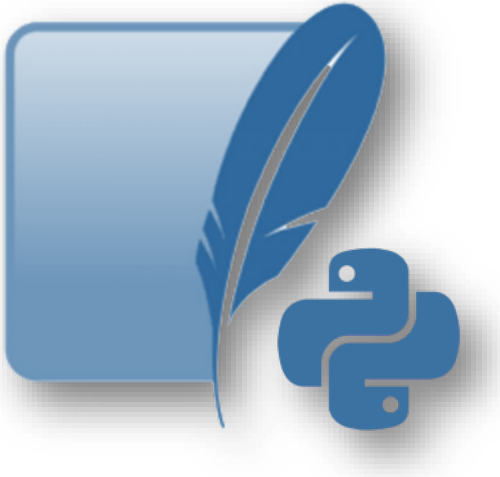
Python and SQLite



Content

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

Content

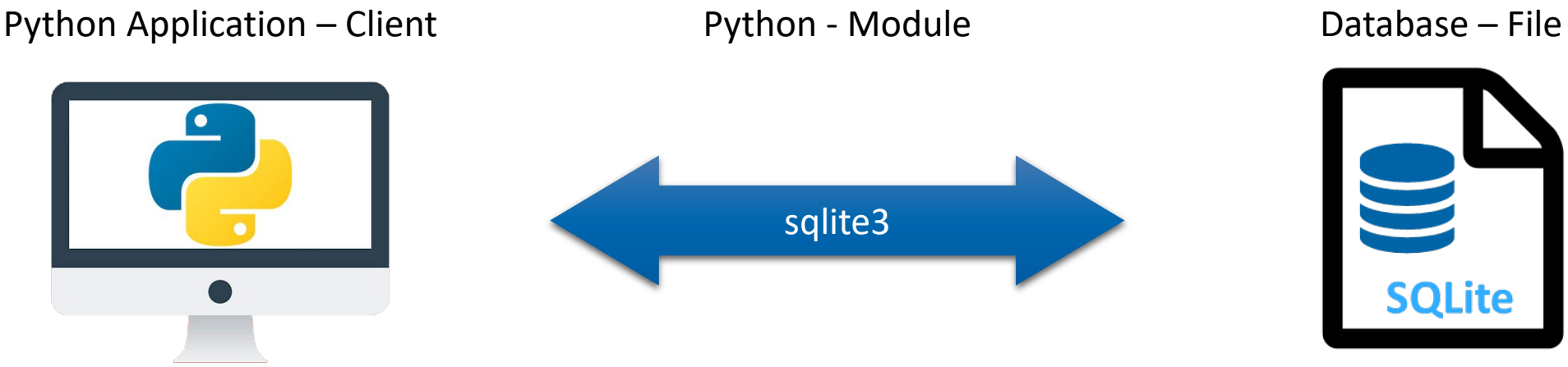
- Introduction
- Install – Connect
- Create – Insert – Query

Introduction

Connector

To be able to connect to a SQLite database in Python, the Python module sqlite3 has to be imported.

The module sqlite3 is already included in a standard Python 3 distribution.



Introduction

Guidelines – same as for MySQL

ACLS | DDAS

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.

Content

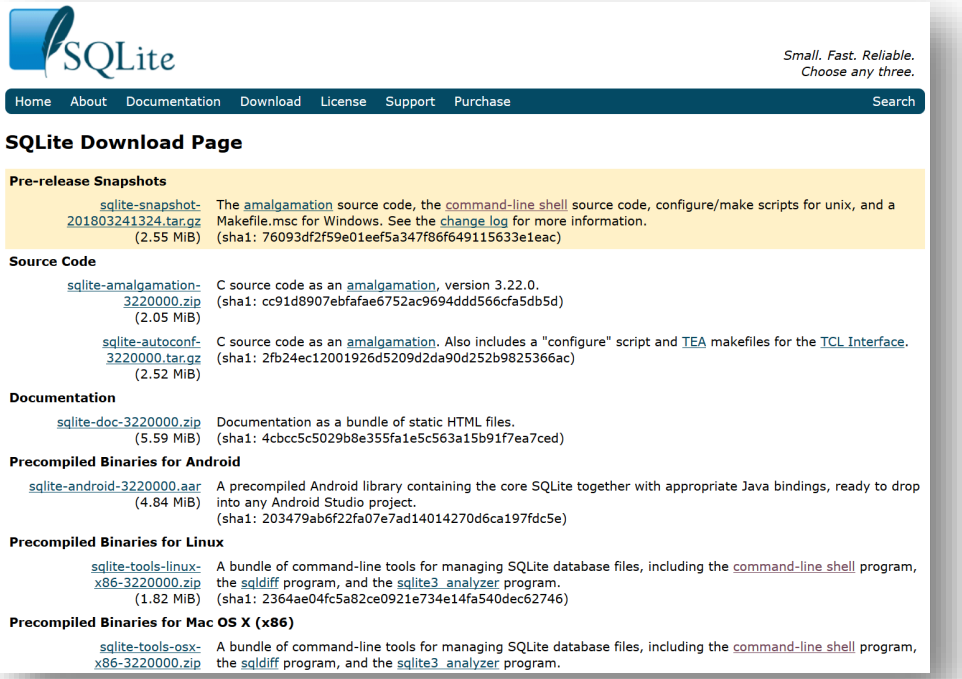
- Introduction
- Install – Connect
- Create – Insert – Query

Install

Two things have to be installed to work with Python and SQLite :

- Python 3
- SQLite

No additional connector/driver/interpreter/etc. is needed.



