024-02-01';
SELECT @a, LENGTH('hello'), @b;
```

```
SELECT @@version;
"""

with cnx.cursor() as cur:
    # Execute SQL; it can contain one or multiple statements
    cur.execute(sql_operation)

    # Fetch result set, see other examples for additional information
```

Custom delimiters are also supported (as of Connector/Python 9.2.0), including in scripts that include delimiters and multiple statements. The Sakila sample database file `sakila-schema.sql` is an example:

```
with cnx.cursor() as cur:
    with open(
        os.path.join("/path/to/files", "sakila-schema.sql"), encoding="utf-8"
    ) as code:
        cur.execute(code.read())

    # Fetch result set, see other examples for additional information
```

Multiple Statement Result Mapping

The optional `map_results` option (defaults to `False`) makes each statement relate to its corresponding result set.

```
sql_operation = ...

with cnx.cursor() as cur:
    # Execute SQL; it can contain one or multiple statements
    cur.execute(sql_operation, map_results=True)

    # Fetch result set, see other examples for additional information
```

A MySQL multi statement or script is composed of one or more single statements. There are two types of single statements:

- `Simple`: these do not include a `BEGIN-END` body declaration.
- `Compound`: these do include a `BEGIN-END` body declaration, such as:

  ```
  CREATE PROCEDURE dorepeat(p1 INT)
  BEGIN
      SET @x = 0;
      REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;
  END;
  ```

Connector/Python uses custom delimiters to break up a multi statement into individual statements when handling compound single statements, like how the MySQL client does. Simple single statements do not require custom delimiters but they can be used.

If no delimiters are utilized when working with compound single statements, the statement-result mapping may cause unexpected results. If mapping is disabled, compound single statements may or may not utilize delimiters.

An example using a mix of simple and compound statements:

```
DROP PROCEDURE IF EXISTS dorepeat;

DELIMITER //

CREATE PROCEDURE dorepeat(p1 INT)
BEGIN
    SET @x = 0;
    REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;
END//

DELIMITER ;
```

```
SELECT @x;
```

Connector/Python carries on a pre-processing step for handling delimiters that may affect performance for large scripts. There are also limitations when working with custom delimiters:

- `Unsupported delimiters`: the following characters are not supported by the connector in `DELIMITER` statements:

  ```
  double quote: "
  single quote: '
  hash: #
  slash plus star: /*
  star plus slash: */
  ```

  Avoid using these symbols as part of a string representing a delimiter.

- `DELIMITER`: the word DELIMITER and any of its lower and upper case combinations such as delimiter, DeLiMiter, and so on, are considered reserved words by the connector. Users must quote these when included in multi statements for other purposes different from declaring an actual statement delimiter; such as names for tables, columns, variables, in comments, and so on. Example:

  ```
  CREATE TABLE `delimiter` (begin INT, end INT); -- I am a `DELimiTer` comment
  ```

Fetching Result Sets

Basic usage (mapping disabled):

```
sql_operation = """
SET @a=1, @b='2024-02-01';
SELECT @a, LENGTH('hello'), @b;
SELECT @@version;
"""

with cnx.cursor() as cur:
    # Execute a statement; it can be single or multi.
    cur.execute(sql_operation)

    # Fetch result sets and do something with them
    result_set = cur.fetchall()

    # do something with result set
    ...

    while cur.nextset():
        result_set = cur.fetchall()
        # do something with result set
        ...
```

The multi statement execution generates one or more result sets, in other words a set of result sets. The first result set is loadable after execution completes. You might fetch (using `fetchall()`) the current result set and process it, or not, and move onto the next one.

Alternatively, use the nextset() cursor API method to traverse a result set. This method makes the cursor skip to the next available set, discarding any remaining rows from the current set.

For executions generating only one result set, which happens when your script only includes one statement, the call to nextset() can be omitted as at most one result set is expected. Calling it returns `None` as there are no more sets.

With Statement-ResultSet mapping usage:

```
sql_operation = ...

with cnx.cursor() as cur:
    # Execute a statement; it can be single or multi.
    cur.execute(sql_operation, map_results=True)
```

```
    # Fetch result sets and do something with them.
    # statement 1 is `SET @a=1, @b='2025-01-01'`,
    # result set from statement 1 is `[]` - aka, an empty set.
    result_set, statement = cur.fetchall(), cur.statement
    # do something with result set
    ...

    # 1st call to `nextset()` will load the result set from statement 2,
    # statement 2 is `SELECT @a, LENGTH('hello'), @b`,
    # result set from statement 2 is `[(1, 5, '2025-01-01')]`.
    #
    # 2nd call to `nextset()` will load the result set from statement 3,
    # statement 3 is `SELECT @@version`,
    # result set from statement 3 is `[('9.2.0',)]`.
    #
    # 3rd call to `nextset()` will return `None` as there are no more sets,
    # leading to the end of the consumption process of result sets.
    while cur.nextset():
        result_set, statement = cur.fetchall(), cur.statement
        # do something with result set
        ...
```

When the mapping is disabled (`map_results=False`), all result sets are related to the same statement, which is the one provided when calling `execute()`. In other words, the `statement` property does not change while result sets are consumed, which differs from when mapping is enabled, when the `statement` property returns the statement that caused the current result set. Therefore, the value of statement changes accordingly while the result sets are traversed.

Shortcut for consuming result sets

A fetch-related API command shortcut is available to consume result sets, this example is equivalent to the previously presented workflow.

