



Les bases de données en Python

- Sqlite3
- MySQL

1.Introduction

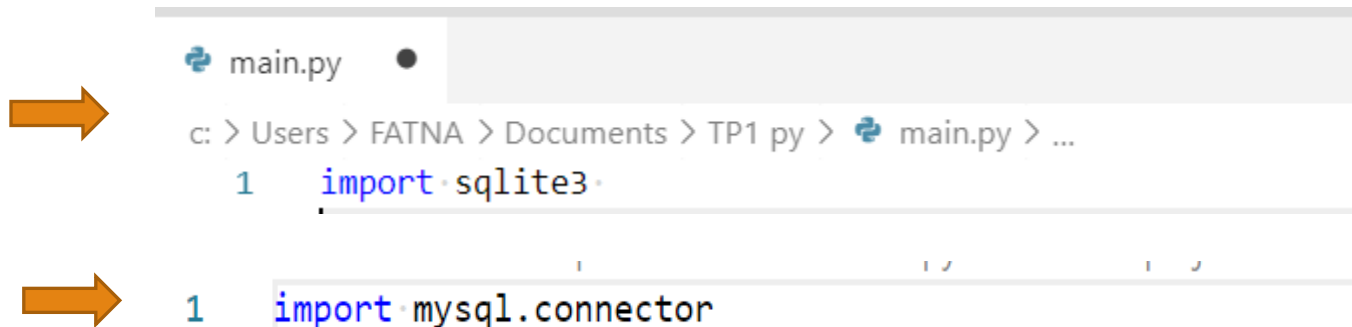
- les Bases De Données (BDD) s'imposent comme une forme efficace de stockage. Il est alors plutôt aisé d'interagir avec celles-ci en utilisant un Système de Gestion de Base de Données (SGBD), un logiciel spécialement conçu pour les gérer et les manipuler à l'aide d'un langage normalisé tel que le Structured 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
- SQLite fait partie de la famille des SGBD dits« Relationnelles », car les données sont alors placées dans des tables et traitées comme des ensembles.
- MySQL est l'un des SGBDR les plus utilisés au monde. Il est gratuit et très puissant et 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.

2. Fonctionnalités de base

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

1. Se connecter et se déconnecter

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




```
main.py
c: > Users > FATNA > Documents > TP1 py > main.py > ...
1 import sqlite3

1 import mysql.connector
```

2. 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 :




```
2 #connexion à la base de données base.db
3 connection=sqlite3.connect("base.db")
4
```

```
import mysql.connector

mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword"
)

print(mydb)
```


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.



```
4 #base de données dans la RAM
5 connexion = sqlite3.connect(":memory:")
6
```

3. Déconnexion

il ne faut pas oublier de se déconnecter. Pour cela, il suffit de faire appel à la méthode close de notre objet Connection



```
13      #Déconnexion  
14      connection.close()
```

4. Les types de champ

il est important de connaître les types disponibles, avec leur correspondance en Python.

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 restreignant. Alors, il est possible d'ajouter nos propres types de données.


5. 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`.

```
16  ###modifications....  
17  connection.commit() #Validation des modifications 3 ###modifications....  
18  connection.rollback() #Retour à l'état du dernier commit, les modifications effectuées depuis sont perdues  
19
```

6.Exécuter une requête

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



```
6 |  
7 cursor=connection.cursor()  
8
```

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

```
1 ###Exécution unique  
2 curseur.execute(''CREATE TABLE IF NOT EXISTS scores(  
3     id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,  
4     pseudo TEXT,  
5     valeur INTEGER  
6 )''')
```


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` :

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

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

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 (?, ?)",
4     donnees)
5 connexion.commit() #Ne pas oublier de valider les modifications
```

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)",
8     donnees)
9 connexion.commit() #Ne pas oublier de valider les modifications
```

Exécuter un script

il est aussi possible d'exécuter un script directement à l'aide de la méthode `executescript`. 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
```

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

```
1 import sqlite3
2
3 #Connexion
4 connexion = sqlite3.connect('basededonnees.db')
5
6 #Récupération d'un curseur
7 curseur = connexion.cursor()
8
9 #Création de la table scores
10 curseur.execute("""
11     CREATE TABLE IF NOT EXISTS scores(
12         id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
13         pseudo TEXT,
14         valeur INTEGER);
15     """)
16
17 #Suppression des éléments de scores
18 curseur.execute("""DELETE FROM scores""")
19
20 #Préparation des données à ajouter
21 donnees = [
22     ("toto", 1000),
23     ("tata", 750),
24     ("titi", 500),
25     ("toto", 250),
26     ("tata", 150),
27     ("tete", 0)
28 ]
29
30 #Insertion des données
31 curseur.executemany('INSERT INTO scores (pseudo, valeur) VALUES (?, ?)',
32     donnees)
33
34 #Validation
35 connexion.commit()
```

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



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.

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

```
1 donnee = ("tata", )
2 curseur.execute("SELECT valeur FROM scores WHERE pseudo = ?",
    donnee)
3 result = curseur.fetchone()
4 while result:
5     print(result)
6     result = curseur.fetchone()
7 ###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

```
1 print(curseur.arraysize) #Affiche "1"
2 donnee = (400, )
3
4 curseur.execute("SELECT pseudo FROM scores WHERE valeur > ?",
    donnee)
5 print(curseur.fetchmany()) #Affiche "[('toto',)]"
6 print(curseur.fetchmany()) #Affiche "[('tata',)]"
7
8 curseur.execute("SELECT pseudo FROM scores WHERE valeur > ?",
    donnee)
9 print(curseur.fetchmany(2)) #Affiche "[('toto',), ('tata',)]"
```

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.

