SQLite Python tutorial

The tutorial was superseded with the Python SQLite tutorial.

This is a Python programming tutorial for the SQLite database. It covers the basics of SQLite programming with the Python language. You might also want to check the Python tutorial, SQLite tutorial or MySQL Python tutorial or PostgreSQL Python tutorial on ZetCode.

To work with this tutorial, we must have Python language, SQLite database, pysqlite language binding and the sqlite command line tool installed on the system.

If we have Python 2.5+ then we only need to install the sqlite3 command line tool. Both the SQLite library and the pysqlite language binding are built into the Python languge.

```
$ python2
Python 2.7.12 (default, Nov 12 2018, 14:36:49)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.

>>> import sqlite3
>>> sqlite3.version
'2.6.0'
>>> sqlite3.sqlite_version
'3.16.2'
```

In the shell, we launch the Python interactive interpreter. We can see the Python version. In our case it is Python 2.7.12. The sqlite.version is the version of the pysqlite (2.6.0), which is the binding of the Python language to the SQLite database. The sqlite3.sqlite_version gives us the version of the SQLite database library. In our case the version is 3.16.2.

SQLite

SQLite is an embedded relational database engine. The documentation calls it a self-contained, serverless, zero-configuration and transactional SQL database engine. It is very popular and there are hundreds of millions copies worldwide in use today. Several programming languages have built-in support for SQLite including Python and PH

Creating SQLite database

Now we are going to use the sqlite3 command line tool to create a new database.

```
$ sqlite3 test.db
SQLite version 3.16.2 2017-01-06 16:32:41
Enter ".help" for usage hints.
sqlite>
```

We provide a parameter to the sqlite3 tool; test.db is a database name. It is a file on our disk. If it is present, it is opened. If not, it is created.

```
sqlite> .tables
sqlite> .exit
$ ls
test.db
```

The .tables command gives a list of tables in the test.db database. There are currently no tables. The .exit command terminates the interactive session of the sqlite3 command line tool. The ls Unix command shows the contents of the current working directory. We can see the test.db file. All data will be stored in this single file.

SQLite version example

In the first code example, we will get the version of the SQLite database.

version.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys

con = None

try:
    con = lite.connect('test.db')

    cur = con.cursor()
    cur.execute('SELECT SQLITE_VERSION()')

    data = cur.fetchone()[0]

    print "SQLite version: {}".format(data)

except lite.Error, e:

    print "Error {}:".format(e.args[0])
    sys.exit(1)

finally:
```

```
    if con:
        con.close()
```

In the above Python script we connect to the previously created test.db database. We execute an SQL statement whic returns the version of the SQLite database.

```
import sqlite3 as lite
```

The sqlite3 module is used to work with the SQLite database.

```
con = None
```

We initialise the con variable to None. In case we could not create a connection to the database (for example the dis is full), we would not have a connection variable defined. This would lead to an error in the finally clause.

```
con = lite.connect('test.db')
```

Here we connect to the test.db database. The connect() method returns a connection object.

```
cur = con.cursor()
cur.execute('SELECT SQLITE_VERSION()')
```

From the connection, we get the cursor object. The cursor is used to traverse the records from the result set. We cal the execute() method of the cursor and execute the SQL statement.

```
data = cur.fetchone()[0]
```

We fetch the data. Since we retrieve only one record, we call the fetchone() method.

```
print "SQLite version: {}".format(data)
```

We print the data that we have retrieved to the console.

```
except lite.Error, e:

    print "Error {}:".format(e.args[0])
    sys.exit(1)
```

In case of an exception, we print an error message and exit the script with an error code 1.

```
finally:

    if con:
        con.close()
```

In the final step, we release the resources.

In the second example, we again get the version of the SQLite database. This time we will use the with keyword.

version2.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite

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

with con:

    cur = con.cursor()
    cur.execute('SELECT SQLITE_VERSION()')

    data = cur.fetchone()[0]

    print "SQLite version: {}".format(data)
```

The script returns the current version of the SQLite database. With the use of the with keyword. The code is more compact.

```
with con:
```

With the with keyword, the Python interpreter automatically releases the resources. It also provides error handling.

```
$ ./version2.py
SQLite version: 3.16.2
```

This is the output.