```
sql_operation = '''
SET @a=1, @b='2025-01-01';
SELECT @a, LENGTH('hello'), @b;
SELECT @@version;
'''
with cnx.cursor() as cur:
    cur.execute(sql_operation, map_results=True)
    for statement, result_set in cur.fetchsets():
        # do something with result set
```

The fetchsets() method returns a generator where each item is a 2-tuple; the first element is the statement that caused the result set, and the second is the result set itself. If mapping is disabled, statement will not change as result sets are consumed.

If statement is not needed, then consider this simpler option:

```
sql_operation = ...
with cnx.cursor() as cur:
    cur.execute(...)
    for _, result_set in cur.fetchsets():
        # do something with result set
```

9.4 Asynchronous Connectivity

Installing Connector/Python also installs the `mysql.connector.aio` package that integrates asyncio with the connector to allow integrating asynchronous MySQL interactions with an application.

Here are code examples that integrate `mysql.connector.aio` functionality:

Basic Usage:

```
from mysql.connector.aio import connect

# Connect to a MySQL server and get a cursor
cnx = await connect(user="myuser", password="mypass")
```

```
cur = await cnx.cursor()

# Execute a non-blocking query
await cur.execute("SELECT version()")

# Retrieve the results of the query asynchronously
results = await cur.fetchall()
print(results)

# Close cursor and connection
await cur.close()
await cnx.close()
```

Usage with context managers:

```
from mysql.connector.aio import connect

# Connect to a MySQL server and get a cursor
async with await connect(user="myuser", password="mypass") as cnx:
    async with await cnx.cursor() as cur:
        # Execute a non-blocking query
        await cur.execute("SELECT version()")

        # Retrieve the results of the query asynchronously
        results = await cur.fetchall()
        print(results)
```

Running Multiple Tasks Asynchronously

This example showcases how to run tasks asynchronously and the usage of to_thread, which is the backbone to asynchronously run blocking functions:

> **Note**
>
> The synchronous version of this example implements coroutines instead of following a common synchronous approach; this to explicitly demonstrate that only awaiting coroutines does not make the code run asynchronously. Functions included in the asyncio API must be used to achieve asynchronicity.

```
import asyncio
import os
import time

from mysql.connector.aio import connect

# Global variable which will help to format the job sequence output.
# DISCLAIMER: this is an example for showcasing/demo purposes,
# you should avoid global variables usage for production code.
global indent
indent = 0

# MySQL Connection arguments
config = {
    "host": "127.0.0.1",
    "user": "root",
    "password": os.environ.get("MYPASS", ":("),
    "use_pure": True,
    "port": 3306,
}


async def job_sleep(n):
    """Take a nap for n seconds.

    This job represents any generic task - it may be or not an IO task.
    """
    # Increment indent
    global indent
    offset = "\t" * indent
    indent += 1
```

```
    # Emulating a generic job/task
    print(f"{offset}START_SLEEP")
    await asyncio.sleep(n)
    print(f"{offset}END_SLEEP")

    return f"I slept for {n} seconds"


async def job_mysql():
    """Connect to a MySQL Server and do some operations.

    Run queries, run procedures, insert data, etc.
    """
    # Increment indent
    global indent
    offset = "\t" * indent
    indent += 1

    # MySQL operations
    print(f"{offset}START_MYSQL_OPS")
    async with await connect(**config) as cnx:
        async with await cnx.cursor() as cur:
            await cur.execute("SELECT @@version")
            res = await cur.fetchone()
            time.sleep(1)  # for simulating that the fetch isn't immediate
    print(f"{offset}END_MYSQL_OPS")

    # return server version
    return res


async def job_io():
    """Emulate an IO operation.

    `to_thread` allows to run a blocking function asynchronously.

    References:
        [asyncio.to_thread]: https://docs.python.org/3/library/asyncio-task.html#asyncio.to_thread
    """

    # Emulating a native blocking IO procedure
    def io():
        """Blocking IO operation."""
        time.sleep(5)

    # Increment indent
    global indent
    offset = "\t" * indent
    indent += 1

    # Showcasing how a native blocking IO procedure can be awaited,
    print(f"{offset}START_IO")
    await asyncio.to_thread(io)
    print(f"{offset}END_IO")

    return "I am an IO operation"


async def main_asynchronous():
    """Running tasks asynchronously.

    References:
        [asyncio.gather]: https://docs.python.org/3/library/asyncio-task.html#asyncio.gather
    """
    print("-------------------- ASYNCHRONOUS --------------------")

    # reset indent
    global indent
    indent = 0

    clock = time.time()
```

```
    # `asyncio.gather()` allows to run awaitable objects
    # in the aws sequence asynchronously.\

    # If all awaitables are completed successfully,
    # the result is an aggregate list of returned values.
    aws = (job_io(), job_mysql(), job_sleep(4))
    returned_vals = await asyncio.gather(*aws)

    print(f"Elapsed time: {time.time() - clock:0.2f}")

    # The order of result values corresponds to the
    # order of awaitables in aws.
    print(returned_vals, end="\n" * 2)

    # Example expected output
    # -------------------- ASYNCHRONOUS --------------------
    # START_IO
    #         START_MYSQL_OPS
    #                 START_SLEEP
    #         END_MYSQL_OPS
    #                 END_SLEEP
    # END_IO
    # Elapsed time: 5.01
    # ['I am an IO operation', ('8.3.0-commercial',), 'I slept for 4 seconds']


async def main_non_asynchronous():
    """Running tasks non-asynchronously"""
    print("------------------- NON-ASYNCHRONOUS -------------------")

    # reset indent
    global indent
    indent = 0

    clock = time.time()

    # Sequence of awaitable objects
    aws = (job_io(), job_mysql(), job_sleep(4))

    # The line below this docstring is the short version of:
    #     coro1, coro2, coro3 = *aws
    #     res1 = await coro1
    #     res2 = await coro2
    #     res3 = await coro3
    #     returned_vals = [res1, res2, res3]
    # NOTE: Simply awaiting a coro does not make the code run asynchronously!
    returned_vals = [await coro for coro in aws]  # this will run synchronously

    print(f"Elapsed time: {time.time() - clock:0.2f}")

    print(returned_vals, end="\n")

    # Example expected output
    # ------------------- NON-ASYNCHRONOUS -------------------
    # START_IO
    # END_IO
    #         START_MYSQL_OPS
    #         END_MYSQL_OPS
    #                 START_SLEEP
    #                 END_SLEEP
    # Elapsed time: 10.07
    # ['I am an IO operation', ('8.3.0-commercial',), 'I slept for 4 seconds']


if __name__ == "__main__":
    # `asyncio.run()`` allows to execute a coroutine (`coro`) and return the result.
    # You cannot run a coro without it.

