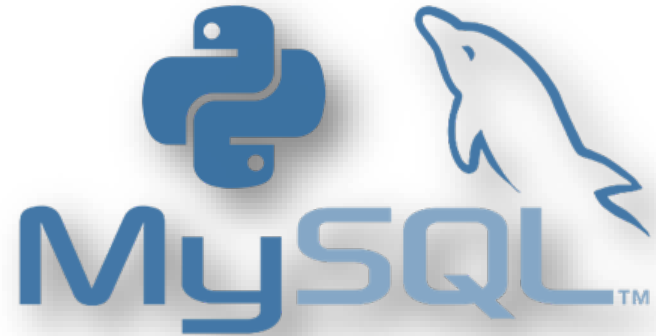




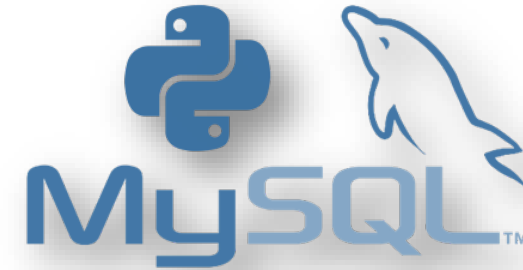
# Python and MySQL





# Content

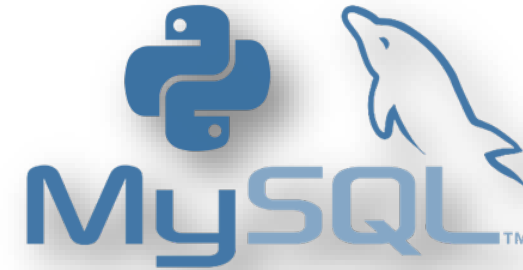
- Introduction
- Install – Connect
- Create – Insert – Query





# Content

- Introduction
- Install – Connect
- Create – Insert – Query





# Introduction

## Connector

To be able to connect to a MySQL database in Python, a database driver called [MySQL Connector/Python](#) is required.

MySQL offers connectors (drivers) for all of the most common programming environments.

Python Application – Client



Connector – Driver



Database – Server





# Introduction

## Guidelines

→ [MySQL Python Developer Guide](#)

- Do not hardcode login information 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.
- Any application that accepts input must expect to handle bad data.
- Data that you choose to store in MySQL is likely to have special characteristics:
  - Too large to all fit in memory at one time.
  - Too complex to be represented by a single data structure.
  - Updated frequently - perhaps by multiple users simultaneously.
- Issuing SQL statements from Python typically involves declaring very long, possibly multi-line string literals.



# Introduction

## Guidelines – expect to handle bad data (1/2)

The bad data might be **accidental**, such as out-of-range values or misformatted strings. The MySQL server has most likely already some checks – such as unique constraints and NOT NULL constraints – in place to keep the bad data from ever reaching the database.

In Python, use techniques such as **exception handlers** to report any problems and take corrective action:

```
# catch bad data
try:
    some SQL action
except:
    show error/warning message
    some corrective action
```



# Introduction

## Guidelines – expect to handle bad data (2/2)

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** (i.e. put a backslash \ in front) 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.

```
# escape special characters
import re
my_text = 'blabla % ; blabla "'
escaped_text = re.escape(my_text)
```



# Introduction

## Guidelines – special characteristics (1/3)

- Too large to all fit in memory at one time.
- Too complex to be represented by a single data structure.
- Updated frequently - perhaps by multiple users simultaneously.

Use **SELECT** statements to query only the precise items you need.





# Introduction

## Guidelines – special characteristics (2/3)

- Too large to all fit in memory at one time.
- Too complex to be represented by a single data structure.
- Updated frequently - perhaps by multiple users simultaneously.

The data is divided between different SQL tables. Recombine data from multiple tables by using a **JOIN** query. Make sure that related data is kept in sync between different tables by setting up **FOREIGN KEY** relationships.



