

# 06- Sqlite3 MySQL Postgres and other DBs with Python

October 31, 2020

---

---

Coursework delivered by: Alison Mukoma

Copyright: Evelyn Hone College cc DevsBranch.

## 1 Sqlite3 Database with python

(working with the rest of the other databases will be demonstrated in a separate class and shared in a separate notebook) ## Before to Start: Importing the Libraries and Packages and Checking the Versions

A Quick Introduction to SQLite with Python

- **First**, connect to the database using the database library's `connect` method.
- **Second**, get a `cursor` which will let us execute SQL commands
- **Third**, We can now execute any SQL commands that we want in the database using the cursor's `execute` method. Querying the database simply involves writing the appropriate SQL and placing it inside a string in the `execute` method call.
- **Fourth**, if you saved the cursor in a variable then close it and then close the database connection as well

```
[5]: # import sqlite3
# or we can choose to import and give it a placeholder name that we will then
# use, it means the same
import sqlite3 as sql3

db = sql3.connect('data.db')

with db:
    c = db.cursor()
    c.execute('SELECT SQLITE_VERSION()')
    data = c.fetchone()
    print "SQLite version: %s" % data
```

SQLite version: 3.6.21

```
[6]: c = db.cursor()
```

```
[7]: c.execute('CREATE TABLE test (i INTEGER, j TEXT)')
```

```
[7]: <sqlite3.Cursor at 0x5c13960>
```

```
[8]: n = 5
     m = 'some text'

     c.execute('INSERT INTO test(i,j) VALUES (?,?)', (n,m))
```

```
[8]: <sqlite3.Cursor at 0x5c13960>
```

```
[9]: n = 100
     m = 'more text'

     c.execute('INSERT INTO test(i,j) VALUES (?,?)', (n,m))
```

```
[9]: <sqlite3.Cursor at 0x5c13960>
```

```
[10]: c.execute('SELECT * FROM test')
```

```
[10]: <sqlite3.Cursor at 0x5c13960>
```

```
[11]: results = c.fetchall()
     print results
```

```
[(5, u'some text'), (100, u'more text')]
```

```
[12]: for (i,j) in results:
     print i,j
```

```
5 some text
100 more text
```

```
[13]: c.execute('SELECT * FROM test WHERE i=5')
     print c.fetchall()
```

```
[(5, u'some text')]
```

```
[14]: c.execute('UPDATE test SET j=\'yet more test\' WHERE i=5')
     c.execute('SELECT * FROM test')
     print c.fetchall()
```

```
[(5, u'yet more test'), (100, u'more text')]
```

```
[15]: c.execute('DELETE FROM test WHERE i=5')
```

```
[15]: <sqlite3.Cursor at 0x5c13960>
```

```
[16]: c.execute('SELECT * FROM test')
      print c.fetchall()
```

```
[(100, u'more text')]
```

### 1.0.1 1. Inserting and Querying Data

```
[25]: # os.unlink('test.db')
      con = sql3.connect('test.db')

      with con:

          cur = con.cursor()
          cur.execute("DROP TABLE IF EXISTS Cars")
          cur.execute("CREATE TABLE Cars(Id INT, Name TEXT, Price INT)")
          cur.execute("INSERT INTO Cars VALUES(1,'Audi',52642)")
          cur.execute("INSERT INTO Cars VALUES(2,'Mercedes',57127)")
          cur.execute("INSERT INTO Cars VALUES(3,'Skoda',9000)")
          cur.execute("INSERT INTO Cars VALUES(4,'Volvo',29000)")
          cur.execute("INSERT INTO Cars VALUES(5,'Bentley',350000)")
          cur.execute("INSERT INTO Cars VALUES(6,'Citroen',21000)")
          cur.execute("INSERT INTO Cars VALUES(7,'Hummer',41400)")
          cur.execute("INSERT INTO Cars VALUES(8,'Volkswagen',21600)")
```

In Python, we can use the `fetchall()` method to fetch all the records in the table:

```
[30]: con = sql3.connect('test.db')
      cur = con.cursor()

      cur.execute('SELECT * FROM Cars')

      rows = cur.fetchall()
      for row in rows:
          print row

      # or, you can do also:
      # print cur.fetchall()
```

```
(1, u'Audi', 52642)
(2, u'Mercedes', 57127)
(3, u'Skoda', 9000)
(4, u'Volvo', 29000)
(5, u'Bentley', 350000)
(6, u'Citroen', 21000)
(7, u'Hummer', 41400)
(8, u'Volkswagen', 21600)
```

