Contents

[Introduction 2](#_Toc107912601)

[Install Connector/Python 2](#_Toc107912602)

[More Complex Samples 3](#_Toc107912603)

# Introduction

<https://www.w3schools.com/python/python_mysql_getstarted.asp>

MySQL Connector/Python enables Python programs to access MySQL databases, using an API.

MySQL Connector/Python includes support for:

* Almost all features provided by MySQL Server up to and including MySQL Server version 8.0.
* Connector/Python 8.0 also supports X DevAPI.
* 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.

## Install Connector/Python

Use **pip** to install Connector/Python on most any operating system:

$ python -m pip install mysql-connector-python

**For Windows**:

* MySQL Installer (recommended): When executing [MySQL Installer](https://dev.mysql.com/doc/refman/8.0/en/mysql-installer.html), choose MySQL Connector/Python as one of the products to install. MySQL Installer installs the Windows MSI Installer described in this documentation.
* You can download and install from <https://dev.mysql.com/downloads/connector/python/>

**Linux Using the MySQL Yum Repository:**

$ sudo yum update mysql-community-release

**Sample Employees Database**: https://github.com/datacharmer/test\_db

Sample code: <https://github.com/AjaySingala/PythonSamples/tree/main/MySQL>

Demo code files:

1. TestConnection.py
2. CreateDB.py
3. CreateTable.py
4. Insert.py
5. GetInsertedId.py
6. InsertMany.py
7. Select.py
8. SelectWhere.py (*also has Wildcard demo*)
9. Limit.py
10. Update.py
11. OrderBy.py
12. Delete.py
13. DropTable.py
14. Join.py

For the Join.py demo:

You can combine rows from two or more tables, based on a related column between them, by using a JOIN statement.

Consider you have a "users" table and a "products" table:

**users**

{id: 1, name: 'John', fav: 154},

{id: 2, name: 'Peter', fav: 154},

{id: 3, name: 'Amy', fav: 155},

{id: 4, name: 'Hannah', fav:},

{id: 5, name: 'Michael', fav:}

**products**

{id: 154, name: 'Chocolate Heaven'},

{id: 155, name: 'Tasty Lemons'},

{id: 156, name: 'Vanilla Dreams'}

These two tables can be combined by using users' fav field and products' id field.

## More Complex Samples

<https://dev.mysql.com/doc/connector-python/en/connector-python-examples.html>

The Employees database is available from [Employees DB on GitHub](https://github.com/datacharmer/test_db) (<https://github.com/datacharmer/test_db>). You can download a pre-packaged archive of the data, or access the information through Git.

To create the database with the data, clone the repo to a local directory and then from the test\_db folder, execute the following command:

$> mysql -t < employees.sql -uUSER\_NAME -pYOUR\_PWD

You could also load the data into the tables individually. Create the DB and the tables and then execute the following commands in the given order:

$> mysql employees < load\_departments.dump

$> mysql employees < load\_employees.dump

$> mysql employees < load\_dept\_emp.dump

$> mysql employees < load\_dept\_manager.dump

$> mysql employees < load\_titles.dump

$> mysql employees < load\_salaries1.dump

$> mysql employees < load\_salaries2.dump

$> mysql employees < load\_salaries3.dump

Source code: <https://github.com/AjaySingala/PythonSamples/tree/main/MySQL/EmployeesDemo>

Demo code files:

1. Connect.py
2. CreateDB\_Tables.py
3. InsertData.py (see notes below)
4. QueryData.py (see notes below)
5. BufferedCursor.py (see notes below)

**Notes for InsertData.py**:

* 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.
* 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](https://dev.mysql.com/doc/refman/8.0/en/glossary.html" \l "glos_autocommit" \t "_top) 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](https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_rollback) using the rollback() method.
* A future statement is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python. The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

**Notes for QueryData.py**:

* The task is to select all employees hired in the year 1999 and print their names and hire dates to the console.
* 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()](https://dev.mysql.com/doc/connector-python/en/connector-python-api-mysqlcursor-execute.html) 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.

**Notes for BufferedCursor.py**:

* The 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*). This way, it is unnecessary to fetch the rows in a new variables. Instead, the cursor can be used as an iterator.