    # References:
    #     [asyncio.run]: https://docs.python.org/3/library/asyncio-runner.html#asyncio.run
    assert asyncio.run(main_asynchronous()) == asyncio.run(main_non_asynchronous())
```

It shows these three jobs running asynchronously:

- `job_io`: Emulate an I/O operation; with to_thread to allow running a blocking function asynchronously.

  Starts first, and takes five seconds to complete so is the last job to finish.

- `job_mysql`: Connects to a MySQL server to perform operations such as queries and stored procedures.

  Starts second, and takes one second to complete so is the first job to finish.

- `job_sleep`: Sleeps for n seconds to represent a generic task.

  Starts last, and takes four seconds to complete so is the second job to finish.

> **Note**
>
> A lock/mutex wasn't added to the `indent` variable because multithreading isn't used; instead the unique active thread executes all of the jobs. Asynchronous execution is about completing other jobs while waiting for the result of an I/O operation.

Asynchronous MySQL Queries

This is a similar example that uses MySQL queries instead of generic jobs.

> **Note**
>
> While cursors are not utilized in the these examples, the principles and workflow could apply to cursors by letting every connection object create a cursor to operate from.

Synchronous code to create and populate hundreds of tables:

```
import os
import time
from typing import TYPE_CHECKING, Callable, List, Tuple

from mysql.connector import connect

if TYPE_CHECKING:
    from mysql.connector.abstracts import (
        MySQLConnectionAbstract,
    )


# MySQL Connection arguments
config = {
    "host": "127.0.0.1",
    "user": "root",
    "password": os.environ.get("MYPASS", ":("),
    "use_pure": True,
    "port": 3306,
}

exec_sequence = []


def create_table(
    exec_seq: List[str], table_names: List[str], cnx: "MySQLConnectionAbstract", i: int
) -> None:
    """Creates a table."""
    if i >= len(table_names):
        return False

    exec_seq.append(f"start_{i}")
    stmt = f"""
```

```
    CREATE TABLE IF NOT EXISTS {table_names[i]} (
        dish_id INT(11) UNSIGNED AUTO_INCREMENT UNIQUE KEY,
        category TEXT,
        dish_name TEXT,
        price FLOAT,
        servings INT,
        order_time TIME
    )
    """
    cnx.cmd_query(f"DROP TABLE IF EXISTS {table_names[i]}")
    cnx.cmd_query(stmt)
    exec_seq.append(f"end_{i}")
    return True


def drop_table(
    exec_seq: List[str], table_names: List[str], cnx: "MySQLConnectionAbstract", i: int
) -> None:
    """Drops a table."""
    if i >= len(table_names):
        return False

    exec_seq.append(f"start_{i}")
    cnx.cmd_query(f"DROP TABLE IF EXISTS {table_names[i]}")
    exec_seq.append(f"end_{i}")
    return True


def main(
    kernel: Callable[[List[str], List[str], "MySQLConnectionAbstract", int], None],
    table_names: List[str],
) -> Tuple[List, List]:

    exec_seq = []
    database_name = "TABLE_CREATOR"

    with connect(**config) as cnx:
        # Create/Setup database
        cnx.cmd_query(f"CREATE DATABASE IF NOT EXISTS {database_name}")
        cnx.cmd_query(f"USE {database_name}")

        # Execute Kernel: Create or Delete tables
        for i in range(len(table_names)):
            kernel(exec_seq, table_names, cnx, i)

        # Show tables
        cnx.cmd_query("SHOW tables")
        show_tables = cnx.get_rows()[0]

    # Return execution sequence and table names retrieved with `SHOW tables;`.
    return exec_seq, show_tables


if __name__ == "__main__":
    # with num_tables=511 -> Elapsed time ~ 25.86
    clock = time.time()
    print_exec_seq = False
    num_tables = 511
    table_names = [f"table_sync_{n}" for n in range(num_tables)]

    print("-------------------- SYNC CREATOR --------------------")
    exec_seq, show_tables = main(kernel=create_table, table_names=table_names)
    assert len(show_tables) == num_tables
    if print_exec_seq:
        print(exec_seq)

    print("-------------------- SYNC DROPPER --------------------")
    exec_seq, show_tables = main(kernel=drop_table, table_names=table_names)
    assert len(show_tables) == 0
    if print_exec_seq:
        print(exec_seq)
```

```
    print(f"Elapsed time: {time.time() - clock:0.2f}")

    # Expected output with num_tables = 11:
    # -------------------- SYNC CREATOR --------------------
    # [
    #     "start_0",
    #     "end_0",
    #     "start_1",
    #     "end_1",
    #     "start_2",
    #     "end_2",
    #     "start_3",
    #     "end_3",
    #     "start_4",
    #     "end_4",
    #     "start_5",
    #     "end_5",
    #     "start_6",
    #     "end_6",
    #     "start_7",
    #     "end_7",
    #     "start_8",
    #     "end_8",
    #     "start_9",
    #     "end_9",
    #     "start_10",
    #     "end_10",
    # ]
    # -------------------- SYNC DROPPER --------------------
    # [
    #     "start_0",
    #     "end_0",
    #     "start_1",
    #     "end_1",
    #     "start_2",
    #     "end_2",
    #     "start_3",
    #     "end_3",
    #     "start_4",
    #     "end_4",
    #     "start_5",
    #     "end_5",
    #     "start_6",
    #     "end_6",
    #     "start_7",
    #     "end_7",
    #     "start_8",
    #     "end_8",
    #     "start_9",
    #     "end_9",
    #     "start_10",
    #     "end_10",
    # ]
```

That script creates and deletes {num_tables} tables, and is fully sequential in that it creates and deletes table_{i} before moving to table_{i+1}.

An asynchronous code example for the same task:

```
import asyncio
import os
import time
from typing import TYPE_CHECKING, Callable, List, Tuple

from mysql.connector.aio import connect

if TYPE_CHECKING:
    from mysql.connector.aio.abstracts import (
        MySQLConnectionAbstract,
    )
```