Or, alternatively, to get the results into Python we then use either the `fetchone()` method to fetch

one record at a time (it returns None when there are no more records to fetch so that you know when to stop)

```
[29]: con = sql3.connect('test.db')
      cur = con.cursor()

      cur.execute('SELECT * FROM Cars')
      record = cur.fetchone()

      while record:
          print record
          record = cur.fetchone()
```

```
(1, u'Audi', 52642)
(2, u'Mercedes', 57127)
(3, u'Skoda', 9000)
(4, u'Volvo', 29000)
(5, u'Bentley', 350000)
(6, u'Citroen', 21000)
(7, u'Hummer', 41400)
(8, u'Volkswagen', 21600)
```

Another possibility ...

```
[33]: con = sql3.connect('test.db')

      with con:

          cur = con.cursor()
          cur.execute("SELECT * FROM Cars")

          rows = cur.fetchall()

          for row in rows:
              print row[0], row[1], row[2]
```

```
1 Audi 52642
2 Mercedes 57127
3 Skoda 9000
4 Volvo 29000
5 Bentley 350000
6 Citroen 21000
7 Hummer 41400
8 Volkswagen 21600
```

A technically better version of the previous code to retrieve data is

```
[32]: con = sql3.connect('test.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM Cars")

    while True:

        row = cur.fetchone()

        if row == None:
            break

        print row[0], row[1], row[2]
```

```
1 Audi 52642
2 Mercedes 57127
3 Skoda 9000
4 Volvo 29000
5 Bentley 350000
6 Citroen 21000
7 Hummer 41400
8 Volkswagen 21600
```

We are going to create the same table. This time using the convenience `executemany()` method.

```
[23]: cars = (
    (1, 'Audi', 52642),
    (2, 'Mercedes', 57127),
    (3, 'Skoda', 9000),
    (4, 'Volvo', 29000),
    (5, 'Bentley', 350000),
    (6, 'Hummer', 41400),
    (7, 'Volkswagen', 21600)
)

con = sql3.connect('test.db')

with con:

    cur = con.cursor()
    # This script drops a Cars table if it exists and (re)creates it.
    cur.execute("DROP TABLE IF EXISTS Cars")
    cur.execute("CREATE TABLE Cars(Id INT, Name TEXT, Price INT)")
    # The first SQL statement drops the Cars table, if it exists.
    # The second SQL statement creates the Cars table.
    cur.executemany("INSERT INTO Cars VALUES(?, ?, ?)", cars)
```

Another way to create our Cars table: We commit the changes manually and provide our own **error handling**. In the script below we re-create the Cars table using the `executescript()` method

```
[24]: try:
    con = sql3.connect('test.db')

    cur = con.cursor()

    # The executescript() method allows us to execute the whole SQL code in one
    ↳ step.

    cur.executescript("""
        DROP TABLE IF EXISTS Cars;
        CREATE TABLE Cars(Id INT, Name TEXT, Price INT);
        INSERT INTO Cars VALUES(1,'Audi',52642);
        INSERT INTO Cars VALUES(2,'Mercedes',57127);
        INSERT INTO Cars VALUES(3,'Skoda',9000);
        INSERT INTO Cars VALUES(4,'Volvo',29000);
        INSERT INTO Cars VALUES(5,'Bentley',350000);
        INSERT INTO Cars VALUES(6,'Citroen',21000);
        INSERT INTO Cars VALUES(7,'Hummer',41400);
        INSERT INTO Cars VALUES(8,'Volkswagen',21600);
    """)

    con.commit()

except lite.Error, e:

    if con:
        con.rollback()

    print "Error %s:" % e.args[0]
    sys.exit(1)

finally:

    if con:
        con.close()
```

## ##2. Parameterized queries

When we use parameterized queries, we use placeholders instead of directly writing the values into the statements. Parameterized queries increase security and performance.

The Python **SQLite3** module supports two types of placeholders. Question marks and named placeholders.

```
[36]: uId = 1
      uPrice = 62300

      con = sql3.connect('test.db')

      with con:

          cur = con.cursor()

          cur.execute("UPDATE Cars SET Price=? WHERE Id=?", (uPrice, uId))
          con.commit()

          print "Number of rows updated: %d" % cur.rowcount
```

Number of rows updated: 1

The second example uses parameterized statements with named placeholders:

```
[35]: uId = 4

      con = sql3.connect('test.db')

      with con:

          cur = con.cursor()

          cur.execute("SELECT Name, Price FROM Cars WHERE Id=:Id",
                      {"Id": uId})
          con.commit()

          row = cur.fetchone()
          print row[0], row[1]
```

Volvo 29000

### ###3. Metadata

Metadata is information about the data in the database. Metadata in a SQLite contains information about the tables and columns, in which we store data. Number of rows affected by an SQL statement is a metadata. Number of rows and columns returned in a result set belong to metadata as well.