SQLite Python create table

We create a cars table and insert several rows to it.

create_table.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite

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

with con:

    cur = con.cursor()

    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)")
```

The above script creates a cars table and inserts 8 rows into the table.

```
cur.execute("CREATE TABLE cars(id INT, name TEXT, price INT)")
```

This SQL statement creates a new cars table. The table has three columns.

```
cur.execute("INSERT INTO cars VALUES(1,'Audi',52642)")
cur.execute("INSERT INTO cars VALUES(2,'Mercedes',57127)")
```

These two lines insert two cars into the table. Using the with keyword, the changes are automatically committed. Otherwise, we would have to commit them manually.

```
sqlite> .mode column
sqlite> .headers on
```

We verify the written data with the sqlite3 tool. First we modify the way the data is displayed in the console. We use the column mode and turn on the headers.

```
sqlite>  select * from cars;
id          name        price
----------  ----------  ----------
1           Audi        52642
2           Mercedes    57127
3           Skoda       9000
4           Volvo       29000
5           Bentley     350000
6           Citroen     21000
7           Hummer      41400
8           Volkswagen  21600
```

This is the data that we have written to the cars table.

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

create_table2.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite

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

con = lite.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.executemany("INSERT INTO cars VALUES(?, ?, ?)", cars)
```

The program drops the cars table if it exists and recreates 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)
```

We insert 8 rows into the table using the convenience executemany() method. The first parameter of this method is a parameterized SQL statement. The second parameter is the data, in the form of tuple of tuples.

We provide another way to create our cars table. We commit the changes manually and provide our own error handling.

create_table3.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys

try:
    con = lite.connect('test.db')

    cur = con.cursor()

    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 {}:".format(e.args[0])
    sys.exit(1)

finally:

    if con:
        con.close()
```

In the above script we (re)create the cars table using the executescript() method.

```
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);
...
```

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

```
con.commit()
```

Without the with keyword, the changes must be committed using the commit() method.

```
except lite.Error, e:

    if con:
        con.rollback()

    print "Error {}:".format(e.args[0])
    sys.exit(1)
```

In case of an error, the changes are rolled back and an error message is printed to the terminal.

SQLite Python lastrowid

Sometimes, we need to determine the id of the last inserted row. In Python SQLite, we use the lastrowid attribute of the cursor object.

lastrowid.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite

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

with con:

    cur = con.cursor()
    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');")

    last_row_id = cur.lastrowid
    print "The last Id of the inserted row is {}".format(last_row_id)
```

We create a friends table in memory. The Id is automatically incremented.

```
cur.execute("CREATE TABLE friends(id INTEGER PRIMARY KEY, name TEXT);")
```

In SQLite, INTEGER PRIMARY KEY column is auto incremented. There is also an AUTOINCREMENT keyword. When used in INTEGER PRIMARY KEY AUTOINCREMENT a slightly different algorithm for Id creation is used.

```
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');")
```

When using auto-increment, we have to explicitly state the column names, omitting the one that is auto-incremented. The four statements insert four rows into the friends table.

```
last_row_id = cur.lastrowid
```

Using the lastrowid we get the last inserted row id.

```
$ ./lastrowid.py
The last Id of the inserted row is 4
```

We see the output of the program.

SQLite Python retrieve data

Now that we have inserted some data into the database, we want to fetch it back.

select_all.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite

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

with con:

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

    rows = cur.fetchall()

    for row in rows:
        print row
```

In this example, we retrieve all data from the cars table.

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

This SQL statement selects all data from the cars table.

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

The fetchall() method gets all records. It returns a result set. Technically, it is a tuple of tuples. Each of the inner tuples represent a row in the table.

```
for row in rows:
    print row
```

We print the data to the console, row by row.

```
$ ./select_all.py
(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)
```

This is the output of the example.

Returning all data at a time may not be feasible. We can fetch rows one by one.

fetch_one.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite

con = lite.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]
```

In this script we connect to the database and fetch the rows of the cars table one by one.

```
while True:
```

We access the data from the while loop. When we read the last row, the loop is terminated.