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

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

Les deux codes ont le même effet et affichent

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

Récupérer quelques informations

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

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.

```
1 ###modifications...
2 print(connexion.in_transaction) #Affiche "True"
3 connexion.commit()
4 print(connexion.in_transaction) #Affiche "False"
```

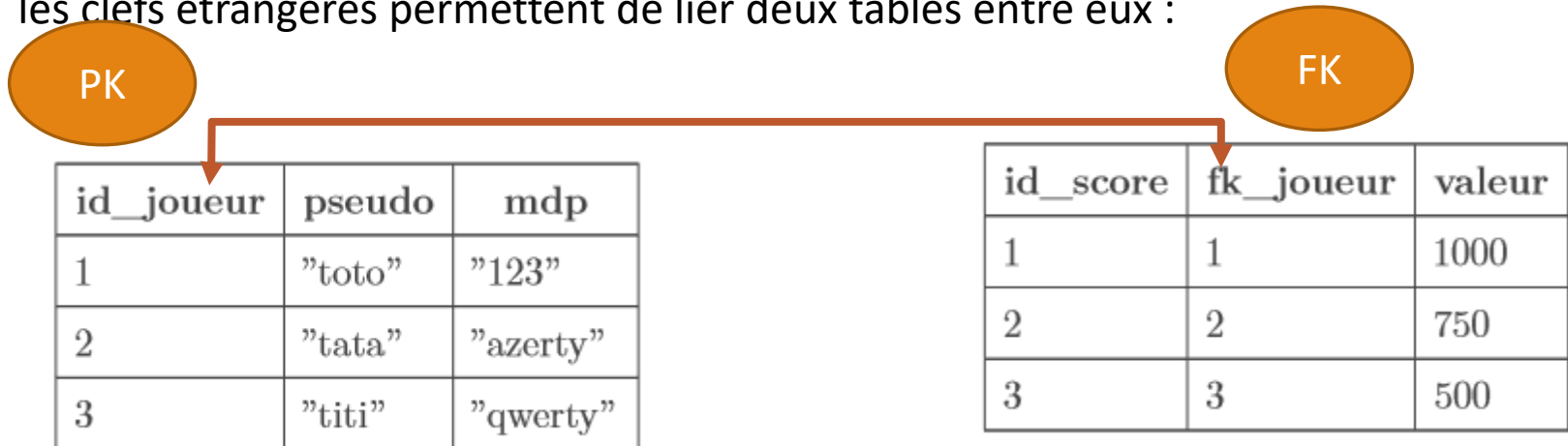

les clefs étrangères

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

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

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

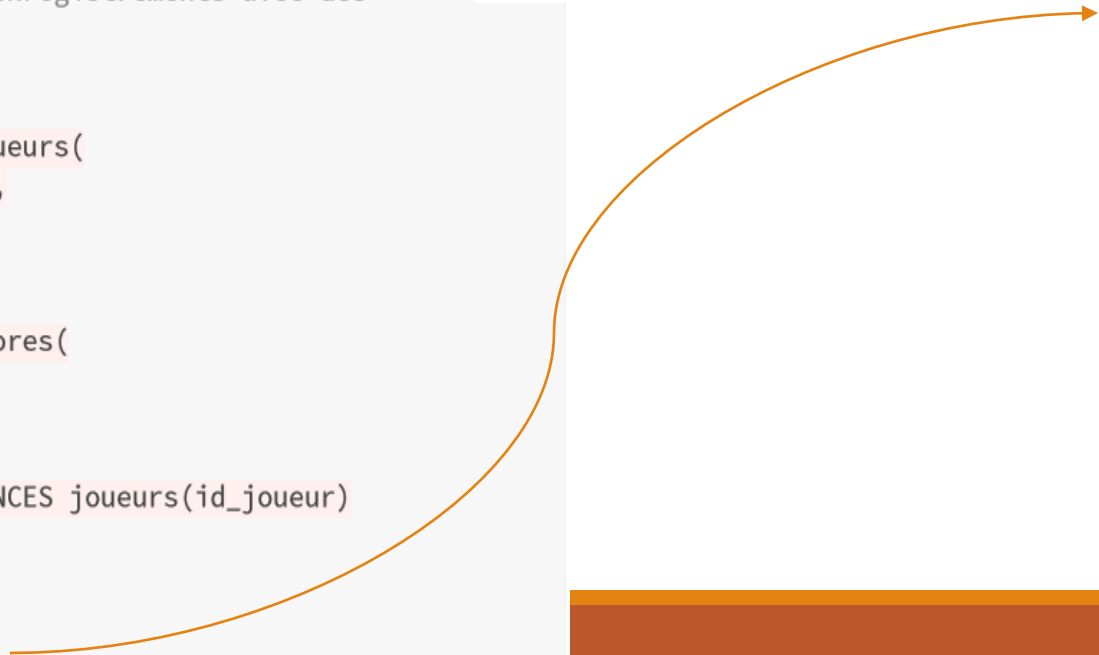
les clefs étrangères permettent de lier deux tables entre eux :



les clefs étrangères code SQL

```
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 = [
```



les clefs étrangères code SQL

```
39     (1, 1000),
40     (2, 750),
41     (3, 500)
42 ]
43
44 #Insertion des données dans table joueur puis score
45 curseur.executemany("INSERT INTO joueurs (pseudo, mdp) VALUES (?, ?)",
46     donnees_joueur)
47
48 #Validation des ajouts
49 connexion.commit()
50
51 #Affichage des données
52 for joueur in curseur.execute("SELECT * FROM joueurs"):
53     print("joueur :", joueur)
54
55 for score in curseur.execute("SELECT * FROM scores"):
56     print("score :", score)
57
58 #Déconnexion
59 connexion.close()
```