Accéder à une base SQLite avec Python

Piloter ecole.db depuis un programme Python avec le module sqlite3 de la bibliothèque standard : connexion, requêtes, écriture, transactions — et la règle de sécurité qui sépare un programme correct d'une porte ouverte.

5TTR 6TTR Découverte

Jusqu'ici, tu tapais tes requêtes à la main dans DB Browser. Un vrai programme doit les exécuter tout seul, avec des valeurs qui viennent de l'utilisateur. C'est là que le module `sqlite3` entre en jeu — et avec lui, une règle de sécurité qu'on n'enfreint jamais.

🎯 Objectifs

À la fin de ce cours, tu seras capable de :

1. **Ouvrir** une base SQLite depuis Python et **fermer** proprement la connexion.
2. **Exécuter** une requête et **récupérer** les résultats sous plusieurs formes.
3. **Passer des valeurs** à une requête avec des paramètres — et expliquer pourquoi jamais autrement.
4. **Reconnaître** une injection SQL et montrer ce qu'elle permet.
5. **Écrire** dans la base : `INSERT`, `commit`, `executemany`.
6. **Gérer** les erreurs de contrainte et comprendre ce que fait le bloc `with`.

Le module est déjà là

`sqlite3` fait partie de la **bibliothèque standard** de Python : rien à installer, rien à télécharger.

```
import sqlite3
```

C'est possible parce que SQLite est une bibliothèque : Python l'embarque directement, il n'y a aucun serveur à joindre. Pour un système de bases de données fonctionnant en serveur, comme MySQL, il faudrait installer un pilote **et** disposer du serveur lui-même.

Se connecter et exécuter

```
import sqlite3

con = sqlite3.connect("ecole.db")
con.execute("PRAGMA foreign_keys = ON")

cur = con.execute("SELECT nom, annee FROM classe ORDER BY annee")
for ligne in cur:
    print(ligne)

con.close()
```

```
('3TTR', 3)
('4TTR', 4)
('5TTR', 5)
('6TTR', 6)
```

Trois points à relever :

- `connect()` **crée le fichier s'il n'existe pas.** Une faute de frappe dans le nom ne provoque donc aucune erreur : tu obtiens une base vide et des requêtes qui échouent sur des tables absentes. Si ton programme ne trouve « plus rien », vérifie d'abord le chemin.
- `PRAGMA foreign_keys = ON` **est à refaire ici.** Le réglage coché dans DB Browser ne concernait que DB Browser. Chaque connexion repart avec les clés étrangères désactivées. Mets cette ligne juste après chaque `connect()`, systématiquement.
- **Chaque ligne est un tuple.** On y accède par index : `ligne[0]`, `ligne[1]`.

Récupérer les résultats

Une ligne, toutes les lignes

```python
cur = con.cursor()

cur.execute("SELECT COUNT(*) FROM eleve")
print(cur.fetchone())        # (33,)   ← un tuple d'une seule valeur
print(cur.fetchone()[0])     # 33      ← la valeur elle-même

cur.execute("SELECT nom, prenom FROM eleve WHERE classe_id = 4 ORDER BY nom")
print(cur.fetchall())
```

```
[('Bruyère', 'Victor'), ('Collard', 'Maya'), ('Dethier', 'Gaspard'), ...]
```

Méthode	Renvoie	Quand l'utiliser
`fetchone()`	La ligne suivante, ou `None`	Un résultat unique : un compte, une recherche par identifiant
`fetchall()`	La liste de toutes les lignes restantes	Résultat court, à manipuler ensuite
Boucle `for` sur le curseur	Les lignes une par une	Gros résultats — rien ne s'accumule en mémoire

> ⚠️ **`fetchone()` sur un `COUNT` renvoie un tuple, pas un nombre.** L'oubli du `[0]` est l'erreur classique : `print(f"{cur.fetchone()} élèves")` affiche `(33,) élèves`.