# Introduction

## Guidelines – special characteristics (3/3)

- Too large to all fit in memory at one time.
- Too complex to be represented by a single data structure.
- 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. Use the SQL **INSERT**, **UPDATE**, and **DELETE** statements to update different items concurrently, writing only the changed values to disk.



# Introduction

## Guidelines – very long, multi-line strings

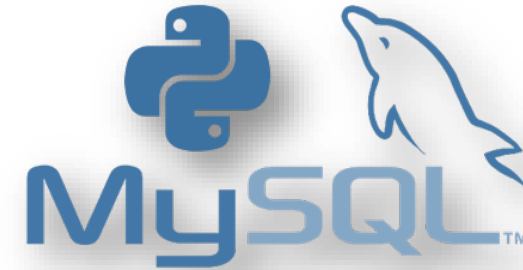
Because string literals within the SQL statements could be enclosed by single quotation, double quotation marks, or contain either of those characters, for simplicity you can use Python's triple-quoting mechanism to enclose the entire statement:

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



# Content

- Introduction
- **Install – Connect**
- Create – Insert – Query





# Install

## Download

MySQL

The world's most popular open source database

MySQL.COM

DOWNLOADS

DOCUMENTATION

DEVELOPER ZONE

Contact MySQL

Login

Register

Enterprise

Community

Yum Repository

APT Repository

SUSE Repository

Windows

Archives

MySQL on Windows

MySQL Yum Repository

MySQL APT Repository

MySQL SUSE Repository

MySQL Community Server

MySQL Cluster

MySQL Router

MySQL Utilities

MySQL Shell

MySQL Workbench

MySQL Connectors

- Connector/ODBC
- Connector/Net
- Connector/J
- Connector/Node.js
- Connector/Python

Download Connector/Python

MySQL Connector/Python is a standardized database driver for Python platforms and development.

Online Documentation:

- MySQL Connector/Python Installation Instructions
- Documentation
- MySQL Connector/Python X DevAPI Reference
- Change History

Please report any bugs or inconsistencies you observe to our [Bugs Database](#).  
Thank you for your support!

Generally Available (GA) Releases

Development Releases

Connector/Python 2.1.7

Select Operating System:  

Microsoft Windows

Select OS Version:  

All

Looking for previous GA versions?

MySQL open source software is provided under the GPL License.  
OEMs, ISVs and VARs can purchase commercial licenses.

## Install

MySQL

The world's most popular open source database

MySQL.COM

DOWNLOADS

DOCUMENTATION

DEVELOPER ZONE

Contact MySQL

Login

Register

MySQL Server

MySQL Enterprise

Workbench

InnoDB Cluster

MySQL NDB Cluster

Connectors

More

Search this Manual

Documentation Home

MySQL Connector/Python Developer Guide

- Preface and Legal Notices
- Introduction to MySQL Connector/Python
- Guidelines for Python Developers
- Connector/Python Versions
- Connector/Python Installation
  - Obtaining Connector/Python
  - Installing Connector/Python from a Binary Distribution
  - Installing Connector/Python from a Source Distribution
  - Verifying Your Connector/Python Installation
- Connector/Python Coding Examples
- Connector/Python Tutorials
- Connector/Python Connection Establishment
- The Connector/Python C Extension
- Connector/Python Other Topics

4.2 Installing Connector/Python from a Binary Distribution

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

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

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

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

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



# Connect

The first thing to do in Python to access a MySQL database is to [establish a connection](#) to the server.

```
# import mysql connector
import mysql.connector

# create a new connection to the database
myConn = mysql.connector.connect(user='user',
                                password='password',
                                host='server')

# close the connection to the database
myConn.close()
```



# Connect

config.py

config.py

```
# store information
DATABASE = {'host': 'localhost',
            'user': 'user',
            'password': 'password'}
```

myCode.py

```
# import
import config
import mysql.connector

# create a new connection to the database
myConn = mysql.connector.connect(user=config.DATABASE['user'],
                                password=config.DATABASE['password'],
                                host=config.DATABASE['host'])
```



