Smokiev

Zeste de savoir

Des bases de données en Python avec sqlite3

20 mars 2019

Table des matières

1. Introduction

Dans le joyeux monde de la programmation, il est souvent nécessaire de stocker des informations.

À petite comme à grande échelle, les Bases De Données (BDD) s'imposent comme une forme efficace de stockage. Il est alors plutôt aisé d'interagir avec celles-ci en utilisant un Système de Gestion de Base de Données (SGBD), un logiciel spécialement conçu pour les gérer et les manipuler à l'aide d'un langage normalisé tel que le *Structured Query Language* (SQL).

Parmi les SGBD, nous pouvons trouver SQLite qui utilise un sous-ensemble de SQL. Sa légèreté et le fait que les données se trouvent sur le terminal du client et non sur un serveur distant, en font un outil apprécié pour des applications personnelles ou encore dans l'embarqué. Toutefois, il est relativement lent. SQLite fait partie de la famille des SGBD dits « Relationnelles », car les données sont alors placées dans des tables et traitées comme des ensembles.

En Python, le module sqlite3 permet de travailler avec ce moteur, mais ne supporte pas le multi-thread.

À travers ce tutoriel, nous allons donc apprendre à utiliser ce dernier tout en pratiquant.

Ce tutoriel, n'est ni une introduction aux BDD ni une introduction au langage SQL. Il est donc recommandé de vous référer à ce tutoriel pour vous familiariser avec ces concepts. De plus, des bases en Python, que vous pouvez acquérir avec ce tutoriel par exemple, sont nécessaires pour être à l'aise.

Pour ce tutoriel, j'utiliserai la version 3.5 de Python. Il est possible qu'il y ait quelques petites différences concernant le module selon votre version de Python, c'est pourquoi il faut que vous choisissiez la documentation adaptée à votre version.

Prérequis
Bases en programmation et connaissances en Python
Connaissances en BDD et en SQL
Objectifs
Faire découvrir le module sqlite3

Vous êtes prêt ? Alors, en route !

2. Fonctionnalités de base

À travers cette partie nous allons nous familiariser avec les bases de sqlite3 : comment créer une base de données, exécuter une requête ou encore utiliser des clefs étrangères.

2.1. Se connecter et se déconnecter

Avant de commencer, il convient d'importer le module, comme il est coutume de faire avec Python :

```
1 import sqlite3
```

2.1.0.1. Connexion

Cela fait, nous pouvons nous connecter à une BDD en utilisant la méthode `connect` et en lui passant l'emplacement du fichier de stockage en paramètre. Si ce dernier n'existe pas, il est alors créé :

```
1 connexion = sqlite3.connect("basededonnees.db")  #BDD dans le
      fichier "basededonnees.db"
```

Comme vous pouvez le voir, nous récupérons un objet retourné par la fonction. Celui-ci est de type *Connection* et il nous permettra de travailler sur la base.

Par ailleurs, il est aussi possible de stocker la BDD directement dans la *RAM* en utilisant la chaîne clef `":memory:"`. Dans ce cas, il n'y aura donc pas de persistance des données après la déconnexion.

```
1 connexion = sqlite3.connect(":memory:")  #BDD dans la RAM
```

?

Mais... en quoi est-ce utile de stocker des informations dans la *RAM* puisque celles-ci sont perdues quand on se déconnecte ?

C'est une bonne question ! Eh bien, premièrement ce qui est stocké dans la *RAM* est plus rapide d'accès que ce qu'il y a sur le disque dur. Ainsi, certains utiliseront la *RAM* de sorte à gagner en performance. Ensuite, les bases temporaires sont aussi très utiles pour effectuer des tests, par exemple des tests unitaires qui sont ainsi reproductibles aisément et n'altèrent pas d'éventuelles BDD persistantes.

2.1.0.2. Déconnexion

Que nous soyons connectés avec la *RAM* ou non, il ne faut pas oublier de nous déconnecter. Pour cela, il nous suffit de faire appel à la méthode `close` de notre objet *Connection*.

```
1 connexion.close()  #Déconnexion
```

2.1.0.3. Un mot sur les types de champ

Comme nous allons bientôt voir comment exécuter des requêtes, il est important de connaître les types disponibles, avec leur correspondance en Python. Voici ci-dessous, un tableau récapitulatif :

SQLite	Python
NULL	None
INTEGER	int
REAL	float
TEXT	str par défaut
BLOB	bytes

Dans le sens inverse, les types Python du tableau seront utilisables avec leur correspondance SQLite. Il est vrai que la liste peut s'avérer restreignante. Heureusement, il est possible d'ajouter nos propres types de données.