Metadata in SQLite can be obtained using the PRAGMA command. SQLite objects may have attributes, which are metadata. Finally, we can also obtain specific metadata from querying the SQLite system sqlite\_master table.

```
[38]: con = sql3.connect('test.db')

      with con:

          cur = con.cursor()
```

```

cur.execute('PRAGMA table_info(Cars)')

data = cur.fetchall()

for d in data:
    print d[0], d[1], d[2]

```

```

0 Id INT
1 Name TEXT
2 Price INT

```

Next we will print all rows from the Cars table with their column names.

```

[46]: con = sql3.connect('test.db')

with con:

    cur = con.cursor()
    cur.execute('SELECT * FROM Cars')

    col_names = [cn[0] for cn in cur.description]

    rows = cur.fetchall()

    print "%-5s %-15s %s" % (col_names[0], col_names[1], col_names[2])

    for row in rows:
        print "%-5s %-15s %s" % row

```

Id	Name	Price
1	Audi	62300
2	Mercedes	57127
3	Skoda	9000
4	Volvo	29000
5	Bentley	350000
6	Citroen	21000
7	Hummer	41400
8	Volkswagen	21600

Another example related to the metadata, we list all tables in the test.db database.

```

[49]: con = sql3.connect('test.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT name FROM sqlite_master WHERE type='table'")

```



```

rows = cur.fetchall()

for row in rows:
    print row[0]

```

Cars

##4. Export and Import of Data

We can dump data in an SQL format to create a simple backup of our database tables

```

[51]: cars = (
    (1, 'Audi', 52643),
    (2, 'Mercedes', 57642),
    (3, 'Skoda', 9000),
    (4, 'Volvo', 29000),
    (5, 'Bentley', 350000),
    (6, 'Hummer', 41400),
    (7, 'Volkswagen', 21600)
)

# The data from the table is being written to the file:

def writeData(data):

    f = open('cars.sql', 'w')

    with f:
        f.write(data)

# We create a temporary table in the memory:

con = sql3.connect(':memory:')

# These lines create a Cars table, insert values and delete rows,
# where the Price is less than 30000 units.

with con:

    cur = con.cursor()

    cur.execute("DROP TABLE IF EXISTS Cars")
    cur.execute("CREATE TABLE Cars(Id INT, Name TEXT, Price INT)")
    cur.executemany("INSERT INTO Cars VALUES(?, ?, ?)", cars)
    cur.execute("DELETE FROM Cars WHERE Price < 30000")

    # The con.iterdump() returns an iterator to dump the database
    # in an SQL text format. The built-in join() function takes

```

```
# the iterator and joins all the strings in the iterator separated  
# by a new line. This data is written to the cars.sql file in  
# the writeData() function.
```

```
data = '\n'.join(con.iterdump())
```

```
writeData(data)
```

```
[53]: print data
```

```
BEGIN TRANSACTION;  
CREATE TABLE Cars(Id INT, Name TEXT, Price INT);  
INSERT INTO "Cars" VALUES(1,'Audi',52643);  
INSERT INTO "Cars" VALUES(2,'Mercedes',57642);  
INSERT INTO "Cars" VALUES(5,'Bentley',350000);  
INSERT INTO "Cars" VALUES(6,'Hummer',41400);  
COMMIT;
```

Now we are going to perform a reverse operation. We will import the dumped table back into memory.

```
[55]: def readData():
```

```
    f = open('cars.sql', 'r')
```

```
    with f:
```

```
        data = f.read()
```

```
        return data
```

```
con = sql3.connect(':memory:')
```

```
with con:
```

```
    cur = con.cursor()
```

```
    sql_query = readData()
```

```
    cur.executescript(sql_query)
```

```
    cur.execute("SELECT * FROM Cars")
```

```
    rows = cur.fetchall()
```

```
    for row in rows:
```

```
        print row
```

```
(1, u'Audi', 52643)
```

```
(2, u'Mercedes', 57642)
```

```
(5, u'Bentley', 350000)
(6, u'Hummer', 41400)
```

### ##5. Transactions

A transaction is an atomic unit of database operations against the data in one or more databases. The effects of all the **SQL** statements in a transaction can be either all committed to the database or all rolled back.

In **SQLite**, any command other than the **SELECT** will start an implicit transaction. Also, within a transaction a command like **CREATE TABLE ...**, **VACUUM**, **PRAGMA**, will commit previous changes before executing.