```
# MySQL Connection arguments
config = {
    "host": "127.0.0.1",
    "user": "root",
    "password": os.environ.get("MYPASS", ":("),
    "use_pure": True,
    "port": 3306,
}

exec_sequence = []


async def create_table(
    exec_seq: List[str], table_names: List[str], cnx: "MySQLConnectionAbstract", i: int
) -> None:
    """Creates a table."""
    if i >= len(table_names):
        return False

    exec_seq.append(f"start_{i}")
    stmt = f"""
    CREATE TABLE IF NOT EXISTS {table_names[i]} (
        dish_id INT(11) UNSIGNED AUTO_INCREMENT UNIQUE KEY,
        category TEXT,
        dish_name TEXT,
        price FLOAT,
        servings INT,
        order_time TIME
    )
    """
    await cnx.cmd_query(f"DROP TABLE IF EXISTS {table_names[i]}")
    await cnx.cmd_query(stmt)
    exec_seq.append(f"end_{i}")
    return True


async def drop_table(
    exec_seq: List[str], table_names: List[str], cnx: "MySQLConnectionAbstract", i: int
) -> None:
    """Drops a table."""
    if i >= len(table_names):
        return False

    exec_seq.append(f"start_{i}")
    await cnx.cmd_query(f"DROP TABLE IF EXISTS {table_names[i]}")
    exec_seq.append(f"end_{i}")
    return True


async def main_async(
    kernel: Callable[[List[str], List[str], "MySQLConnectionAbstract", int], None],
    table_names: List[str],
    num_jobs: int = 2,
) -> Tuple[List, List]:
    """The asynchronous tables creator...
    Reference:
        [as_completed]: https://docs.python.org/3/library/asyncio-task.html#asyncio.as_completed
    """
    exec_seq = []
    database_name = "TABLE_CREATOR"

    # Create/Setup database
    # ---------------------
    # No asynchronous execution is done here.
    # NOTE: observe usage WITH context manager.
    async with await connect(**config) as cnx:
        await cnx.cmd_query(f"CREATE DATABASE IF NOT EXISTS {database_name}")
        await cnx.cmd_query(f"USE {database_name}")
    config["database"] = database_name

    # Open connections
    # ----------------
```

```
    # `as_completed` allows to run awaitable objects in the `aws` iterable asynchronously.
    # NOTE: observe usage WITHOUT context manager.
    aws = [connect(**config) for _ in range(num_jobs)]
    cnxs: List["MySQLConnectionAbstract"] = [
        await coro for coro in asyncio.as_completed(aws)
    ]

    # Execute Kernel: Create or Delete tables
    # -------------
    # N tables must be created/deleted and we can run up to `num_jobs` jobs asynchronously,
    # therefore we execute jobs in batches of size num_jobs`.
    returned_values, i = [True], 0
    while any(returned_values):  # Keep running until i >= len(table_names) for all jobs
        # Prepare coros: map connections/cursors and table-name IDs to jobs.
        aws = [
            kernel(exec_seq, table_names, cnx, i + idx) for idx, cnx in enumerate(cnxs)
        ]
        # When i >= len(table_names) coro simply returns False, else True.
        returned_values = [await coro for coro in asyncio.as_completed(aws)]
        # Update table-name ID offset based on the number of jobs
        i += num_jobs

    # Close cursors
    # -------------
    # `as_completed` allows to run awaitable objects in the `aws` iterable asynchronously.
    for coro in asyncio.as_completed([cnx.close() for cnx in cnxs]):
        await coro

    # Load table names
    # ----------------
    # No asynchronous execution is done here.
    async with await connect(**config) as cnx:
        # Show tables
        await cnx.cmd_query("SHOW tables")
        show_tables = (await cnx.get_rows())[0]

    # Return execution sequence and table names retrieved with `SHOW tables;`.
    return exec_seq, show_tables


if __name__ == "__main__":
    # `asyncio.run()`` allows to execute a coroutine (`coro`) and return the result.
    # You cannot run a coro without it.

    # References:
    #     [asyncio.run]: https://docs.python.org/3/library/asyncio-runner.html#asyncio.run

    # with num_tables=511 and num_jobs=3 -> Elapsed time ~ 19.09
    # with num_tables=511 and num_jobs=12 -> Elapsed time ~ 13.15
    clock = time.time()
    print_exec_seq = False
    num_tables = 511
    num_jobs = 12
    table_names = [f"table_async_{n}" for n in range(num_tables)]

    print("-------------------- ASYNC CREATOR --------------------")
    exec_seq, show_tables = asyncio.run(
        main_async(kernel=create_table, table_names=table_names, num_jobs=num_jobs)
    )
    assert len(show_tables) == num_tables
    if print_exec_seq:
        print(exec_seq)

    print("-------------------- ASYNC DROPPER --------------------")
    exec_seq, show_tables = asyncio.run(
        main_async(kernel=drop_table, table_names=table_names, num_jobs=num_jobs)
    )
    assert len(show_tables) == 0
    if print_exec_seq:
        print(exec_seq)

    print(f"Elapsed time: {time.time() - clock:0.2f}")
```

```
# Expected output with num_tables = 11 and num_jobs = 3:
# -------------------- ASYNC CREATOR --------------------
# 11
# [
#     "start_2",
#     "start_1",
#     "start_0",
#     "end_2",
#     "end_0",
#     "end_1",
#     "start_5",
#     "start_3",
#     "start_4",
#     "end_3",
#     "end_5",
#     "end_4",
#     "start_8",
#     "start_7",
#     "start_6",
#     "end_7",
#     "end_8",
#     "end_6",
#     "start_10",
#     "start_9",
#     "end_9",
#     "end_10",
# ]
# -------------------- ASYNC DROPPER --------------------
# [
#     "start_1",
#     "start_2",
#     "start_0",
#     "end_1",
#     "end_2",
#     "end_0",
#     "start_3",
#     "start_5",
#     "start_4",
#     "end_4",
#     "end_5",
#     "end_3",
#     "start_6",
#     "start_8",
#     "start_7",
#     "end_7",
#     "end_6",
#     "end_8",
#     "start_10",
#     "start_9",
#     "end_9",
#     "end_10",
# ]
```

This output shows how the job flow isn't sequential in that up to {num_jobs} can be executed asynchronously. The jobs are run following a batch-like approach of {num_jobs} and waits until all terminate before launching the next batch, and the loop ends once no tables remain to create.

Performance comparison for these examples: the asynchronous implementation is about 26% faster when using 3 jobs, and 49% faster using 12 jobs. Note that increasing the number of jobs does add job management overhead which at some point evaporates the initial speed-up. The optimal number of jobs is problem-dependent, and is a value determined with experience.

As demonstrated, the asynchronous version requires more code to function than the non-asynchronous variant. Is it worth the effort? It depends on the goal as asynchronous code better optimizes performance, such as CPU usage, whereas writing standard synchronous code is simpler.

For additional information about the asyncio module, see the official Asynchronous I/O Python Documentation.

9.5 Connector/Python Connection Pooling

Simple connection pooling is supported 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.
- websites 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 or equal to `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, “pooling.PooledMySQLConnection Class”). `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, “pooling.MySQLConnectionPool Class”):

```
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": "password"
}

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.