2.2. Exécuter des requêtes

Pour exécuter nos requêtes, nous allons nous servir d'un objet *Cursor*, récupéré en faisant appel à la méthode `cursor` de notre objet de type *Connection*.

```
1 curseur = connexion.cursor()  #Récupération d'un curseur
```

2.2.0.1. Valider ou annuler les modifications

Lorsque nous effectuons des modifications sur une table (insertion, modification ou encore suppression d'éléments), celles-ci ne sont pas automatiquement validées. Ainsi, sans validation, les modifications ne sont pas effectuées dans la base et ne sont donc pas visibles par les autres connexions. Pour résoudre cela, il nous faut donc utiliser la méthode `commit` de notre objet de type *Connection*.

En outre, si nous effectuons des modifications puis nous souhaitons finalement revenir à l'état du dernier `commit`, il suffit de faire appel à la méthode `rollback`, toujours de notre objet de type *Connection*.

Voici un petit morceau de code résumant cela :

```
###modifications....
connexion.commit()  #Validation des modifications
###modifications....
connexion.rollback()  #Retour à l'état du dernier commit, les
    modifications effectuées depuis sont perdues
```

2.2.0.2. Exécuter une requête

Pour exécuter une requête il suffit de passer celle-ci à la méthode `execute` :

```
###Exécution unique
curseur.execute('''CREATE TABLE IF NOT EXISTS scores(
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
    pseudo TEXT,
    valeur INTEGER
)''')
```

Comme vous pouvez le voir, nous venons d'ajouter une table **scores** dans notre base jusqu'à présent vide.

2.2.0.3. Exécuter plusieurs requêtes

Pour exécuter plusieurs requêtes, comme pour ajouter des éléments à une table par exemple, nous pouvons faire appel plusieurs fois à la méthode `execute` :

```
donnees = [("toto", 1000), ("tata", 750), ("titi", 500)]
###Exécutions multiples
for donnee in donnees:
```

```
4
        curseur.execute('''INSERT INTO scores (pseudo, valeur) VALUES (?, ?)''
        donnee)
5 connexion.commit()  #Ne pas oublier de valider les modifications
```

Ou nous pouvons aussi passer par la méthode `executemany` :

```
1 donnees = [("toto", 1000), ("tata", 750), ("titi", 500)]
2 ###Exécutions multiples
3 curseur.executemany("INSERT INTO scores (pseudo, valeur) VALUES (?, ?)",
      donnees)
4 connexion.commit()  #Ne pas oublier de valider les modifications
```

Remarquez que nous utilisons ici, l'opérateur `?` couplé à des tuples pour passer des paramètres aux requêtes, mais nous pouvons aussi utiliser des dictionnaires et l'opérateur `:` avec le nom des clefs :

```
1 donnees = (
2     {"psd" : "toto", "val" : 1000},
3     {"psd" : "tata", "val" : 750},
4     {"psd" : "titi", "val" : 500}
5 )
6 ###Exécutions multiples
7 curseur.executemany("INSERT INTO scores (pseudo, valeur) VALUES (:psd, :val)",
      donnees)
8 connexion.commit()  #Ne pas oublier de valider les modifications
```

2.2.0.4. Exécuter un script

Enfin, il est aussi possible d'exécuter un script directement à l'aide de la méthode `executes cript`. Si celui-ci contient plusieurs requêtes, celles-ci doivent être séparées par des points-virgules.

```
1 ###Exécution d'un script
2 curseur.executescript("""
3
4     DROP TABLE IF EXISTS scores;
5
6     CREATE TABLE IF NOT EXISTS scores(
7     id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
8     pseudo TEXT,
9     valeur INTEGER);
```

```
10
11      INSERT INTO scores(pseudo, valeur) VALUES ("toto", 1000);
12      INSERT INTO scores(pseudo, valeur) VALUES ("tata", 750);
13      INSERT INTO scores(pseudo, valeur) VALUES ("titi", 500)
14  """)
15  connexion.commit()  #Ne pas oublier de valider les modifications
```

i Remarquez que comme notre table **scores** se trouvera sur la machine du joueur et qu'il n'y aura pas de communication avec l'extérieur, ce sera un classement en local et non global.

