Vk

Zeste de savoir

Des bases de données en Python avec sqlite3

samedi 05 octobre 2024

Table des matières

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.

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 3.12 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.

> ***i***
>
> **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 ! 🧙

1. Fonctionnalités de base

Introduction

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.

1.1. 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
```

1.1.1. 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.

1.1.2. 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
```

1.1.3. 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.

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
```

1.2.1. 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
```

1.2.2. 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.

1.2.3. 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:
    curseur.execute(
        "INSERT INTO scores (pseudo, valeur) VALUES (?, ?)",
        donnee)
connexion.commit() # Ne pas oublier de valider les modifications
```

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

```
donnees = [("toto", 1000), ("tata", 750), ("titi", 500)]
# Exécutions multiples
curseur.executemany(
    "INSERT INTO scores (pseudo, valeur) VALUES (?, ?)",
    donnees)
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 :

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

1.2.4. Exécuter un script

Enfin, 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.

```
# Exécution d'un script
curseur.executescript("""
    DROP TABLE IF EXISTS scores;

```

```
    CREATE TABLE scores(
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
    pseudo TEXT,
    valeur INTEGER);

    INSERT INTO scores(pseudo, valeur) VALUES ("toto", 1000);
    INSERT INTO scores(pseudo, valeur) VALUES ("tata", 750);
    INSERT INTO scores(pseudo, valeur) VALUES ("titi", 500)
""")
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.

1.3. 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

1.3.1. 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,)"
```

1.3.2. 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.

1.3.3. 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)
```

1.4. Récupérer quelques informations

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

1.4.1. 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"
```

1.4.2. 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"
```

1.4.3. 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")
```

```
15 print(curseur.rowcount) # Affiche "9" (0 si version SQLite < 3.6.5)
```

1.4.4. 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* :

```
1 from random import randint
2
3 # ...
4
5 donnee = (randint(1, 1000), "toto", 1000)
6 print(donnee[0]) # Affiche "589"
7 curseur.execute(
      "INSERT INTO scores (id, pseudo, valeur) VALUES (?, ?, ?)",
      donnee)
8 curseur.execute("SELECT * FROM scores WHERE id = ?",
      (curseur.lastrowid, ))
9 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.

1.5. Utiliser des clefs étrangères

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

1.5.1. 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 :

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

1.5.2. 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 .

Conclusion

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.executescript("""
    DROP TABLE IF EXISTS scores;

    CREATE TABLE 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),
```

```
28      ("tata", 150),
29      ("tete", 0)
30      ]
31
32  # Insertion des données
33  curseur.executemany(
        "INSERT INTO scores (pseudo, valeur) VALUES (?, ?)",
        donnees)
34
35  # Validation
36  connexion.commit()
37
38  # Parcours des enregistrements....
39
40  # Déconnexion
41  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      DROP TABLE IF EXISTS scores;
17      DROP TABLE IF EXISTS joueurs;
18
19      CREATE TABLE joueurs(
20      id_joueur INTEGER PRIMARY KEY,
21      pseudo TEXT,
22      mdp TEXT);
23
24      CREATE TABLE scores(
```

```
    id_score INTEGER PRIMARY KEY,
    fk_joueur INTEGER NOT NULL,
    valeur INTEGER,
    FOREIGN KEY(fk_joueur) REFERENCES joueurs(id_joueur)
    ON DELETE CASCADE);
""")

# Préparation des données
donnees_joueur = [
    ("toto", "123"),
    ("tata", "azerty"),
    ("titi", "qwerty")
]
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.

2. Fonctionnalités avancées

Introduction

Au fil de cette partie nous allons explorer des fonctionnalités plus avancées de sqlite3 : comment gérer les exceptions, ajouter ses propres types de données ou encore créer une copie sauvegardée.

2.1. Gérer les exceptions

Lors de notre utilisation de sqlite3, nous pouvons rencontrer des exceptions.

2.1.1. Mécanismes de gestion des erreurs

Celles-ci se gèrent comme il est coutume de faire en Python (`try ... except ... finally ...`) :