Accéder aux colonnes par leur nom

Les tuples deviennent illisibles dès quatre colonnes : personne ne sait ce que `ligne[3]` désigne. La solution tient en une ligne :

```python
con.row_factory = sqlite3.Row

ligne = con.execute("SELECT id, nom, prenom FROM eleve WHERE id = 1").fetchone()

print(ligne["nom"], ligne["prenom"])   # Adam Lucas
print(ligne[1])                        # Adam        ← l'index marche toujours
print(ligne.keys())                    # ['id', 'nom', 'prenom']
print(dict(ligne))                     # {'id': 1, 'nom': 'Adam', 'prenom': 'Lucas'}
```

Ça ne coûte rien et ça rend le code compréhensible. **Prends-en l'habitude.**

Les paramètres : la partie à ne pas rater

La mauvaise façon

Tu veux chercher les élèves d'une classe choisie par l'utilisateur. La tentation :

```python
# ⛔ NE FAIS JAMAIS ÇA
nom = input("Nom à chercher : ")
requete = f"SELECT COUNT(*) FROM eleve WHERE nom = '{nom}'"
resultat = con.execute(requete).fetchone()[0]
```

Ça fonctionne. Tant que l'utilisateur tape un nom normal.

Ce qui se passe quand il n'en tape pas

Saisis ceci : `x' OR '1'='1`

La chaîne construite devient :

```
SELECT COUNT(*) FROM eleve WHERE nom = 'x' OR '1'='1'
```

Le `'` saisi a **fermé** la chaîne de la requête, et la suite est devenue du **code SQL**. La condition `'1'='1'` est toujours vraie, donc la requête renvoie **toute la table** :

```
avec f-string      : 33 lignes     ← toute la base
avec place-holder  : 0 lignes      ← aucun élève ne s'appelle ainsi
```

C'est une **injection SQL**. Ici elle ne fait que compter des lignes ; la même faille permet, selon la requête, de lire les données d'autres tables, de contourner un mot de passe, ou de supprimer une table entière.

⚠️ **LA RÈGLE, SANS EXCEPTION.** Une valeur qui vient de l'extérieur — saisie clavier, formulaire web, fichier, capteur, API — **ne se colle jamais dans une requête**. Ni avec `+`, ni avec une f-string, ni avec `.format()`, ni avec `%`.

La bonne façon : les place-holders

```python
classe = 4
cur = con.execute("SELECT nom, prenom FROM eleve WHERE classe_id = ?", (classe,))
```

Le `?` marque un emplacement ; les valeurs arrivent **à part**, dans un tuple. SQLite reçoit alors deux choses distinctes : d'un côté la requête, de l'autre les données. Une valeur reste une valeur, même si elle contient des apostrophes ou des mots-clés SQL — elle ne peut plus devenir du code.

💡 **LA VIRGULE QUI COMPTE.** `(classe)` n'est pas un tuple, c'est juste un nombre entre parenthèses. Le tuple d'un seul élément s'écrit `(classe,)`. Sans cette virgule : `Parameters are of unsupported type`.

Variante avec des paramètres nommés, plus lisible quand il y en a plusieurs :

```python
cur = con.execute(
    "SELECT nom FROM eleve WHERE classe_id = :classe AND nom LIKE :debut",
    {"classe": 4, "debut": "D%"})
print(cur.fetchall())        # [('Dethier',)]
```

⚠️ **UN PLACE-HOLDER REMPLACE UNE *VALEUR*, JAMAIS UN NOM DE TABLE OU DE COLONNE.** `ORDER BY ?` ne fonctionne pas. Si l'utilisateur choisit la colonne de tri, valide sa saisie contre une **liste blanche** écrite dans ton code :

```python
colonnes_permises = {"nom", "prenom", "date_naissance"}
if tri not in colonnes_permises:
    raise ValueError("colonne de tri invalide")
requete = f"SELECT * FROM eleve ORDER BY {tri}"   # sûr : tri vient de ta liste
```

Écrire dans la base

`INSERT` et `commit`

```python
cur = con.execute(
    "INSERT INTO eleve (nom, prenom, date_naissance, classe_id) VALUES (?, ?, ?, ?)",
    ("Martin", "Sacha", "2008-05-02", 4))

print("nouvel id :", cur.lastrowid)      # 34
con.commit()
```

- `commit()` **est obligatoire.** Sans lui, la modification reste dans une transaction ouverte et disparaît à la fermeture du programme. « Mon insertion ne marche pas » signifie neuf fois sur dix « j'ai oublié le `commit` ».
- `lastrowid` donne l'identifiant que SQLite vient d'attribuer — indispensable pour créer ensuite les lignes liées.

Plusieurs lignes d'un coup

```python
inscriptions = [(34, 5), (34, 6), (34, 7)]
con.executemany("INSERT INTO inscription (eleve_id, cours_id) VALUES (?, ?)", inscriptions)
con.commit()
```

`executemany` prend une **liste** de tuples et exécute la requête pour chacun. C'est plus court et nettement plus rapide qu'une boucle Python autour de `execute`.