2.3. Parcourir des enregistrements

Pour récupérer des éléments, nous allons évidemment passer par une requête SQL. Il faudra ensuite parcourir le résultat et nous nous servirons de notre objet de type *Cursor* pour cela.

Mais avant de le faire, reprenons notre table **scores** et ajoutons quelques éléments afin d'avoir un exemple pratique.

Le **code** :

Contenu masqué n°1

Le **résultat** :

identifiant	pseudo	valeur
1	”toto”	1000
2	”tata”	750
3	”titi”	500
4	”toto”	250
5	”tata”	150
6	”tete”	0

2.3.0.1. Un résultat à la fois

Pour parcourir un résultat à la fois, il suffit d'utiliser la méthode `fetchone` qui retourne un résultat sous forme de tuple, ou `None`, s'il n'y en a pas.

```
donnee = ("titi", )
curseur.execute("SELECT valeur FROM scores WHERE pseudo = ?",
    donnee)
print(curseur.fetchone())  #affiche "(500,)"
```

2.3.0.2. Plusieurs résultats d'un coup

Vous comprendrez que cette technique montre vite ses limites quand le nombre de résultats augmente, et ce même si nous pouvons procéder ainsi :

```
donnee = ("tata", )
curseur.execute("SELECT valeur FROM scores WHERE pseudo = ?",
    donnee)
result = curseur.fetchone()
while result:
    print(result)
    result = curseur.fetchone()
###affiche "(750,)" puis "(150,)"
```

Or, `fetchmany`, utilisable de la même manière, permet justement de récupérer plusieurs résultats d'un coup. Le nombre de résultats prend par défaut la valeur de l'attribut `arraysize` du curseur, mais nous pouvons aussi passer un nombre à la méthode. S'il n'y a pas de résultat, la liste retournée est vide :

```
print(curseur.arraysize)  #Affiche "1"
donnee = (400, )

curseur.execute("SELECT pseudo FROM scores WHERE valeur > ?",
    donnee)
print(curseur.fetchmany())  #Affiche "[('toto',)]"
print(curseur.fetchmany())  #Affiche "[('tata',)]"

curseur.execute("SELECT pseudo FROM scores WHERE valeur > ?",
    donnee)
print(curseur.fetchmany(2)) #Affiche "[('toto',), ('tata',)]"
```

Comme vous pouvez le constater, cela revient à utiliser `fetchone` si l'attribut `arraysize` du curseur vaut 1, ce qui n'est pas très utile.

2.3.0.3. Tout ou rien

Enfin, pour récupérer directement tous les résultats d'une requête, nous pouvons faire appel à la méthode `fetchall`. Là encore, elle retourne une liste vide s'il n'y a pas de résultats.

```
curseur.execute("SELECT * FROM scores")
resultats = curseur.fetchall()
for resultat in resultats:
    print(resultat)
```

Par ailleurs, nous pouvons aussi utiliser le curseur comme un itérable :

```
curseur.execute("SELECT * FROM scores")
for resultat in curseur:
    print(resultat)
```

Les deux codes ont le même effet et affichent :

```
(1, "toto", 1000)
(2, "tata", 750)
(3, "titi", 500)
(4, "toto", 250)
(5, "tata", 150)
(6, "tete", 0)
```

2.4. Récupérer quelques informations

Avec sqlite3, nous pouvons récupérer quelques informations sur l'état de notre base.

2.4.0.1. En transaction ou pas

Tout d'abord, pour savoir si des modifications ont été apportées sans être validées, il suffit de récupérer la valeur de l'attribut `in_transaction` de notre objet de type *Connection*. En effet, celui-ci vaut `True` si c'est le cas et `False` sinon.

```
###modifications...
print(connexion.in_transaction)  #Affiche "True"
connexion.commit()
print(connexion.in_transaction)  #Affiche "False"
```

2.4.0.2. Connaître le nombre de modifications depuis le dernier commit

Ensuite, pour être au courant du nombre de modifications (ajouts, mises à jour ou suppressions) apportées depuis notre connexion à la base, il suffit de récupérer la valeur de l'attribut `total_changes` de notre objet de type *Connection*.