```
1  import sqlite3
2
3  connexion = sqlite3.connect('basededonnees.db')
4
5  curseur = connexion.cursor()
6
7  curseur.execute("""CREATE TABLE IF NOT EXISTS livres(
8      id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
9      titre TEXT,
10     nombre_pages INTEGER
11 )""")
12
13 # Insertion données avec erreur d'intégrité puis sans
14 try:
15     donnees = (
16         {"id": 1, "titre": "Les Raisins de la colère",
               "nombre_pages": 640},
17         {"id": 1, "titre": "Rhinocéros", "nombre_pages": 246},
18     )
19     curseur.executemany(
```

```
20          ⌋
                "INSERT INTO livres (id, titre, nombre_pages) VALUES (:id, :titre,
                ,
21          donnees
22      )
23      connexion.commit()
24  except sqlite3.IntegrityError:
25      print("Erreur d'intégrité") # Affichage "Erreur d'intégrité"
26      connexion.rollback()
27      curseur.executemany(
28          ⌋
                "INSERT INTO livres (titre, nombre_pages) VALUES (:titre, :nombre_
                ,
29          donnees
30      )
31      connexion.commit()
32      print(curseur.rowcount) # Affichage "2"
33
34
35  connexion.close()
```

2.1.2. Erreurs disponibles

Voici un listing des classes d'erreur disponibles :

Classe d'erreur	Classe parente	Des
sqlite3.Warning	Exception	Une mod
sqlite3.Error	Exception	L'exc tions
sqlite3.InterfaceError	sqlite3.Error	Exce lisati
sqlite3.DatabaseError	sqlite3.Error	Exce BDD
sqlite3.DataError	sqlite3.DatabaseError	Exce valeu exem
sqlite3.OperationalError	sqlite3.DatabaseError	Exce la B lectu
sqlite3.IntegrityError	sqlite3.DatabaseError	Exce tégri par e
sqlite3.InternalError	sqlite3.DatabaseError	Exce cas d

sqlite3.ProgrammingError	sqlite3.DatabaseError	Exce tion thod
sqlite3.NotSupportedError	sqlite3.DatabaseError	Erre par l

L'objet `sqlite3.Error` possède deux attributs pour en apprendre plus sur l'erreur (`sqlite_errorcode` pour le code d'erreur et `sqlite_errorname` pour le nom de l'erreur).

Dans la documentation, il est parfois précisé les exceptions pouvant être levées pour certaines méthodes.

2.2. Utiliser ses propres fabriques

La bibliothèque met à disposition des moyens d'élaborer ses propres fabriques.

2.2.1. Fabrique de ligne

De base, sqlite3 présente une ligne de donnée lue sous forme de tuple.

Nous pouvons personnaliser ce comportement avec une fabrique de ligne (*row factory*). En effet, un objet de type *Connection* ainsi qu'un objet de type *Cursor* possèdent tous les deux un attribut `row_factory` dans ce sens. Il est alors préférable d'utiliser l'objet de type *Connection* si nous voulons que tous les curseurs créés par la suite soient impactés.

2.2.1.1. Classe Row

Nous pourrions vouloir accéder aux données via des clefs.

Justement, il y a une classe `Row` pour cela. Avec celle-ci, les données lues sont accessibles par positions ainsi que par clefs (peu importe la casse d'ailleurs). Il y a même une méthode `keys`.

```
1  import sqlite3
2
3  connexion = sqlite3.connect('basededonnees.db')
4
5  # Utilisation de la classe Row pour la fabrique de ligne
6  connexion.row_factory = sqlite3.Row
7
8  curseur = connexion.cursor()
9
10 curseur.execute("SELECT * FROM livres")
11
12 ligne = curseur.fetchone()
```

```
13
14 print(ligne[1]) # Affiche "Les Raisins de la colère"
15 print(ligne.keys()) # Affiche "['id', 'titre', 'nombre_pages']"
16 print(ligne['tItrE']) # Affiche "Les Raisins de la colère"
17
18 connexion.close()
```

2.2.1.2. Fonction personnalisée

Nous pourrions vouloir obtenir la ligne sous forme de dictionnaire ou autre. Il suffit alors de créer une fonction, prenant le curseur et les données, qui va formater la ligne comme voulue, puis de l'associer à l'attribut `row_factory` :

