Sqlite

3 articles

http://apprendre-python.com/page-database-data-base-donnees-query-sql-mysql-postgre-sqlite

http://pythoncentral.io/introduction-to-sqlite-in-python/

http://pythoncentral.io/advanced-sqlite-usage-in-python/

http://apprendre-python.com/page-database-data-base-donnees-query-sql-mysql-postgre-sqlite

Base de données et Python

Accueil › Python avancé › Base de données

Réseau / socketCréer un serveur web

Vous pouvez stocker des informations dans des fichiers XML mais très rapidement, suivant l'ampleur de votre projet, cette technologie atteindra ses limites que ce soit en terme de performances ou d'utilisation au quotidien. Si plusieurs personnes souhaitent modifier un fichier texte de données en même temps les erreurs seront inévitables. Il existe une solution pour stocker des informations et pouvoir travailler avec : **les bases de données**.

Une base de données c'est quoi?

Une base de données (*database* en anglais) est un conteneur dans lequel il est possible de stocker des données de façon structurée. Cette structure permet au programme informatique connectée à celle-ci de **faire des recherches complexes**.

Un langage standardisé -SQL- est dédié à cette structure et permet aussi bien de faire des recherches mais aussi des modifications ou des suppressions.

Les logiciels de gestion de bases de données les plus utilisées aujourd'hui sont des **SGBDR** -*Système de gestion de base de données relationnelles*-, c'est à dire que les données sont liées

les unes aux autres, par exemple on peut définir que si on supprime une information, d'autres informations dépendantes de cette dernière soient elles-aussi automatiquement supprimées. Cela garantit une **cohérence de données**.

Il ne faut donc pas confondre une **base de données** qui est un conteneur et le **SGBDR** qui est un logiciel de gestion de bases de données.

Quels sont les SGBDR les plus connus?

MySQL	Mysql est l'un des SGBDR les plus utilisés au monde. Il est gratuit et très puissant. Il possède la **double licence GPL et propriétaire** depuis son rachat par *Sun Microsystem* eux-mêmes racheté par *Oracle* (concurrent direct de MySQL). Le logiciel reste cependant entièrement gratuit et libre. Il répond à une **logique client/serveur**, c'est à dire que plusieurs clients (ordinateurs distants) peuvent se connecter sur un seul serveur qui héberge les données.
PostgreSQL	PostgreSQL ressemble à MySQL, moins connu mais possède des fonctionnalités en plus.
SQLite	SQLite est une **bibliothèque écrite en C**. SQLite est parfait pour les petits projets. Sa particularité est d'être intégré directement à un programme et ne répond donc pas à la logique client-serveur. Il est le moteur de base de données le plus distribué au monde puiqu'il est intégré à de nombreux logiciels grand public comme FireFox, Skype, Adobe, etc. Le logiciel pèse moins de 300 ko et peut donc être intégré à des projets tournant sur de petites supports comme les smartphones. Souvent aucune installation n'est nécessaire pour l'utiliser.
Oracle	Oracle Database est sous licence propriétaire, c'est à dire payant. Il est souvent utilisé pour les projets à gros budget nécessitant de réaliser des actions complexes.
Microsoft SQL Server	Produit *Microsoft* ne tourne que sur un OS Windows, payant n'apporte rien de plus que les logiciels concurrents libre de droit. Si vous avez trop d'argent à la limite...

MySQL vs Postgre

Qui est le meilleur entre postgre et mysql? Telle est la question...

Il existe toujours des faux débats -*troll*- pour savoir quelle technologie est meilleure que l'autre. Dire que MySQL est meilleur que Postgre n'a aucun sens, chacun est bon dans ce pour quoi il a été crée.

On préférera MySQL pour des projet plus modestes où le nombre d'utilisateurs est faible avec un petit volume de données. Dans ce cas là, oui c'est lui le meilleur.

Par contre pour des projets plus ambitieux avec des données dépassant la dizaine de téraoctets (très gros projets), c'est lui le meilleur. Par contre il faut optimiser la configuration, etc. bref cela demande un certain travail.

NoSQL

Pour info, il existe un autre type de SGBD: les **NoSQL** -pour *Not Only SQL*- où la structure n'est plus pensée en tables / SQL. Ces **SGBD** permettent de travailler avec d'énormes données, sont utilisés par *Google*, *Amazon*, *Ubuntu One*, etc. Dans ce cas pour augmenter les performances il faut augmenter le nombre de serveurs.