9.6 Connector/Python Django Back End

Connector/Python includes a `mysql.connector.django` module that provides a Django back end for MySQL. This back end supports new features found as of 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 back end, the example found in the Django manual can be used as a basis:

```
DATABASES = {
    'default': {
        'NAME': 'user_data',
        'ENGINE': 'mysql.connector.django',
        'HOST': '127.0.0.1',
        'PORT': 3306,
        'USER': 'mysql_user',
        'PASSWORD': 'password',
        'OPTIONS': {
          'autocommit': True,
          'use_oure': True,
          'init_command': "SET foo='bar';"
        },
    }
}
```

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 back end 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.

Using a custom class for data type conversation is supported as a subclass of *mysql.connector.django.base.DjangoMySQLConverter*. This support was added in Connector/Python 8.0.29.

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.

> **Note**
>
> Python 2.7 support was removed in Connector/Python 8.0.24.

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 mysql.connector Module

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

10.1.1 mysql.connector.connect() Method

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, “connection.MySQLConnection Class”.

10.1.2 mysql.connector.apilevel Property

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

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

10.1.3 mysql.connector.paramstyle Property

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

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

10.1.4 mysql.connector.threadsafety Property

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

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

10.1.5 mysql.connector.__version__ Property

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 mysql.connector.__version_info__ Property

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 connection.MySQLConnection Class

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 connection.MySQLConnection() Constructor

Syntax:

```
cnx = 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 MySQLConnection.close() Method

Syntax:

```
cnx.close()
```

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

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 9.5, “Connector/Python Connection Pooling”.

10.2.3 MySQLConnection.commit() Method

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), (%s)", ('Jane', 'Mary'))
>>> cnx.commit()
```

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

10.2.4 MySQLConnection.config() Method

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 9.5, “Connector/Python Connection Pooling”.

10.2.5 MySQLConnection.connect() Method

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 9.5, “Connector/Python Connection Pooling”.

10.2.6 MySQLConnection.cursor() Method

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, “cursor.MySQLCursor Class”, and Section 10.6, “Subclasses cursor.MySQLCursor”.

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` 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, “cursor.MySQLCursorBuffered Class”.

- 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 `prepared` is `True`, the cursor is used for executing prepared statements. This argument is available as of Connector/Python 1.1.2. The C extension supports this as of Connector/Python 8.0.17.

- 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 MySQLConnection.cmd_change_user() Method

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.

Syntax:

```
cnx.cmd_change_user(username='', password='', database='', charset=33)
```

Returns a dictionary containing the OK packet information.

10.2.8 MySQLConnection.cmd_debug() Method

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 MySQLConnection.cmd_init_db() Method

Syntax:

```
cnx.cmd_init_db(db_name)
```

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 MySQLConnection.cmd_ping() Method

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 MySQLConnection.cmd_process_info() Method

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

> **Deprecation**
>
> This MySQL Server functionality is deprecated.

10.2.12 MySQLConnection.cmd_process_kill() Method

Syntax:

```
cnx.cmd_process_kill(mysql_pid)
```

> **Deprecation**
>
> This MySQL Server functionality is deprecated.

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 MySQLConnection.cmd_query() Method

Syntax:

```
cnx.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 MySQLConnection.cmd_query_iter() Method

Syntax:

```
cnx.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 MySQLConnection.cmd_quit() Method

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 MySQLConnection.cmd_refresh() Method

Syntax:

```
cnx.cmd_refresh(options)
```

> **Deprecation**
>
> This MySQL Server functionality is deprecated.

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, “constants.RefreshOption Class”.

Example:

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

10.2.17 MySQLConnection.cmd_reset_connection() Method

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 MySQLConnection.cmd_shutdown() Method

> **Deprecation**
>
> This MySQL Server functionality is deprecated.

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 MySQLConnection.cmd_statistics() Method

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 MySQLConnection.disconnect() Method

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 MySQLConnection.get_row() Method

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 MySQLConnection.get_rows() Method

Syntax:

```
cnx.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 MySQLConnection.get_server_info() Method

> **Deprecation**
>
> This method has been deprecated as of 9.3.0. Use the property method Section 10.2.49, “MySQLConnection.server_info Property” instead.

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 MySQLConnection.get_server_version() Method

> **Deprecation**
>
> This method has been deprecated as of 9.3.0. Use the property method Section 10.2.51, “MySQLConnection.server_version Property” instead.

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

10.2.25 MySQLConnection.is_connected() Method

> **Deprecation**
>
> This method has been deprecated as of 9.3.0. Use the property method Section 10.2.41, “MySQLConnection.connected Property” instead.

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 MySQLConnection.isset_client_flag() Method

Syntax:

```
cnx.isset_client_flag(flag)
```

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

10.2.27 MySQLConnection.ping() Method

Syntax:

```
cnx.ping(reconnect=False, 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, and these options are forwarded to 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 MySQLConnection.reconnect() Method

Syntax:

```
cnx.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 MySQLConnection.reset_session() Method

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. For MySQL servers 5.7 or higher, the `cmd_reset_connection()` method is a more lightweight alternative.

This method was added in Connector/Python 1.2.1.

10.2.30 MySQLConnection.rollback() Method

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), (%s)", ('Jane', 'Mary'))
>>> cnx.rollback()
```

To commit modifications, see the commit() method.