```
1  import sqlite3
2
3  # Définition fonction pour fabrique de ligne
4  def dict_factory(curseur, ligne):
5      noms_colonnes = [colonne[0] for colonne in curseur.description]
6      return {key: value for key, value in zip(noms_colonnes, ligne)}
7
8  connexion = sqlite3.connect('basededonnees.db')
9
10 # Utilisation de la fonction définie pour la fabrique de ligne
11 connexion.row_factory = dict_factory
12
13 curseur = connexion.cursor()
14
15 curseur.execute("SELECT * FROM livres")
16
17 ligne = curseur.fetchone()
18
19 print(type(ligne)) # Affiche "<class 'dict'>"
20 print(ligne) # Affiche "{'id': 1, 'titre': 'Les Raisins de la
       colère', 'nombre_pages': 640}"
21
22 connexion.close()
```

2.2.2. Fabrique de texte

Dans la première partie de ce cours, nous avons dit que la correspondance pour un texte SQL était une chaine `str` par défaut. Nous avons la possibilité d'interférer dans cette transposition via une fabrique de texte (*text factory*).

Cela est parfois nécessaire. En effet, cette correspondance fonctionne bien dans le cas d'un encodage `utf-8`, mais ce n'est pas toujours le cas quand la base de données utilise d'autres encodages comme expliqué dans la documentation.

L'objet de type *Connection* possède un attribut `text_factory` pour gérer cela :

```
# Fabrique de texte via fonction
connexion.text_factory = lambda valeur_texte: str(
        valeur_texte,
        encoding="utf-8",
        errors="gestion_erreur"
)
```

2.3. Ajouter ses propres types

En plus des correspondances de base, il est possible d'ajouter des adaptateurs pour convertir ses propres valeurs dans un type accepté par SQLite, mais aussi des convertisseurs pour l'opération inverse.

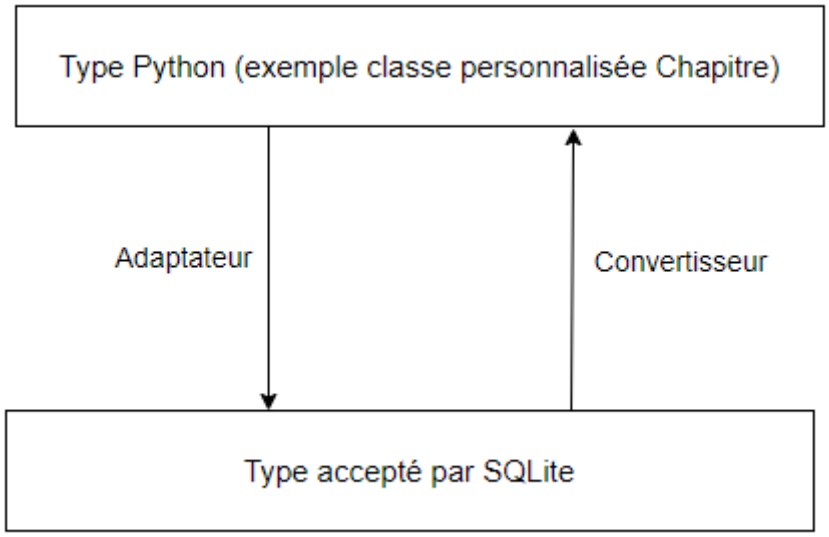

FIGURE 2.1. – Schéma adaptateur et convertisseur.

La documentation liste quelques exemples concernant les dates notamment.

Pour les besoins de cette section, nous commençons par créer notre classe.

