Introduction to SQLite in Python

This article is part 1 of 2 in the series Python SQLite Tutorial

SQLite3 is a very easy to use database engine. It is self-contained, serverless, zero-configuration and transactional. It is very fast and lightweight, and the entire database is stored in a single disk file. It is used in a lot of applications as internal data storage. The Python Standard Library includes a module called "sqlite3" intended for working with this database. This module is a SQL interface compliant with the DB-API 2.0 specification.

Using Python's SQLite Module

To use the SQLite3 module we need to add an import statement to our python script:

```
1 import sqlite3
```

Connecting SQLite to the Database

We use the function `sqlite3.connect` to connect to the database. We can use the argument ":memory:" to create a temporary DB in the RAM or pass the name of a file to open or create it.

```
1 # Create a database in RAM
2 db = sqlite3.connect(':memory:')
3 # Creates or opens a file called mydb with a SQLite3 DB
4 db = sqlite3.connect('data/mydb')
```

When we are done working with the DB we need to close the connection:

```
1 db.close()
```

Creating (CREATE) and Deleting (DROP) Tables

In order to make any operation with the database we need to get a cursor object and pass the SQL statements to the cursor object to execute them. Finally it is necessary to commit the changes. We are going to create a users table with name, phone, email and password columns.

```
```

```
# Get a cursor object
cursor = db.cursor()
cursor.execute('''
    CREATE TABLE users(id INTEGER PRIMARY KEY, name TEXT,
                       phone TEXT, email TEXT unique, password TEXT)
''')
db.commit()
```

To drop a table:

```
# Get a cursor object
cursor = db.cursor()
cursor.execute('''DROP TABLE users''')
db.commit()
```

Please note that the commit function is invoked on the db object, not the cursor object. If we type `cursor.commit` we will get `AttributeError: 'sqlite3.Cursor' object has no attribute 'commit'`

Inserting (INSERT) Data into the Database

To insert data we use the cursor to execute the query. If you need values from Python variables it is recommended to use the "?" placeholder. Never use string operations or concatenation to make your queries because is very insecure. In this example we are going to insert two users in the database, their information is stored in python variables.

```
cursor = db.cursor()
name1 = 'Andres'
phone1 = '3366858'
email1 = 'user@example.com'
# A very secure password
password1 = '12345'

name2 = 'John'
phone2 = '5557241'
email2 = 'johndoe@example.com'
password2 = 'abcdef'

# Insert user 1
cursor.execute('''INSERT INTO users(name, phone, email, password)
                  VALUES(?,?,?,?)''', (name1,phone1, email1, password1))
print('First user inserted')

# Insert user 2
cursor.execute('''INSERT INTO users(name, phone, email, password)
                  VALUES(?,?,?,?)''', (name2,phone2, email2, password2))
print('Second user inserted')

db.commit()
```

The values of the Python variables are passed inside a tuple. Another way to do this is passing a dictionary using the ":keyname" placeholder:

```
cursor.execute('''INSERT INTO users(name, phone, email, password)
                  VALUES(:name,:phone, :email, :password)''',
                  {'name':name1, 'phone':phone1, 'email':email1, 'password':password1})
```

If you need to insert several users use `executemany` and a list with the tuples:

```
users = [(name1,phone1, email1, password1),
         (name2,phone2, email2, password2),
         (name3,phone3, email3, password3)]
cursor.executemany(''' INSERT INTO users(name, phone, email, password) VALUES(?,?,?,?)''', users)
db.commit()
```

If you need to get the id of the row you just inserted use `lastrowid`:

```
id = cursor.lastrowid
print('Last row id: %d' % id)
```

Retrieving Data (SELECT) with SQLite

To retrieve data, execute the query against the cursor object and then use `fetchone()` to retrieve a single row or `fetchall()` to retrieve all the rows.

```
cursor.execute('''SELECT name, email, phone FROM users''')
user1 = cursor.fetchone() #retrieve the first row
print(user1[0]) #Print the first column retrieved(user's name)
all_rows = cursor.fetchall()
for row in all_rows:
    # row[0] returns the first column in the query (name), row[1] returns email column.
    print('{0} : {1}, {2}'.format(row[0], row[1], row[2]))
```