10.2.31 MySQLConnection.set_charset_collation() Method

Syntax:

```
cnx.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 MySQLConnection.set_client_flags() Method

> **Deprecation**
>
> This method has been deprecated as of 9.3.0. Use the property method Section 10.2.39, “MySQLConnection.client_flags Property” instead.

Syntax:

```
cnx.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, “constants.ClientFlag Class”).

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 MySQLConnection.shutdown() Method

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 MySQLConnection.start_transaction() Method

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 Statements. 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 MySQLConnection.autocommit Property

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 MySQLConnection.unread_results Property

Indicates whether there is an unread result. It is set to `False` if there is not an unread result, otherwise `True`. This is used by cursors to check whether another cursor still needs to retrieve its result set.

Do not set the value of this property, as only the connector should change the value. In other words, treat this as a read-only property.

10.2.37 MySQLConnection.can_consume_results Property

This property indicates the value of the `consume_results` connection parameter that controls whether result sets produced by queries are automatically read and discarded. See Section 7.1, “Connector/Python Connection Arguments”.

This method was added in Connector/Python 2.1.1.

10.2.38 MySQLConnection.charset Property

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

10.2.39 MySQLConnection.client_flags Property

Syntax:

```
>>> cnx.client_flags=flags
>>> cnx.clieng_flags
```

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

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.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.

10.2.40 MySQLConnection.collation Property

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

10.2.41 MySQLConnection.connected Property

Reports whether the connection to MySQL Server is available.

This read-only property checks whether the connection to MySQL is available using the ping() method; but unlike `ping()`, `connected` returns `True` when the connection is available, and `False` otherwise.

10.2.42 MySQLConnection.connection_id Property

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

10.2.43 MySQLConnection.converter-class Property

This property sets and returns the converter class to use when configuring the connection.

```
# get the current converter class being used
print(cnx.converter_class)
>> <class 'mysql.connector.conversion.MySQLConverter'>

class TestConverter(MySQLConverterBase): ...

# set the custom converter class
cnx.converter_class = TestConverter
print(cnx.converter_class)
>> <class '__main__.TestConverter'>
```

10.2.44 MySQLConnection.database Property

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.

10.2.45 MySQLConnection.get_warnings Property

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`.

10.2.46 MySQLConnection.in_transaction Property

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 an 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.

10.2.47 MySQLConnection.raise_on_warnings Property

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 10.2.52, “MySQLConnection.sql_mode Property”). 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`.

10.2.48 MySQLConnection.server_host Property

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

Returns a string.

10.2.49 MySQLConnection.server_info Property

This read-only property returns the MySQL server information verbatim as a string: for example `8.4.0-log`, or `None` when not connected.

10.2.50 MySQLConnection.server_port Property

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

Returns an integer.

10.2.51 MySQLConnection.server_version Property

This read-only property returns the MySQL server version as a tuple, or `None` when not connected.

10.2.52 MySQLConnection.sql_mode Property

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.

10.2.53 MySQLConnection.time_zone Property

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.

10.2.54 MySQLConnection.use_unicode Property

This property sets and returns whether the connection uses Unicode with the value `True` or `False`.

```
# gets whether the connector returns string fields as unicode or not
print(cnx.use_unicode)
>> True

# set or update use_unicode property
cnx.use_unicode = False
print(cnx.use_unicode)
>> False
```

10.2.55 MySQLConnection.unix_socket Property

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

Returns a string.

10.2.56 MySQLConnection.user Property

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

Returns a string.

10.3 pooling.MySQLConnectionPool Class

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

10.3.1 pooling.MySQLConnectionPool Constructor

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 MySQLConnectionPool.add_connection() Method

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 MySQLConnectionPool.get_connection() Method

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 MySQLConnectionPool.set_config() Method

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": "password",
}

cnxpool.set_config(**dbconfig)
```

10.3.5 MySQLConnectionPool.pool_name Property

Syntax:

```
cnxpool.pool_name
```

This property returns the connection pool name.

Example:

```
name = cnxpool.pool_name
```

10.4 pooling.PooledMySQLConnection Class

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 9.5, “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 by Section 9.5, “Connector/Python Connection Pooling”.
- A pooled connection has a `pool_name` property that returns the pool name.

10.4.1 pooling.PooledMySQLConnection Constructor

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 PooledMySQLConnection.close() Method

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 PooledMySQLConnection.config() Method

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

10.4.4 PooledMySQLConnection.pool_name Property

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 cursor.MySQLCursor Class

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, “cursor.MySQLCursorBuffered Class”.

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

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

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

- `MySQLCursorDict` creates a cursor that returns rows as dictionaries. See Section 10.6.3, “cursor.MySQLCursorDict Class”.

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

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

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

- `MySQLCursorPrepared` creates a cursor for executing prepared statements. See Section 10.6.5, “cursor.MySQLCursorPrepared Class”.

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

10.5.1 cursor.MySQLCursor Constructor

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 MySQLCursor.add_attribute() Method

Syntax:

```
cursor.add_attribute(name, value)
```

Adds a new named query attribute to the list, as part of MySQL server's Query Attributes functionality.

`name`: The name must be a string, but no other validation checks are made; attributes are sent as is to the server and errors, if any, will be detected and reported by the server.

`value`: a value converted to the MySQL Binary Protocol, similar to how prepared statement parameters are converted. An error is reported if the conversion fails.

Query attributes must be enabled on the server, and are disabled by default. A warning is logged when setting query attributes server connection that does not support them. See also Prerequisites for Using Query Attributes for enabling the query_attributes MySQL server component.

Example query attribute usage:

```
#  Each invocation of `add_attribute` method will add a new query attribute:
    cur.add_attribute("foo", 2)
    cur.execute("SELECT first_name, last_name FROM clients")
 # The query above sent attibute "foo" with value 2.

    cur.add_attribute(*("bar", "3"))
    cur.execute("SELECT * FROM products WHERE price < ?", 10)
 # The query above sent attributes ("foo", 2)  and ("bar", "3").

 my_attributes = [("page_name", "root"), ("previous_page", "login")]
    for attribute_tuple in my_attributes:
  cur.add_attribute(*attribute_tuple)
    cur.execute("SELECT * FROM offers WHERE publish = ?", 0)
 # The query above sent 4 attributes.