Manual transactions are started with the **BEGIN TRANSACTION** statement and finished with the **COMMIT** or **ROLLBACK** statements.

**SQLite** supports three non-standard transaction levels. **DEFERRED**, **IMMEDIATE** and **EXCLUSIVE**. **SQLite** Python module also supports an autocommit mode, where all changes to the tables are immediately effective.

```
[66]: # We create a friends table and try to fill it with data. However, the data is
      ↪not committed...
      # because the commit() method is commented.
      # If we uncomment the line, the line will be written to the table:

      #import sqlite3 as sql

      try:
          con = sql3.connect('test.db')
          cur = con.cursor()
          cur.execute("DROP TABLE IF EXISTS Friends")
          cur.execute("CREATE TABLE Friends(Id INTEGER PRIMARY KEY, Name TEXT)")
          cur.execute("INSERT INTO Friends(Name) VALUES ('Tom')")
          cur.execute("INSERT INTO Friends(Name) VALUES ('Rebecca')")
          cur.execute("INSERT INTO Friends(Name) VALUES ('Jim')")
          cur.execute("INSERT INTO Friends(Name) VALUES ('Robert')")

          #---> con.commit()

      except sql3.error, e:

          if con:
              con.rollback()

          print "Error %s:" % e.args[0]
          sys.exit(1)

      finally:

          if con:
```

```
con.close()
```

#### ##Code Example 04: A Data Base of Movies

The `pandas.io.sql` module provides a collection of query wrappers to both facilitate data retrieval and to reduce dependency on DB-specific API. These wrappers only support the Python database adapters which respect the Python DB-API.

Let us use, in the following example, a list of the 10.000 movies made since 1950 with the most IMDB user ratings. Download the data at [http://bit.ly/cs109\\_imdb](http://bit.ly/cs109_imdb) and save it as text file in your working directory.

```
[112]: names = ['imdbID', 'title', 'year', 'score', 'votes', 'runtime', 'genres']
movies = pd.read_csv('imdb_top_10000.txt', delimiter='\t', names = names).
        ↳dropna()

print movies.head()

clean_runtime = [float(r.split(' ')[0]) for r in movies.runtime]
movies['runtime'] = clean_runtime

# determine the unique genres

genres = set()

for m in movies.genres:
    genres.update(g for g in m.split('|'))

genres = sorted(genres)

# make a column for each genre

for genre in genres:
    movies[genre] = [genre in movie.split('|') for movie in movies.genres]

movies['title'] = [t[0:-7] for t in movies.title]

print movies.head() # print the first 10 rows
```

	imdbID	title	year	score	votes	runtime	\
0	tt0111161	The Shawshank Redemption (1994)	1994	9.2	619479	142 mins.	
1	tt0110912	Pulp Fiction (1994)	1994	9.0	490065	154 mins.	
2	tt0137523	Fight Club (1999)	1999	8.8	458173	139 mins.	
3	tt0133093	The Matrix (1999)	1999	8.7	448114	136 mins.	
4	tt1375666	Inception (2010)	2010	8.9	385149	148 mins.	

  

	genres
0	Crime Drama
1	Crime Thriller

```

2          Drama|Mystery|Thriller
3          Action|Adventure|Sci-Fi
4 Action|Adventure|Sci-Fi|Thriller
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5 entries, 0 to 4
Data columns (total 31 columns):
imdbID      5  non-null values
title       5  non-null values
year        5  non-null values
score       5  non-null values
votes       5  non-null values
runtime     5  non-null values
genres      5  non-null values
Action      5  non-null values
Adult       5  non-null values
Adventure   5  non-null values
Animation   5  non-null values
Biography   5  non-null values
Comedy      5  non-null values
Crime       5  non-null values
Drama       5  non-null values
Family      5  non-null values
Fantasy     5  non-null values
Film-Noir   5  non-null values
History     5  non-null values
Horror      5  non-null values
Music       5  non-null values
Musical     5  non-null values
Mystery     5  non-null values
News        5  non-null values
Reality-TV  5  non-null values
Romance     5  non-null values
Sci-Fi      5  non-null values
Sport       5  non-null values
Thriller    5  non-null values
War         5  non-null values
Western     5  non-null values
dtypes: bool(24), float64(2), int64(2), object(3)

```

```

[98]: cnx = sql3.connect('movies.db')

with con:

    cur = con.cursor()

    cur.execute('PRAGMA table_info(Movies)')

```

```
data = cur.fetchall()

for d in data:
    print d[0], d[1], d[2]
```