SGDB-Python-3

1. Avec sqlite3 : adapté pour les petits projets, ne nécessite pas de système client/serveur, SGBD le plus distribué au monde

1.1. Importation de la bibliothèque

```
import sqlite3
```

Connexion

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

« connexion » est un objet qui contient le résultat de la méthode « connect « pour de les objets de type sqlite3

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

1.2. 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*.

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

- Exécution d'une seule requête

```
#Exécution unique
curseur.execute('''CREATE TABLE IF NOT EXISTS palmares(
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
    pseudo TEXT,
    valeur INTEGER
) ;''')
#La requête est une chaine de caractères il faut donc
#au moins un guillemet simple ou double aux extrémités
```

- Exécution de plusieurs requêtes

Note importante : Dans un programme concret, les données à enregistrer se présenteront la plupart du temps dans des variables Python. Vous devrez donc construire la chaîne de caractères contenant la requête SQL, en y incluant les valeurs tirées de ces variables. Il est cependant fortement déconseillé de faire appel dans ce but aux techniques ordinaires de formatage des chaînes, car cela peut ouvrir une faille de sécurité dans vos

programmes, en y autorisant les intrusions par la méthode de piratage connue sous le nom d' **injection SQL** https://fr.wikipedia.org/wiki/Injection_SQL . Il faut donc plutôt confier le formatage de vos requêtes au module d'interface lui-même. C'est ce que l'on appelle des requêtes préparées dans d'autres langages (PHP)

L'opérateur ? est couplé à des tuples pour passer des paramètres aux requêtes

Faire appel plusieurs fois à la méthode **execute**

```
donnees = [("toto", 200), ("tata", 50), ("titi", 100)]
#Exécutions multiples
for donnee in donnees:
      curseur.execute('''INSERT INTO palmares (pseudo, valeur) VALUES (?, ?)''
      ', donnees)
connexion.commit()  #Ne pas oublier de valider les modifications
```

Faire appel une fois à la méthode **executemany**

```
donnees = [("toto", 200), ("tata", 50), ("titi", 100)]
#Exécutions multiples
curseur.executemany("INSERT INTO scores (pseudo, valeur) VALUES (?, ?)", donne
es)
connexion.commit()  #Ne pas oublier de valider les modifications
```

Utilisation d'un dictionnaire

```
scores = (
{"spdo":"toto","val": 200},
{"spdo":"tata","val": 50},
{"spdo":"titi","val": 100}
)
```

```
#Exécutions multiples
curseur.executemany("INSERT INTO scores (pseudo, valeur) VALUES ( :spdo, :val)
", donnees)
connexion.commit()  #Ne pas oublier de valider les modifications
```

Utilisation d'un script entier avec **executescript**

On peut effectuer plusieurs requêtes dans un même script. Celles-ci doivent être séparées par un « ; ».

```
#Exécution d'un script
curseur.executescript("""

    DROP TABLE IF EXISTS palmares;

    CREATE TABLE IF NOT EXISTS palmares(
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
    pseudo TEXT,
    valeur INTEGER);

    INSERT INTO scores(pseudo, valeur) VALUES ("toto", 200);
    INSERT INTO scores(pseudo, valeur) VALUES ("tata", 50);
    INSERT INTO scores(pseudo, valeur) VALUES ("titi", 100)
""")
connexion.commit()  #Ne pas oublier de valider les modifications
```

- Valider ou annuler les modifications

Lorsque qu'une modification est appliquée sur une table (insertion, modification ou encore suppression d'éléments), celle-ci ne sont pas automatiquement validée. Ainsi, sans validation, la modification n'est pas effectuée dans la base et n'est donc pas visible 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 c*onnection*.

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

1.3. Clôture de la connexion

Un fois le travail effectué RAM ou pas il faut arrêter la connexion.

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

1.4. Exploitation d'une requête SELECT

Après l'exécution d'une requête SELECT vous pouvez :

- Pour une requête retournant un résultat unique méthode **fetchone()** sur l'objet cursor

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

retourne **un tuple** ou retournera **None**

- Pour une requête retournant un résultat unique méthode **fetchall()** sur l'objet cursor

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

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

Retourne une liste

2. Avec MYSQL :

La structure du programme dans son déroulement est identique :

2.1. Connexion / déconnexion

```
import mysql.connector

connexion = mysql.connector.connect(host="localhost",user="root",password="XXX
", database="test1")
curseur = connexion.cursor()
#requêtes

connexion.close()
```

- host : adresse où se trouve la base de données peut être une @IP ou une URL
- user nom : de l'utilisateur qui pourra utilisé que les droits qui lui auront été définis
- password : mot de passe

2.2. Création d'une table de données

```
curseur.execute("""
CREATE TABLE IF NOT EXISTS palmares (
    id int(5) NOT NULL AUTO_INCREMENT,
    pseudo varchar(50) DEFAULT NULL,
    valeur INTEGER DEFAULT NULL,
    PRIMARY KEY(id)
);
""")
```

2.3. Insertion des données

```
#tuple

jr1 = ("toto", "200")
curseur.execute("""INSERT INTO pamares (pseudo, valeur) VALUES(%s, %s)""", jr1
)
#ou dictionnaire
jr1 = {"psdo": "toto", "val" : "200"}
curseur.execute("""INSERT INTO users (pseudo, valeur) VALUES(%(psdo)s, %(val)s
)""", jr1)
```

%s : l'argument est traité et présenté comme une chaîne de caractères.

2.4. Traitement d'une requête SELECT

Idem SQLite : les méthodes fetchone() et fetchmany() sont aussi présentent . Elles utilisent de la même manière cursor comme un itérateur

```
# Using a while loop
cursor.execute("SELECT * FROM employees")
row = cursor.fetchone()
while row is not None:
  print(row)
  row = cursor.fetchone()

# Using the cursor as iterator
cursor.execute("SELECT * FROM employees")
for row in cursor:
  print(row)


#retourne un tuple ou None
```

```
rows = cursor.fetchmany(size=1)

#retourne une liste de tuples ou une liste vide
```

```
rows = cursor.fetchall()

#retourne une liste de tuples ou une liste vide
```