Prérequis

Pour pouvoir exploiter les base de données que nous allons étudier, il est nécessaire de maîtriser le SQL. Ce n'est pas le sujet de cette page, vous trouverez donc de la documentation en suivant ces liens:

Créer une table SQL
Insérer des données
Sélectionner des données
Modifier des données
Supprimer des données

SQLite

Notre premier exemple concernera SQLite.

SQLite a été conçu pour être intégré dans le programme même. Pour des projets plus ambitieux / projets web le choix de MySQL serait plus judicieux.

Utiliser le module SQLite

Pour importer le module SQLite:

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

import sqlite3
```

Créer une base de données avec SQLite

La aussi pour créer une base de données avec SQLite, rien de plus simple:

```
conn = sqlite3.connect('ma_base.db')
```

Lorsque vous executerez votre programme vous remarquerez que si la base n'existe pas encore, un fichier sera crée dans le dossier de votre programme. Et si celui-ci existe déjà il sera réutilisé. Vous pouvez bien évidemment choisir l'emplacement de votre base de données en renseignant un path, exemple: "/data/ma_base.db" . Il vous faudra cependant vérifier que le dossier existe avant de l'utiliser.

Il est également possible de travailler avec une base de données de manière temporaire:

```
conn = sqlite3.connect(':memory:')
```

Lorsque le travail que vous attendiez est terminé, pensez à fermer la connection vers la base:

```
db.close()
```

Créer une table avec SQLite

Voici un exemple de création de table:

```
cursor = conn.cursor()
cursor.execute("""
CREATE TABLE IF NOT EXISTS users(
     id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
     name TEXT,
     age INTERGER
)
""")
conn.commit()
```

Supprimer une table avec SQLite:

```
cursor = conn.cursor()
cursor.execute("""
DROP TABLE users
""")
conn.commit()
```

Insérer des données

Il existe plusieurs manière d'insérer des données, la plus simple étant celle-ci:

```
cursor.execute("""
INSERT INTO users(name, age) VALUES(?, ?)""", ("olivier", 30))
```

Vous pouvez passer par un dictionnaire:

```
data = {"name" : "olivier", "age" : 30}
cursor.execute("""
INSERT INTO users(name, age) VALUES(:name, :age)""", data)
```

Vous pouvez récupérer l'id de la ligne que vous venez d'insérer de cette manière:

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

Il est également possible de faire plusieurs **insert** en une seule fois avec la fonction **executemany**:

```
users = []
users.append(("olivier", 30))
users.append(("jean-louis", 90))
cursor.executemany("""
INSERT INTO users(name, age) VALUES(?, ?)""", users)
```

Récupérer des données

Vous pouvez récupérer la première ligne correspondant à votre recherche à l'aide de la fonction **fetchone**.

```
cursor.execute("""SELECT name, age FROM users""")
user1 = cursor.fetchone()
print(user1)
```

Le résultat est un tuple:

```
('olivier', 30)
```

Vous pouvez récupérer plusieurs données de la même recherche en utilisant la fonction **fetchall()**.

```
cursor.execute("""SELECT id, name, age FROM users""")
rows = cursor.fetchall()
for row in rows:
    print('{0} : {1} - {2}'.format(row[0], row[1], row[2]))
```

L'objet curseur fonctionne comme un itérateur, invoquant la méthode **fetchall()** automatiquement:

```
cursor.execute("""SELECT id, name, age FROM users""")
for row in cursor:
    print('{0} : {1}, {2}'.format(row[0], row[1], row[2]))
```

Pour la recherche spécifique,on utilise la même logique vu précédemment:

```
id = 2
cursor.execute("""SELECT id, name FROM users WHERE id=?""", (id,))
response = cursor.fetchone()
```

Modifier des entrées

Pour modifier des entrées:

```
cursor.execute("""UPDATE users SET age = ? WHERE id = 2""", (31,))
```

SQLite transactions : rollback

Pour revenir au dernier **commit**, utilisez la méthode **rollback**.