```
row = cur.fetchone()

if row == None:
    break
```

The fetchone() method returns the next row from the table. If there is no more data left, it returns None. In this case we break the loop.

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

The data is returned in the form of a tuple. Here we select records from the tuple. The first is the Id, the second is th car name and the third is the price of the car.

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

This is the output of the example.

SQLite Python dictionary cursor

The default cursor returns the data in a tuple of tuples. When we use a dictionary cursor, the data is sent in the forn of Python dictionaries. This way we can refer to the data by their column names.

dictionary_cursor.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite

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

with con:

    con.row_factory = lite.Row

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

    rows = cur.fetchall()

    for row in rows:
        print "{} {} {}".format(row["id"], row["name"], row["price"])
```

In this example, we print the contents of the cars table using the dictionary cursor.

```
con.row_factory = lite.Row
```

We select a dictionary cursor. Now we can access records by the names of columns.

```
for row in rows:
    print "{} {} {}".format(row["id"], row["name"], row["price"])
```

The data is accessed by the column names.

SQLite Python parameterized queries

Now we will concern ourselves with parameterized queries. When we use parameterized queries, we use placeholde 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.

parameterized_query.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite

uId = 1
uPrice = 62300

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

with con:

    cur = con.cursor()

    cur.execute("UPDATE cars SET price=? WHERE id=?", (uPrice, uId))

    print "Number of rows updated: {}".format(cur.rowcount)
```

We update a price of one car. In this code example, we use the question mark placeholders.

```
cur.execute("UPDATE cars SET price=? WHERE id=?", (uPrice, uId))
```

The question marks ? are placeholders for values. The values are added to the placeholders.

```
print "Number of rows updated: {}".format(cur.rowcount)
```

The rowcount property returns the number of updated rows. In our case one row was updated.

The second example uses parameterized statements with named placeholders.

parameterized_query2.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite

uId = 4

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

with con:

    cur = con.cursor()

    cur.execute("SELECT name, price FROM cars WHERE Id=:Id", {"Id": uId})

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

We select a name and a price of a car using named placeholders.

```
cur.execute("SELECT name, price FROM cars WHERE Id=:Id", {"Id": uId})
```

The named placeholders start with a colon character.

SQLite Python insert image

In this section, we are going to insert an image to the SQLite database. Note that some people argue against putting images into databases. Here we only show how to do it. We do not dwell into technical issues of whether to save images in databases or not.

```
sqlite> CREATE TABLE images(id INTEGER PRIMARY KEY, data BLOB);
```

For this example, we create a new table called Images. For the images, we use the BLOB data type, which stands for Binary Large Objects.

insert_image.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys

def readImage():

    fin = None

    try:
        fin = open("sid.png", "rb")
        img = fin.read()
        return img

    except IOError, e:

        print e
        sys.exit(1)

    finally:

        if fin:
```

```
            fin.close()


try:
    con = lite.connect('test.db')

    cur = con.cursor()

    data = readImage()
    binary = lite.Binary(data)
    cur.execute("INSERT INTO images(data) VALUES (?)", (binary,) )

    con.commit()

except lite.Error, e:

    if con:
        con.rollback()

    print e
    sys.exit(1)

finally:

    if con:
        con.close()
```

In this script, we read an image from the current working directory and write it into the images table of the SQLite test.db database.

```
try:
    fin = open("sid.png", "rb")
    img = fin.read()
    return img
```

We read binary data from the filesystem. We have a JPG image called sid.png.

```
binary = lite.Binary(data)
```

The data is encoded using the SQLite Binary object.

```
cur.execute("INSERT INTO images(data) VALUES (?)", (binary,) )
```

This SQL statement is used to insert the image into the database.

SQLite Python read image

In this section, we are going to perform the reverse operation: we read an image from the database table.

read_image.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys
```

```
def writeImage(data):

    fout = None

    try:
        fout = open('sid2.png','wb')
        fout.write(data)

    except IOError, e:

        print e
        sys.exit(1)

    finally:

        if fout:
            fout.close()


try:
    con = lite.connect('test.db')

    cur = con.cursor()
    cur.execute("SELECT data FROM images LIMIT 1")
    data = cur.fetchone()[0]

    writeImage(data)


except lite.Error, e:

    print e
    sys.exit(1)

finally:

    if con:
        con.close()
```

We read image data from the Images table and write it to another file, which we call woman2.jpg.