# To check the current query attributes:

 print(cur.get_attributes())
 # prints:
 [("foo", 2), ("bar", "3"), ("page_name", "root"), ("previous_page", "login")]

# Query attributes are not cleared until the cursor is closed or
# of the clear_attributes() method is invoked:

 cur.clear_attributes()
    print(cur.get_attributes())
 # prints:
 []
    cur.execute("SELECT first_name, last_name FROM clients")
    # The query above did not send any attibute.
```

This method was added in Connector/Python 8.0.26.

10.5.3 MySQLCursor.clear_attributes() Method

Syntax:

```
cursor.clear_attributes()
```

Clear the list of query attributes on the connector's side, as set by Section 10.5.2, “MySQLCursor.add_attribute() Method”.

This method was added in Connector/Python 8.0.26.

10.5.4 MySQLCursor.get_attributes() Method

Syntax:

```
cursor.get_attributes()
```

Return a list of existing query attributes, as set by Section 10.5.2, “MySQLCursor.add_attribute() Method”.

This method was added in Connector/Python 8.0.26.

10.5.5 MySQLCursor.callproc() Method

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])
```

10.5.6 MySQLCursor.close() Method

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.

10.5.7 MySQLCursor.execute() Method

Syntax:

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

# Allowed before 9.2.0
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).

Before Connector/Python 9.2.0, `execute()` accepted a `multi` option and returned an iterator if set to `True`. That option was removed in 9.2.0, and Section 9.3, “Executing Multiple Statements” was added.

> **Note**
>
> In Python, a tuple containing a single value must include a comma. For example, *('abc')* is evaluated as a scalar while *('abc',)* is evaluated as a tuple.

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

10.5.8 MySQLCursor.executemany() Method

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`.

> **Note**
>
> In Python, a tuple containing a single value must include a comma. For example, *('abc')* is evaluated as a scalar while *('abc',)* is evaluated as a tuple.

In most cases, the `executemany()` method iterates through the sequence of parameters, each time passing the current parameters to 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 Section 9.3, "Executing Multiple Statements" instead.

10.5.9 MySQLCursor.fetchall() Method

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.

10.5.10 MySQLCursor.fetchmany() Method

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.

10.5.11 MySQLCursor.fetchone() Method

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, "cursor.MySQLCursorRaw Class".

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.

10.5.12 MySQLCursor.nextset() Method

Syntax:

```
row = cursor.nextset()
```

This method makes the cursor skip to the next available set, discarding any remaining rows from the current set. It returns `None` if there are no more sets or returns `True` and subsequent calls to the cursor.fetch*() methods returns rows from the next result set.

This method can be used as part of the multi statement execution workflow.

```
sql_operation = '''
SET @a=1, @b='2025-01-01';
SELECT @a, LENGTH('hello'), @b;
SELECT @@version;
'''
with cnx.cursor() as cur:
    cur.execute(sql_operation)

    result_set = cur.fetchall()
    # do something with result set
    ...

    while cur.nextset():
        result_set = cur.fetchall()
        # do something with result set
```

This method was added in Connector/Python 9.2.0.

10.5.13 MySQLCursor.fetchsets() Method

Syntax:

```
for statement, result_set in cursor.fetchsets():
    # do something with statement and/or result set
```

This method generates a set of result sets caused by the last cursor.execute*(). It returns a generator where each item is a 2-tuple; the first element is the statement that caused the result set, and the second is the result set itself.

This method can be used as part of the multi statement execution workflow.

```
    sql_operation = '''
    SET @a=1, @b='2025-01-01';
    SELECT @a, LENGTH('hello'), @b;
    SELECT @@version;
    '''
    with cnx.cursor() as cur:
        cur.execute(sql_operation)
        for statement, result_set in cur.fetchsets():
            # do something with statement and/or result set
```

This method was added in Connector/Python 9.2.0.

10.5.14 MySQLCursor.fetchwarnings() Method

> **Deprecation**
>
> This method has been deprecated as of 9.3.0. Use the property method Section 10.5.18, “MySQLCursor.warnings Property” instead.

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.

10.5.15 MySQLCursor.stored_results() Method

> **Deprecation**
>
> This method has been deprecated as of 9.3.0.

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,)]
```

10.5.16 MySQLCursor.column_names Property

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 10.6.3, “cursor.MySQLCursorDict Class”.

10.5.17 MySQLCursor.description Property

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
```

10.5.18 MySQLCursor.warnings Property

Syntax:

```
tuples = cursor.warnings
```

This property 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,)]
>>> print(cursor.warnings)
[(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.

10.5.19 MySQLCursor.lastrowid Property

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().

10.5.20 MySQLCursor.rowcount Property

Syntax:

```
count = cursor.rowcount
```

This read-only property returns the number of rows returned for `SELECT` statements, or the number of rows affected by DML statements such as `INSERT` or `UPDATE`. For an example, see Section 10.5.7, “MySQLCursor.execute() Method”.

For nonbuffered cursors, the row count cannot be known before the rows have been fetched. In this case, the number of rows is -1 immediately after query execution and is incremented as rows are fetched.

The `rowcount` property is like the `mysql_affected_rows()` C API function; see mysql_affected_rows().

10.5.21 MySQLCursor.statement Property

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.

10.5.22 MySQLCursor.with_rows Property

Syntax:

```
boolean = cursor.with_rows
```

This read-only property returns `True` or `False` to indicate whether the most recently executed operation could have 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):
  if result.with_rows:
    result.fetchall()
  else:
    print("Number of affected rows: {}".format(result.rowcount))
```

10.6 Subclasses cursor.MySQLCursor

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

10.6.1 cursor.MySQLCursorBuffered Class

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 cursor.MySQLCursorRaw Class

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 cursor.MySQLCursorDict Class

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))
```

10.6.4 cursor.MySQLCursorBufferedDict Class

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, "cursor.MySQLCursorBuffered Class".