# Connect

## Errors

```
# import
import mysql.connector

# catch errors
try:
    myConn = mysql.connector.connect(user='user',
                                     password='password',
                                     host='server')
except mysql.connector.Error as err:
    if err.errno == mysql.connector.errorcode.ER_ACCESS_DENIED_ERROR:
        print("Something is wrong with your user name or password")
    else:
        print(err)
else:
    myConn.close()
```



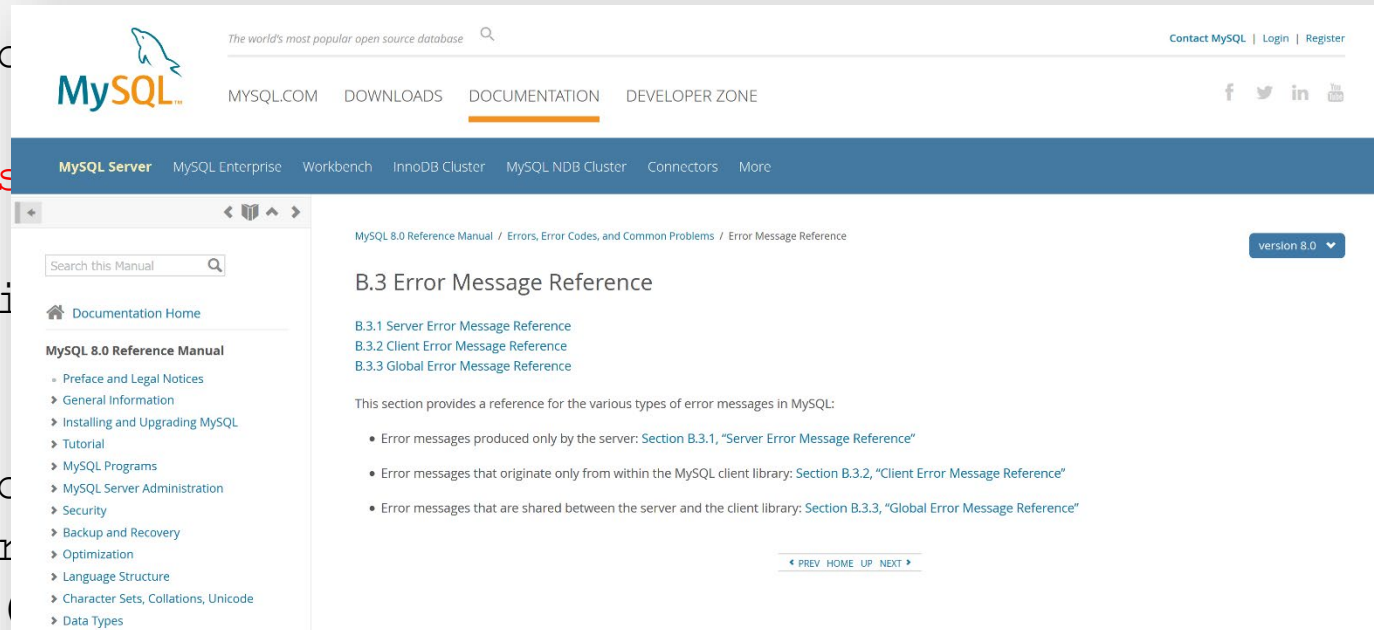


# Connect

## Errors

```
# import
import mysql.connector

# catch errors
try:
    myConnection = mysql.connector.connect(
        host='localhost',
        user='root',
        password='password'
    )
except mysql.connector.Error as err:
    if err.errno == 1044:
        print("Access denied for user")
    else:
        print(err)
else:
    myConnection.close()
```