```
conn.rollback()
```

Gestion des erreurs

Il est recommandé de toujours encadrer les opérations sur des bases de données et d'anticiper des erreurs:

```
import sqlite3

try:
    conn = sqlite3.connect('data/users.db')
    cursor = conn.cursor()
    cursor.execute("""
CREATE TABLE users(
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
    name TEXT,
    age INTERGER
)
""")
    conn.commit()
except sqlite3.OperationalError:
    print('Erreur la table existe déjà')
except Exception as e:
    print("Erreur")
    conn.rollback()
    # raise e
finally:
    conn.close()
```

Les erreurs que vous pouvez intercepter:

```
Error
DatabaseError
DataError
IntegrityError
InternalError
NotSupportedError
OperationalError
ProgrammingError
InterfaceError
Warning
```

MySQL

MySQL est le logiciel idéal pour vos projets de sites web.

Contrairement à SQLite, il est nécessaire de l'installer et de le configurer.

Installation de MySQL pour python

Tout d'abord il vous faudra installer le packet **mysql-server**

```
sudo apt-get install mysql-server
```

Un mot de passe super-utilisateur vous sera demandé lors de l'installation.

Une fois l'installation terminée, il vous faudra créer une base de données. Pour cela vous devez vous connecter à MySQL:

```
mysql -u root -p
```

On vous demandera votre mot de passe.

Une fois la **console MySQL** ouverte, vous pouvez créer votre base de données.

Dans notre cas, nous la nommerons **test1**:

```
mysql> CREATE DATABASE test1;
Query OK, 1 row affected (0.00 sec)
```

Il faudra installer les packets suivants:

```
sudo apt-get install python-pip libmysqlclient-dev python-dev python-
mysqldb
```

Et la librairie MySQL:

```
pip install mysql
pip install MySQL-python
pip install mysql-connector-python --allow-external mysql-connector-python
```

Pour python 3 il vous faudra installer le packet **python3-mysql.connector**:

```
sudo apt-get install python3-mysql.connector
```

Connection au serveur MySQL

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

import mysql.connector

conn = mysql.connector.connect(host="localhost",user="root",password="XXX",
database="test1")
cursor = conn.cursor()
conn.close()
```

Créer une table de données MySQL

```
cursor.execute("""
CREATE TABLE IF NOT EXISTS visiteurs (
    id int(5) NOT NULL AUTO_INCREMENT,
    name varchar(50) DEFAULT NULL,
    age INTEGER DEFAULT NULL,
```

```
    PRIMARY KEY(id)
);
""")
```

Insérer des données

Il existe deux manières d'ajouter des données dans une table:

```
user = ("olivier", "34")
cursor.execute("""INSERT INTO users (name, age) VALUES(%s, %s)""", user)

user = {"name": "olivier", "age" : "34"}
cursor.execute("""INSERT INTO users (name, age) VALUES(%(name)s,
%(age)s)""", user)
```

Trouver des données avec mysql

La logique est la même que pour SQLite:

```
cursor.execute("""SELECT id, name, age FROM users WHERE id = %s""", ("5",
))
rows = cursor.fetchall()
for row in rows:
    print('{0} : {1} - {2}'.format(row[0], row[1], row[2]))
```

La solution ORM

On remarque rapidement que changer de SGBD signifie changer de code. On peut éviter ce problème en ajoutant une couche d'abstraction: travailler avec un ORM. Il existe plusieurs **ORM** python mais le plus connu/utilisé est **SQLAlchemy**, que nous verrons dans un prochain chapitre.

Pensez à consulter l'aide des packages pour approfondir vos connaissances:

```
help(sqlite3)
help(mysql.connector)
```

http://pythoncentral.io/introduction-to-sqlite-in-python/

Introduction to SQLite in Python

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

Published: Thursday 11th April 2013
Last Updated: Thursday 12th December 2013

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.

```
1 # Get a cursor object
2 cursor = db.cursor()
3 cursor.execute('''
4     CREATE TABLE users(id INTEGER PRIMARY KEY, name TEXT,
5                        phone TEXT, email TEXT unique, password TEXT)
6 ''')
7 db.commit()
```

To drop a table:

```
1 # Get a cursor object
2 cursor = db.cursor()
3 cursor.execute('''DROP TABLE users''')
4 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.

```
1  cursor = db.cursor()
2  name1 = 'Andres'
3  phone1 = '3366858'
4  email1 = 'user@example.com'
5  # A very secure password
6  password1 = '12345'
7
8  name2 = 'John'
9  phone2 = '5557241'
10 email2 = 'johndoe@example.com'
11 password2 = 'abcdef'
12
13 # Insert user 1
14 cursor.execute('''INSERT INTO users(name, phone, email, password)
15                    VALUES(?,?,?,?)''', (name1,phone1, email1, password1))
16 print('First user inserted')
17
18 # Insert user 2
19 cursor.execute('''INSERT INTO users(name, phone, email, password)
20                    VALUES(?,?,?,?)''', (name2,phone2, email2, password2))
21 print('Second user inserted')
22
23 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:

```
1 cursor.execute('''INSERT INTO users(name, phone, email, password)
2                   VALUES(:name,:phone, :email, :password)''',
```

```
3                   {'name':name1, 'phone':phone1, 'email':email1,
  'password':password1})
```