Erreurs et transactions

Les deux erreurs qu'on rencontre

```
try:
    con.execute("INSERT INTO eleve (nom, prenom, classe_id) VALUES (?, ?, ?)",
                ("Test", "X", 999))
    con.commit()
except sqlite3.IntegrityError as e:
    print("IntegrityError :", e)      # FOREIGN KEY constraint failed

try:
    con.execute("SELECT * FROM eleves")          # faute de frappe
except sqlite3.OperationalError as e:
    print("OperationalError :", e)    # no such table: eleves
```

Exception	Cause	Que faire
`sqlite3.IntegrityError`	Une **contrainte** refuse la donnée : clé étrangère, `UNIQUE`, `NOT NULL`, `CHECK`	La base te protège : afficher un m
`sqlite3.OperationalError`	Un problème de **requête ou de fichier** : table inconnue, SQL invalide, `database is locked`	C'est un bug de ton programme :

🧠 **UNE `INTEGRITYERROR` EST UNE BONNE NOUVELLE.** Elle prouve que tes contraintes fonctionnent et qu'une donnée incohérente vient d'être refusée. Le vrai danger, c'est la base qui accepte tout — comme celle où le pragma `foreign_keys` a été oublié.

Ce que `with` fait — et ne fait pas

```
try:
    with con:
        con.execute("INSERT INTO classe (nom, annee) VALUES ('TEST', 3)")
        con.execute("INSERT INTO classe (nom, annee) VALUES ('TEST', 4)")   # viole UNIQUE
except sqlite3.IntegrityError as e:
    print("erreur attrapée :", e)          # UNIQUE constraint failed: classe.nom

print(con.execute("SELECT COUNT(*) FROM classe WHERE nom='TEST'").fetchone()[0])
```

```
erreur attrapée : UNIQUE constraint failed: classe.nom
0     ← la PREMIÈRE insertion a été annulée elle aussi
```

Le bloc `with con:` forme une **transaction** : si tout se passe bien il `commit`, si une exception survient il annule **tout le bloc**. C'est exactement ce qu'on veut quand plusieurs écritures forment un ensemble indissociable — créer un élève *et* ses inscriptions.

⚠️ **`WITH CON:` NE FERME PAS LA CONNEXION.** Contrairement à `with open(...)` pour les fichiers, la connexion reste ouverte après le bloc. Il faut toujours un `con.close()`. C'est une différence que beaucoup de tutoriels passent sous silence.

Fermer, quoi qu'il arrive

```
def moyennes_par_classe(chemin):
    con = sqlite3.connect(chemin)
    con.row_factory = sqlite3.Row
    try:
        return con.execute("""
            SELECT c.nom AS classe,
                   COUNT(DISTINCT e.id) AS nb_eleves,
                   ROUND(AVG(n.valeur), 2) AS moyenne
            FROM classe c
            JOIN eleve e ON e.classe_id = c.id
            JOIN note  n ON n.eleve_id  = e.id
            GROUP BY c.id
            ORDER BY moyenne DESC
        """).fetchall()
    finally:
        con.close()
```

Le `finally` garantit la fermeture même si la requête lève une exception. Une connexion laissée ouverte garde un verrou sur le fichier — et le prochain programme qui veut écrire reçoit `database is locked`.

Un programme complet

```python
import sqlite3

def moyennes_par_classe(chemin):
    con = sqlite3.connect(chemin)
    con.row_factory = sqlite3.Row
    try:
        return con.execute("""
            SELECT c.nom AS classe,
                   COUNT(DISTINCT e.id) AS nb_eleves,
                   ROUND(AVG(n.valeur), 2) AS moyenne
            FROM classe c
            JOIN eleve e ON e.classe_id = c.id
            JOIN note  n ON n.eleve_id  = e.id
            GROUP BY c.id
            ORDER BY moyenne DESC
        """).fetchall()
    finally:
        con.close()

if __name__ == "__main__":
    print(f"{'Classe':8} {'Élèves':>7} {'Moyenne':>8}")
    for r in moyennes_par_classe("ecole.db"):
        print(f"{r['classe']:8} {r['nb_eleves']:>7} {r['moyenne']:>8}")
```

```
Classe    Élèves  Moyenne
5TTR           7    13.35
4TTR           9    13.22
3TTR          10    13.14
6TTR           6    12.95
```

> 💡 **REMARQUE LA RÉPARTITION DU TRAVAIL.** Le regroupement, la moyenne et le tri sont faits par SQLite, pas par Python. C'est presque toujours le bon choix : le moteur est écrit pour ça, et il ne transporte que le résultat. Charger 367 points en Python pour les moyenner à la main serait plus long à écrire et plus lent à exécuter.