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)
```

10.6.5 cursor.MySQLCursorPrepared Class

The `MySQLCursorPrepared` class inherits from `MySQLCursor`.

> **Note**
>
> This class is available as of Connector/Python 1.1.0. The C extension supports it as of Connector/Python 8.0.17.

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 Statement Interface.

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.

The MySQL client/server protocol has an option to send prepared statement parameters via the `COM_STMT_SEND_LONG_DATA` command. To use this from Connector/Python scripts, send the parameter in question using the `IOBase` interface. Example:

```
from io import IOBase

...

cur = cnx.cursor(prepared=True)
cur.execute("SELECT (%s)", (io.BytesIO(bytes("A", "latin1")), ))
```

10.7 constants.ClientFlag Class

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, “MySQLConnection.set_client_flags() Method” and the connection argument `client_flag`.

The `ClientFlag` class cannot be instantiated.

10.8 constants.FieldType Class

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 constants.SQLMode Class

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 10.2.52, “MySQLConnection.sql_mode Property”.

The `SQLMode` class cannot be instantiated.

10.10 constants.CharacterSet Class

This class provides all known MySQL characters sets and their default collations. For examples, see Section 10.2.31, “MySQLConnection.set_charset_collation() Method”.

The `CharacterSet` class cannot be instantiated.

10.11 constants.RefreshOption Class

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.REPLICA`

  On a replica replication server, reset the source server information and restart the replica, like `RESET SLAVE`. This constant was named "RefreshOption.SLAVE" before v8.0.23.

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 Message Reference). 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 Mapping of Server Errors to Python Exceptions

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
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 errorcode Module

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
```

For more information about MySQL errors, see Error Messages and Common Problems.

10.12.2 errors.Error Exception

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 errors.DataError Exception

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 errors.DatabaseError Exception

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 errors.IntegrityError Exception

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 errors.InterfaceError Exception

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 errors.InternalError Exception

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 errors.NotSupportedError Exception

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 errors.OperationalError Exception

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 errors.PoolError Exception

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

10.12.11 errors.ProgrammingError Exception

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 errors.Warning Exception

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 errors.custom_error_exception() Function

Syntax:

```
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 _mysql_connector Module

The `_mysql_connector` module provides classes.

11.2 _mysql_connector.MySQL() Class

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='password',
             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 _mysql_connector.MySQL.affected_rows() Method

Syntax:

```
count = ccnx.affected_rows()
```

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

11.4 _mysql_connector.MySQL.autocommit() Method

Syntax:

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

Sets the autocommit mode.

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

11.5 _mysql_connector.MySQL.buffered() Method

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 _mysql_connector.MySQL.change_user() Method

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 _mysql_connector.MySQL.character_set_name() Method

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 _mysql_connector.MySQL.close() Method

Syntax:

```
ccnx.close()
```

Closes the MySQL connection.

11.9 _mysql_connector.MySQL.commit() Method

Syntax:

```
ccnx.commit()
```

Commits the current transaction.

11.10 _mysql_connector.MySQL.connect() Method

Syntax:

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

Connects to a MySQL server.

```
import _mysql_connector

ccnx = _mysql_connector.MySQL()
ccnx.connect(user='scott', password='password',
             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 _mysql_connector.MySQL.connected() Method

Syntax:

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

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

11.12 _mysql_connector.MySQL.consume_result() Method

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 _mysql_connector.MySQL.convert_to_mysql() Method

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 _mysql_connector.MySQL.escape_string() Method

Syntax:

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

Uses the `mysql_escape_string()` C API function to create an 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 _mysql_connector.MySQL.fetch_fields() Method

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 _mysql_connector.MySQL.fetch_row() Method

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 _mysql_connector.MySQL.field_count() Method

Syntax:

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

Returns the number of columns in the active result set.

11.18 _mysql_connector.MySQL.free_result() Method

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 _mysql_connector.MySQL.get_character_set_info() Method

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 _mysql_connector.MySQL.get_client_info() Method

Syntax:

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

Returns the MySQL client library version as a string.

11.21 _mysql_connector.MySQL.get_client_version() Method

Syntax:

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

Returns the MySQL client library version as a tuple.

11.22 _mysql_connector.MySQL.get_host_info() Method

Syntax:

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

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

11.23 _mysql_connector.MySQL.get_proto_info() Method

Syntax:

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

Returns the protocol version used by the current session.

11.24 _mysql_connector.MySQL.get_server_info() Method

Syntax:

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

Returns the MySQL server version as a string.

11.25 _mysql_connector.MySQL.get_server_version() Method

Syntax:

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

Returns the MySQL server version as a tuple.

11.26 _mysql_connector.MySQL.get_ssl_cipher() Method

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 _mysql_connector.MySQL.hex_string() Method

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 _mysql_connector.MySQL.insert_id() Method

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 _mysql_connector.MySQL.more_results() Method

Syntax:

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

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

11.30 _mysql_connector.MySQL.next_result() Method

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 _mysql_connector.MySQL.num_fields() Method

Syntax:

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

Returns the number of columns in the active result set.

11.32 _mysql_connector.MySQL.num_rows() Method

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 _mysql_connector.MySQL.ping() Method

Syntax:

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

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

11.34 _mysql_connector.MySQL.query() Method

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 _mysql_connector.MySQL.raw() Method

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 _mysql_connector.MySQL.refresh() Method

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 _mysql_connector.MySQL.reset_connection() Method

Syntax:

```
ccnx.reset_connection()
```

Resets the user variables and session variables for a connection session.

11.38 _mysql_connector.MySQL.rollback() Method

Syntax:

```
ccnx.rollback()
```

Rolls back the current transaction.

Raises a `MySQLInterfaceError` exception on errors.

11.39 _mysql_connector.MySQL.select_db() Method

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.

11.40 _mysql_connector.MySQL.set_character_set() Method

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`.

11.41 _mysql_connector.MySQL.shutdown() Method

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.

11.42 _mysql_connector.MySQL.stat() Method

Syntax:

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

Returns the server status as a string.

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

11.43 _mysql_connector.MySQL.thread_id() Method

Syntax:

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

Returns the current thread or connection ID.

11.44 _mysql_connector.MySQL.use_unicode() Method

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.

11.45 _mysql_connector.MySQL.warning_count() Method

Syntax:

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

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

11.46 _mysql_connector.MySQL.have_result_set Property

Syntax:

```
has_rows = ccnx.have_result_set
```

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

Index

Symbols

C

D

E

F

M

P

S