Dans l'exemple ci-dessous, nous insérons autant de scores qu'il y a de lettres dans la chaîne de caractères :

```
print(connexion.total_changes)  #Affiche "0"
chaine = "azertyuiopmlkjhgfdsqwxcvbnmlkjhgfdsqazertyuiopnbvcxw"
print(len(chaine))  #Affiche "52"
for donnee in enumerate(chaine):

        curseur.execute("INSERT INTO scores (pseudo, valeur) VALUES (?, ?)",
        donnee[::-1])
print(connexion.total_changes)  #Affiche "52"
```

2.4.0.3. Connaitre le nombre de lignes impactées par une exécution

De même, pour connaître le nombre de lignes impactées par une exécution, il suffit d'utiliser l'attribut `rowcount` de notre objet de type *Cursor*. S'il n'y a eu aucune exécution ou que le nombre de de lignes ne peut pas être déterminé (comme pour une sélection par exemple), il vaut -1. De plus, pour les versions de SQLite antérieure à la 3.6.5, la valeur vaut 0 après une suppression totale des éléments d'une table.

Voici un exemple :

```
print(curseur.rowcount)  #Affiche "-1"

donnee = ("toto", 1000)
curseur.execute("INSERT INTO scores (pseudo, valeur) VALUES (?, ?)",
    donnee)
print(curseur.rowcount)  #Affiche "1"

donnees = [("tata", 750), ("titi", 500)]
curseur.executemany("INSERT INTO scores (pseudo, valeur) VALUES (?, ?)",
    donnees)
print(curseur.rowcount)  #Affiche "2"

curseur.execute("SELECT * FROM scores")
print(curseur.rowcount)  #Affiche "-1"

curseur.execute("DELETE FROM scores")
print(curseur.rowcount)  #Affiche "-3" (0 si version SQLite <
    3.6.5)
```

2.4.0.4. Récupérer l'identifiant de la dernière ligne insérée

Par ailleurs, nous pouvons aussi récupérer l'identifiant du dernier enregistrement dans une table à l'aide de l'attribut `lastrowid` de notre objet de type *Connection* :

```
from random import randint

###...

donnee = (randint(1, 1000), "toto", 1000)
print(donnee[0])  #Affiche "589"
curseur.execute("INSERT INTO scores (id, pseudo, valeur) VALUES (?, ?, ?)",
    donnee)
curseur.execute("SELECT * FROM scores WHERE id = ?",
    (curseur.lastrowid, ))
print(curseur.fetchone())  #Affiche "(589, 'toto', 1000)"
```

Dans l'exemple ci-dessus, nous insérons un enregistrement avec un identifiant aléatoire puis nous récupérons ce même enregistrement grâce à la valeur de l'attribut.

2.5. Utiliser des clefs étrangères

Dès que le nombre de tables augmente, il est souvent primordial de les lier à l'aide de clefs étrangères.

2.5.0.1. Activer les clefs étrangères

Avec sqlite3, les clefs étrangères ne sont pas activées de base. Il nous faut donc y remédier avec la requête adéquate :

```
curseur.execute("PRAGMA foreign_keys = ON")  #Active les clés
    étrangères
```

2.5.0.2. Lier deux tables

Maintenant que c'est fait, nous pouvons ajouter une table **joueurs**, donc créer une nouvelle table **scores** (veillez à supprimer l'ancienne si jamais), puis remplir celles-ci et récupérer les enregistrements, avec une bonne utilisation des clefs étrangères :

Le **code** :

Contenu masqué n°2

Le **résultat** :

joueurs

id_joueur	pseudo	mdp
1	”toto”	”123”
2	”tata”	”azerty”
3	”titi”	”qwerty”

scores

id_score	fk_joueur	valeur
1	1	1000
2	2	750
3	3	500

```
joueur : (1, 'toto', '123')
joueur : (2, 'tata', 'azerty')
joueur : (3, 'titi', 'qwerty')
score : (1, 1, 1000)
score : (2, 2, 750)
score : (3, 3, 500)
```

Vous remarquerez que les mots de passe ne sont pas chiffrés ce qui, comme vous le savez, est une pratique fortement déconseillée.

Une fois, nos tables créées et remplies, nous pouvons facilement travailler dessus à l'aide de jointures, comme pour récupérer le meilleur score (pseudo et valeur) par exemple :

```
###Récupération du meilleur score
curseur.execute("""SELECT j.pseudo, s.valeur FROM joueurs as j INNER JOIN
    scores as s ON j.id_joueur = s.fk_joueur
    ORDER BY s.valeur DESC LIMIT 1""")
print(curseur.fetchone())  #Affiche "('toto', 1000)"
```

Dans l'exemple ci-dessus, nous utilisons le type de *join* le plus répandu (le `INNER JOIN`), mais nous aurions pu en utiliser d'autres .