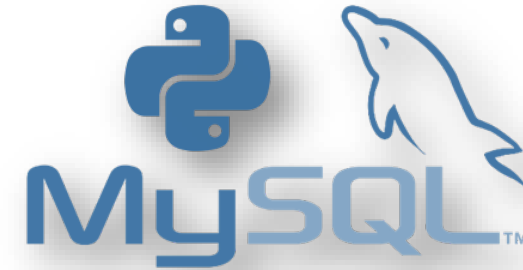


ERROR:  
ord")



# Content

- Introduction
- Install – Connect
- Create – Insert – Query





# Create

## Cursor

The *MySQL Connector/Python* provides the `cursor` object to pass statements to the server.

In a first step, the cursor object has to be created:

```
# connect
myConn = mysql.connector.connect (user='user',
                                   password='password',
                                   host='server')

# create cursor
myCursor = myConn.cursor()
```



# Create

## Database

Any SQL statement can be sent to the server by calling the `execute` method of the cursor object.

```
# create database  
myCursor.execute('CREATE DATABASE firstPythonDB')
```

no semicolon

Implementing your own method to create databases:

```
def create_database(cursor, dbname):  
    try:  
        cursor.execute('CREATE DATABASE {}'.format(dbname))  
    except mysql.connector.Error as err:  
        print('Failed creating database: {}'.format(err))  
        exit(1)
```



# Create

## Database

To tell the server which database to use, the database name should be given to the connector.

```
# use database firstPythonDB
myConn.database = 'firstPythonDB'
```

WARNING: If no database with the given name exists, an error is thrown.

```
try:
    myConn.database = 'firstPythonDB'
except mysql.connector.Error as err:
    if err.errno == mysql.connector.errorcode.ER_BAD_DB_ERROR:
        print('Database does not exist: {}'.format(err))
        exit(1)
```



# Create

## Tables

It's always a good idea to define longer strings outside of the code that uses them.

```
# define table
employeesTable = ("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) "
    ") ")
```



# Create

## Tables

Again the cursor object is used to send the CREATE TABLE statement to the server:

```
try:
    myCursor.execute(employeesTable)
except mysql.connector.Error as err:
    if err.errno == mysql.connector.errorcode.ER_TABLE_EXISTS_ERROR:
        print("Table already exists.")
    else:
        print(err.msg)
```



# Insert

## Define String

INSERT statements are executed on the server using the cursor object's execute method:

```
# define new record
addEmployee = ("INSERT INTO employees ("
               "first_name, last_name, "
               "hire_date, gender, birth_date) "
               "VALUES ("
               "'Tom', 'Smith', "
               "'1987-02-14', 'M', '1955-05-07') "
               ")")

# insert record into table
myCursor.execute(addEmployee)
```





# Insert

## Values

Values should be separated from the SQL string:

```
# define new record
addEmployee = ("INSERT INTO employees ("
               "first_name, last_name, "
               "hire_date, gender, birth_date) "
               "VALUES (%s, %s, %s, %s, %s)")

# set data
dataEmployee = ('Tom', 'Smith', '1987-02-14', 'M', '1955-05-07')

# insert record into table
myCursor.execute(addEmployee, dataEmployee)
```



# Insert

## Commit

SQL commands that **modify the database** in any way (INSERT, ALTER, UPDATE, DROP, etc.) do not have an immediate effect after running the execute function. To make these commands effective, the [commit method of the connector](#) has to be executed.:

```
# insert record into table
myCursor.execute(addEmployee, dataEmployee)

# commit changes
myConn.commit()
```



# Query

## fetchall

To retrieve data, a SELECT command has to be executed. The resulting table can be fetched in Python using the cursor's `fetchall` method. The `fetchall` method returns the data as a list of lists:

```
# get all employees
myCursor.execute("SELECT * FROM employees")
employeesList = myCursor.fetchall()

# get the first employee
firstEmployee = employeesList[0]

# get the first attribute of the first employee
firstAttr = firstEmployee[0]
```