```
try:
    fout = open('sid2.png','wb')
    fout.write(data)
```

We open a binary file in a writing mode. The data from the database is written to the file.

```
cur.execute("SELECT data FROM images LIMIT 1")
data = cur.fetchone()[0]
```

These two lines select and fetch data from the images table. We obtain the binary data from the first row.

SQLite Python 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 metatada from querying the SQLite system sqlite_master table.

column_names.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite

con = lite.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]
```

In this example, we issue the PRAGMA table_info(tableName) command, to get some metadata info about our cars table.

```
cur.execute('PRAGMA table_info(cars)')
```

The PRAGMA table_info(tableName) command returns one row for each column in the cars table. Columns in the result set include the column order number, column name, data type, whether or not the column can be NULL, and the default value for the column.

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

From the provided information, we print the column order number, column name and column data type.

```
$ ./column_names.py
0 id INT
1 name TEXT
2 price INT
```

Output of the example.

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

column_names2.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
```

```
import sqlite3 as lite

con = lite.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 "{:3} {:10} {:7}".format(col_names[0], col_names[1], col_names[2])

    for row in rows:
        print "{:<3} {:<10} {:7}".format(row[0], row[1], row[2])
```

We print the contents of the cars table to the console. Now, we include the names of the columns too. The records ar aligned with the column names.

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

We get the column names from the description property of the cursor object.

```
print "{:3} {:10} {:7}".format(col_names[0], col_names[1], col_names[2])
```

This line prints three column names of the cars table.

```
for row in rows:
    print "{:<3} {:<10} {:7}".format(row[0], row[1], row[2])
```

We print the rows using the for loop. The data is aligned with the column names.

```
$  ./column_names2.py
id  name       price
1   Audi         62300
2   Mercedes     57127
3   Skoda         9000
4   Volvo        29000
5   Bentley     350000
6   Hummer       41400
7   Volkswagen   21600
```

This is the output.

In our last example related to the metadata, we will list all tables in the test.db database.

list_tables.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite

con = lite.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]
```

The code example prints all available tables in the current database to the terminal.

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

The table names are stored inside the system sqlite_master table.

```
$ ./list_tables.py
cars
images
```

These were the tables on our system.

SQLite Python data export & import

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

export_table.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite

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

def writeData(data):

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

    with f:
        f.write(data)


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

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")

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

    writeData(data)
```

In the above example, we recreate the cars table in the memory. We delete some rows from the table and dump the current state of the table into a cars.sql file. This file can serve as a current backup of the table.

```
def writeData(data):

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

    with f:
        f.write(data)
```

The data from the table is being written to the file.

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

We create a temporary table in the memory.

```
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")
```

These lines create a cars table, insert values and delete rows, where the price is less than 30000 units.

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

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.

```
$ cat cars.sql
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;
```

The contents of the dumped in-memory cars table.

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

import_table.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite
```

```
def readData():

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

    with f:

        data = f.read()

        return data


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

with con:

    cur = con.cursor()

    sql = readData()
    cur.executescript(sql)

    cur.execute("SELECT * FROM cars")

    rows = cur.fetchall()

    for row in rows:
        print row
```

In this script, we read the contents of the cars.sql file and execute it. This will recreate the dumped table.

```
def readData():

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

    with f:

        data = f.read()

        return data
```

Inside the readData() method we read the contents of the cars.sql table.

```
cur.executescript(sql)
```

We call the executescript() method to launch the SQL script.

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

rows = cur.fetchall()

for row in rows:
    print row
```

We verify the data.

```
$ ./import_table.py
(1, u'Audi', 52643)
```

```
(2, u'Mercedes', 57642)
(5, u'Bentley', 350000)
(6, u'Hummer', 41400)
```

The output shows that we have successfully recreated the saved cars table.

Python SQLite transactions

A transaction is an atomic unit of database operations against the data in one or more databases. The effects of all t 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.

no_commit.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys


try:
    con = lite.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 lite.Error, e:

    if con:
        con.rollback()

    print e
    sys.exit(1)

finally:

    if con:
        con.close()
```

We create a friends table and try to fill it with data. However, as we will see, the data is not committed.