🧪 Exercices

Travaille sur une **copie** de `ecole.db`.

Exercice 1 — Lire ★★☆☆☆

Écris un programme qui affiche, pour une classe demandée à l'utilisateur (`input`), la liste de ses élèves par ordre alphabétique et leur nombre.

Contraintes : `sqlite3.Row`, un place-holder, la connexion fermée dans un `finally`. Si la classe n'existe pas, affiche un message clair plutôt qu'une liste vide.

Exercice 2 — Démontrer l'injection ★★★☆☆

a) Écris volontairement une fonction `chercher_mauvais(nom)` qui construit la requête avec une f-string. b) Trouve une saisie qui te renvoie **tous** les élèves. c) Trouve une saisie qui provoque une `OperationalError`. d) Réécris `chercher_bon(nom)` avec un place-holder et vérifie que les deux saisies précédentes ne donnent plus rien d'anormal. e) Explique en trois lignes, à un camarade, pourquoi le place-holder règle le problème alors que « filtrer les apostrophes » ne le règle pas.

Exercice 3 — Écrire ★★★☆☆

Écris `inscrire_eleve(nom, prenom, naissance, classe, cours_ids)` qui, en **une seule transaction** :

1. insère l'élève,
2. récupère son identifiant,
3. l'inscrit à tous les cours de la liste.

Si l'un des cours n'existe pas, **rien** ne doit être enregistré — ni l'élève, ni les autres inscriptions. Teste avec une liste contenant un identifiant valide et un identifiant inexistant, puis vérifie que l'élève n'a pas été créé.

Exercice 4 — Encoder des points ★★★☆☆

Écris un programme qui encode les points d'un cours pour toute une classe :

- il demande la classe et le cours ;

- il affiche les élèves **inscrits à ce cours** dans cette classe ;
- il demande une note pour chacun, en refusant toute valeur hors de 0-20 **côté Python** ;
- il enregistre tout en une transaction avec `executemany`.

Question : la base a déjà un `CHECK (valeur BETWEEN 0 AND 20)`. Pourquoi vérifier aussi côté Python ? Réponds en deux lignes dans un commentaire du code.

Exercice 5 — Bulletin en fichier ★★★★☆

Écris un programme qui génère un fichier `bulletin-6TTR.csv` contenant, pour chaque élève de la classe : nom, prénom, moyenne par cours (une colonne par cours), moyenne générale.

Utilise le module `csv`. Les élèves sans aucun point doivent apparaître avec des cellules vides, pas être omis — un `LEFT JOIN` sera nécessaire.

✍️ À retenir

- `sqlite3` est dans la **bibliothèque standard** : `import sqlite3`, rien à installer.
- `connect()` **crée** le fichier s'il n'existe pas — d'où les « bases vides » inexpliquées.
- `PRAGMA foreign_keys = ON` est à réémettre **à chaque connexion**.
- `fetchone()` renvoie un **tuple** : ne pas oublier `[0]`.
- `con.row_factory = sqlite3.Row` permet `ligne["nom"]` — à mettre par défaut.
- **Jamais de f-string dans une requête.** Les valeurs passent par `?` ou `:nom`, dans un tuple ou un dictionnaire.
- Le tuple d'un seul élément prend une virgule : `(valeur,)`.
- Un place-holder remplace une **valeur**, jamais un nom de table ou de colonne : pour ça, liste blanche.
- **Pas de `commit()`, pas d'écriture.**
- `with con:` gère la **transaction**, pas la fermeture : `close()` reste nécessaire.
- Laisse SQLite faire les regroupements et les calculs : c'est son métier.

Suite

Tu écris maintenant du SQL depuis Python. Le cours suivant, Utiliser un ORM avec SQLite : Peewee, montre l'approche inverse : manipuler des objets Python et laisser une bibliothèque écrire le SQL à ta place — avec ce qu'on y gagne et ce qu'on y perd.