Au terme de cette partie, vous savez désormais tout ce qui est nécessaire pour créer une base de données et gérer celle-ci avec sqlite3 !

Contenu masqué

Contenu masqué n°1

```
import sqlite3

#Connexion
connexion = sqlite3.connect('basededonnees.db')

#Récupération d'un curseur
curseur = connexion.cursor()

#Création de la table scores
curseur.execute("""
    CREATE TABLE IF NOT EXISTS scores(
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
    pseudo TEXT,
    valeur INTEGER);
    """)

#Suppression des éléments de scores
curseur.execute("""DELETE FROM scores""")

#Préparation des données à ajouter
donnees = [
    ("toto", 1000),
    ("tata", 750),
    ("titi", 500),
    ("toto", 250),
    ("tata", 150),
    ("tete", 0)
    ]

#Insertion des données
curseur.executemany('''INSERT INTO scores (pseudo, valeur) VALUES (?, ?)''',
    donnees)

#Validation
connexion.commit()

#Parcours des enregistrements....

#Déconnexion
```

```
39 connexion.close()
```

Retourner au texte.

Contenu masqué n°2

```
1  import sqlite3
2
3  #Connexion
4  connexion = sqlite3.connect("basededonnees.db")
5
6  #Récupération d'un curseur
7  curseur = connexion.cursor()
8
9  #Activation clés étrangères
10 curseur.execute("PRAGMA foreign_keys = ON")
11
12 #Création table joueur puis score si elles n'existent pas encore
13 #Puis suppression des données dans joueurs (et dans scores aussi
       par cascade)
14 #afin d'éviter les répétitions d'enregistrements avec des
       exécutions multiples
15 curseur.executescript("""
16
17     CREATE TABLE IF NOT EXISTS joueurs(
18     id_joueur INTEGER PRIMARY KEY,
19     pseudo TEXT,
20     mdp TEXT);
21
22     CREATE TABLE IF NOT EXISTS scores(
23     id_score INTEGER PRIMARY KEY,
24     fk_joueur INTEGER NOT NULL,
25     valeur INTEGER,
26     FOREIGN KEY(fk_joueur) REFERENCES joueurs(id_joueur)
27     ON DELETE CASCADE);
28
29     DELETE FROM joueurs;
30 """)
31
32 #Préparation des données
33 donnees_joueur = [
34     ("toto", "123"),
35     ("tata", "azerty"),
36     ("titi", "qwerty")
37 ]
38 donnees_score = [
```

```
    (1, 1000),
    (2, 750),
    (3, 500)
]

#Insertion des données dans table joueur puis score
curseur.executemany("INSERT INTO joueurs (pseudo, mdp) VALUES (?, ?)",
    donnees_joueur)
curseur.executemany("INSERT INTO scores (fk_joueur, valeur) VALUES (?, ?)",
    donnees_score)

#Validation des ajouts
connexion.commit()

#Affichage des données
for joueur in curseur.execute("SELECT * FROM joueurs"):
    print("joueur :", joueur)

for score in curseur.execute("SELECT * FROM scores"):
    print("score :", score)

#Déconnexion
connexion.close()
```

Retourner au texte.

3. Conclusion

Au cours de ce tutoriel, nous avons appris à utiliser sqlite3. Comme vous avez pu le constater, c'est un outil à la fois puissant et flexible.

Ce tutoriel n'est pas terminé. Une seconde partie, portant sur des fonctionnalités plus avancées (gestion des erreurs ou encore utilisation de ses propres types et de ses propres fonctions) devrait venir le compléter.

Si vous souhaitez en apprendre plus ou renforcer vos connaissances, vous pouvez vous référer aux liens suivants (liste non exhaustive) :

— la documentation
— le chapitre 16 du livre de Swinnen
— le site officiel de SQLite

Enfin, il est possible que vous recherchiez des alternatives à SQLite pour diverses raisons (si vous voulez stocker vos données sur un serveur distant par exemple). Dans ce cas là, vous pouvez poursuivre avec un autre SGBDR tel que MySQL et il vous sera assez facile de porter votre code. D'autre part, vous pouvez aussi vouloir vous tourner vers d'autres types de SGBD, notamment ceux dits NoSQL tel que MongoDB (utilisable en Python avec la distribution PyMongo), ou encore vers d'autres formats (xml, json, etc.).

Comme vous le savez, votre choix dépendra de vos besoins.

À bientôt !

Merci notamment à Aabu, germinolegrand et nohar pour leurs retours.
Merci à artragis pour la validation.