The cursor object works as an iterator, invoking `fetchall()` automatically:

```
cursor.execute('''SELECT name, email, phone FROM users''')
for row in cursor:
    # row[0] returns the first column in the query (name), row[1] returns email column.
    print('{0} : {1}, {2}'.format(row[0], row[1], row[2]))
```

To retrive data with conditions, use again the "?" placeholder:

```
user_id = 3
cursor.execute('''SELECT name, email, phone FROM users WHERE id=?''', (use
r_id,))
user = cursor.fetchone()
```

Updating (UPDATE) and Deleting (DELETE) Data

The procedure to update or delete data is the same as inserting data:

```
# Update user with id 1
newphone = '3113093164'
userid = 1
cursor.execute('''UPDATE users SET phone = ? WHERE id = ? ''',
 (newphone, userid))

# Delete user with id 2
delete_userid = 2
cursor.execute('''DELETE FROM users WHERE id = ? ''', (delete_userid,))

db.commit()
```

Using SQLite Transactions

Transactions are an useful property of database systems. It ensures the atomicity of the Database. Use `commit` to save the changes:

```
cursor.execute('''UPDATE users SET phone = ? WHERE id = ? ''',
 (newphone, userid))
db.commit() #Commit the change
```

Or `rollback` to roll back any change to the database since the last call to `commit`:

```
cursor.execute('''UPDATE users SET phone = ? WHERE id = ? ''',
(newphone, userid))
# The user's phone is not updated
db.rollback()
```

Please remember to always call `commit` to save the changes. If you close the connection using `close` or the connection to the file is lost (maybe the program finishes unexpectedly), not committed changes will be lost.

SQLite Database Exceptions

For best practices always surround the database operations with a try clause or a context manager:

```
import sqlite3 #Import the SQLite3 module
try:
    # Creates or opens a file called mydb with a SQLite3 DB
    db = sqlite3.connect('data/mydb')
    # Get a cursor object
    cursor = db.cursor()
    # Check if table users does not exist and create it
    cursor.execute('''CREATE TABLE IF NOT EXISTS
                      users(id INTEGER PRIMARY KEY, name TEXT, phone TEX
T, email TEXT unique, password TEXT)''')
    # Commit the change
    db.commit()
# Catch the exception
except Exception as e:
    # Roll back any change if something goes wrong
    db.rollback()
    raise e
finally:
    # Close the db connection
    db.close()
```

In this example we used a try/except/finally clause to catch any exception in the code. The `finally` keyword is very important because it always closes the database connection correctly. Please refer to this article to find more about exceptions. Please take a look to:

```
# Catch the exception
except Exception as e:
    raise e
```

This is called a catch-all clause, This is used here only as an example, in a real application you should catch a specific exception such as `IntegrityError` or `DatabaseError`, for more information please refer to DB-API 2.0 Exceptions.

We can use the Connection object as context manager to automatically commit or rollback transactions:

```
name1 = 'Andres'
phone1 = '3366858'
email1 = 'user@example.com'
# A very secure password
password1 = '12345'

try:
    with db:
        db.execute('''INSERT INTO users(name, phone, email, password)
                   VALUES(?,?,?,?)''', (name1,phone1, email1, password1))
except sqlite3.IntegrityError:
    print('Record already exists')
finally:
    db.close()
```

In the example above if the insert statement raises an exception, the transaction will be rolled back and the message gets printed; otherwise the transaction will be committed. Please note that we call `execute` on the `db` object, not the `cursor` object.

SQLite Row Factory and Data Types

The following table shows the relation between SQLite datatypes and Python datatypes:

- `None` type is converted to `NULL`
- `int` type is converted to `INTEGER`
- `float` type is converted to `REAL`
- `str` type is converted to `TEXT`
- `bytes` type is converted to `BLOB`

The row factory class `sqlite3.Row` is used to access the columns of a query by name instead of by index:

```
db = sqlite3.connect('data/mydb')
db.row_factory = sqlite3.Row
cursor = db.cursor()
cursor.execute('''SELECT name, email, phone FROM users''')
for row in cursor:
    # row['name'] returns the name column in the query, row['email'] retur
ns email column.
    print('{0} : {1}, {2}'.format(row['name'], row['email'], row['phon
e']))
db.close()
```