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

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

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

```
1 id = cursor.lastrowid
2 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.

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

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

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

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

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

Updating (UPDATE) and Deleting (DELETE) Data

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

```
1  # Update user with id 1
2  newphone = '3113093164'
3  userid = 1
```

```
4  cursor.execute('''UPDATE users SET phone = ? WHERE id = ? ''',
5  (newphone, userid))
6
7  # Delete user with id 2
8  delete_userid = 2
9  cursor.execute('''DELETE FROM users WHERE id = ? ''', (delete_userid,))
10
11 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:

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

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

```
1 cursor.execute('''UPDATE users SET phone = ? WHERE id = ? ''',
2 (newphone, userid))
3 # The user's phone is not updated
4 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:

```
1  import sqlite3 #Import the SQLite3 module
2  try:
3      # Creates or opens a file called mydb with a SQLite3 DB
4      db = sqlite3.connect('data/mydb')
5      # Get a cursor object
6      cursor = db.cursor()
7      # Check if table users does not exist and create it
8      cursor.execute('''CREATE TABLE IF NOT EXISTS
9                        users(id INTEGER PRIMARY KEY, name TEXT, phone
10 TEXT, email TEXT unique, password TEXT)''')
11     # Commit the change
12     db.commit()
13 # Catch the exception
14 except Exception as e:
15     # Roll back any change if something goes wrong
16     db.rollback()
17     raise e
18 finally:
19     # 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']
```

```
7 returns email column.
8     print('{0} : {1}, {2}'.format(row['name'], row['email'],
  row['phone']))
  db.close()
```

To Practice: Try this interactive course on the basics of Lists, Functions, Packages and NumPy in Python.

http://pythoncentral.io/advanced-sqlite-usage-in-python/

Advanced SQLite Usage in Python

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

Published: Tuesday 16th April 2013
Last Updated: Thursday 12th December 2013

Following the SQLite3 series, this post is about some advanced topics when we are working with the SQLite3 module. If you missed the first part, you can find it here.

Using SQLite's date and datetime Types

Sometimes we need to insert and retrieve some `date` and `datetime` types in our SQLite3 database. When you execute the insert query with a date or datetime object, the `sqlite3` module calls the default adapter and converts them to an ISO format. When you execute a query in order to retrieve those values, the `sqlite3` module is going to return a string object:

```
   >>> import sqlite3
1  >>> from datetime import date, datetime
2  >>>
3  >>> db = sqlite3.connect(':memory:')
4  >>> c = db.cursor()
5  >>> c.execute('''CREATE TABLE example(id INTEGER PRIMARY KEY, created_at
6  DATE)''')
7  >>>
8  >>> # Insert a date object into the database
9  >>> today = date.today()
10 >>> c.execute('''INSERT INTO example(created_at) VALUES(?)''', (today,))
11 >>> db.commit()
12 >>>
13 >>> # Retrieve the inserted object
14 >>> c.execute('''SELECT created_at FROM example''')
15 >>> row = c.fetchone()
16 >>> print('The date is {0} and the datatype is {1}'.format(row[0],
17 type(row[0])))
18 # The date is 2013-04-14 and the datatype is <class 'str'>
   >>> db.close()
```

The problem is that if you inserted a date object in the database, most of the time you are expecting a date object when you retrieve it, not a string object. This problem can be solved passing `PARSE_DECLTYPES` and `PARSE_COLNAMES` to the `connect` method:

```
1  >>> import sqlite3
2  >>> from datetime import date, datetime
3  >>>
```

```
>>> db = sqlite3.connect(':memory:',
detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_COLNAMES)
>>> c = db.cursor()
>>> c.execute('''CREATE TABLE example(id INTEGER PRIMARY KEY, created_at
DATE)''')
>>> # Insert a date object into the database
>>> today = date.today()
>>> c.execute('''INSERT INTO example(created_at) VALUES(?)''', (today,))
>>> db.commit()
>>>
>>> # Retrieve the inserted object
>>> c.execute('''SELECT created_at FROM example''')
>>> row = c.fetchone()
>>> print('The date is {0} and the datatype is {1}'.format(row[0],
type(row[0])))
# The date is 2013-04-14 and the datatype is <class 'datetime.date'>
>>> db.close()
```

Changing the connect method, the database now is returning a date object. The `sqlite3` module uses the column's type to return the correct type of object. So, if we need to work with a `datetime` object, we must declare the column in the table as a `timestamp` type:

```
>>> c.execute('''CREATE TABLE example(id INTEGER PRIMARY KEY, created_at
timestamp)''')
>>> # Insert a datetime object
>>> now = datetime.now()
>>> c.execute('''INSERT INTO example(created_at) VALUES(?)''', (now,))
>>> db.commit()
>>>
>>> # Retrieve the inserted object
>>> c.execute('''SELECT created_at FROM example''')
>>> row = c.fetchone()
>>> print('The date is {0} and the datatype is {1}'.format(row[0],
type(row[0])))
# The date is 2013-04-14 16:29:11.666274 and the datatype is <class
'datetime.datetime'>
```

In case you have declared a column type as `DATE`, but you need to work with a `datetime` object, it is necessary to modify your query in order to parse the object correctly:

```
c.execute('''CREATE TABLE example(id INTEGER PRIMARY KEY, created_at
DATE)''')
# We are going to insert a datetime object into a DATE column
now = datetime.now()
c.execute('''INSERT INTO example(created_at) VALUES(?)''', (now,))
db.commit()

# Retrieve the inserted object
c.execute('''SELECT created_at as "created_at [timestamp]" FROM
example''')
```

Using `as "created_at [timestamp]"` in the SQL query will make the adapter to parse the object correctly.

Insert Multiple Rows with SQLite's executemany

Sometimes we need to insert a sequence of objects in the database, the `sqlite3` module provides the `executemany` method to execute a SQL query against a sequence.

```
# Import the SQLite3 module
import sqlite3
db = sqlite3.connect(':memory:')
c = db.cursor()
c.execute('''CREATE TABLE users(id INTEGER PRIMARY KEY, name TEXT, phone
TEXT)''')
users = [
    ('John', '5557241'),
    ('Adam', '5547874'),
    ('Jack', '5484522'),
    ('Monthy',' 6656565')
]

c.executemany('''INSERT INTO users(name, phone) VALUES(?,?)''', users)
db.commit()

# Print the users
c.execute('''SELECT * FROM users''')
for row in c:
    print(row)

db.close()
```

Please note that each element of the sequence must be a tuple.

Execute SQL File with SQLite's executescript

The `execute` method only allows you to execute a single SQL sentence. If you need to execute several different SQL sentences you should use `executescript` method:

```
# Import the SQLite3 module
import sqlite3
db = sqlite3.connect(':memory:')
c = db.cursor()
script = '''CREATE TABLE users(id INTEGER PRIMARY KEY, name TEXT, phone
TEXT);
            CREATE TABLE accounts(id INTEGER PRIMARY KEY, description
TEXT);

            INSERT INTO users(name, phone) VALUES ('John', '5557241'),
             ('Adam', '5547874'), ('Jack', '5484522');'''
c.executescript(script)

# Print the results
c.execute('''SELECT * FROM users''')
for row in c:
    print(row)

db.close()
```

If you need to read the script from a file:

```
1 fd = open('myscript.sql', 'r')
2 script = fd.read()
3 c.executescript(script)
4 fd.close()
```

Please remember that it is a good idea to surround your code with a `try/except/else` clause in order to catch the exceptions. To learn more about the `try/except/else` keywords, checkout the Catching Python Exceptions – The try/except/else keywords article.

Defining SQLite SQL Functions

Sometimes we need to use our own functions in a statement, specially when we are inserting data in order to accomplish some specific task. A good example of this is when we are storing passwords in the database and we need to encrypt those passwords:

```
   import sqlite3 #Import the SQLite3 module
1  import hashlib
2
3  def encrypt_password(password):
4      # Do not use this algorithm in a real environment
5      encrypted_pass = hashlib.sha1(password.encode('utf-8')).hexdigest()
6      return encrypted_pass
7
8  db = sqlite3.connect(':memory:')
9  # Register the function
10 db.create_function('encrypt', 1, encrypt_password)
11 c = db.cursor()
12 c.execute('''CREATE TABLE users(id INTEGER PRIMARY KEY, email TEXT,
13 password TEXT)''')
14 user = ('johndoe@example.com', '12345678')
15 c.execute('''INSERT INTO users(email, password) VALUES (?,encrypt(?))''',
   user)
```

The `create_function` takes 3 parameters: `name` (the name used to call the function inside the statement), the number of parameters the function expects (1 parameter in this case) and a callable object (the function itself). To use our registered function, we called it using `encrypt()` in the statement.

Finally, PLEASE use a true encryption algorithm when you are storing passwords!

To Practice: Try this interactive course on the basics of Lists, Functions, Packages and NumPy in Python.