```
class Chapitre:
    def __init__(self, numero, titre):
        self.numero, self.titre = numero, titre
```

2.3.1. Écriture en base avec adaptateur

Maintenant, nous allons ajouter une table en faisant simple :

```
curseur.execute("""CREATE TABLE IF NOT EXISTS chapitres(
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
    chapitre TEXT
)""")
```

Pour adapter notre objet à un type supporté par SQLite (d'un objet chapitre à un `TEXT` ici), il y a deux possibilités.

2.3.1.1. Utiliser un objet qui s'adapte

La première consiste à faire en sorte que l'objet soit adaptable en lui implémentant une méthode spéciale `__conform__`. Cette méthode doit vérifier que le protocole fourni en paramètre est bien `sqlite3.PrepareProtocol` pour retourner une valeur d'un type accepté par SQLite :

```
class Chapitre:
    def __init__(self, numero, titre):
        self.numero, self.titre = numero, titre

    # Implémentation de la méthode spéciale __conform__
    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return f"{self.numero};{self.titre}"

curseur.execute("INSERT INTO chapitres(chapitre) VALUES (?)",
    (Chapitre(1, "Préface"), ))
connexion.commit()
curseur.execute("SELECT * FROM chapitres ORDER BY id DESC LIMIT 1")
print(curseur.fetchone()) # Affiche "(1, '1;Préface')"

connexion.close()
```

2.3.1.2. Enregistrer une fonction d'adaptation

La seconde possibilité est de d'abord créer une fonction prenant l'objet et faisant cette conversion. Puis ensuite d'enregistrer celle-ci avec la fonction `register_adapter` du module en indiquant la classe et la fonction d'adaptation :

```
class Chapitre:
    def __init__(self, numero, titre):
        self.numero, self.titre = numero, titre

# Définition d'une fonction d'adaptation
def adapt_chapitre(chapitre):
    return f"{chapitre.numero};{chapitre.titre}"

# Enregistrement de la fonction d'adaptation
sqlite3.register_adapter(Chapitre, adapt_chapitre)

curseur.execute("INSERT INTO chapitres(chapitre) VALUES (?)",
    (Chapitre(20, "Mot de fin"), ))
connexion.commit()
curseur.execute("SELECT * FROM chapitres ORDER BY id DESC LIMIT 1")
print(curseur.fetchone()) # Affiche "(2, '20;Mot de fin')"

connexion.close()
```

2.3.2. Lecture de base avec convertisseur

Au contraire, pour construire un type personnalisé à partir d'un type SQLite, il faut passer par un convertisseur.

2.3.2.1. Fonction de conversion

C'est un peu le même principe qu'avec une fonction d'adaptation. Nous devons créer une fonction qui va convertir l'objet de type `bytes` passé en un objet personnalisé voulu puis le retourner.

Il faut ensuite enregistrer cette fonction définie via `register_converter` en indiquant un nom sous forme de chaîne de caractères et la fonction.

```
# Définition d'une fonction de conversion
def convert_chapitre(s):
    valeurs = s.split(b";", 1)
    return Chapitre(int(valeurs[0]), valeurs[1].decode("utf-8"))

# Enregistrement de la fonction de conversion
sqlite3.register_converter("chapitre", convert_chapitre)
```

2.3.2.2. Détection de type

Pour que cela fonctionne, il faut également paramétrer la détection de type au niveau de l'objet de type `Connection` (désactivé par défaut).

Le constructeur de celui-ci peut prendre une valeur `detect_types`, soit `PARSE_DECLTYPES` (l'indication de conversion se fait au niveau du type déclaré dans la table), soit `PARSE_COLNAMES` (l'indication de conversion se fait au niveau du nom de colonne), soit les deux séparés par une barre verticale (dans ce cas-là c'est `PARSE_COLNAMES` qui prend la priorité).

2.3.2.2.1. PARSE_DECLTYPES

Avec la constante `PARSE_DECLTYPES` de sqlite3, nous indiquons la conversion de type au niveau de la création de la table. Ce nom de type doit correspondre avec le nom utilisé pour l'enregistrement de la fonction bien entendu.

Voici un exemple complet :