SQLite

Small. Fast. Reliable.
Choose any three.

Home | About | Documentation | Download | License | Support | Purchase | Search

SQLite Download Page

Pre-release Snapshots

sqlite-snapshot-201803241324.tar.gz (2.55 MiB)	The amalgamation source code, the command-line shell source code, configure/make scripts for unix, and a Makefile.msc for Windows. See the change log for more information. (sha1: 76093df2f59e01eef5a347f86f649115633e1eac)

Source Code

sqlite-amalgamation-3220000.zip (2.05 MiB)	C source code as an amalgamation, version 3.22.0. (sha1: cc91d8907ebfafae6752ac9694ddd566cfa5db5d)
sqlite-autoconf-3220000.tar.gz (2.52 MiB)	C source code as an amalgamation. Also includes a "configure" script and TEA makefiles for the TCL Interface. (sha1: 2fb24ec12001926d5209d2da90d252b9825366ac)

Documentation

sqlite-doc-3220000.zip (5.59 MiB)	Documentation as a bundle of static HTML files. (sha1: 4cbcc5c5029b8e355fa1e5c563a15b91f7ea7ced)

Precompiled Binaries for Android

sqlite-android-3220000.aar (4.84 MiB)	A precompiled Android library containing the core SQLite together with appropriate Java bindings, ready to drop into any Android Studio project. (sha1: 203479ab6f22fa07e7ad14014270d6ca197fdc5e)

Precompiled Binaries for Linux

sqlite-tools-linux-x86-3220000.zip (1.82 MiB)	A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff program, and the sqlite3_analyzer program. (sha1: 2364ae04fc5a82ce0921e734e14fa540dec62746)

Precompiled Binaries for Mac OS X (x86)

sqlite-tools-osx-x86-3220000.zip	A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff program, and the sqlite3_analyzer program.

Connect

Connect to the file

The first thing to do in Python to access a SQLite database is to establish a connection to the file.

```
# import SQLite module
import sqlite3

# create a new connection to the database
myConn = sqlite3.connect('filename.sqlite')
```

no authentication

If the database file does not exist, it will be created automatically (no error!).

Connect

Free the file

Since we are working in SQLite with a file, the file has to be freed in the end:

```
# import SQLite module
import sqlite3

# create a new connection to the database
myConn = sqlite3.connect('filename.sqlite')

# free the file
myConn.close()
```

Connect

Errors

The module sqlite3 does <u>not</u> provide specific error handling.

```
# import
import sqlite3

# catch errors
try:
    [ some sqlite code ]

# something went wrong
except sqlite3.Error as err:
    # do something
    [ not many options here ]
    print('An error occured')
    exit(1)
```

Content

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

Create

Cursor

The module sqlite3 provides the curser object to execute SQL statements.

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

```
# connect
myConn = sqlite3.connect('filename.sqlite')

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

Create

Database

SQLite stores a single database in a single file. Thus, a database is implicitly created by connecting to a database/file that does not exists.

```
# import SQLite module
import sqlite3

# create a new connection to the database
myConn = sqlite3.connect('newfilename.sqlite')

# free the file
myConn.close()
```

```
# import
import os.path

# database file
dbfile = 'firstSQLiteDB.sqlite'

# check if file exists
if os.path.isfile(dbfile):
    [ do something ]
```

Create

Tables

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

defining the primary key is done directly on the same line

```
# define table
employeesTable = ("CREATE TABLE employees ("
                        "emp_no integer NOT NULL PRIMARY KEY, "
                        "birth_date date NOT NULL, "
                        "first_name varchar(14) NOT NULL, "
                        "last_name varchar(16) NOT NULL, "
                        "gender varchar(1) NOT NULL, "
                        "hire_date date NOT NULL "
                        ")")
```

sqlite does not support the datatype *enum*

Create

Tables

The cursor object is used to execute the CREATE TABLE statement:

```
# define table
employeesTable = ("CREATE TABLE employees ("
                        "emp_no integer NOT NULL PRIMARY KEY, "
                        "birth_date date NOT NULL, "
                        "first_name varchar(14) NOT NULL, "
                        "last_name varchar(16) NOT NULL, "
                        "gender varchar(1) NOT NULL, "
                        "hire_date date NOT NULL "
                        ")")

# create employees table
myCursor.execute(employeesTable)
```

Insert

Define String

INSERT statements are executed 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 (?, ?, ?, ?, ?)")

# 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

Use the cursor

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]
```