```
import sqlite3

# Connexion en renseignant detect_types à sqlite3.PARSE_DECLTYPES
connexion = sqlite3.connect('basededonnees.db',
    detect_types=sqlite3.PARSE_DECLTYPES)

curseur = connexion.cursor()

class Chapitre:
    def __init__(self, numero, titre):
        self.numero, self.titre = numero, titre

def adapt_chapitre(chapitre):
    return f"{chapitre.numero};{chapitre.titre}"

def convert_chapitre(s):
    valeurs = s.split(b";", 1)
    return Chapitre(int(valeurs[0]), valeurs[1].decode("utf-8"))

sqlite3.register_adapter(Chapitre, adapt_chapitre)
sqlite3.register_converter("chapitre", convert_chapitre)

# Création de la table en indiquant le type pour la conversion
curseur.executescript("""
    DROP TABLE IF EXISTS chapitres;

    CREATE TABLE chapitres(
        id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
        chap chapitre
    );
""")

curseur.execute("INSERT INTO chapitres(chap) VALUES (?)",
    (Chapitre(20, "Mot de fin"), ))

curseur.execute("SELECT chap FROM chapitres")
chapitre = curseur.fetchone()[0]
```

```
print(chapitre.numero, chapitre.titre) # Affiche "20 Mot de fin"

connexion.close()
```

2.3.2.2.2. PARSE_COLNAMES Avec la constante `PARSE_COLNAMES` de sqlite3, nous indiquons la conversion de type au niveau d'utilisation de la colonne de la table via la syntaxe `nom_colonne AS 'nom_colonne [nom_conversion]'`. Ce nom de type doit correspondre avec le nom utilisé pour l'enregistrement de la fonction bien entendu.

Voici un exemple complet :

```
import sqlite3

# Connexion en renseignant detect_types à sqlite3.PARSE_COLNAMES
connexion = sqlite3.connect('basededonnees.db',
    detect_types=sqlite3.PARSE_COLNAMES)

curseur = connexion.cursor()

class Chapitre:
    def __init__(self, numero, titre):
        self.numero, self.titre = numero, titre

def adapt_chapitre(chapitre):
    return f"{chapitre.numero};{chapitre.titre}"

def convert_chapitre(s):
    valeurs = s.split(b";", 1)
    return Chapitre(int(valeurs[0]), valeurs[1].decode("utf-8"))

sqlite3.register_adapter(Chapitre, adapt_chapitre)
sqlite3.register_converter("chapitre", convert_chapitre)

curseur.executescript("""
    DROP TABLE IF EXISTS chapitres;

    CREATE TABLE chapitres(
        id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
        chap
    );
""")

curseur.execute("INSERT INTO chapitres(chap) VALUES (?)",
    (Chapitre(20, "Mot de fin"), ))

# Requête en indiquant le type de colonne pour la conversion
curseur.execute("SELECT chap AS 'chap [chapitre]' FROM chapitres")
```

```
35 chapitre = curseur.fetchone()[0]
36 print(chapitre.numero, chapitre.titre) # Affiche "20 Mot de fin"
37
38 connexion.close()
```

Remarquons que dans ce cas, nous n'avons même pas à spécifier de type lors de la création de la table pour les colonnes à convertir.

? Que se passe-t-il si l'objet chapitre contient des caractères spécifiques à SQL ? Est-ce qu'il y a un risque d'injection SQL ? 🍊

Je vous rassure : non ! Comme nous utilisons le système de *placeholders* de la bibliothèque (les fameux `?` dans la requête SQL sont une approche possible), le pilote va se charger de protéger notre requête des injections.

2.4. Créer une copie sauvegardée

Il peut être judicieux voire nécessaire d'effectuer des sauvegardes.

2.4.1. Dans un fichier SQL

La méthode `iterdump` de notre objet de connexion permet de parcourir la base de données et d'en extraire la représentation SQL, ligne par ligne, à écrire dans un fichier :

```
1 # Exemple dump avec connexion.iterdump()
2 with open('dump.sql', 'w') as f:
3     for ligne in connexion.iterdump():
4         f.write('%s\n' % ligne)
```

Les instructions SQL du fichier pourront ensuite être utilisées pour recréer l'état de la base par exemple.

2.4.2. Dans une autre base de données

Pour faire un copier-coller dans une autre base directement, nous pouvons utiliser la méthode `backup` d'un objet de type *Connection*. Nous devons lui fournir une connexion vers la destination.

```
1 # Backup d'une BDD source vers une BDD destination
2 source = sqlite3.connect('basededonnees.db')
3 destination = sqlite3.connect('backup.db')
```

```
source.backup(destination)
destination.close()
source.close()
```

Il est possible de passer des valeurs pour choisir le pas de page à copier au fur et à mesure (`page`) ou encore avoir un retour de la progression au fur et à mesure (`progress`).

2.5. Simplifier son code

Découvrons maintenant quelques autres façons d'utiliser la connexion.

2.5.1. Se passer d'un curseur

En faisant appel directement aux méthodes d'exécution depuis un objet de connexion, les curseurs sont implicitement créés et retournés sans devoir le faire explicitement :

```
import sqlite3

connexion = sqlite3.connect(":memory:")

# Exécution via la méthode de l'objet de connexion directement
connexion.executescript("""CREATE TABLE IF NOT EXISTS livres(
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
    titre TEXT,
    nombre_pages INTEGER
)""")

donnees = (
    {"titre": "Les Raisins de la colère", "nombre_pages": 640},
    {"titre": "Rhinocéros", "nombre_pages": 246},
)
connexion.executemany(
    "INSERT INTO livres (titre, nombre_pages) VALUES (:titre, :nombre_pages)",
    donnees
)

# Itération via le curseur implicite
for ligne in connexion.execute("select * from livres"):
    print(ligne)
    # (1, 'Les Raisins de la colère', 640)
    # (2, 'Rhinocéros', 246)

connexion.close()
```

2.5.2. Choisir ou non l'auto-validation

Depuis la version 3.12 de Python, sqlite3 recommande de gérer les transactions en renseignant `autocommit` de la connexion à `False`. Il y a soit `True`, soit `False`, soit `sqlite3.LEGACY_TRANSACTION_CONTROL` (qui est la valeur par défaut historique).

C'est donc assez nouveau et il est prévu que la valeur par défaut passe de `sqlite3.LEGACY_TRANSACTION_CONTROL` à `False` dans une version ultérieure du module.

```
import sqlite3
con = sqlite3.connect("basededonnees.db", autocommit=False) #
    False recommandé à partir de Python 3.12
```

? Qu'est-ce que cela change ?

En l'état, pas grand chose entre `sqlite3.LEGACY_TRANSACTION_CONTROL` et `False`. Le module gère les transactions et il faut valider (au risque de perdre ses modifications à la fermeture de la connexion) ou annuler ses modifications explicitement.

Avec `True`, les méthodes `commit` et `rollback` n'ont aucun effet : les modifications sont validées au fur et à mesure automatiquement.

2.5.3. Utiliser le gestionnaire de contexte

Un objet de type `Connection` peut être utilisé avec le gestionnaire de contexte `with`.

Avec l'auto-validation désactivée, cela permet, une fois la fin du corps du gestionnaire de contexte atteint, de valider si tout se passe bien ou au contraire d'annuler si erreur puis d'ouvrir une nouvelle transaction. Le tout de manière automatique.

```
import sqlite3

# Auto-validation désactivée (recommandé à partir de Python 3.12)
connexion = sqlite3.connect("basededonnees.db", autocommit=False)

try:
    with connexion:
        connexion.execute(
            "INSERT INTO livres (titre, nombre_pages) VALUES (:titre, :nomb
            ,
            {"titre": "Le Petit Prince", "nombre_pages": 93}
        )
        # Si aucune erreur alors commit() automatique et ouverture
            d'une nouvelle transaction
```

```
13          # Si erreur alors rollback() automatique et ouverture
                d'une nouvelle transaction
14  except sqlite3.Error as e:
15      print(e.sqlite_errorcode,  e.sqlite_errorname)
16
17  connexion.close()
```

Conclusion

Voilà, vous connaissez maintenant des usages plus avancés de sqlite3 !

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.

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

— la documentation
— 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, nohar et germinolegrand pour leurs retours.
Merci à artragis pour la validation.